

CS 425 / ECE 428

Distributed Systems

Fall 2026

Aishwarya Ganesan & Ram Kesavan

Lecture 20-21: RPCs, Concurrency Control, & Transactions

Ack: Indranil Gupta

All slides © IG

Why RPCs

- **RPC** = Remote Procedure Call
- Proposed by Birrell and Nelson in 1984
- Important abstraction for processes to call functions in other processes
- Allows code reuse
- Implemented and used in most distributed systems, including cloud computing systems
- Counterpart in Object-based settings is called RMI (Remote Method Invocation)

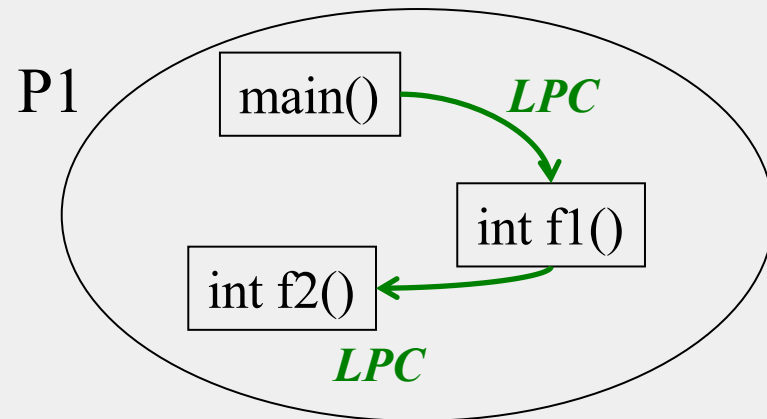
Local Procedure Call (LPC)

- Call from one function to another function within the same process
 - Uses stack/registers to pass arguments and return values
 - Accesses objects via pointers (e.g., C) or by reference (e.g., Java)
- LPC has *exactly-once* semantics
 - If process is alive, called function executed exactly once

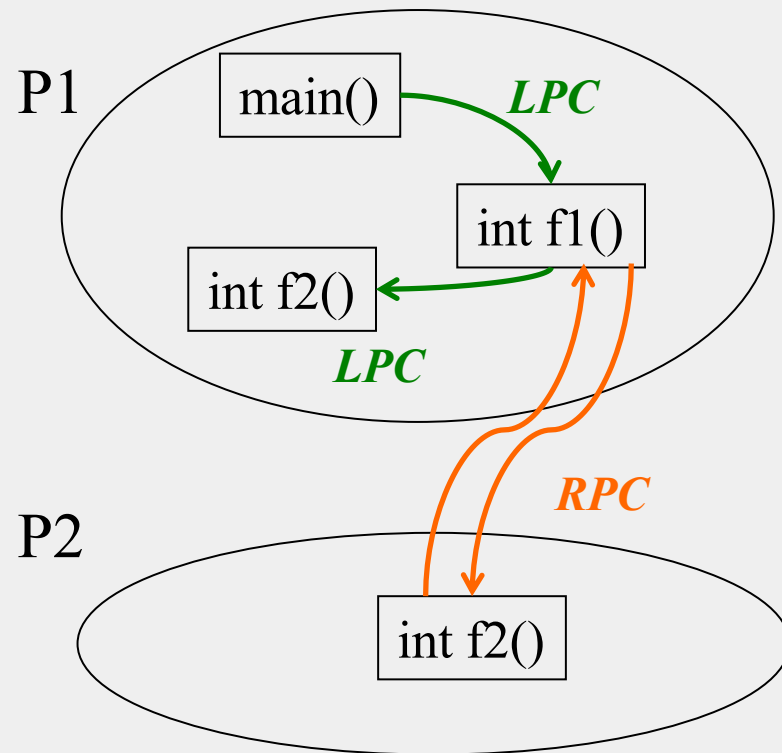
Remote Procedure Call

- Call from one function to another function, where caller and callee function reside in different processes
 - Function call crosses a process boundary
 - Accesses procedures via global references
 - Can't use pointers across processes since a reference address in process P1 may point to a different object in another process P2
 - E.g. procedure address = IP + port + procedure number
- Similarly, RMI (Remote Method Invocation) in Object-based settings

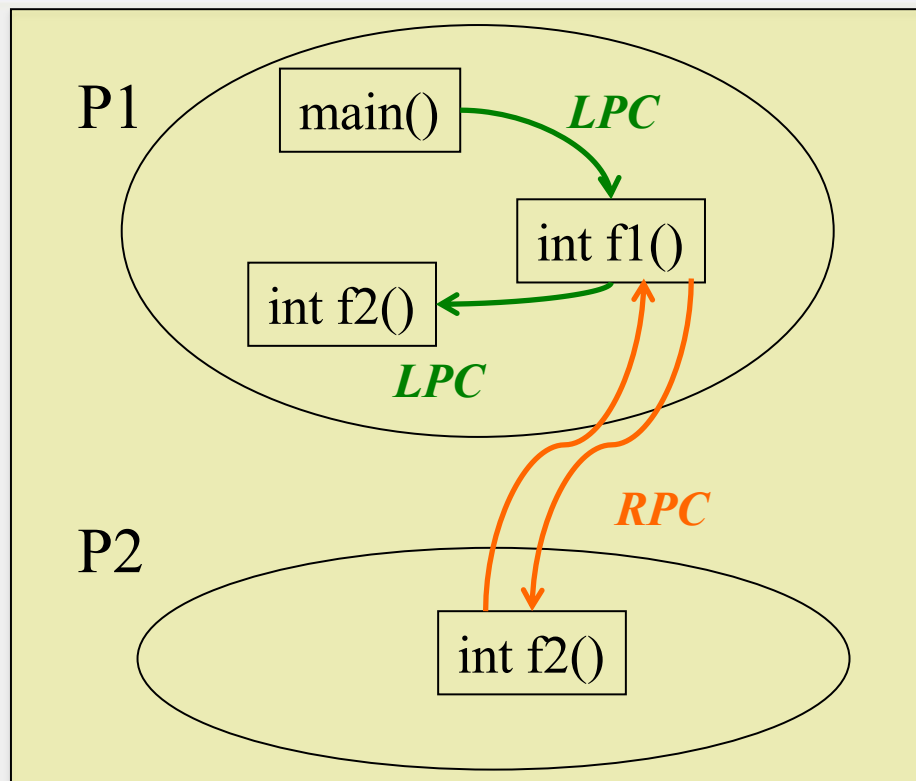
LPCs



RPCs

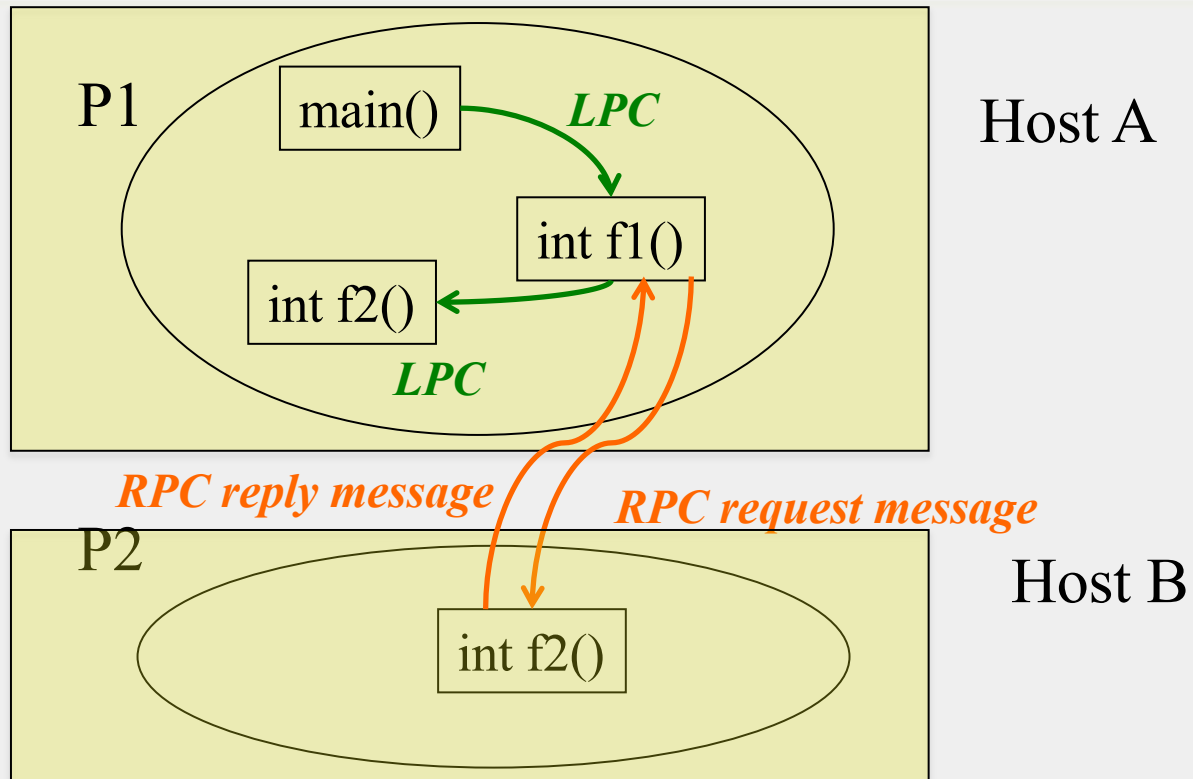


RPCs



Host A

RPCs



RPC Call Semantics

- Under failures, hard to guarantee exactly-once semantics
- Function may not be executed if
 - Request (call) message is dropped
 - Reply (return) message is dropped
 - Callee process fails before executing called function
 - Callee process fails after executing called function
 - Hard for caller to distinguish these cases
- Function may be executed multiple times if
 - Request (call) message is duplicated

Implementing RPC Call Semantics

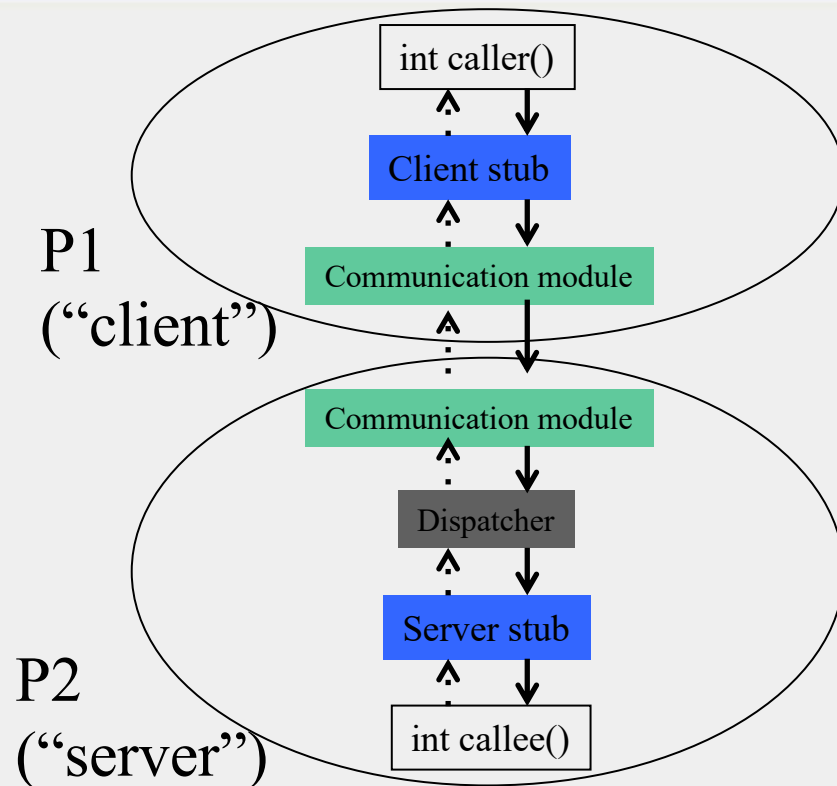
- Possible semantics
 - At least once semantics (e.g., Sun RPC)
 - At most once semantics (e.g., Java RMI)
 - Maybe, i.e., best-effort (e.g., CORBA)

Retransmit request	Filter duplicate requests	Re-execute function or retransmit reply	RPC Semantics
Yes	No	Re-execute	At least once
Yes	Yes	Retransmit	At most once
No	NA	NA	Maybe

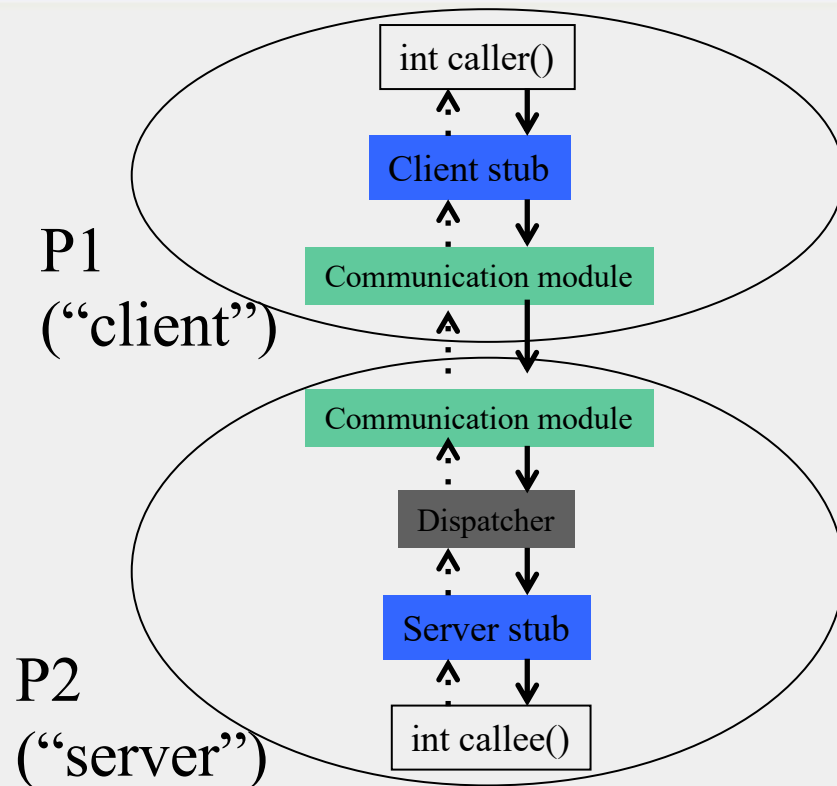
Idempotent Operations

- Idempotent operations are those that can be repeated multiple times, without any side effects
- Examples (x is a variable in P2)
 - $x=1;$
 - $x=(\text{argument})\ y;$
- Non-examples
 - $x=x+1;$
 - $x=x*2$
- Idempotent operations can be used with at-least-once semantics

Implementing RPCs



RPC Components



Client

- **Client stub:** has same function signature as `callee()`
 - Allows same `caller()` code to be used for LPC and RPC
- **Communication Module:** Forwards requests and replies to appropriate hosts

Server

- **Dispatcher:** Selects which server stub to forward request to
- **Server stub:** calls `callee()`, allows it to return a value

Poll everywhere



Generating Code

- Programmer only writes code for caller function and callee function
- Code for remaining components all **generated automatically** from function signatures (or object interfaces in Object-based languages)
 - E.g., Sun RPC system: Sun XDR interface representation fed into rpcgen compiler
- These components together part of a Middleware system
 - E.g., CORBA (Common Object Request Brokerage Architecture)
 - E.g., Sun RPC
 - E.g., Java RMI

Marshalling

- Different architectures use different ways of representing data
 - **Big endian**: Hex 12-AC-33 stored with 12 in lowest address, then AC in next higher address, then 33 in highest address
 - IBM z, System 360
 - **Little endian**: Hex 12-AC-33 stored with 33 in lowest address, then AC in next higher address, then 12
 - Intel
- Caller (and callee) process uses its own *platform-dependent* way of storing data
- Middleware has a common data representation (CDR)
 - *Platform-independent*

Marshalling (2)

- Middleware has a common data representation (CDR)
 - Platform-independent
- Caller process converts arguments into CDR format
 - Called “Marshalling”
- Callee process extracts arguments from message into its own platform-dependent format
 - Called “Unmarshalling”
- Return values are marshalled on callee process and unmarshalled at caller process

Next

- Now that we know RPCs, we can use them as a building block to understand transactions

Transaction

- Series of operations executed by client
- Each operation is an RPC to a server
- Transaction either
 - completes and *commits* all its operations at server
 - Commit = reflect updates on server-side objects
 - Or *aborts* and has no effect on server
- Ops within a transaction are clearly demarcated:
start/open & close/end/commit

Example: Transaction

Client code:

RPCs

```
int transaction_id = openTransaction();  
x = server.getFlightAvailability(ABC, 123, date);  
if (x > 0)  
    y = server.bookTicket(ABC, 123, date);  
server.putSeat(y, "aisle");  
// commit entire transaction or abort  
closeTransaction(transaction_id);
```

Example: Transaction

Client code:

RPCs

```
int transaction_id = openTransaction();  
x = server.getFlightAvailability(ABC, 123, date); // read(ABC, 123, date)  
if (x > 0)  
    y = server.bookTicket(ABC, 123, date); // write(ABC, 123, date)  
    server.putSeat(y, "aisle"); // write(more ticket info)  
    // commit entire transaction or abort  
closeTransaction(transaction_id);
```

Atomicity and Isolation

- **Atomicity:** All or nothing principle: a transaction should either i) complete successfully, so its effects are recorded in the server objects; or ii) the transaction has no effect at all.
- **Isolation:** Need a transaction to be indivisible (atomic) from the point of view of other transactions
 - No access to intermediate results/states of other transactions
 - Free from interference by operations of other transactions
- But...
 - Clients and/or servers might crash
 - Transactions could run concurrently, i.e., with multiple clients
 - Transactions may be distributed, i.e., across multiple servers

ACID Properties for Transactions

- **A**tomicity: All or nothing
- **C**onsistency: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- **I**solation: Each transaction must be performed without interference from other transactions, i.e., non-final effects of a transaction must not be visible to other transactions.
- **D**urability: After a transaction has completed successfully, all its effects are saved in permanent storage.

Multiple Clients, One Server

- What could go wrong?

1. Lost Update Problem

Transaction T1

`x = getSeats(ABC123);`

`// x = 10`

`if(x > 1)`

`x = x - 1;`

`write(x, ABC123);`

`commit`

Transaction T2

`x = getSeats(ABC123);`

`if(x > 1)   // x = 10`

`x = x - 1;`

`write(x, ABC123);`

`commit`

At Server: seats = 10

T1's or T2's update was lost!

seats = 9

seats = 9

2. Inconsistent Retrieval Problem

Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);  
    // ABC123 = 5 now  
  
write(y+5, ABC789);  
  
commit
```

Transaction T2

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
    // x = 5, y = 15  
print("Total:" x+y);  
    // Prints "Total: 20"  
  
commit
```

At Server:

ABC123 = 10
ABC789 = 15

**T2's sum is the wrong value!
Should have been "Total: 25"**

Next

- How to prevent transactions from affecting each other

Concurrent Transactions

- To prevent transactions from affecting each other
 - Could execute them one at a time at server
 - But reduces number of concurrent transactions
 - *Transactions per second* directly related to revenue of companies
 - This metric needs to be maximized
- Goal: increase concurrency (tps) while maintaining correctness (ACID)

Serial Equivalence

- An interleaving (say O) of transactions' operations is serially equivalent iff (if and only if):
 - There is some ordering (O') of those transactions, one after the other, such that
 - operations execute one at a time within each transaction, which
 - **yields the same end-state (for all objects on the server and clients) as with O**
- E.g. Transactions A, B, and C have 5 ops each.
 - O' : B1,..., B5, C1,..., C5, A1,..., A5.
 - An interleaving O of these 15 ops is serially equivalent iff it yields the same end-state as O' .

Checking for Serial Equivalence

- An operation has an **effect** on
 - The server object if it is a write
 - The client (returned value) if it is a read
- Two ops are said to be conflicting ops, if their *combined effect* depends on the **order** in which they are executed
 - read(x) and write(x)
 - write(x) and read(x)
 - write(x) and write(x)
 - NOT read(x) and read(x): *reordering doesn't change the combined effect*
 - NOT read(x) and write(y): *reordering doesn't change the combined effect*
- So, only 2 types of conflicts: read-write OR write-write

Checking for Serial Equivalence (2)

- *An ordering across 2 transactions is serially equivalent iff all pairs (pair: one op from each transaction) of conflicting ops are executed in the same (transaction) order.*
 - Take all pairs of conflicting ops, one from T1 and one from T2
 - If the T1 op was reflected first on the server, mark the pair as “(T1, T2)”, otherwise mark it as “(T2, T1)”
 - All pairs should be marked as either “(T1, T2)” or all pairs should be marked as “(T2, T1)”.

1. Lost Update Problem – Caught!

Transaction T1

x = getSeats(ABC123);

// x = 10

if(x > 1)

x = x - 1;

write(x, ABC123);

commit

Transaction T2

x = getSeats(ABC123);

if(x > 1) // x = 10

(T1, T2)

x = x - 1;

write(x, ABC123);

commit

At Server: seats = 10

T1's or T2's update was lost!

seats = 9

seats = 9

2. Inconsistent Retrieval Problem – Caught!

Transaction T1

```
x = getSeats(ABC123);  
y = getSeats(ABC789);  
write(x-5, ABC123);
```

```
write(y+5, ABC789);
```

```
commit
```

Transaction T2

(T1, T2)

x = getSeats(ABC123);

y = getSeats(ABC789);

(T2, T1) // x = 5, y = 15

```
print("Total:" x+y);
```

```
// Prints "Total: 20"
```

```
commit
```

At Server:

ABC123 = 10

ABC789 = 15

**T2's sum is the wrong value!
Should have been "Total: 25"**

What's Our Response?

- At commit point of a transaction T, check for serial equivalence with all other transactions
 - Can limit to transactions that overlapped in time with T
- If not serially equivalent
 - Abort T
 - Roll back (undo) any writes that T did to server objects

Can We do better?

- Aborting => wasted work
- Can you prevent violations from occurring?

Two Approaches

- Preventing isolation from being violated can be done in two ways
 1. **Pessimistic** concurrency control
 2. **Optimistic** concurrency control

Pessimistic vs. Optimistic

- **Pessimistic**: assume the worst, prevent transactions from accessing the same object
 - E.g., Locking
- **Optimistic**: assume the best, allow transactions to write, but check later
 - E.g., Check at commit time, multi-version approaches

Pessimistic: Exclusive Locking

- Each object has a lock
- At most one transaction can be inside lock
- Before reading or writing object O, transaction T must call **lock(O)**
 - Blocks if another transaction already inside lock
- After entering lock T can read and write O multiple times
- When done (or at commit point), T calls **unlock(O)**
 - If other transactions waiting at lock(O), allows one of them in
- Sound familiar? (This is Mutual Exclusion!)

Can we improve concurrency?

- More concurrency => more transactions per second => more revenue (\$\$\$)
- Real-life workloads have a lot of read-only or read-mostly transactions
 - Exclusive locking reduces concurrency
 - Hint: Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

Another Approach: Read-Write Locks

- Each object has a lock that can be held in one of two modes
 - **Read mode**: multiple transactions allowed in
 - **Write mode**: exclusive lock
- Before first reading O, transaction T calls `read_lock(O)`
 - T allowed in only if *all* transactions inside lock for O all entered via read mode
 - Not allowed if *any* transaction inside lock for O entered via write mode

Read-Write Locks (2)

- Before first writing O, call `write_lock(O)`
 - Allowed in only if no other transaction inside lock
- If T already holds `read_lock(O)`, and wants to write, call `write_lock(O)` to *promote* lock from read to write mode
 - Succeeds only if no other transactions in write mode or read mode
 - Otherwise, T blocks
- `Unlock(O)` called by transaction T releases any lock on O by T

Guaranteeing Serial Equivalence With Locks

- **Two-phase locking**
 - A transaction cannot acquire (or promote) any locks after it has started releasing locks
 - Transaction has two phases
 1. Growing phase: only acquires or promotes locks
 2. Shrinking phase: only releases locks
 - **Strict two phase locking**: releases locks only at commit point

Why Two-phase Locking $\Rightarrow$ Serial Equivalence?

- Proof by contradiction
- Assume two phase locking system where serial equivalence is violated for some two transactions T1, T2
- Two facts must then be true:
 - (A) For some object O1, there were conflicting operations in T1 and T2 such that the time ordering pair is (T1, T2)
 - (B) For some object O2, the conflicting operation pair is (T2, T1)
 - (A) $\Rightarrow$ T1 released O1's lock and T2 acquired it after that
 $\Rightarrow$ T1's shrinking phase is before or overlaps with T2's growing phase
- Similarly, (B) $\Rightarrow$ T2's shrinking phase is before or overlaps with T1's growing phase
- But both these cannot be true!

Downside of Locking

- Deadlocks!

Downside of Locking – Deadlocks!

Transaction T1

Lock(ABC123);

x = write(10, ABC123);

Lock(ABC789);

// Blocks waiting for T2

...

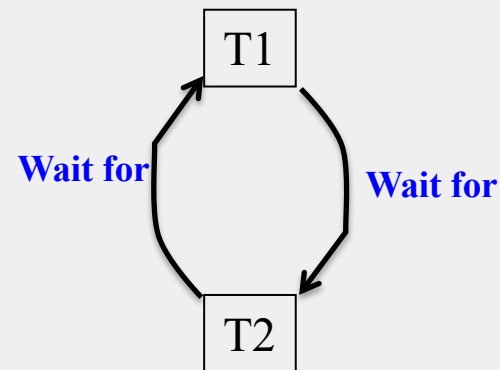
Transaction T2

Lock(ABC789);

y = write(15, ABC789);

Lock(ABC123);

... // Blocks waiting for T1



When do Deadlocks Occur?

- 3 necessary conditions for a deadlock to occur
 1. Some objects are accessed in exclusive lock modes
 2. Transactions holding locks cannot be preempted
 3. There is a circular wait (cycle) in the Wait-for graph
- “Necessary” = if there’s a deadlock, these conditions are all definitely true
- (Conditions not sufficient: if they’re present, it doesn’t imply a deadlock is present.)

Combating Deadlocks

1. Lock **timeout**: abort transaction if lock cannot be acquired within timeout
 - ☹ Expensive; leads to wasted work
2. Deadlock **Detection**:
 - keep track of Wait-for graph (e.g., via Global Snapshot algorithm), and
 - find cycles in it (e.g., periodically)
 - If find cycle, there's a deadlock $\Rightarrow$ Abort one or more transactions to break cycle
 - ☹ Still allows deadlocks to occur

Combating Deadlocks (2)

3. Deadlock Prevention

- Set up the system so one of the *necessary conditions* is violated
 1. *Some objects are accessed in exclusive lock modes*
 - Fix: Allow read-only access to objects
 2. *Transactions holding locks cannot be preempted*
 - Fix: Allow preemption of some transactions
 3. *There is a circular wait (cycle) in the Wait-for graph*
 - Fix: Lock all objects in the beginning; if fail any, abort transaction
=> No cycles in Wait-for graph

Next

- Can we allow more concurrency?
- Optimistic Concurrency Control

Optimistic Concurrency Control

- Increases concurrency more than pessimistic concurrency control
- Increases transactions per second
- For non-transaction systems, increases operations per second and lowers latency
- Used in Dropbox, Google apps, Wikipedia, key-value stores like Cassandra, Riak, and Amazon's Dynamo
- Preferable than pessimistic when conflicts are *expected to be rare*
 - But still need to ensure conflicts are caught!

First-cut Approach

- Most basic approach
 - Write and read objects at will
 - Check for serial equivalence at commit time
 - If abort, roll back updates made
 - An abort may result in other transactions that read dirty data, also being aborted
 - Any transactions that read from *those* transactions also now need to be aborted
- ☹ *Cascading aborts*

Second approach: Timestamp Ordering

- Assign each transaction an id
- Transaction id determines its position in **serialization order**
- Ensure that for a transaction T, both are true:
 1. T's **write** to object O allowed only if **transactions that have read or written O had lower ids than T.**
 2. T's **read** to object O is allowed only if **O was last written by a transaction with a lower id than T.**
- Implemented by maintaining read and write timestamps for the object
- If rule violated, abort!
 - Can we do better?

Third Approach: Multi-version Concurrency Control

- For each object
 - A per-transaction version of the object is maintained
 - Marked as *tentative* versions
 - And a **committed** version
- Each tentative version has a timestamp
 - Some systems maintain both a read timestamp and a write timestamp
- On a read or write, find the “correct” tentative version to read or write from
 - “Correct” based on transaction id, and tries to make transactions only read from “immediately previous” transactions

Eventual Consistency...

- ...in key-value stores...
- ... is a form of optimistic concurrency control
 - In Cassandra key-value store
 - In DynamoDB key-value store
 - In Riak key-value store
- But since non-transaction systems, the optimistic approach looks different

Eventual Consistency in Cassandra and DynamoDB

- Only one version of each data item (key-value pair)
- Last-write-wins (LWW)
 - Timestamp, typically based on *physical time*, used to determine whether to overwrite
 - if(new write's timestamp > current object's timestamp)
 overwrite;
else
 do nothing;
- With unsynchronized clocks
 - If two writes are close by in time, older write might have a newer timestamp, and might win

Eventual Consistency in Riak Key-value Store

- An older version of Riak uses **vector clocks**! (Should sound familiar to you!)
- Implements causal ordering
- Uses vector clocks to detect whether
 1. New write is strictly newer than current value, or
 2. If new write conflicts with existing value
- In case (2), a *sibling value* is created
 - Resolvable by user, or automatically by application (but not by Riak)
- To prevent vector clocks from getting too many entries
 - Size-based pruning
- To prevent vector clocks from having entries updated a long-time ago
 - Time-based pruning

Summary

- RPCs and RMIs
- Transactions
- Serial Equivalence
 - Detecting it via conflicting operations
- Pessimistic Concurrency Control: locking
- Optimistic Concurrency Control