THE UNIVERSITY of EDINBURGH

Distributed Systems
Fall 2024

Yuvraj Patel

Today’s Agenda

Mutual Exclusion

Deadlocks

Transactions

Why Mutual Exclusion?

Balance = Balance + 1 equivalent in assembly

```
mov 0x123, %eax
add %0x1, %eax
mov %eax, 0x123
```

Want 3 instructions to execute as an uninterruptable group

- Want them to be atomic; appears that all execute at once, or none execute

Uninterruptable group of code is called critical section

More general: Need mutual exclusion for critical sections

- If thread A is in critical section C, thread B isn't
- It is fine if other threads do unrelated work

Distributed Locking

Cannot share local lock variables

How do we support mutual exclusion in a distributed system?

Let us start simple with a central solution

Distributed Lock – Central Solution

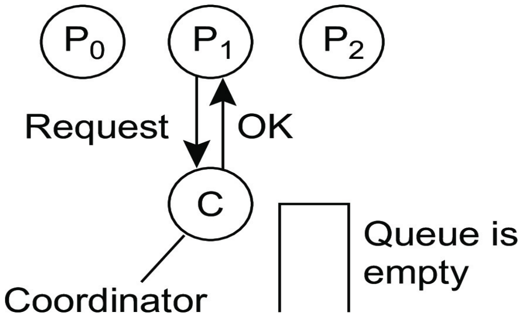
System Model

- Each pair of nodes is connected by reliable channels
- Messages are eventually delivered to recipient, and in FIFO order
- Nodes do not fail

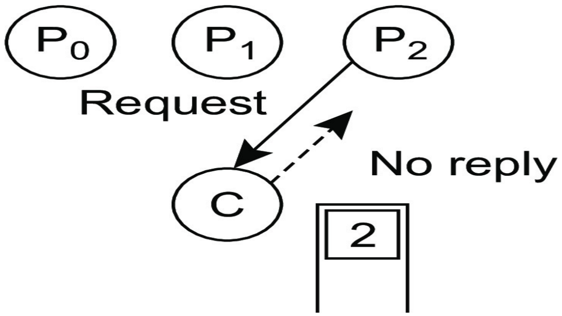
Central Solution

- Elect a central leader using election algorithm
- Leader keeps a queue of waiting requests from nodes who wish to access CS

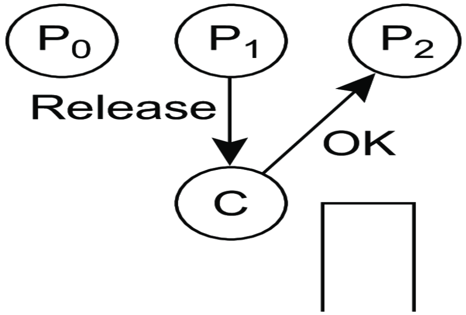
Distributed Lock – Central Solution (contd…)



Step 1



Step 2



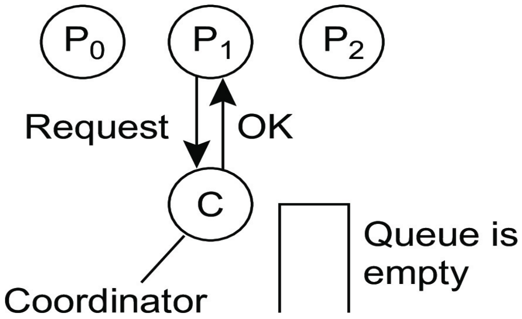
Step 3

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.

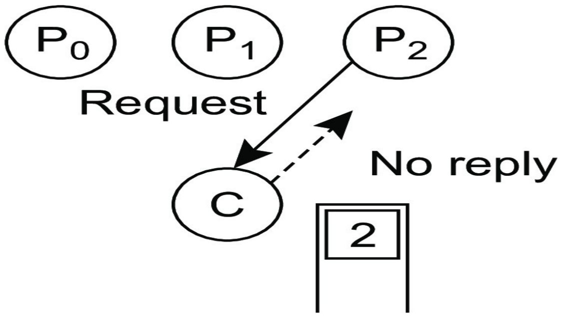
Node P2 then asks permission to access the same resource. The Coordinator does not reply.

When P1 releases the resource, it tells the coordinator, which then replies to P2.

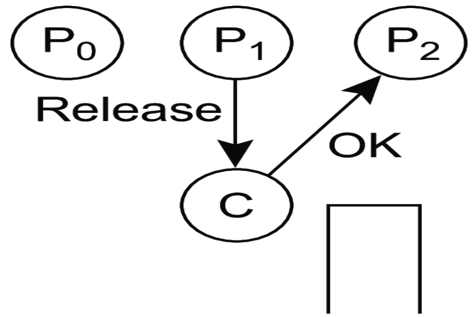
Distributed Lock – Central Solution (contd…)



Step 1



Step 2



Step 3

Node P1 asks the coordinator for permission to access a shared resource. Permission is granted.

Node P2 then asks permission to access the same resource. The Coordinator does not reply.

When P1 releases the resource, it tells the coordinator, which then replies to P2.

Problems????

Distributed Solution

Decentralized approach

- All nodes involved in the decision making of who should access the resource

Ricart-Agarwala Algorithm

Use the notion of causality – rely on logical timestamps

Ricart-Agrawala Algorithm

Requestor

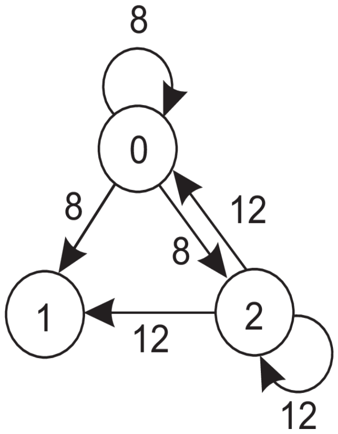
1. Broadcast a message to all receiver (including itself)
   <Resource-Name, Node-Name, Logical Timestamp>

3. Wait for all the OK messages.
4. Access resources once all receivers send OK message.
5. Release the resource; Send OK message to all queue entries

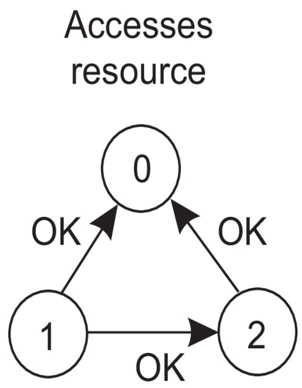
Receiver

2. If receiver not accessing the resource or does not want to access it, send OK message to the sender.
   If the receiver already has access to the resource. Do not reply. Queue the request.
   If receiver wants to access the resource but has not yet done, compare the timestamp
   - If incoming message has lower timestamp:
     - send OK message to the sender
   - Else:
     - Queue the incoming request and send nothing

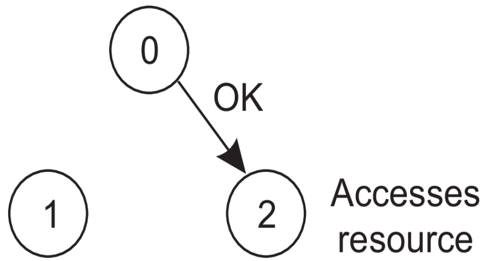
Ricart-Agrawala Algorithm Example



Step 1



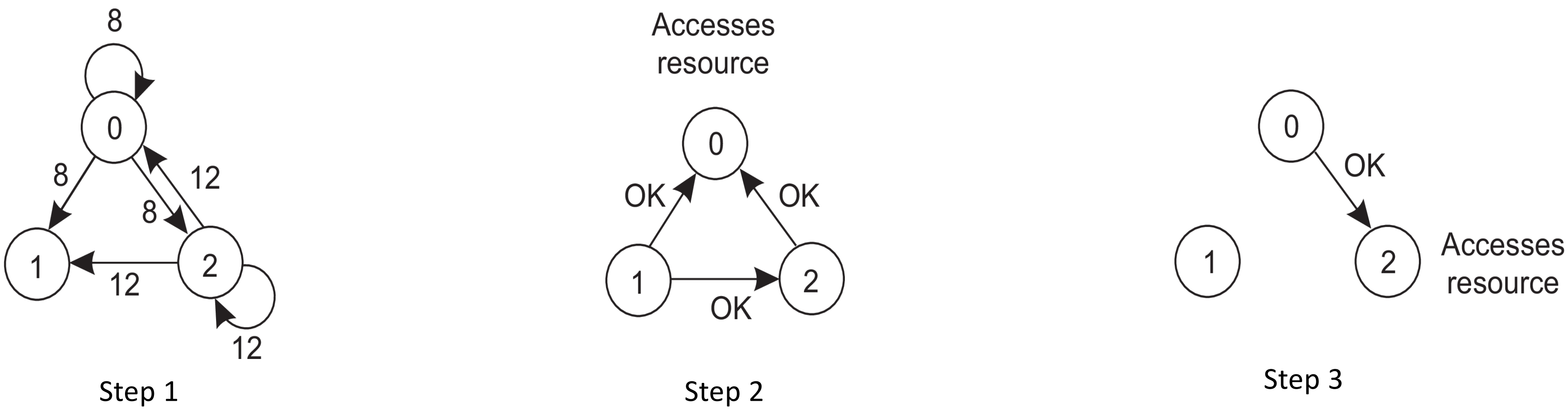
Step 2



Step 3

Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

Ricart-Agrawala Algorithm Example



Step 1: Two nodes want to access a shared resource at the same moment.
Step 2: P0 has the lowest timestamp, so it wins.
Step 3: When process P0 is done, it sends an OK message to P2. P2 can access the resource thereafter.

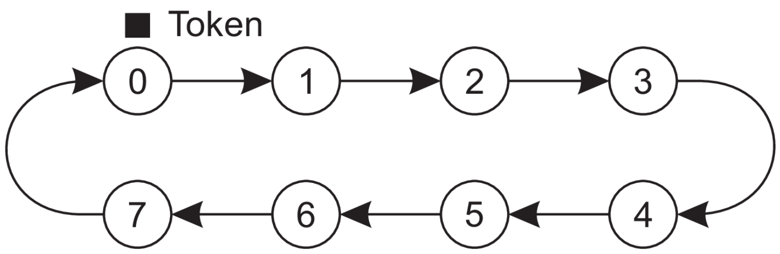
Problems????

Token Ring Algorithm

All nodes arranged in a ring fashion

Use token as a means of ownership

- Whosoever has the token can access the resource
- If no access needed, pass it on to the neighbor
- Token gets passed to all the nodes



Problems????

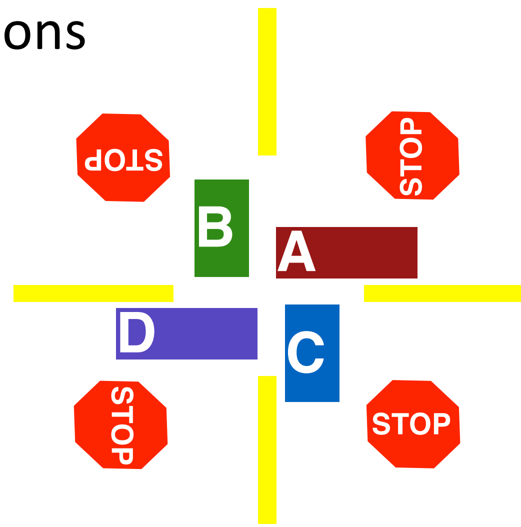
Deadlocks

No progress can be made because two or more nodes are each waiting for another to take some action and thus each never does

Deadlocks can only happen with these four conditions

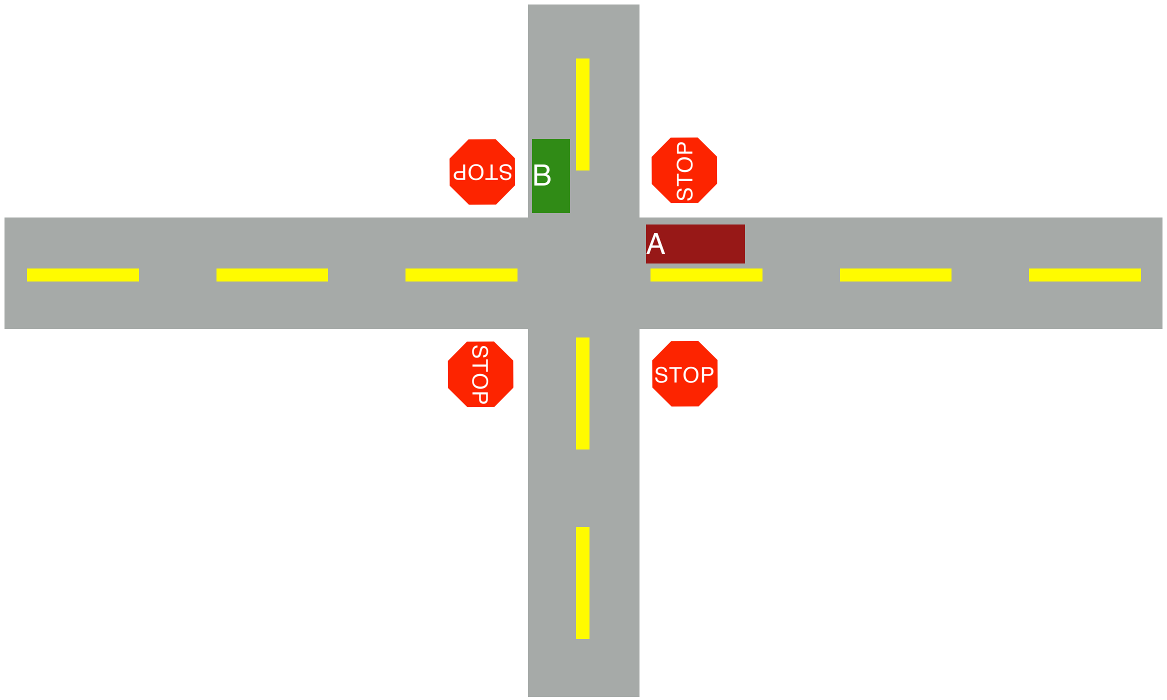
1. Mutual Exclusion
2. Hold-and-wait
3. No preemption
4. Circular Wait

Leads to safety property violation



Deadlock Example – Real World Case

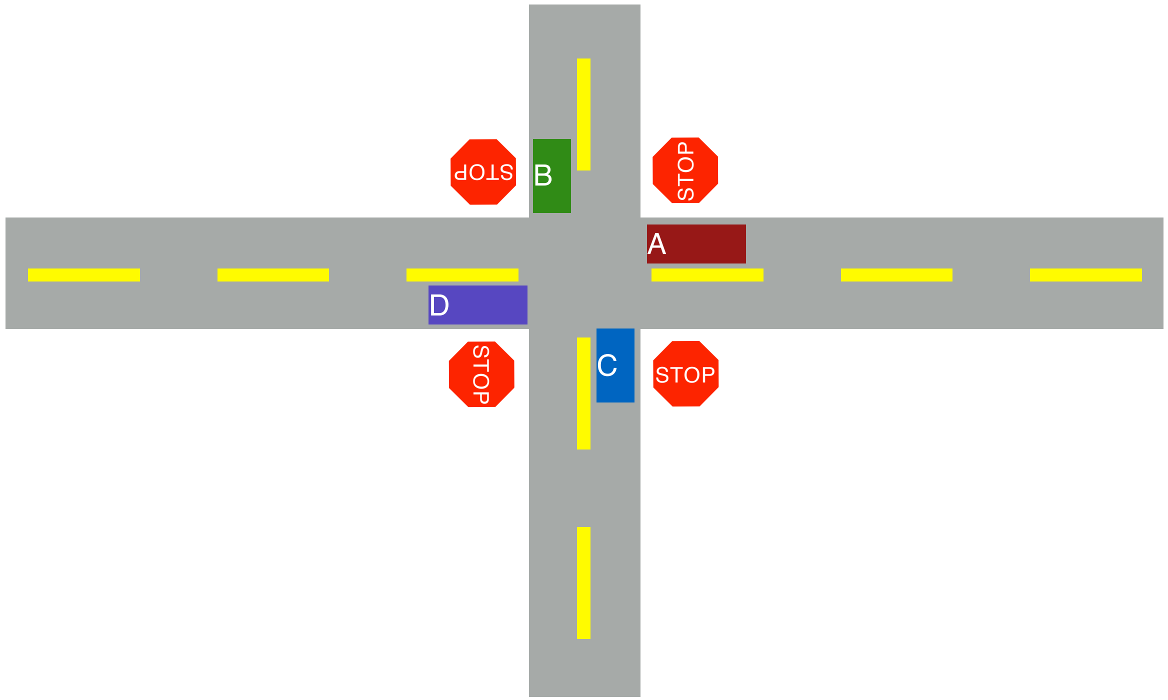
Both cars arrive at same time
Is this deadlocked?



Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlock Example – Real World Case (contd..)

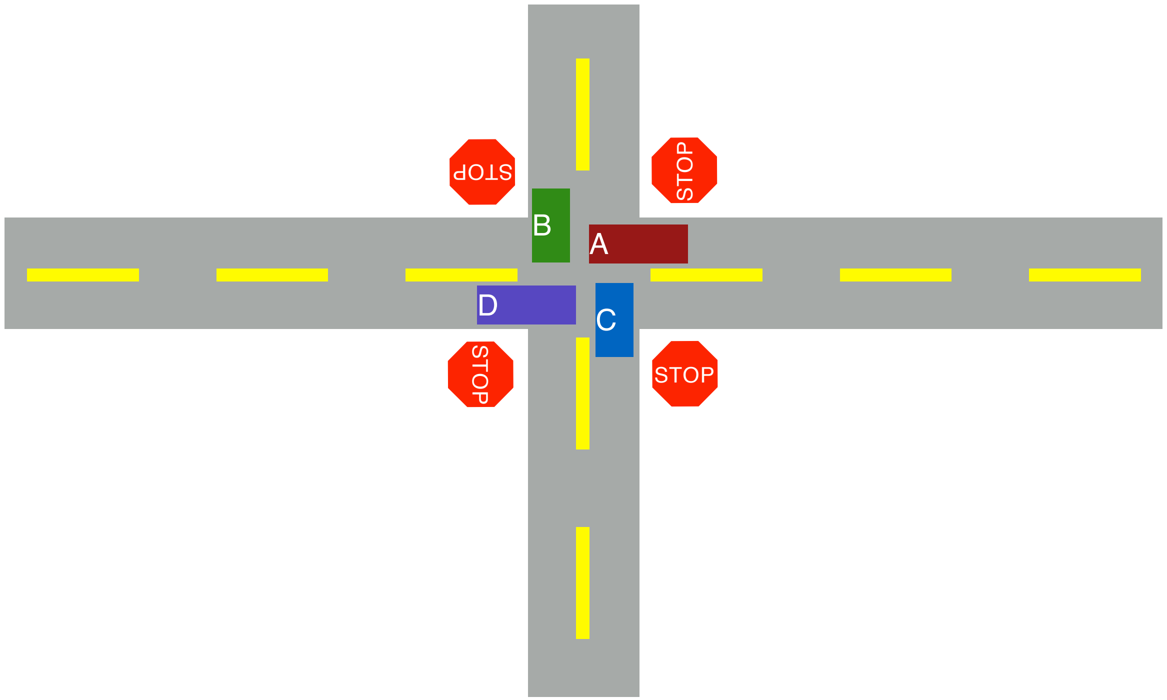
4 cars arrive at same time
Is this deadlocked?



Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlock Example – Real World Case (contd..)

4 cars arrive at same time
Is this deadlocked?

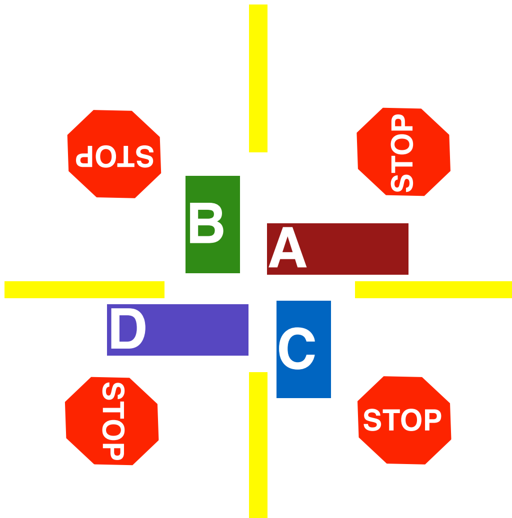


Conditions necessary for a deadlock:
1. mutual exclusion
2. hold-and-wait
3. no preemption
4. circular wait

Deadlocks Handling

Two main strategies

- Prevention – Eliminate any one condition
- Detect and Handle



Eliminate Hold-And-Wait Condition

Problem: Nodes hold resources while waiting for additional resources

Deadlock Prevention Strategy

- Acquire all the locks atomically
- Can release locks over time, but cannot acquire again until all locks have been released

How?

- Use a meta lock

Eliminate Hold-And-Wait Condition (contd…)

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

Eliminate Hold-And-Wait Condition (contd…)

```
lock(&meta);
lock(&L1);
lock(&L2);
lock(&L3);
…
unlock(&meta);
// CS1
unlock(&L1);
// CS 2
Unlock(&L2);
```

```
lock(&meta);
lock(&L2);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);

// CS2
Unlock(&L2);
```

```
lock(&meta);
lock(&L1);
unlock(&meta);

// CS1
unlock(&L1);
```

Disadvantages

- Must know ahead of time which locks will be needed
- Must be conservative (acquire any lock possibly needed)
- Degenerates to just having one big lock (reduces concurrency)

Eliminate No Preemption Condition

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can’t get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Eliminate No Preemption Condition (contd…)

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can't get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Livelock:
No processes make progress, but state of involved processes constantly changes
Classic solution: Exponential random back-off

Eliminate No Preemption Condition

Problem: Locks cannot be forcibly removed from nodes that are held

Strategy: If thread can't get what it wants, release what it holds

```
top:
        lock(A);
        if (trylock (B) == -1) {
                unlock(A);
                goto top;
        }
```

Other strategy: Use timeouts instead of trylock. If lock not acquired within a timeout, release locks

Problem: How long should be the timeout?

Detect Circular Wait Dependency

Another strategy is to detect deadlocks

Find cycles in the wait graphs

- Take snapshots using Global Snapshot Algorithm
- Detect cycles in the snapshot
- Abort tasks to break the cycle

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
- Durability (Persistence) – Remember Everything
  - Once a transaction commits, the changes are permanent

Issues with Transactions – Lost-Update

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)*	
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