Distributed Transactions

Distributed Transactions

- Introduction
- Concurrency model
- Methods for serialization of transactions
- Recoverability from aborts
- **Distributed Transactions**
- Nested Transactions

Introduction 1/3

- **Transaction :** operations that appear to execute as one large, atomic operation.
- **Goal :** ensure that all objects managed by a server remain in a consistent state when they are accessed by multiple transactions
  - **All or Nothing** : all transaction operations must be completed successfully or must have no effect at all in the presence of server crashes.
    - >> Recoverable objects
    - >> changes completed must be in permanent storage
  - **Isolation** : no interference by operations from concurrent clients.
    - >> Synchronization (serialization of transactions)
    - >> maximize concurrency between transactions (No one at a time)

Introduction 2/3 (*ACID Properties*)

- Transaction operations execute with following properties :
- **Atomicity**
  - All or Nothing
- **Consistency**
  - A transaction will leave the system's state to be consistent after a transaction completes.
- **Isolation**
  - Concurrent changes invisible (serializable)
- **Durability**
  - Committed updates persist (saved in permanent storage)

Introduction 3/3

- **Transaction** = cooperation between (client + recoverable objects + coordinator)
- Each transaction is created and managed by a coordinator
- Client call the recoverable object operation in passing *TID*
  - *Recoverable object operation = Deposit(**TID**, amount)*
  - Middleware support Transaction passes TID implicitly with all remote invocation between the transaction begin & end/abort
- Coordinator operations :
  - ***openTransaction()*** → *TID trans;*
  - ***closeTransaction(TID)*** → *(commit, abort);*
  - ***abortTransaction(TID)***

Concurrency model

Concurrency model

- Lost update problem
- Inconsistent retrieval problem
- Serialization of Transactions
  - Locks
  - Optimistic concurrency control
  - Timestamp ordering

Lost update problem

- تحدث عندما مناقلتين تقرأان قيمة قديمة لغرض و تستعملانها لحساب قيمته الجديدة و تكتبها بغض النظر عن الأخرى

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

Inconsistent retrieval problem

- تحدث عندما مناقلة تقرأ قيمة عدة أغراض وذلك بينما هذه الأغراض تتعرض إلى تحديث من قبل مناقلة أخرى.

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	⋮	

Serialization of Transactions

Serialization of Transactions (Lost update problem)

- عند وجود نزاع بين عمليات من مناقلات مختلفة على غرض ما
  - الحل = سلسلة الوصول إلى هذا الغرض بحيث تتم عمليات النزاع لمناقلة بعد تلك الخاصة بمناقلة أخرى

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

Methods for Transaction serialization

Locks

- Server يضع Lock للغرض قبل الوصول إليه معلماً بالـ TID الخاصة بالمناقلة و يتم إزالته عندما تنتهي المناقلة.
- المناقلة التي وضعت القفل هي الوحيدة التي يمكنها الوصول إلى الغرض و الأخريات تنتظر حتى تتم إزالته (أحياناً يمكن تشارك القفل)

Optimistic concurrency control

- المناقلة تعمل حتى تصل إلى commit و عندئذ يفحص الـ server إذا ما وجد نزاع على الأغراض التي وصلتها المناقلة و عندها فإنه abort يلغي المناقلة

Timestamp ordering

- Server يقارن بين زمن الفراءة/الكتابة على غرض مع زمن الغرض لمعرفة إذا العملية يمكن تنفيذها أو تأجيلها (المناقلة تنتظر) أو إلغاؤها.

Locks & Transaction serialization

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

Lock Compatibility

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

Lock implementation 1/2

```
public class Lock {
        private Object object;          // the object being protected by the lock
        private Vector holders;         // the TIDs of current holders
        private LockType lockType;  // the current type
        public synchronized void acquire(TID ,  a LockType  ) {
                while (/*another transaction holds the lock in conflicting mode*/) {
                        try {wait();}catch ( InterruptedException e){/*...*/ }
                }
                if (holders.isEmpty()) { // no TIDs  hold lock
                        holders.addElement(trans);
                        lockType  = aLockType;
                } else if (/*another transaction holds the lock, share it*/ ) ){
                        if(/* this transaction not a holder*/ )
                                holders.addElement(trans);
                } else if (/* this transaction holdes but needs a more exclusive lock*/)
                                lockType.promote();
                }
        }
        public synchronized void release(TID ) {
                holders.removeElement(trans);     // remove this holder
                // set locktype to none
                notifyAll();
        }
}
```

Lock implementation 2/2

```
public class LockManager {
    private Hashtable theLocks;

    public synchronized  void setLock(object, TID , lockType){
            Lock foundLock;
            // find the lock associated with object
             // if there isn't one, create it and add to the hashtable
            foundLock.acquire(TID, lockType);
    }

    public synchronized void unLock(TID) {
            Enumeration e = theLocks.elements();
            while(e.hasMoreElements()){
                    Lock aLock = (Lock)(e.nextElement());
                    if(/* trans is a holder of this lock*/ )
                            aLock.release(trans);
        }
    }
}
```

Recoverability from aborts

Dirty reads

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	

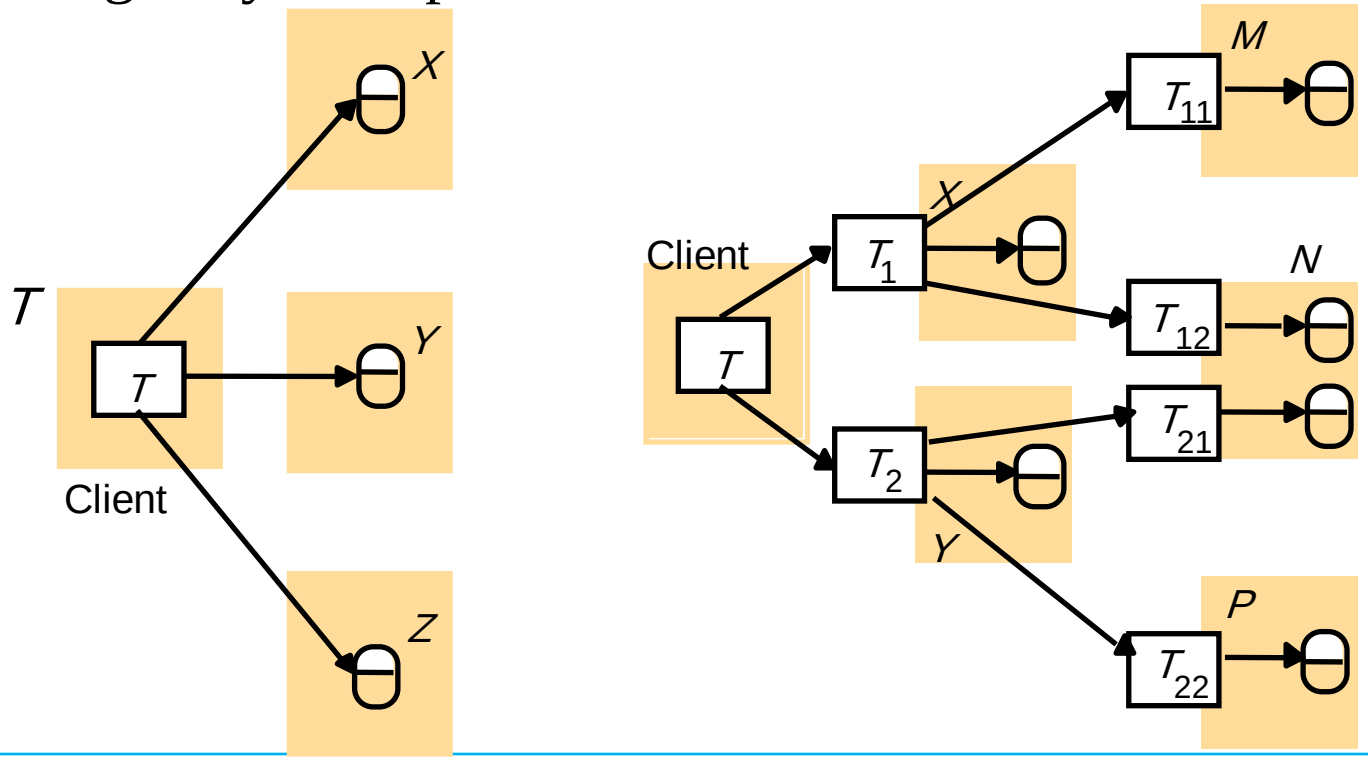
Premature writes

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

Distributed Transactions

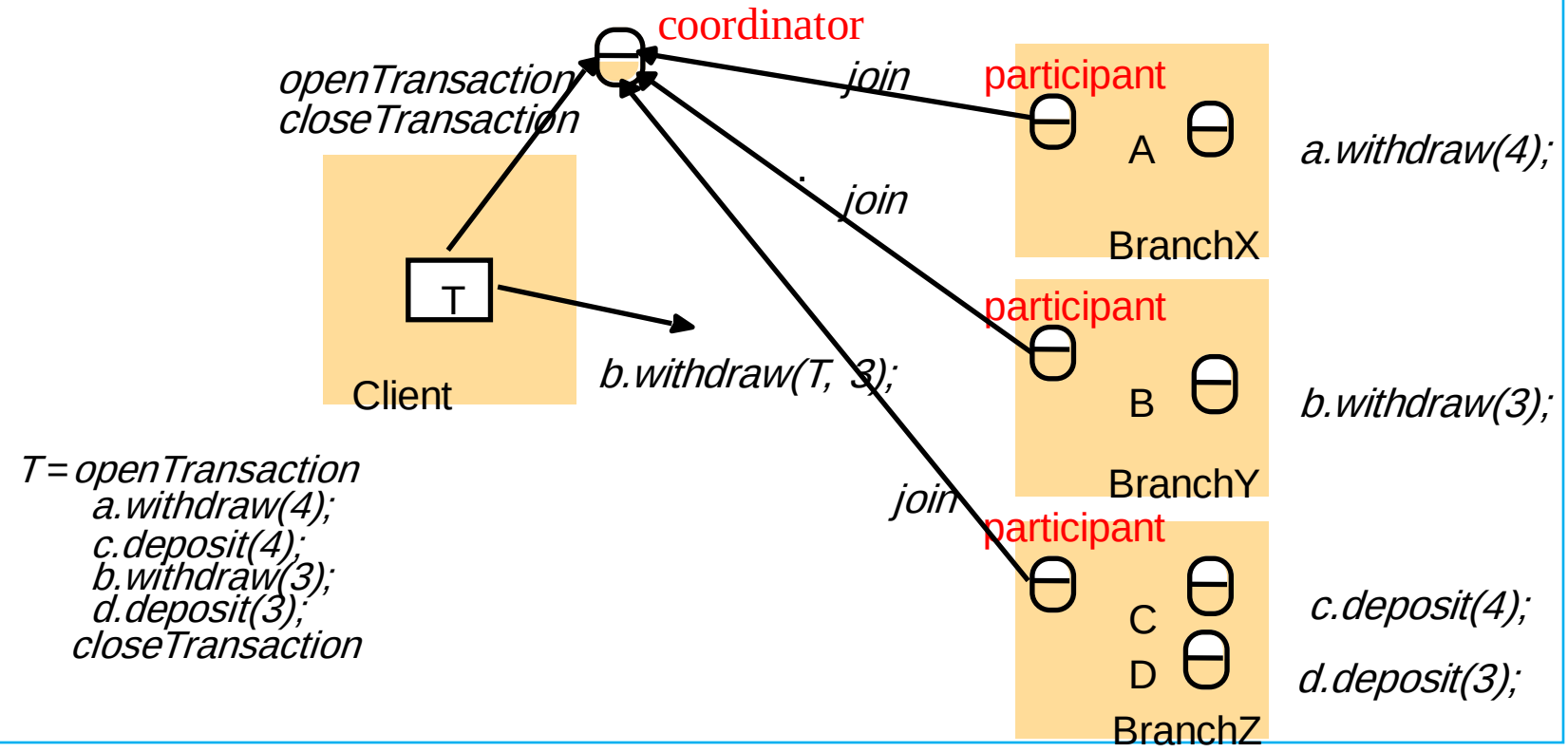
Distributed Transactions

- (flat or nested) Distributed Transaction = accesses objects managed by multiple servers.



Coordinator & Participants of distributed transaction

- One server = coordinator to commit or abort all operations in all servers.



Coordinator & Participants of distributed transaction

- Client starts by calling ***OpenTransaction()*** on Coordinator
- Coordinator returns ***TID***
  - Coordinator is responsible for committing or aborting the transaction
- Each server involved in transaction provides ***participant*** object
  - Participant keeps track of recoverable objects on its server
  - Participant cooperates with coordinator to implement commit protocol
- Coordinator records a list of participant references
- When client call an operation on a server. Then :
  - Object receiving the call, informs participant that it belongs to transaction
  - Participant joins coordinator (if not) by calling ***join(TID, participant ref)***

Distributed Two Phases Commit Protocol

- ***Phase 1 (voting phase):***
  - coordinator sends *canCommit*? to each participant.
  - participant that receives *canCommit*? replies with its vote (*Yes* / *No*) coordinator.
    - *Before sending Yes*, it prepares by saving objects in permanent storage.
    - *No* the participant aborts immediately.
- ***Phase 2 (completion according to outcome of vote):***
  - coordinator collects the votes (including its own).
    - no failures & all votes = *Yes.* Then, coordinator sends a *doCommit* to each participant.
    - Otherwise, coordinator sends *doAbort* to participants voted *Yes*.
  - Participants voted *Yes* wait for *doCommit / doAbort*.
    - participant that receives doAbort → aborts
    - participant that receives doCommit → commits & makes *haveCommitted* confirmation to coordinator.

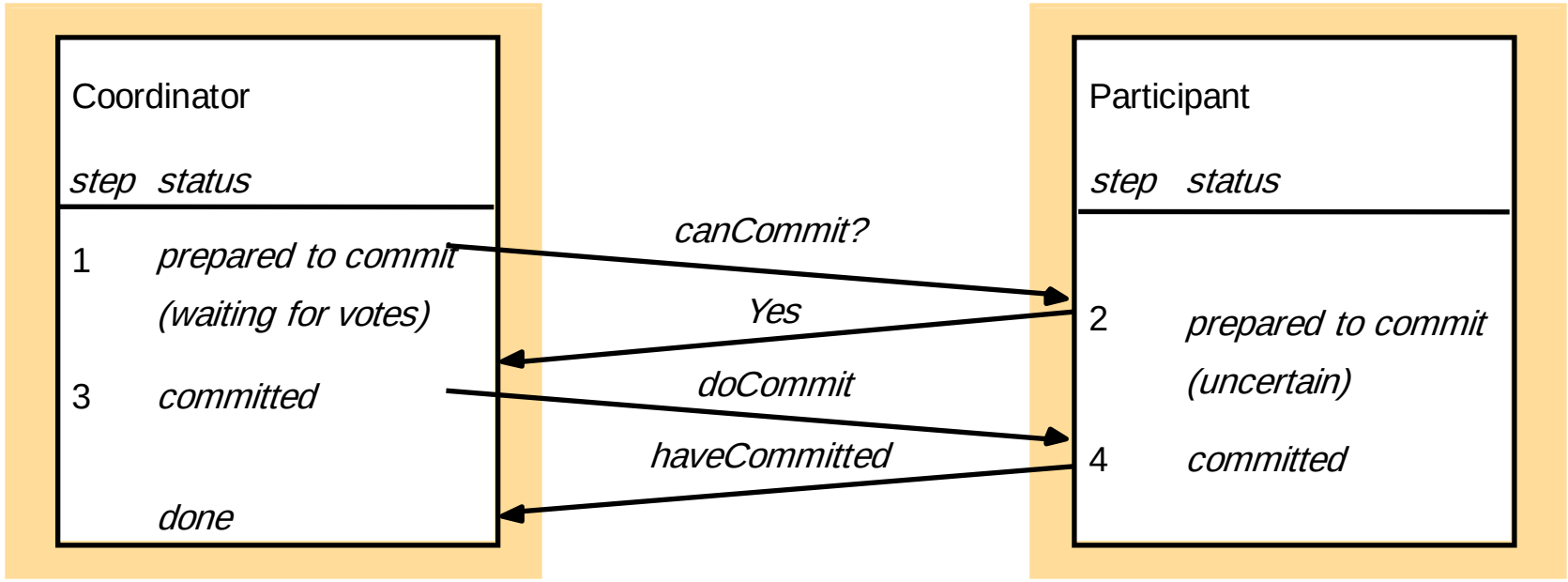
Distributed Two Phases Commit Protocol

Coordinator

haveCommitted*(TID, participant)*
getDecision*(TID) -> Yes / No*

Participant

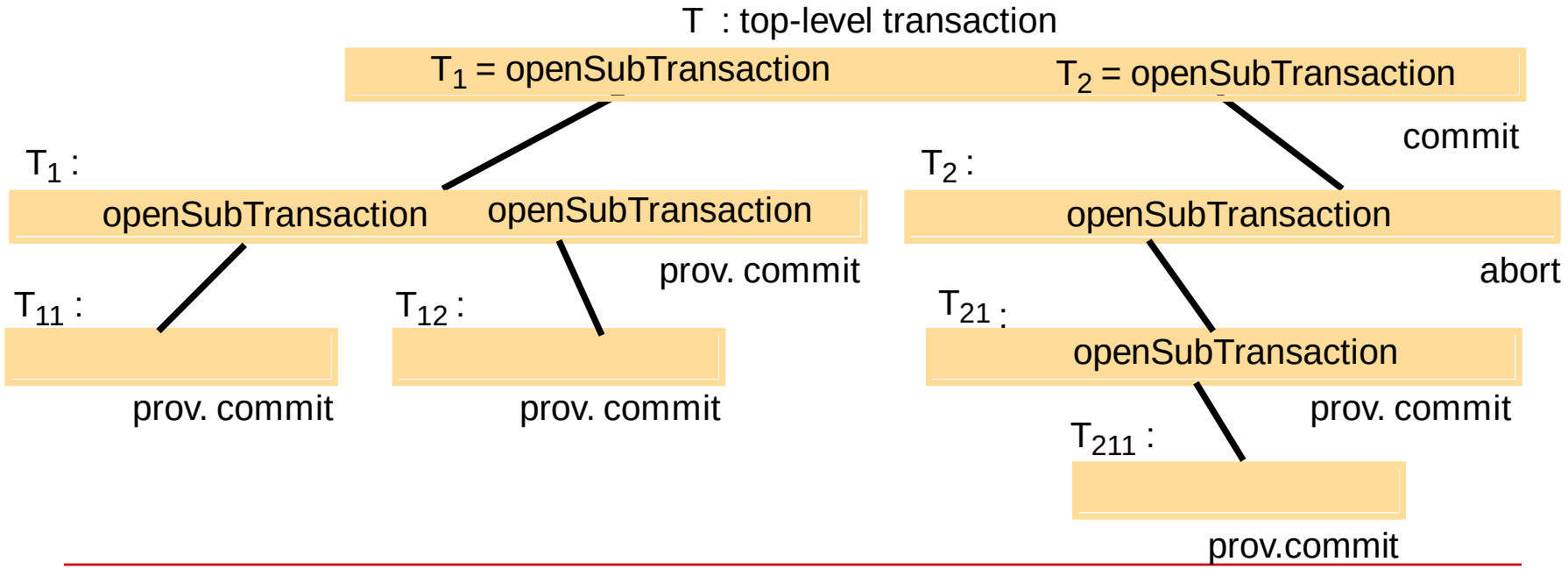
canCommit*(TID)-> Yes / No*
doCommit*(TID)*
doAbort*(TID)*



Nested Transactions

Nested Transactions

- Sub-transactions at one level may run concurrently
- Sub-transactions can commit or abort independently



Committing rules for nested Transactions

- A transaction may commit or abort after its child transactions have completed
- When sub_transaction completes, it may abort (finally) or commit (provisionally)
- When parent transaction abort. Then, its sub-transactions abort
- When sub-transaction aborts. The parent transaction can abort or not
- Top-level transaction commits. Then, all sub-transactions that have provisionally committed can commit too if their ancestors have not aborted.

Questions ?