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)

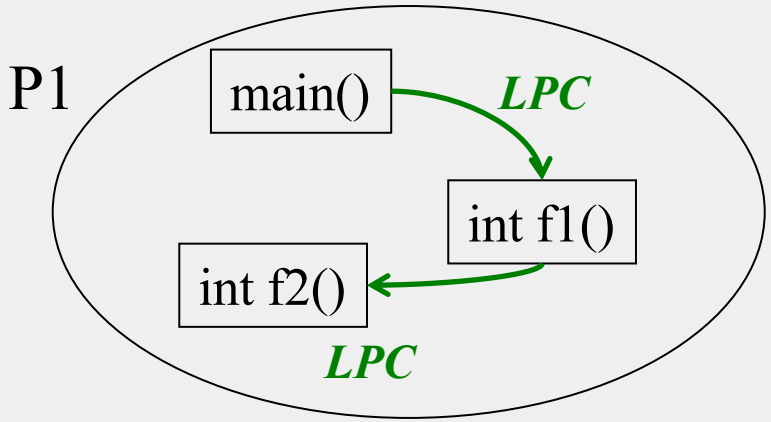
Local Procedure Call (LPC)

- 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

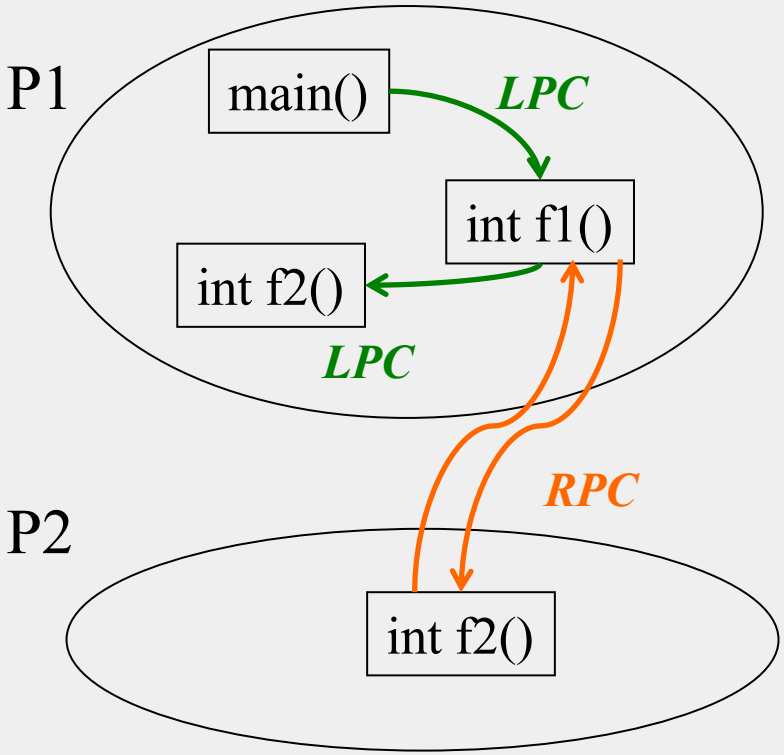
Remote Procedure Call

- 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

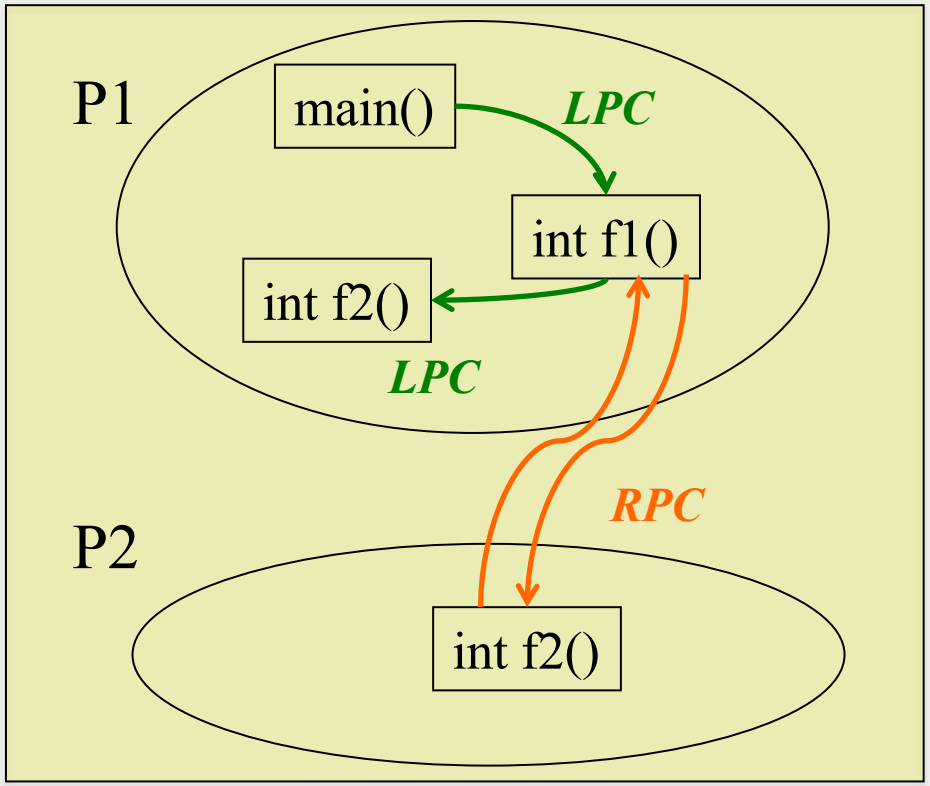
LPCs



RPCs

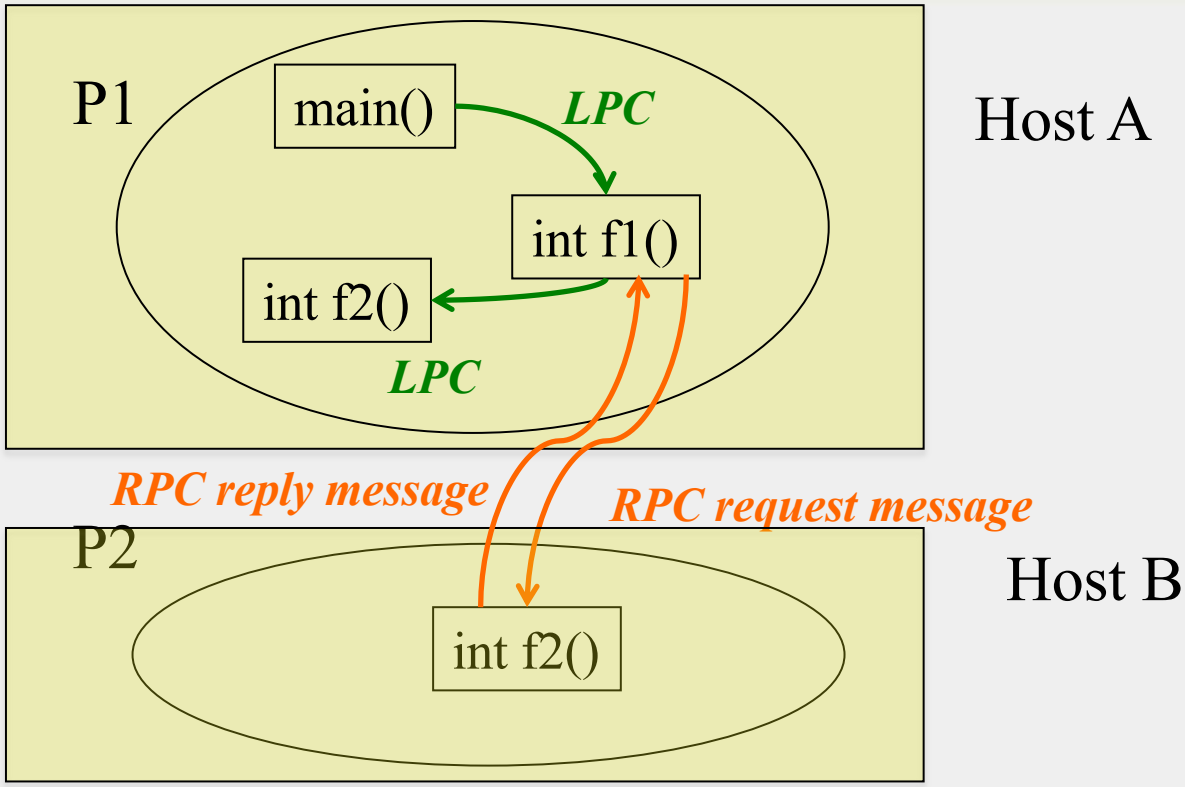


RPCs



Host A

RPCs


P1
main()
LPC
int f1()
int f2()
LPC
RPC reply message
RPC request message
P2
int f2()
Host A
Host B

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

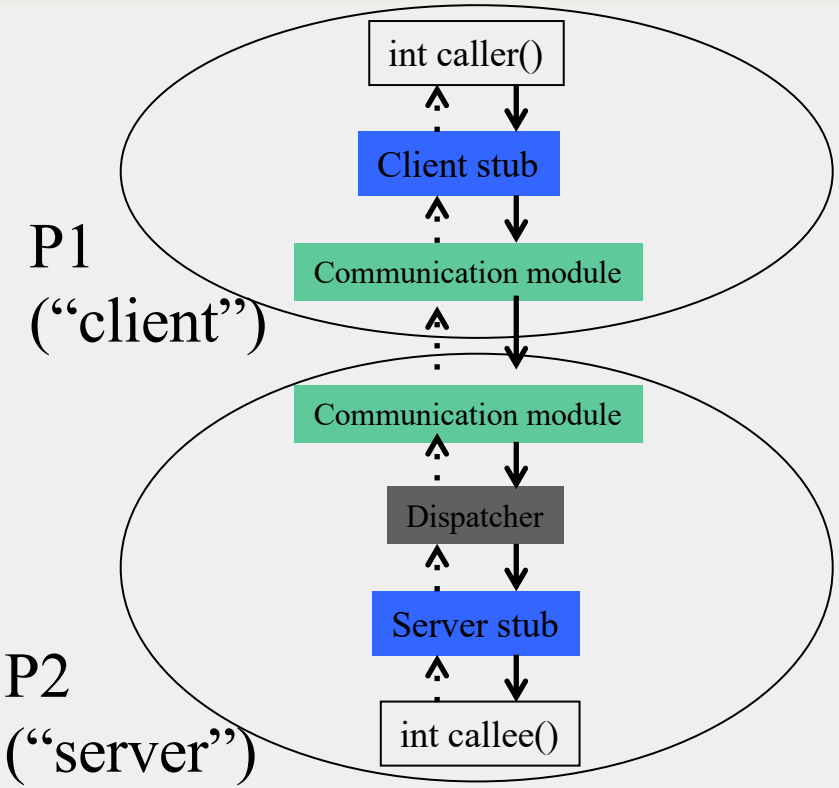
- Possible semantics
  - At least once semantics (e.g., Sun RPC)
  - At most once semantics (e.g., Java RMI)
  - Maybe, i.e., best-effort (e.g., CORBA)

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

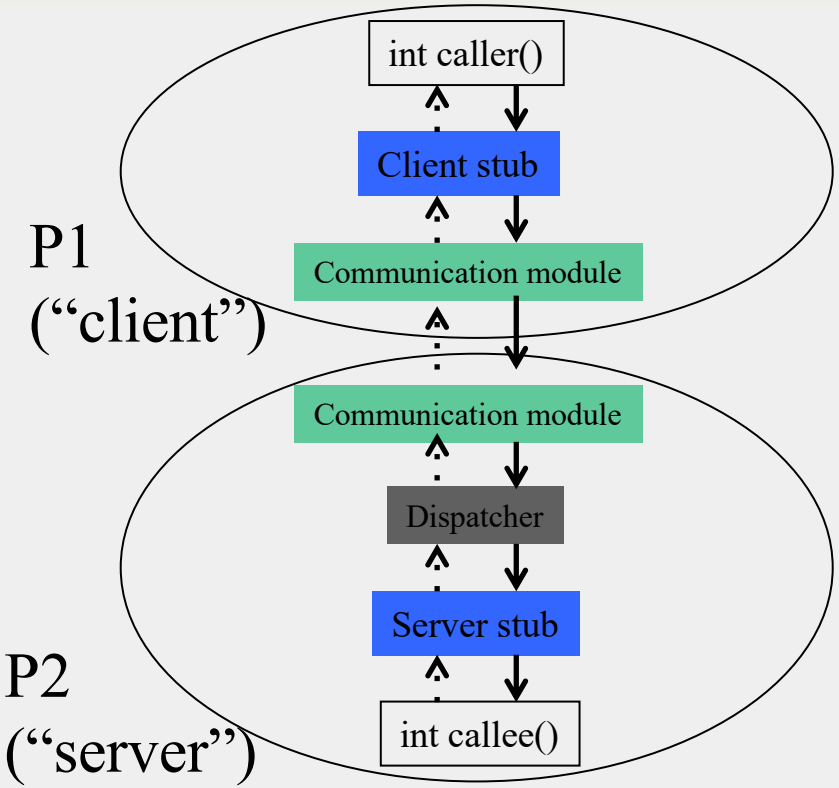
Idempotent Operations

- Idempotent operations are those that can be repeated multiple times, without any side effects
- Examples (x is a variable in P2)
  - x=1;
  - x=(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

Can the callee (P2, the server) invoke an RPC to another process (P3) as part of servicing P1's RPC?

Responses hidden

Yes. RPCs can cascade. So, P2 can become a client for P3 server.

Heck no! P2 cannot be both a client and a server...

Poll everywhere

Can the caller (P1, client) invoke an RPC to a different server (P3)?

Responses hidden

Sure. P1 can invoke RPC2 to P2, and then invoke RPC3 to P3, and so on, as long as they are sequentially done.

Sure. In fact, P1 can invoke 2+ RPCs in parallel to different servers!

No way! A client process shouldn't be allowed to talk to more than one server process.

Auto-Generated Code

- Many popular libraries/frameworks:
  - gRPC (.proto file): Google's protocol buffers
  - Apache Thrift (.thrift file): originally built by FB
  - xmlrpc-c, json-rpc, Cap'nProto, tRPC, etc.
- Some frameworks (gRPC/Thrift) use a formal Interface Definition Language (IDL)
  - Programmers write (1) IDL spec (.proto/.thrift file) (2) caller and callee functions
  - IDL spec used to **generate code for both stubs & dispatcher, and some fraction of the communication modules**

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
- Communication is made possible by translating to/from a common data format
  - *Platform-independent*

Marshalling (2)

- The stubs use a platform-independent data format
  - gRPC: protocol buffers
  - Apache Thrift: TBinaryProtocol or TCompactProtocol
  - xmlrpc-c/jsonrpc: XML/JSON string
- Marshalling: Client stub converts arguments into this format
  - aka serialization
- Unmarshalling: Server stub extracts and converts arguments into its own platform-dependent format
  - aka deserialization
- Response path: marshalling by server stub & unmarshalling by server stub

Next

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

Transaction

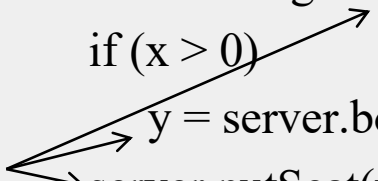
- Series of operations executed by client
  - The ops are demarcated: start/open & close/end/commit
- Each op is an RPC to the server
  - Local ops are not interesting for this discussion
- Transaction either
  - closes and *commits* all its ops at server
    - Commit = reflect updates on server-side objects
  - Or *aborts* and has no effect on server

Example Transaction: Book online airline ticket

Client code:

```
int transaction_id = openTransaction( );
x = server.getFlightAvailability(ABC, 123, date);
if (x > 0)
    y = server.bookTicket(ABC, 123, date);
server.putSeat(y, "aisle");
// more RPCs: e.g. provide payment info
// commit entire transaction or abort
closeTransaction(transaction_id);
```

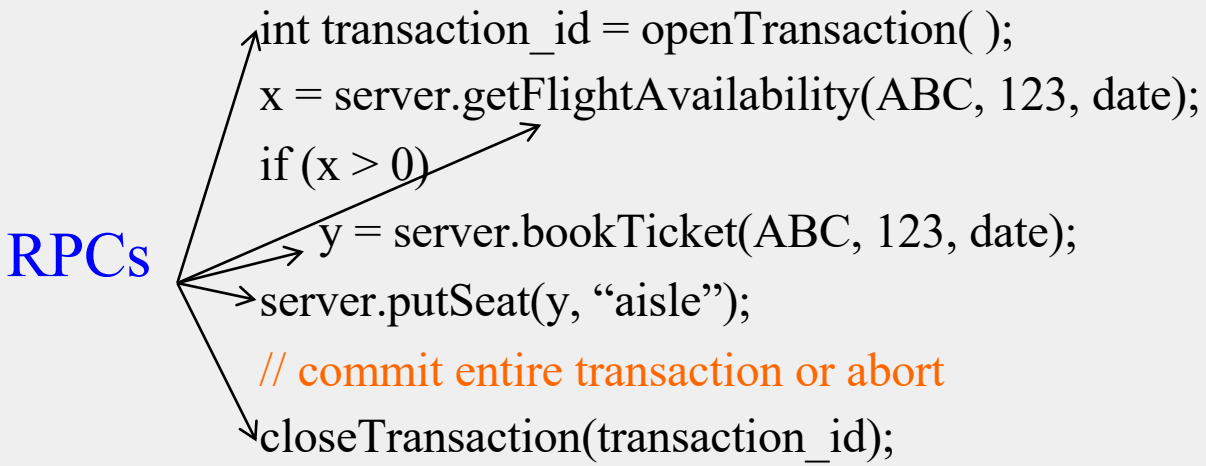
RPCs



Are open & close RPCs too?

Example: Transaction

Client code:



What’s the point of the transaction_id?

An Op is either a Read or a Write

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: Either:
  - Transaction completes successfully, so its effects are recorded in the server objects, or
  - Transaction has no effect at all
- Isolation: Need a transaction to be indivisible (atomic) from the point of view of other transactions/ops
  - No other transaction/op has access to intermediate results/states of this transaction
  - This transaction is free from interference by other ops/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 (prior slides)
- **C**onsistency: If the server is in a consistent state at the start of a transaction, then the server must be in a consistent state after the completion of the transaction.
- **I**solation: Each transaction must be performed without interference from other transactions, and intermediate effects of a transaction must not be visible to other transactions.
- **D**urability: After a transaction has completed successfully, all its effects must be saved in permanent storage.

Multiple Clients, One Server

- What could go wrong?

1. Lost Update Problem

time

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

seats = 9

seats = 9

T1's or T2's update was lost!

Consistency violated on the server

2. Inconsistent Retrieval Problem

time ↓

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
y = getSeats(ABC789);	
write(x-5, ABC123);	
// ABC123 = 5 now	
	x = getSeats(ABC123);
	y = getSeats(ABC789);
write(y+5, ABC789);	
	print("Total:" x+y);
	// Prints "Total: 20"
commit	commit

At Server:
ABC123 = 10
ABC789 = 15

T2's sum is the wrong value!
Should have been "Total: 25"
Consistency violated on the client

// x = 5, y = 15

Next

- How to prevent transactions from interfering with each other

Concurrent Transactions

- Running on different clients
- To prevent transactions from affecting each other
  - Could execute them one at a time at server
    - Booking an airline ticket takes ~5min
  - 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) without sacrificing ACID semantics

Serial Equivalence => Correctness

- An interleaving (say O) of transactions' operations is *serially equivalent* iff:
  - There is some ordering (O') of those transactions, one after the other, such that…
  - operations executing one at a time in O'…
  - yields the exact same end-state (on the server and clients) as O

    ⇒ ***we know O' is correct, and…***

    ⇒ ***we cannot tell the difference between the O' and O!***
- Let's say transactions A, B, and C have 5 ops each.
  - Let O' be: B1,..., B5, C1,…, C5, A1,…, A5.
  - An interleaving O of these 15 ops is serially equivalent iff it yields the exact same end-state as O'.

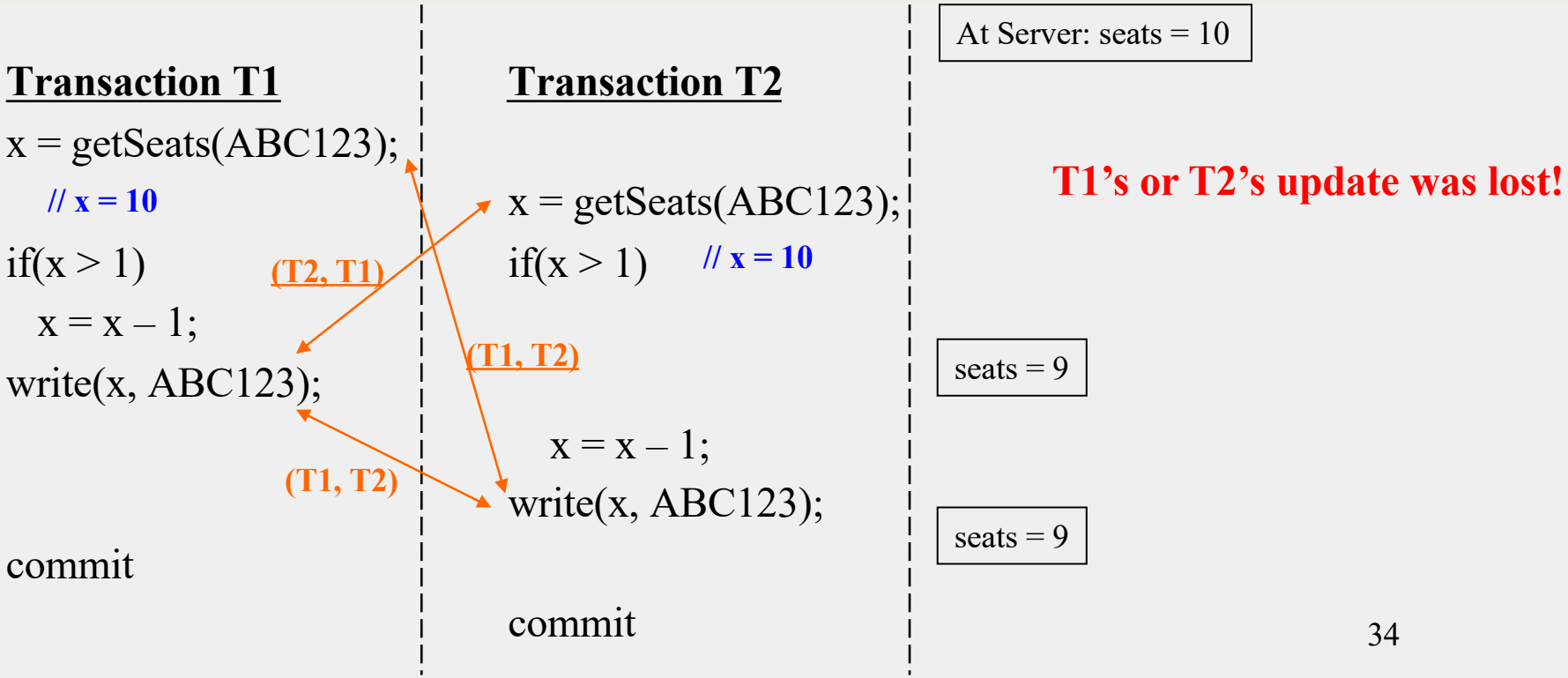
Checking for Serial Equivalence

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

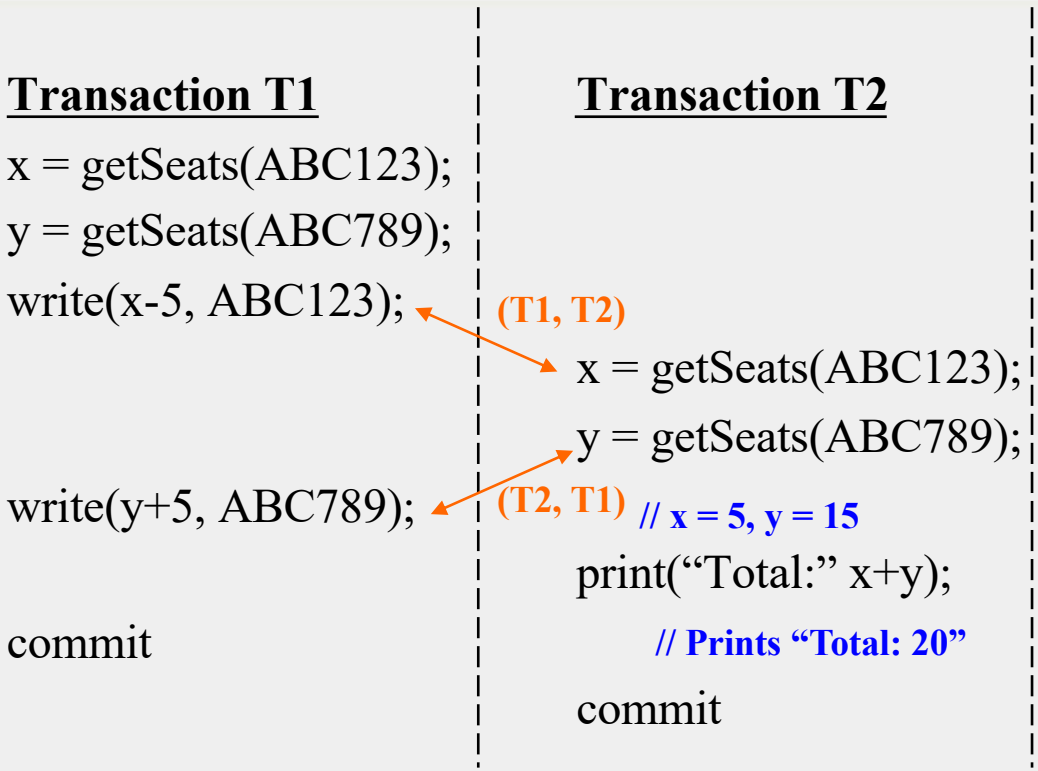
Checking for Serial Equivalence (2)

- *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's op occurred first, mark the pair as "(T1, T2)", otherwise mark it as "(T2, T1)"
  - Either all pairs should be marked "(T1, T2)" or all pairs should be marked "(T2, T1)".
- In other words, a serially equivalent ordering produces the same end-effect as T1 running after T2 or vice-versa.

1. Lost Update Problem – Caught!



2. Inconsistent Retrieval Problem – Caught!



At Server:
ABC123 = 10
ABC789 = 15

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

How Do We Use Serial Equivalence?

What control does the server really have?

Client operations arrive in some indeterminate order.

Server can:

1. Delay executing an RPC
2. Fail an RPC
3. Delay or fail the entire transaction at commit time

How Do We Use Serial Equivalence?

- At commit time of transaction $\boldsymbol{\tau}$
  - Let's say $\boldsymbol{\tau}$ started at time t_s
- Check for serial equivalence with every concurrent transaction
  - Transactions that overlapped in time with $\boldsymbol{\tau}$
  - (i) started after t_s (ii) completed after t_s (iii) currently in progress
- Commit $\boldsymbol{\tau}$ if serially equivalent with *all* concurrent transactions
- If not, and let's say $\boldsymbol{\tau}_c$ is the conflicting transaction
  - If $\boldsymbol{\tau}_c$ already committed, abort $\boldsymbol{\tau}$
  - Else, abort either $\boldsymbol{\tau}_c$ or $\boldsymbol{\tau}$
    - Abort: roll back (undo) all writes to server objects by that transaction

Can We do better?

- Not great for performance
  - An N-min long transaction may have 1000s of concurrent transactions
  - Will need to check many conflicting pairs
  - Abort of a transaction => wasted work
- Can we 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 (aka OCC)

Pessimistic vs. Optimistic

- Pessimistic: assume the worst, expect lots of conflicts, prevent transactions from accessing the same object
  - E.g., Locking
- Optimistic: assume the best, expect very few conflicts, allow transactions to proceed, 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 acquire a lock
- Before reading or writing object O, transaction T must call lock(O)
  - Blocks if another transaction already holds lock
- After acquiring lock, T can read and write O multiple times
- When done (or at commit point), T calls unlock(O)
  - If other transactions waiting on lock(O), allow 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 lots 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

Improved 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* other transactions holding lock for O entered in read mode
  - T is blocked if transaction holding lock for O entered in 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 holding the lock
  - Otherwise, T blocks
- Unlock(O) called by transaction T releases the lock
  - Independent of the mode

Guaranteeing Serial Equivalence With Locks

- Two-phase locking
  - A transaction cannot acquire (or promote) any locks once it has started releasing locks
  - Each 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 => Serial Equivalence?

- Proof by contradiction
- Assume serial equivalence is violated for two transactions T1, T2 in the two-phase locking system
- Two facts must then be true:
  - (*A*) For some object O1, there exist a conflicting operation pair where the time ordering pair is (T1, T2)
    - T1 released O1's lock and T2 acquired it after that
    - => T1's shrinking phase is before or overlaps with T2's growing phase
  - (*B*) For some object O2, there exists a conflicting operation pair with (T2, T1)
    - T2 released O2's lock and T1 acquired is after that
    - => T2's shrinking phase is before or overlaps with T1's growing phase
- Both A & B cannot be true!

Downside of Locking

- Deadlocks!

Downside of Locking – Deadlocks!

Transaction T1

Lock(ABC123);

x = write(…, ABC123);

Lock(ABC789);

// Blocks waiting for T2

…

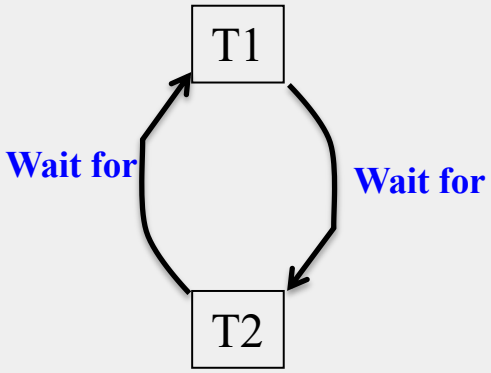
Transaction T2

Lock(ABC789);

y = write(…, ABC789);

Lock(ABC123);

… // Blocks waiting for T1



When do Deadlocks Occur?

- Three 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 => Abort one or more transactions to break cycle

   ☹ Still allows deadlocks to occur

Combating Deadlocks (2)

3. Deadlock Prevention

- Design system so one of the *necessary conditions* is violated:
  1. *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 (OCC)

- 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 to 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 ops can proceed at will
  - Check for serial equivalence at commit time
  - If abort, roll back updates made
  - An abort may result in aborting other transactions that read *dirty data*
    - Any transactions that read dirty data from *those* transactions also now need to be aborted

    ☹ *Cascading aborts*

Second approach: Timestamp Ordering

- Return server's timestamp at openTransaction as transaction 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 each object
- If rule violated, abort!
  - Can we do better?

Third Approach: Multi-version Concurrency Control (MVCC)

- 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…
- … can be considered a form of optimistic concurrency control in
  - Cassandra key-value store
  - DynamoDB key-value store
  - 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 (we studied them!)
- 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

Announcements

- Mid-semester survey participation: 60%
  - Will post a summary & response
- Next week
  - MP3 due on 11/8 (Sunday); demos 11/9
  - **Demo Signup by Friday 11/6, 11:59pm**
- HW4 released, due 12/3 (**Thursday**) at 11:59 pm