Transactions and Concurrency Control

CMPS 4760/6760: Distributed Systems

Overview

- Transactions (16.1-16.2)
- Concurrency control (16.4-16.5)
- Distributed transactions (17.3.1)

Simple synchronization

- Consider a single server that manages multiple remote objects
- The server uses multiple threads to allow the objects to be accessed by multiple clients concurrently

A Banking Example

Operations of the *Account* interface

deposit(amount)
deposit amount in the account

withdraw(amount)
withdraw amount from the account

getBalance() -> amount
return the balance of the account

setBalance(amount)
set the balance of the account to amount

Operations of the *Branch* interface

create(name) -> account
create a new account with a given name

lookUp(name) -> account
return a reference to the account with the given name

branchTotal() -> amount
return the total of all the balances at the branch

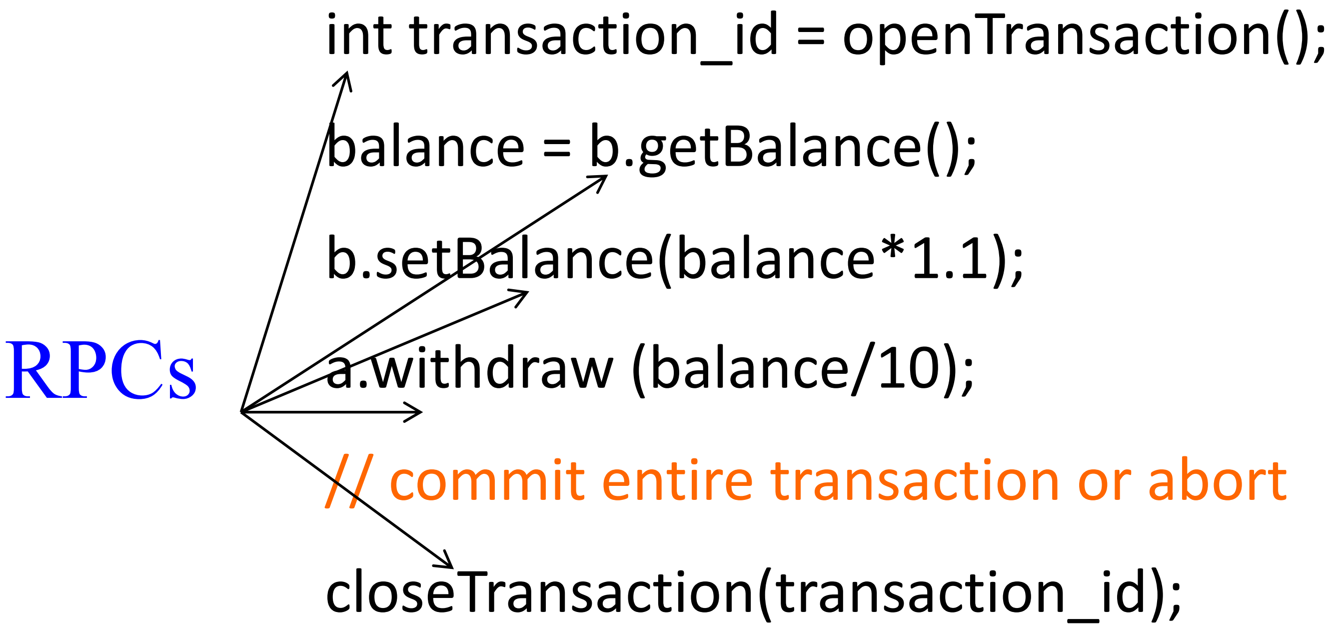
Atomic operations

- A possible implementation of *deposit(amount)*
  1. *read the current balance*
  2. *increase the balance by amount*
- Two separate invocations can be interleaved arbitrarily and have strange effects
- Atomic operations: operations that are free from interference from concurrent operations
  - e.g., synchronized methods in Java + wait/notify methods to enhance communication among threads

Transactions

- Series of operations executed by client
- Each operation is an RPC to a server
- They are free from interference operations from other concurrent clients
- 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();
balance = b.getBalance();                 // read(b)
b.setBalance(balance*1.1);                // write(b)
a.withdraw (balance/10);                  // write(a)
// commit entire transaction or abort
closeTransaction(transaction_id);
```

Operations in *Coordinator* interface

openTransaction() -> trans;

starts a new transaction and delivers a unique TID *trans*. This identifier will be used in the other operations in the transaction.

closeTransaction(trans) -> (commit, abort);

ends a transaction: a *commit* return value indicates that the transaction has committed; an *abort* return value indicates that it has aborted.

abortTransaction(trans);

aborts the transaction.

Transaction life histories

Successful

openTransaction
operation
operation
:
operation
closeTransaction

Transaction life histories

Successful	Aborted by client
openTransaction	openTransaction
operation	operation
operation	operation
:	:
operation	operation
closeTransaction	abortTransaction

Transaction life histories

Successful	Aborted by client	Aborted by server	
openTransaction	openTransaction		openTransaction
operation	operation		operation
operation	operation		operation
:	:	server aborts transaction →	:
operation	operation		operation ERROR reported to client
closeTransaction	abortTransaction		

The lost update problem

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)*

Initial balance

A: 100

B: 200

C: 300

If T and U are run sequentially, then the closing balances would be:

Case 1: (T, U)	Case 2: (U, T)
A: 100-200/10 = 80	A: 100-(200*1.1)/10 = 78
B: 200*1.1*1.1 = 242	B: 200*1.1*1.1 = 242
C: 300-(200*1.1)/10 = 278	C: 300-200/10 = 280

The lost update problem

Transaction _T_ :		**Transaction _U_:**	
balance = b.getBalance(); _b.setBalance(balance*1.1);_ _a.withdraw(balance/10)_		_balance = b.getBalance();_ _b.setBalance(balance*1.1);_ _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

Initial balance

A: 100

B: 200

C: 300

The inconsistent retrievals problem

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
		:	
b.deposit(100)	$300		

Initial balance

A: 200

B: 200

ACID Properties of Transactions

- **Atomicity**: All or nothing: 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.
- **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.

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
- Goal: increase concurrency while maintaining correctness (ACID)

Serial Equivalence

- An interleaving (say O) of transaction operations is serially equivalent if:
  - There is some ordering (O’) of those transactions, one at a time,
  - Where the operations of each transaction occur consecutively (in a batch),
  - Which gives the same end-result (for all objects and transactions) as the interleaving O

A serially equivalent interleaving of *T* and *U*

Transaction *T*:		**Transaction *U*:**	
balance = b.getBalance() *b.setBalance(balance*1.1)* *a.withdraw(balance/10)*		*balance = b.getBalance()* *b.setBalance(balance*1.1)* *c.withdraw(balance/10)*	
balance = b.getBalance()	$200		
*b.setBalance(balance*1.1)*	$220		
		balance = b.getBalance()	$220
		*b.setBalance(balance*1.1)*	$242
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$278

A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	$100		
b.deposit(100)	$300		
		total = a.getBalance()	$100
		total = total+b.getBalance()	$400
		...	

A non-serially equivalent interleaving of operations

Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Initial balance

i: 5

j: 5

- End-result:
  - The interleaving above: i=10, j=20, x=5, y=5, z=10
  - (T, U): i=10, j=30, x=5, y=20, z=10
  - (U, T): i=10, j=20, x=5, y=5, z=5

Checking for Serial Equivalence

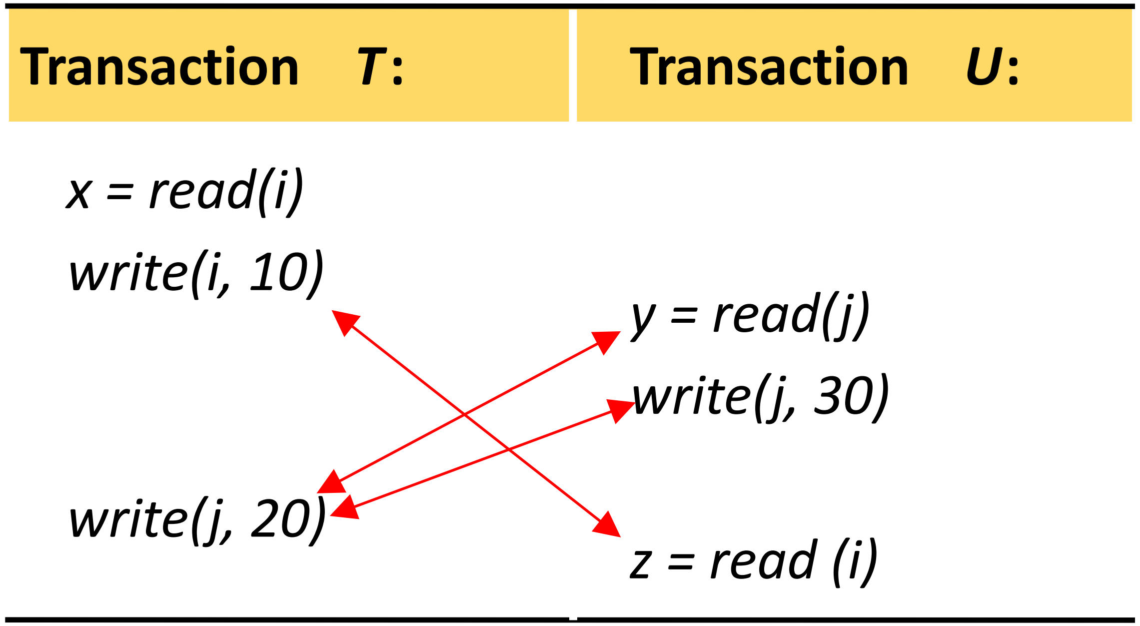
- An operation has an effect on
  - The server object if it is a write
  - The client (returned value) if it is a read
- Two operations are said to be conflicting operations, if their *combined effect* depends on the order they are executed

Operations of different transactions		*Conflict*
read	*read*	No
read	*write*	Yes
write	*write*	Yes

Checking for Serial Equivalence

- 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)”.

A non-serially equivalent interleaving of operations



Transaction *T*:	**Transaction *U*:**
x = read(i)	
write(i, 10)	
	y = read(j)
	write(j, 30)
write(j, 20)	
	z = read (i)

Initial balance

i: 5

j: 5

Recovery from aborts

- Server must record the effects of all committed transactions and none of the effects of aborted transactions.
- Problems due to aborted transactions:
  - dirty reads
  - premature writes.
- Both can occur in serially equivalent executions of transactions.

A dirty read when transaction *T* aborts

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

- Can lead to cascading aborts

Overwriting uncommitted values

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

- Some database systems implement the action of abort by restoring ‘before images’ of all the writes of a transaction.
- If U aborts and T commits, the balance should be $105
- If U commits and then T aborts, what is the balance? ~~$100~~ $110
- To ensure correct results in a recovery scheme that uses before images, write operations must be delayed until earlier transactions that updated the same objects have either committed or aborted.

Strict executions of transactions

- The executions of transactions are called strict if the service delays both read and write operations on an object until all transactions that previously wrote that object have either committed or aborted
- Avoids dirty reads and premature writes
- Enforces the desired property of isolation
- But reduces concurrency

Overview

- Transactions (16.1-16.2)
- Concurrency control (16.4-16.5)
- Distributed transactions (17.3.1)

Concurrency control

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking (16.4)
- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time (16.5)
- Timestamp ordering (16.6)

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!)

Transactions *T* and *U* with exclusive locks

Transaction *T*:	**Transaction *U*:**
balance = b.getBalance()	*balance = b.getBalance()*
*b.setBalance(bal*1.1)*	*b.setBalance(bal*1.1)*
a.withdraw(bal/10)	*c.withdraw(bal/10)*

Operations	Locks	Operations	Locks
openTransaction			
bal = b.getBalance()	lock *B*		
*b.setBalance(bal*1.1)*		*openTransaction*	
a.withdraw(bal/10)	lock *A*	*bal = b.getBalance()*	waits for *T*'s lock on *B*
closeTransaction	unlock *A*, *B*	• • •	
			lock *B*
		*b.setBalance(bal*1.1)*	
		c.withdraw(bal/10)	lock *C*
		closeTransaction	unlock *B*, *C*

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
  - Ok to allow two transactions to concurrently read an object, since read-read is not a conflicting pair

Read-Write Locks

- Each object has a lock that can be held in one of two modes
  - Read mode: multiple transactions allowed in (shared lock)
  - 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

- 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
- It is not safe to demote a lock held by a transaction before it commits as this may allow executions by other transactions that are inconsistent with serial equivalence

Lock compatibility

For one object		*Lock requested*	
		read	*write*
Lock already set	*none*	OK	OK
	read	OK	wait
	write	wait	wait

Two-phase locking

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

Two-phase Locking => Serial Equivalence

- Proof by contradiction
- Assume 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)

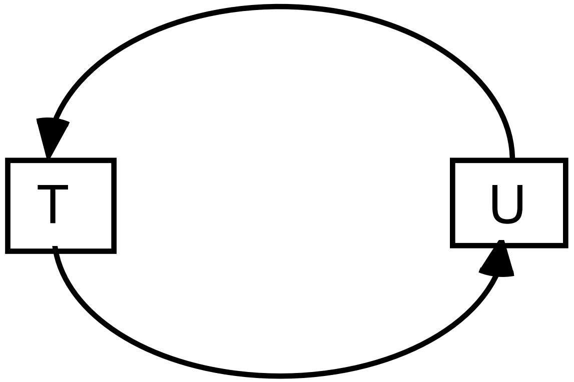
- (*A*) => T1 released O1's lock and T2 acquired it after that

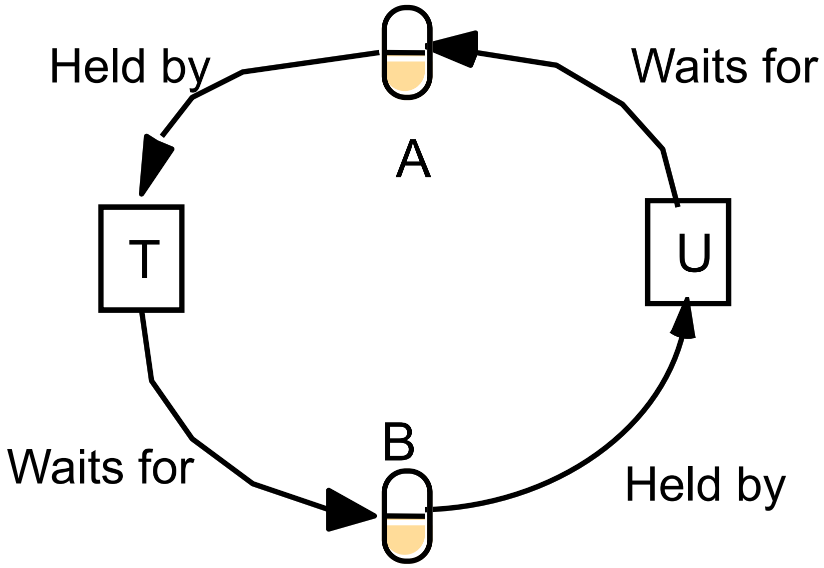
  => T1's shrinking phase is before or overlaps with T2's growing phase

- Similarly, *(B)* => T2's shrinking phase is before or overlaps with T1's growing phase
- A contradiction!!

Deadlock with write locks

Transaction *T*		**Transaction *U***	
Operations	**Locks**	**Operations**	**Locks**
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
•••	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for *T*'s lock on *A*
		•••	
•••		•••	
•••		•••	





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
- Can be used to prevent and detect deadlocks

Timeout

Transaction T		Transaction U	
Operations	Locks	Operations	Locks
a.deposit(100);	write lock *A*		
		b.deposit(200)	write lock *B*
b.withdraw(100)			
• • •	waits for *U*'s lock on *B*	*a.withdraw(200);*	waits for T's lock on *A*
		• • •	
	(timeout elapses)	• • •	
	T's lock on *A* becomes vulnerable, unlock *A*, abort T		
		a.withdraw(200);	write locks *A* unlock *A*, *B*

Downside of Locking

- Overhead: lock may be necessary only in the worst case
    - consider two client processes that are concurrently incrementing the values of n objects. The chances that the two programs will attempt to access the same object at the same time are just 1 in n on average
- To avoid dirty reads and premature writes, locks cannot be released until end of the transaction
- Deadlock

Concurrency control

- Pessimistic: assume the worst, prevent transactions from accessing the same object
  - E.g., Locking (16.4)
- Optimistic: assume the best, allow transactions to write, but check later
  - E.g., Check at commit time (16.5)
- Timestamp ordering (16.6)

Beyond Pessimistic 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!

Opportunistic Concurrency control

- 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

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!

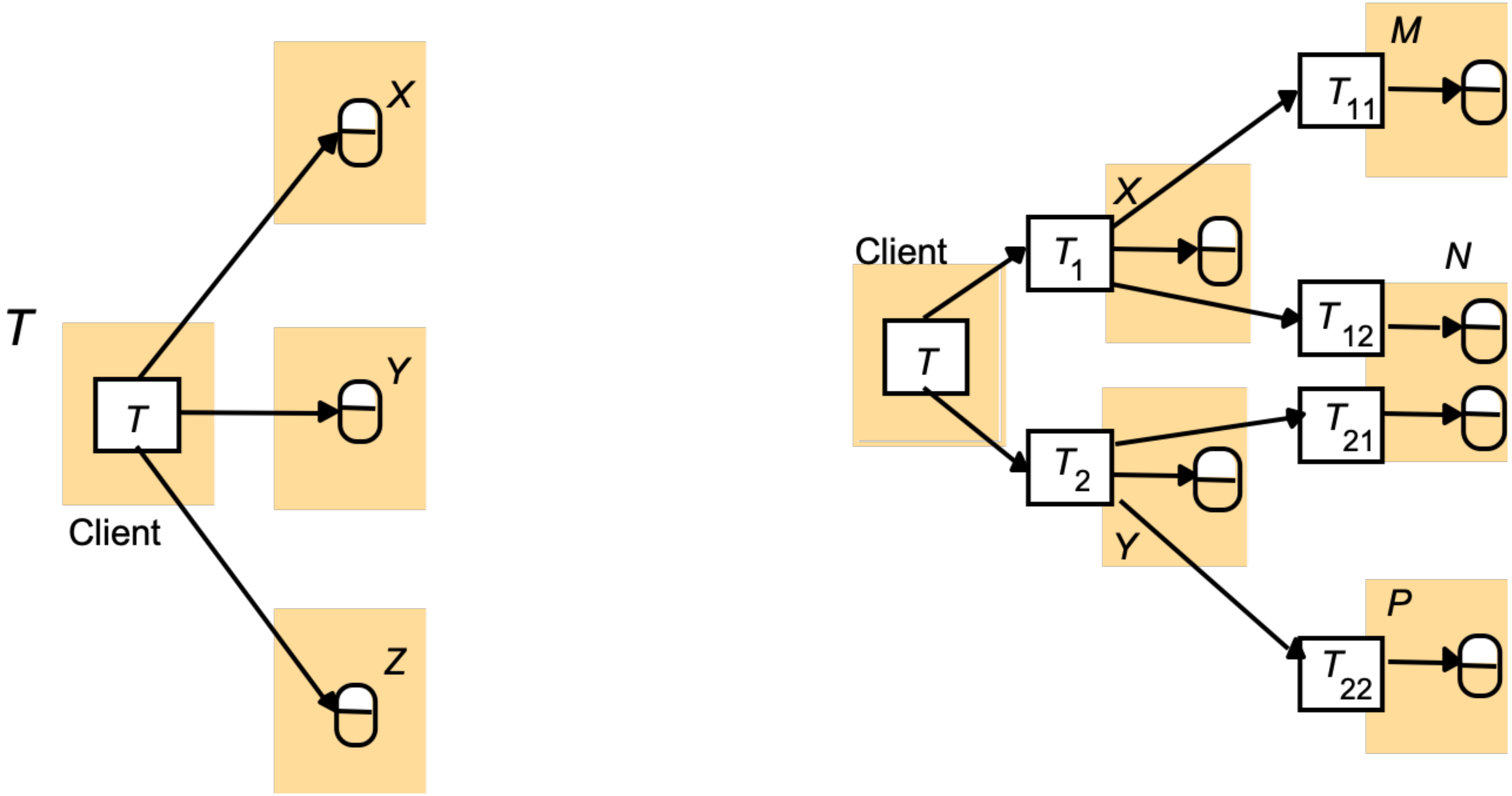
Multi-version Concurrency Control

- 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

Overview

- Transactions (16.1-16.2)
- Concurrency control (16.4-16.5)
- Distributed transactions (17.3.1)

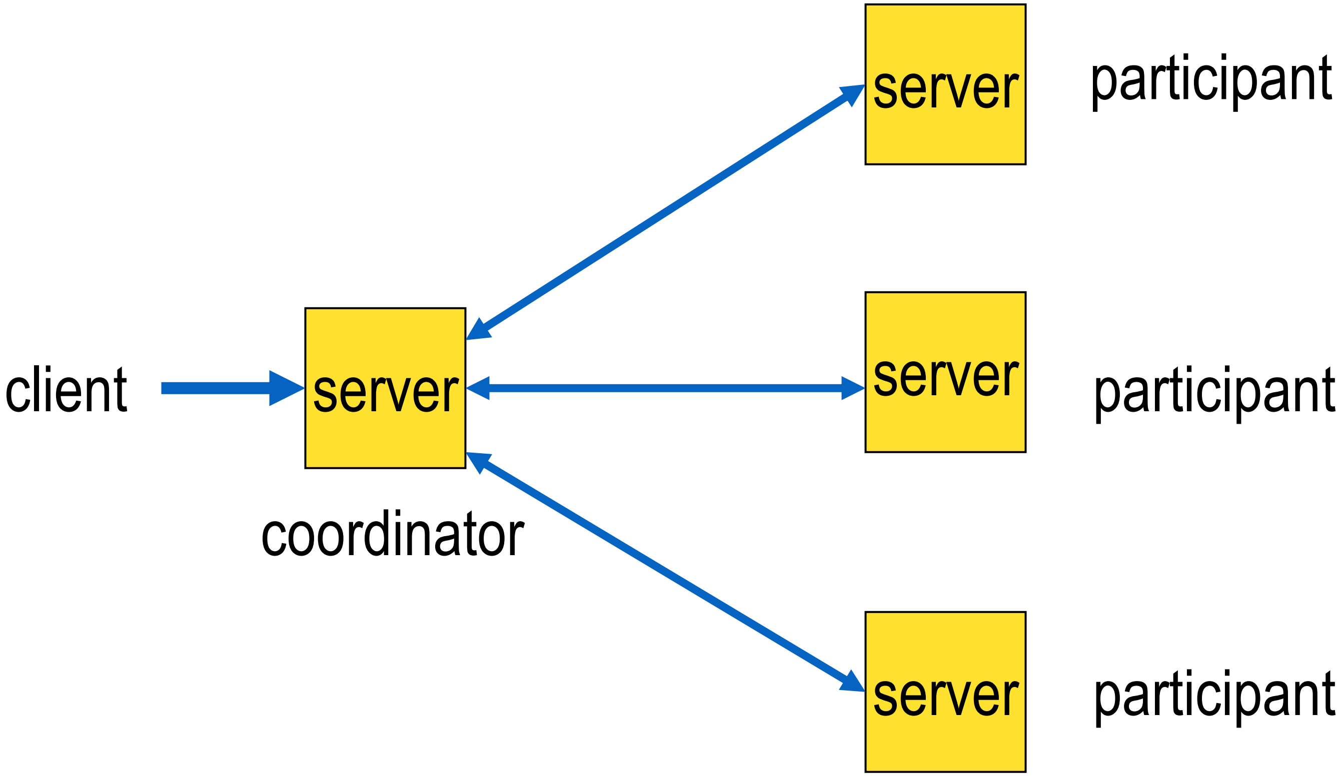
Distributed Transactions



(a) Flat transaction

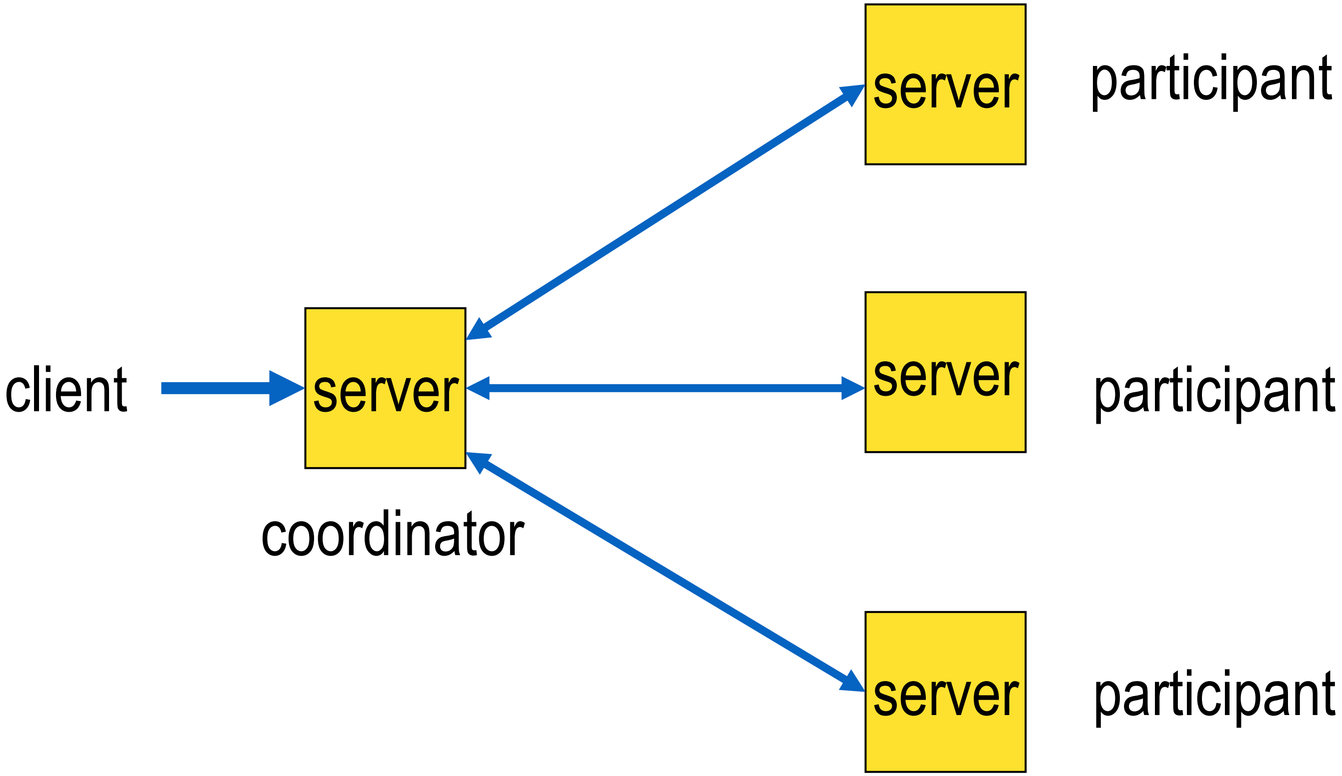
(b) Nested transactions

Atomic Commit Protocols



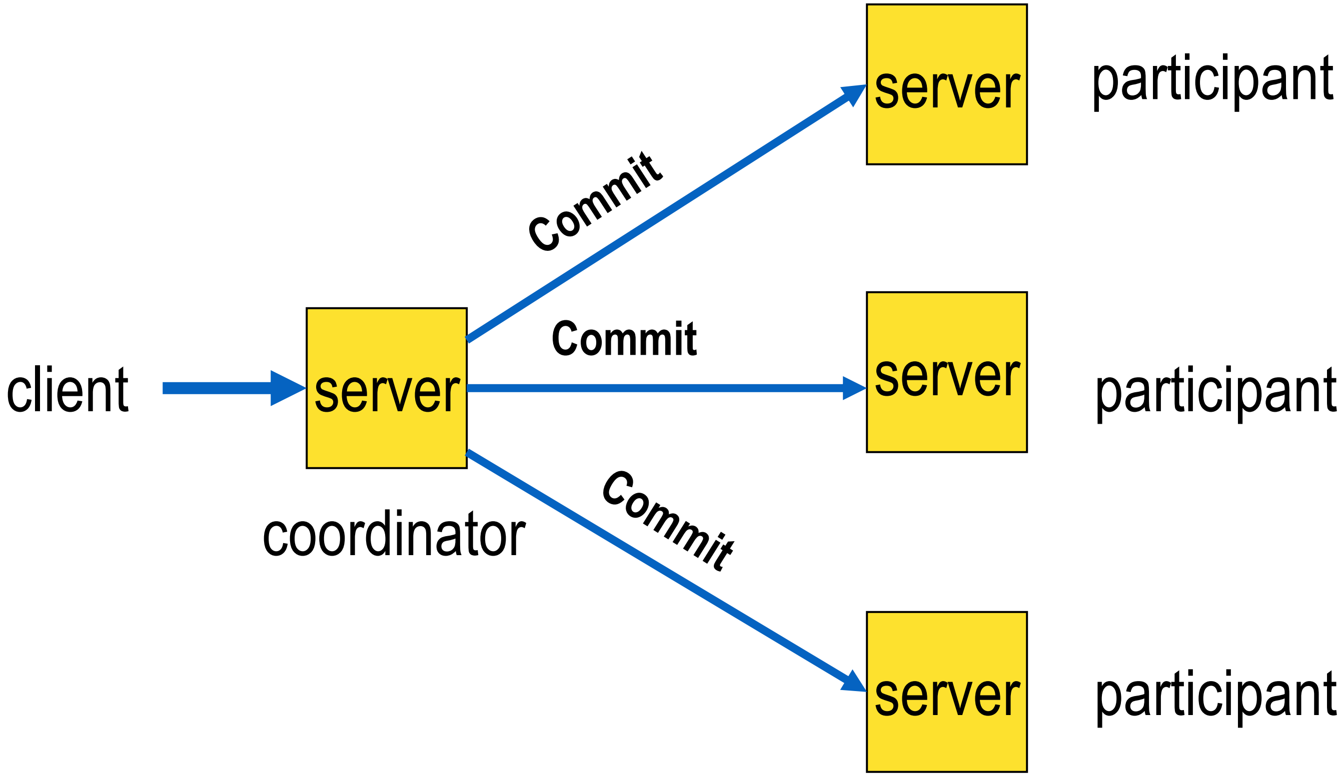
- The initiator of a transaction is called the coordinator, and the remaining servers are participants
- When a distributed transaction comes to an end, either all of its operations are carried out or none of them
- All the servers involved need to reach an agreement
- A consensus problem

Atomic Commit Protocols

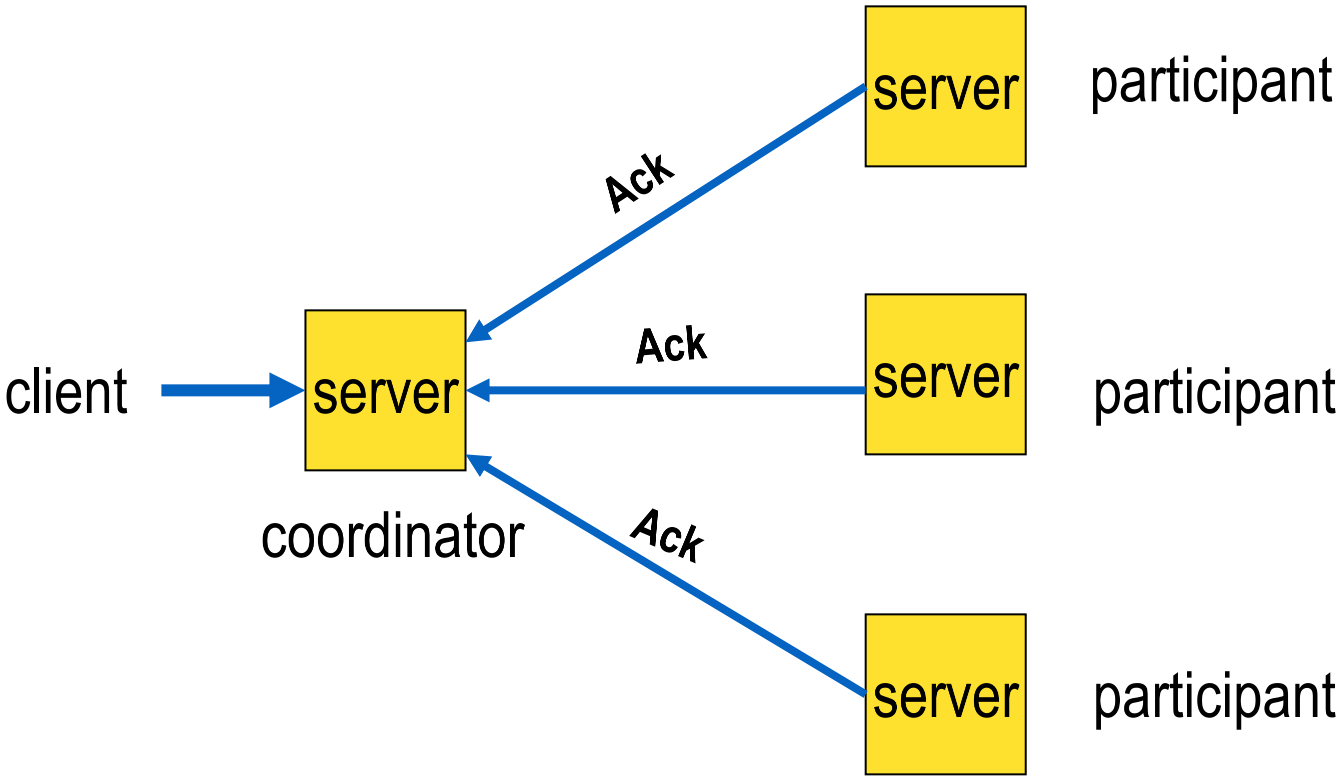


- Designed for an asynchronous system where servers may crash and messages may be lost
- A crashed process is eventually replaced with a new process whose state is set from information saved in permanent storage and information held by other processes
- No Byzantine faults

One-phase Commit



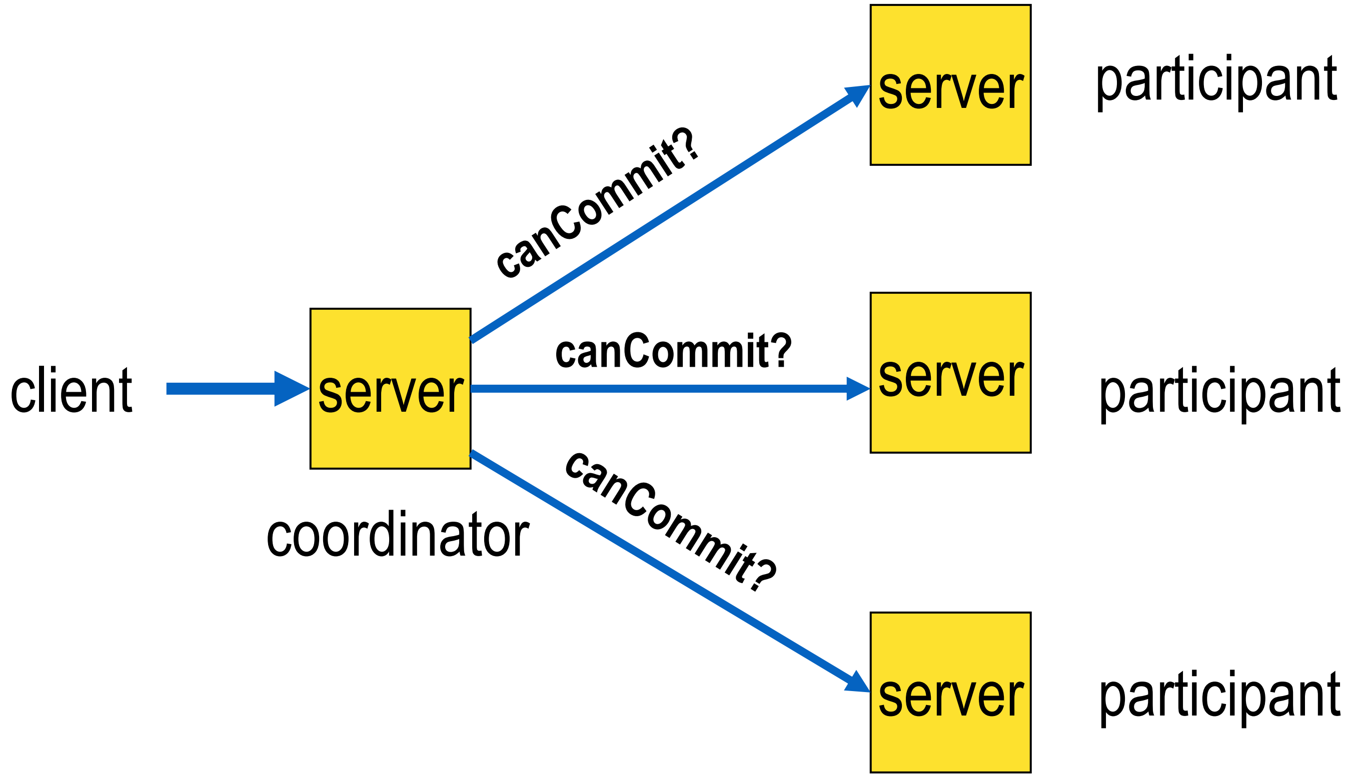
One-phase Commit



- *If a participant deadlocks or faces a local problem then the coordinator may never be able to find it.*

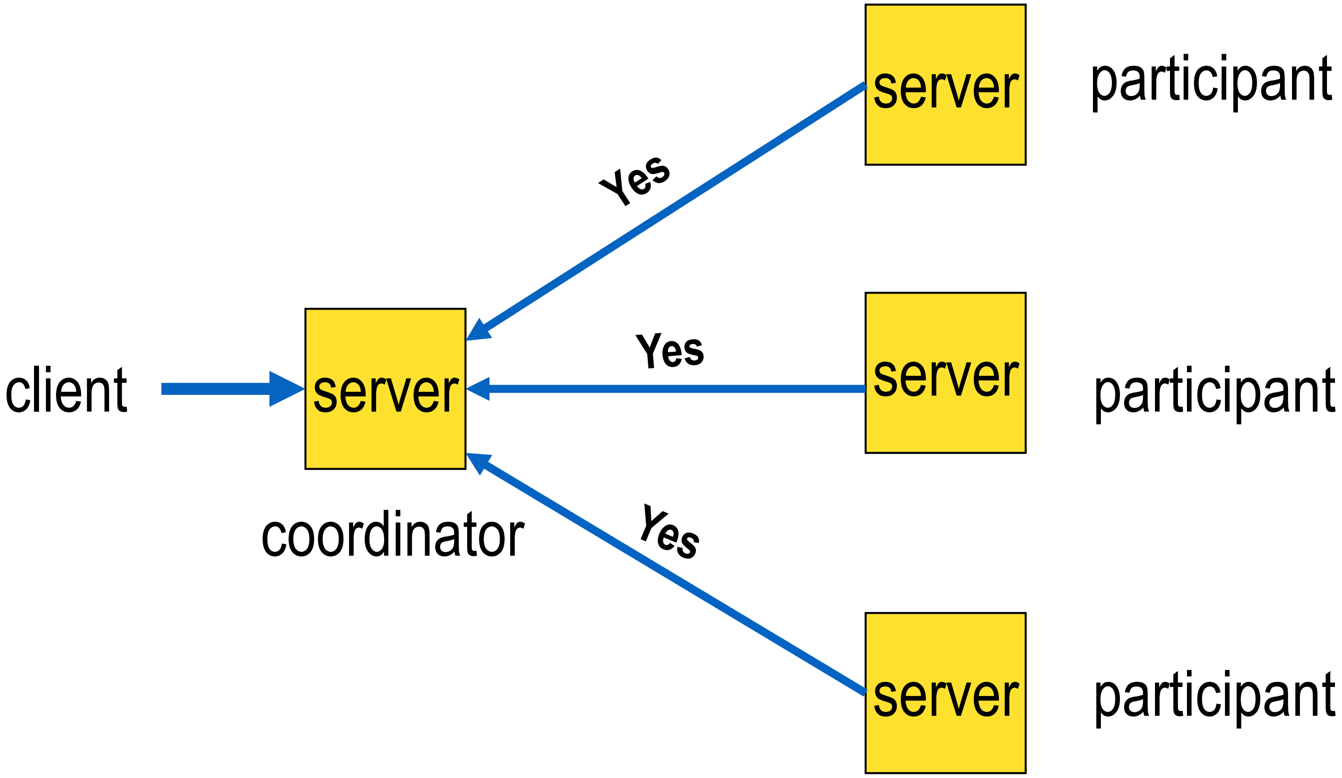
Two-phase commit (2PC)

Phase 1: Voting Phase



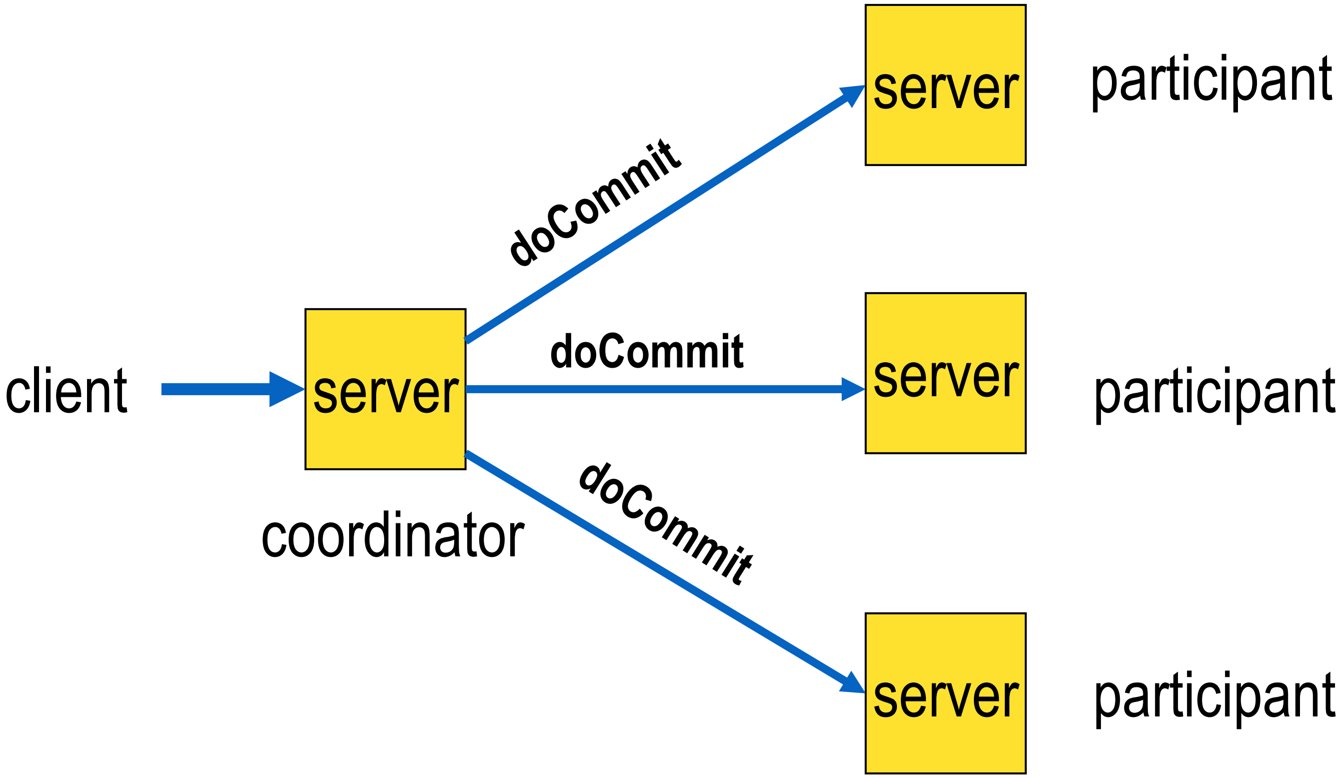
Two-phase commit (2PC)

Phase 1: Voting Phase



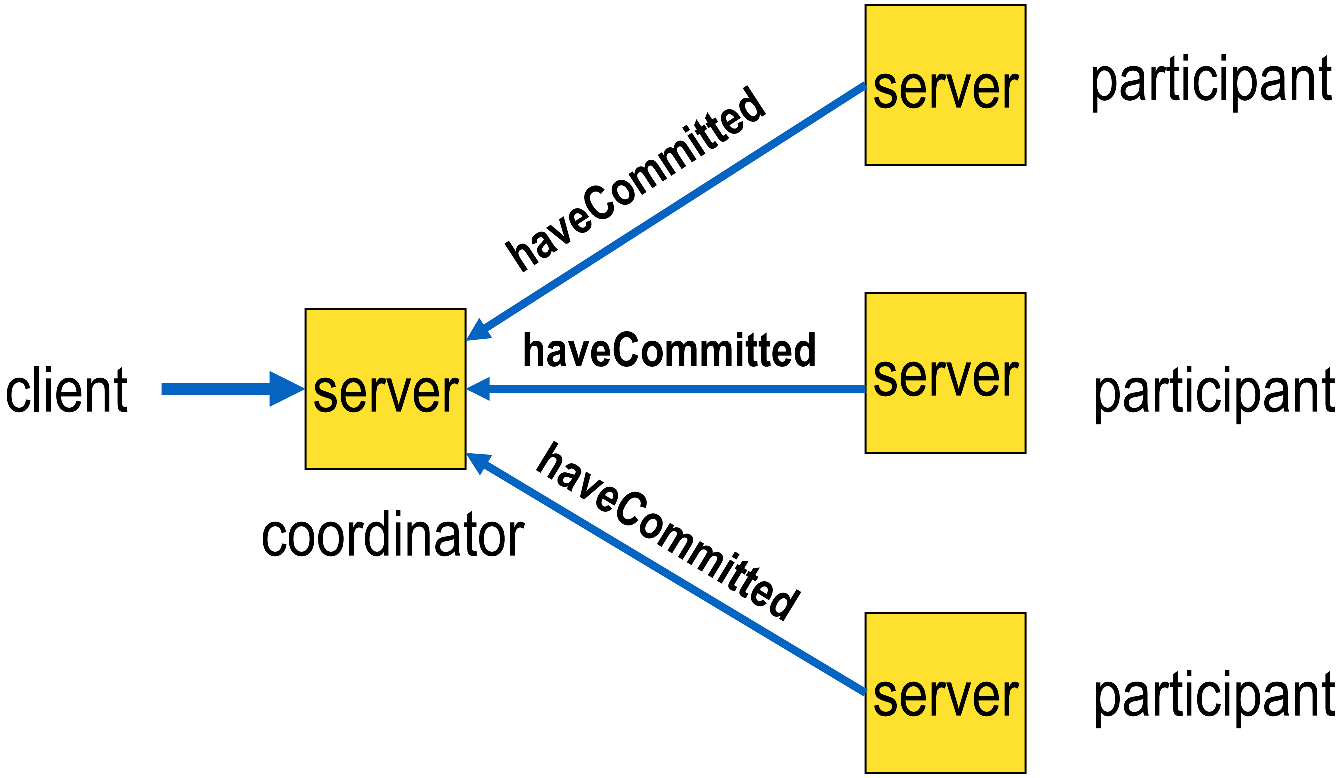
Two-phase commit (2PC)

Phase 2: Commit Phase



Two-phase commit (2PC)

Phase 2: Commit Phase



Two-phase commit protocol

Phase 1: Voting Phase

Coordinator		Participant
		▪ Work on transaction
▪ Write *prepare to commit* to log		▪ Wait for message from coordinator
▪ Send *canCommit?* message	→	▪ Receive the *canCommit?* message
▪ Wait for all participants to respond		▪ When ready, write *agree to commit* or *abort* to the log
	←	▪ Send *Yes* or *No* to the coordinator. If voting *No*, abort immediately

Two-phase commit protocol

Phase 2: Commit Phase

Coordinator

- Write *commit* or *abort* to log
- Send *doCommit or doAbort*
- Wait for all participants to respond
- Clean up all state. Done!

Participant

- Wait for commit/abort message
- Receive *doCommit or doAbort*
- If a *doCommit* was received, write “*commit*” to the log, release all locks, update databases, call *haveComitted* (a method implemented by the coordinator)
- If a *doAbort* was received, undo all changes

Failure scenarios in 2PC

(Phase 1)

Fault: Coordinator did not receive YES / NO:

OR

Participant did not receive VOTE:

Solution: Broadcast ABORT after certain timeout;

Abort local transactions after certain timeout

Failure scenarios in 2PC

(Phase 2)

Fault: A participant does not receive COMMIT or ABORT from the coordinator

- E.g., coordinator crashed after sending ABORT or COMIT to a fraction of the participants.
- Such a participant is *uncertain* of the outcome and cannot decide unilaterally what to do next, and meanwhile the objects used by its transaction cannot be released for use by other transactions
- The participants may query the coordinator or obtain a decision cooperatively.
- In the worst-case when all the active participants are *uncertain*, they remain undecided, until the coordinator is repaired and reinstalled.

A known weakness of 2PC => 3PC (see Homework 4)