Review

- Find x in the following scenarios:
- [Synchronous system, crash failure, up to f failures]
  It takes x rounds to reach consensus.
- [Synchronous system, Byzantine failure, 30 processes, up to f failures]
  Consensus is impossible if f >= x.
- [Asynchronous system, crash failure, 30 processes, up to f failures]
  Consensus is impossible if f >= x.

Correction

- The first presentation is on March 30, not March 28

ECEN 757:
Transactions and Concurrency

Chapter 16

- I want to transfer $100 at an ATM
- The ATM checks the balance in my account, reduces it by 100, and then adds 100 to the destination account
- My account information is stored in a faraway server, not at the ATM
- How does this work?
- What happens when the ATM fails right after the second step?

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)

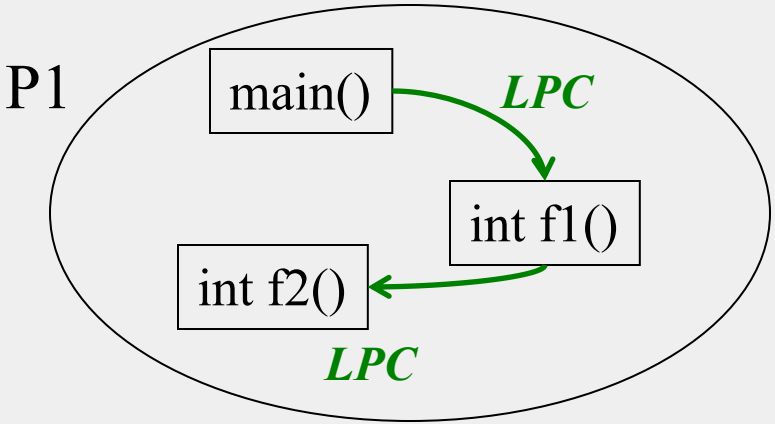
Local Procedure Call (LPC)

- Call from one function to another function within the same process
  - Uses stack 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

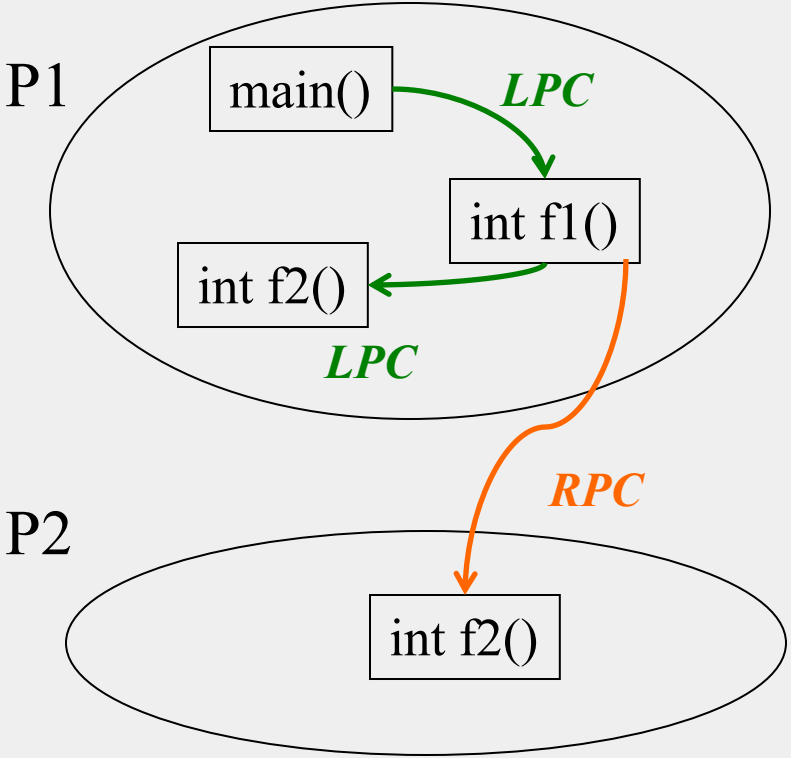
Remote Procedure Call

- 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

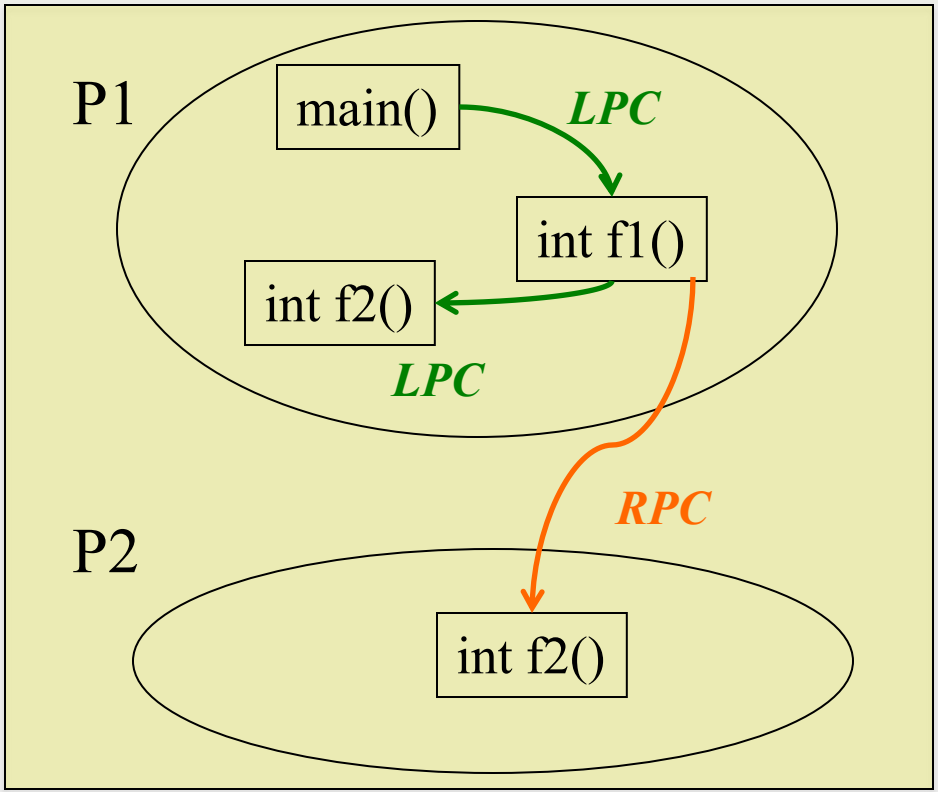
LPCs



RPCs

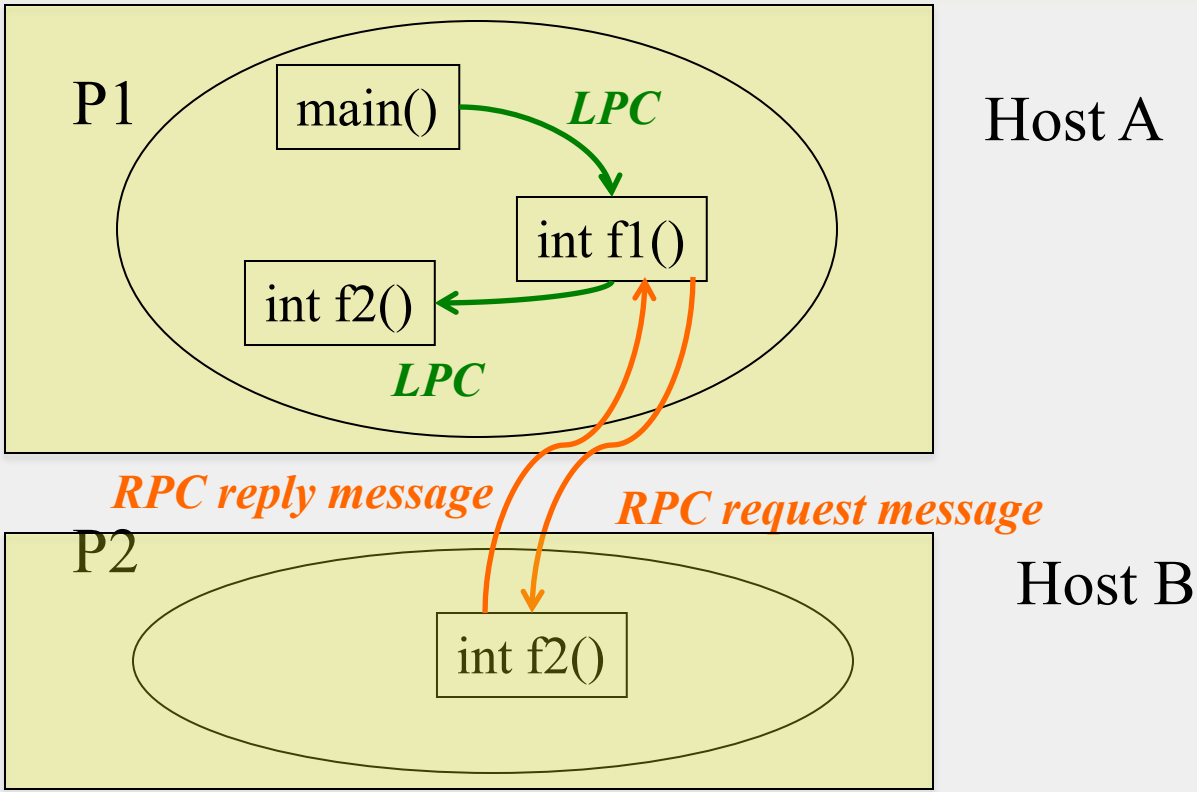


RPCs



Host A

RPCs


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

RPC Call Semantics

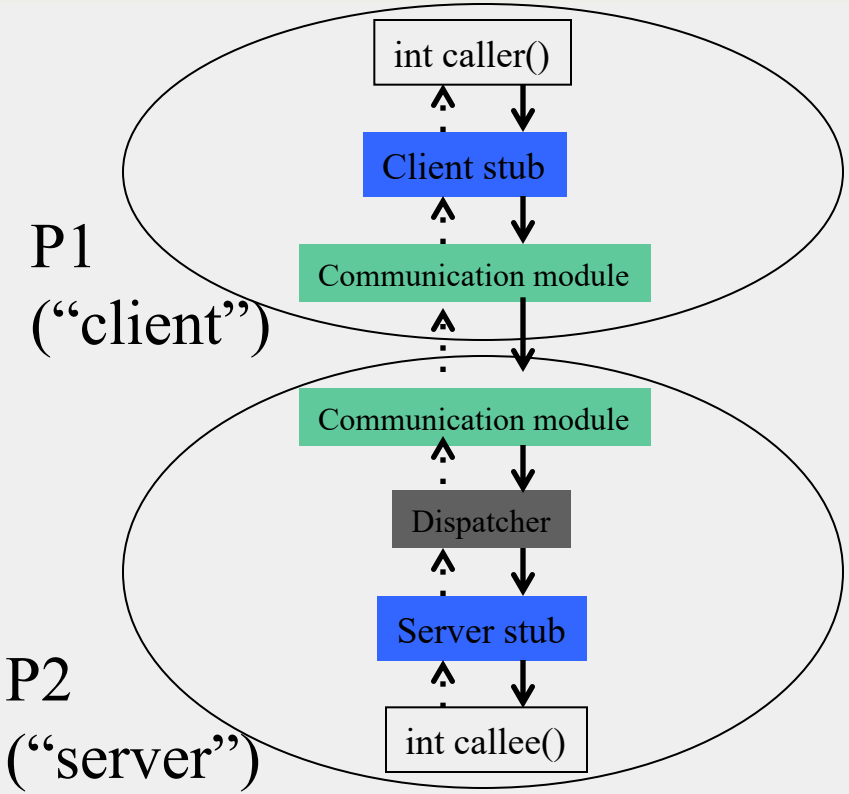
- Under failures, hard to guarantee exactly-once semantics
- Function may not be executed if
  - Request (call) message is dropped
  - Reply (return) message is dropped
  - Called process fails before executing called function
  - Called process fails after executing called function
  - Hard for caller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

Implementing RPC Call Semantics

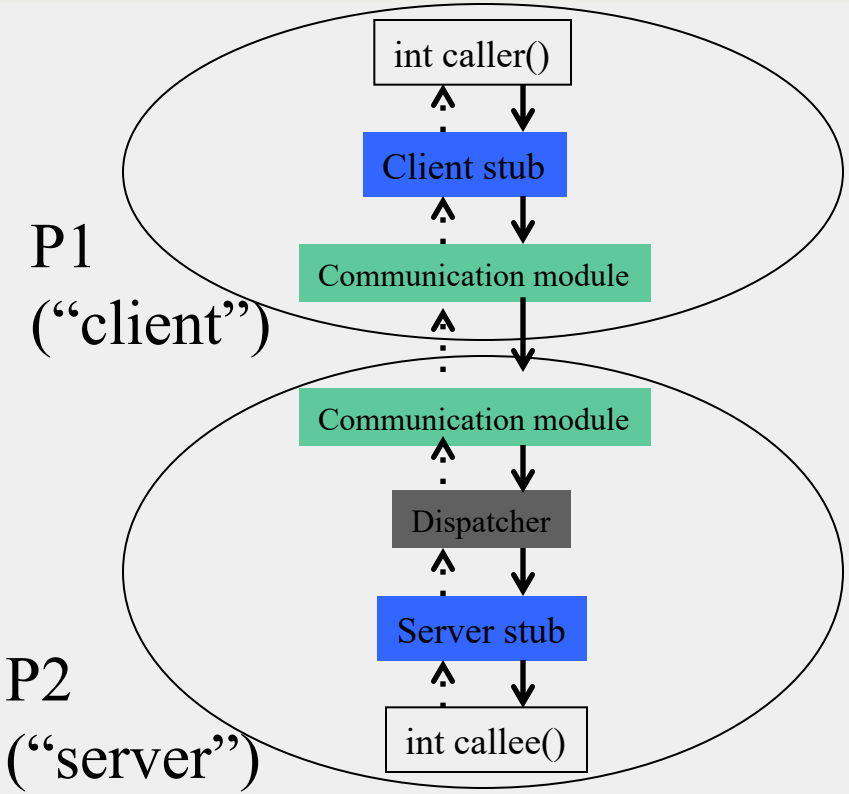
- Possible semantics
  - At most once semantics (e.g., Java RMI)
  - At least once semantics (e.g., Sun RPC)
  - Maybe, i.e., best-effort (e.g., CORBA)

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

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

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

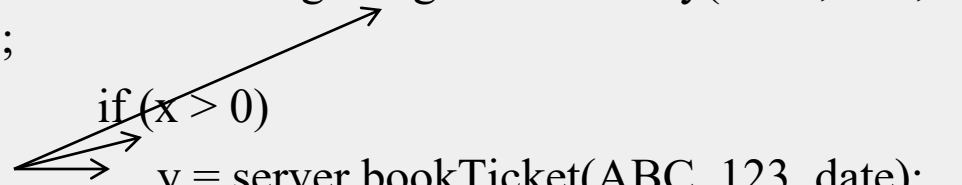
- Series of operations executed by client
- Each operation is an RPC to a server
- Transaction either
  - completes and *commits* all its operations at server
    - Commit = reflect updates on server-side objects
  - Or *aborts* and has no effect on server

Example: Transaction

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”);
// commit entire transaction or abort
closeTransaction(transaction_id);
```

RPCs

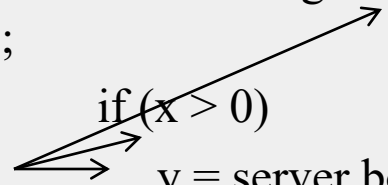


Example: Transaction

Client code:

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

RPCs



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

- **Atomicity**: All or nothing
- **Consistency**: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- **Isolation**: 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.
- **Durability**: 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

seats = 9

seats = 9

T1's or T2's update was lost!

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 while maintaining correctness (ACID)

Serial Equivalence

- An interleaving (say O) of transaction operations is serially equivalent iff (if and only if):
  - There is some ordering (O') of those transactions, one at a time, which
  - Gives the same end-result (for all objects and transactions) as the original interleaving O
  - Where the operations of each transaction occur consecutively (in a batch)
- Says: Cannot distinguish end-result of real operation O from (fake) serial transaction order O'

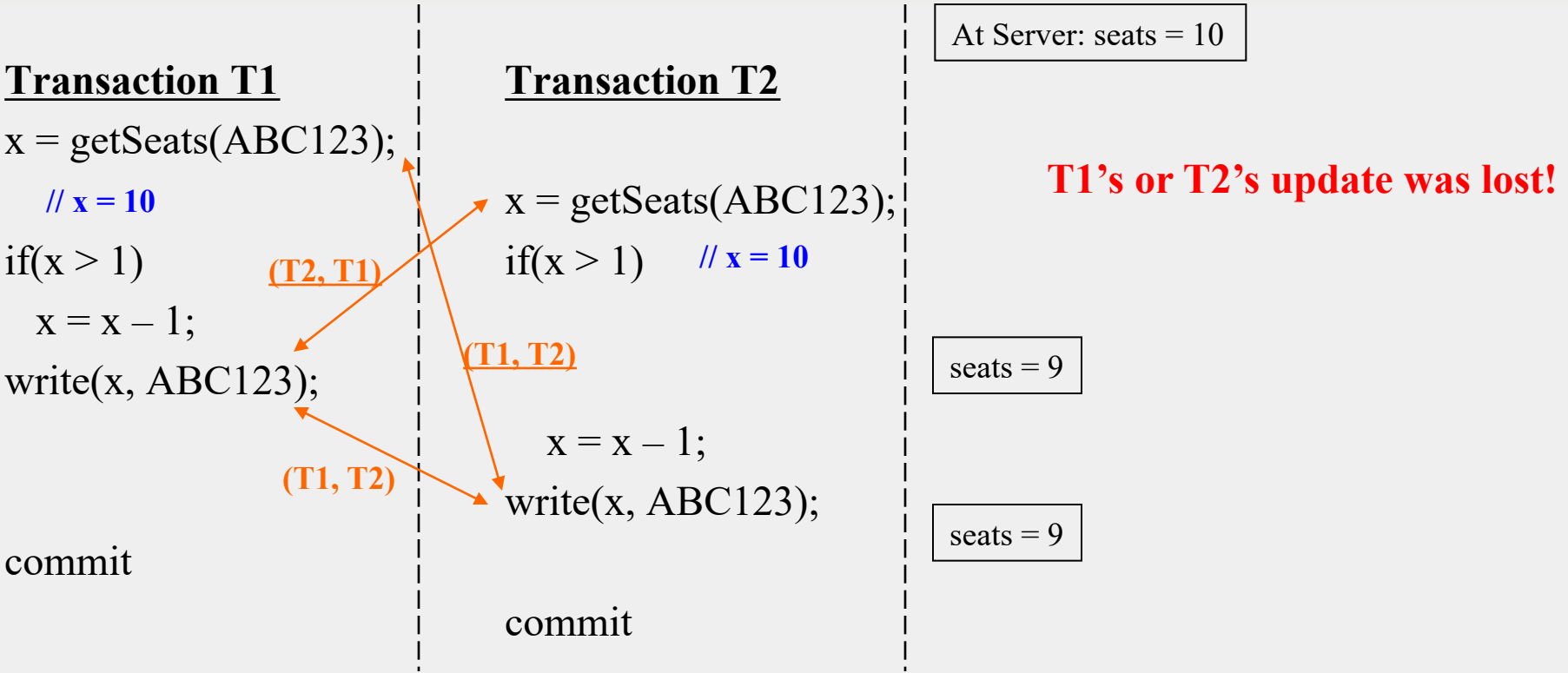
Checking for Serial Equivalence

- An operation has an effect on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed
  - read(x) and write(x)
  - write(x) and read(x)
  - write(x) and write(x)
  - NOT read(x) and read(x): swapping them doesn't change their effects
  - NOT read/write(x) and read/write(y): swapping them ok

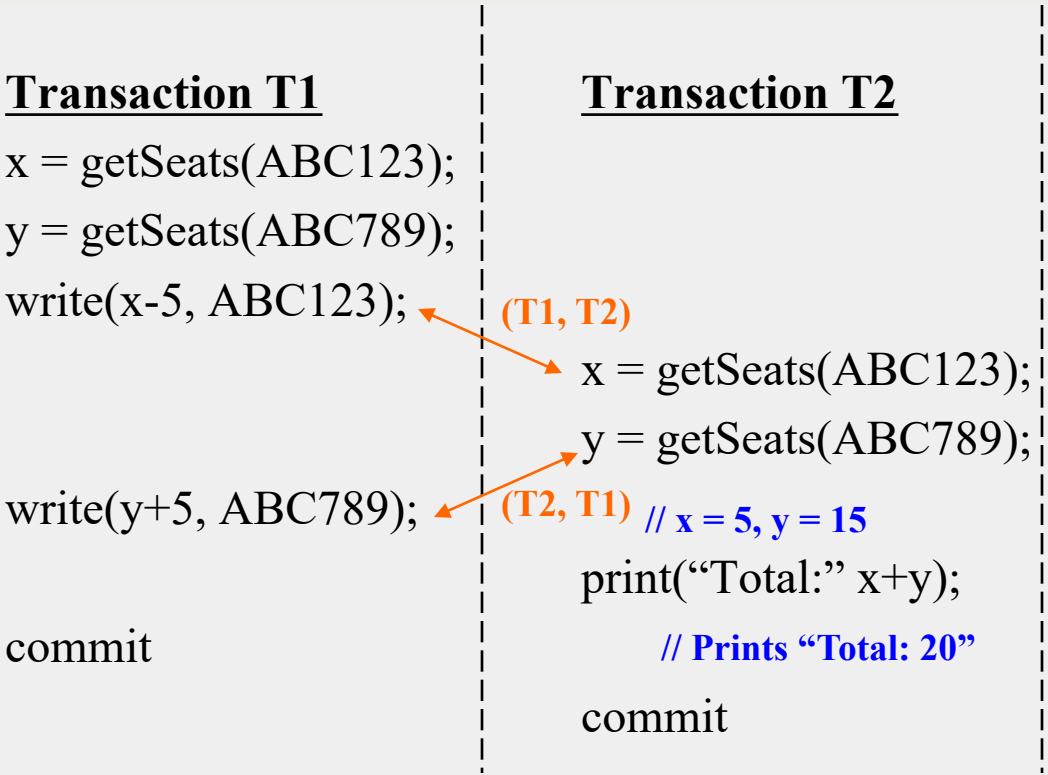
Checking for Serial Equivalence (2)

- *Two transactions are serially equivalent if and only if all pairs of conflicting operations (pair containing one operation from each transaction) are executed in the same order (transaction order) for all objects (data) they both access.*
  - Take all pairs of conflict operations, one from T1 and one from T2
  - If the T1 operation 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!



2. Inconsistent Retrieval Problem – Caught!



At Server:
ABC123 = 10
ABC789 = 15

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

1. Lost Update Problem

Transaction T1	**Transaction T2**
x = getSeats(ABC123);	
if(x > 1)	
x = x – 1;	
write(x, ABC123);	
	x = getSeats(ABC123);
	if(x > 1)
	x = x – 1;
	write(x, ABC123);
commit	
	commit

At Server: seats = 10

Is this interleaving serial equivalent?
What is the resulting x?

2. Inconsistent Retrieval Problem

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

At Server:
ABC123 = 10
ABC789 = 15

Is this interleaving serially equivalent?
What is the output of T2?

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

Recoverability from Aborts

- Processes can abort transactions because…
- There can be failures during transactions
- They may detect violations of serially equivalency
- They may be force to abort because some other processes abort

Dirty Read

Transaction *T*:		**Transaction *U*:**	
a.getBalance()		*a.getBalance()*	
a.setBalance(balance + 10)		*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			

Which Property Is Violated?

- **Atomicity**: All or nothing
- **Consistency**: if the server starts in a consistent state, the transaction ends the server in a consistent state.
- **Isolation**: 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.
- **Durability**: After a transaction has completed successfully, all its effects are saved in permanent storage.

How to Handle Dirty Reads?

- In the previous example, the effects of U cannot be undone after it commits
- In practice, “commit” may mean “overwrite the data on the hard drive”
- Solution: U delays its commit until T commits. If T aborts, U aborts as well.
- Problem – Cascading Aborts: T may trigger U to abort, who in turn makes V abort, ….

Premature Write

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

Premature Write

- The two transactions are serial equivalent! What can go wrong?

Premature Write

- The two transactions are serial equivalent! What can go wrong?
- Suppose T aborts but U commits, the final value should be $110
- Some systems implement “abort” by restoring the value before the transaction
- In this case, the system may decide to restore the value to $100
- The effects of U are erased without U’s knowledge!
- Solution: U needs to wait for T before it commits
  - Luckily, there are no cascading aborts