Universität Stuttgart
Master of Science

Distributed Transactions

Marco Aiello

Transactions

- A *transaction* is a group of operations on an object to be executed coherently according to some policy, typically, atomicity (i.e., all or nothing)

Single server, single client

- If one unique “client” of an “object” at one server then locking (e.g. `synchronized` in java) is enough

ACIDity

A tomicity

C onsistency

I solation

D urability

For atomicity and durability, objects must be *recoverable*

Transactions facts

- The goal of any sever supporting transactions is to maximize concurrency
- *Serializability* is often the requirement
- Transaction support can be part of the middleware (e.g, CORBA's Object Transaction Service, the TID are implicit)
- Transactions are created and managed by a *coordinator*

Figure 16.3
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.

Figure 16.4
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		

Figure 16.1
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

Figure 16.2
A client's banking transaction

Transaction T:
a.withdraw(100);
b.deposit(100);
c.withdraw(200);
b.deposit(200);

Figure 16.5
The lost update problem

a,b,c are \$100, \$200, \$300, resp.

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

Figure 16.7
A serially equivalent interleaving of *T* and *U*

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)*	
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

Figure 16.6
The inconsistent retrievals problem

a,b are $200

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		

Figure 16.8
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
		total = total+c.getBalance()	
		...	

Conflicting operations

- A pair of operations is *conflicting* when their combined effect depends on the order of execution

Figure 16.9
Read and *write* operation conflict rules

Operations of different transactions		*Conflict*	*Reason*
read	*read*	No	Because the effect of a pair of *read* operations does not depend on the order in which they are executed
read	*write*	Yes	Because the effect of a *read* and a *write* operation depends on the order of their execution
write	*write*	Yes	Because the effect of a pair of *write* operations depends on the order of their execution

Figure 16.10
A non-serially equivalent interleaving of operations of transactions *T* and *U*

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

Serially equivalence

Serially equivalence requires either one of:

1. *T* accesses *i* before *U* and *T* access *j* before *U*
2. *U* accesses *i* before *T* and *U* access *j* before *T*

Can be obtained with:

1. locks
2. optimistic concurrency
3. timestamping

Recoverability from aborts

- How to take into account that transactions may abort while executing?
  - *Recoverability:* any possible dirty read forces delayed commit
  - *Avoid cascading aborts:* transactions are allowed only to read values which are committed

Figure 16.11
A dirty read when transaction *T* aborts

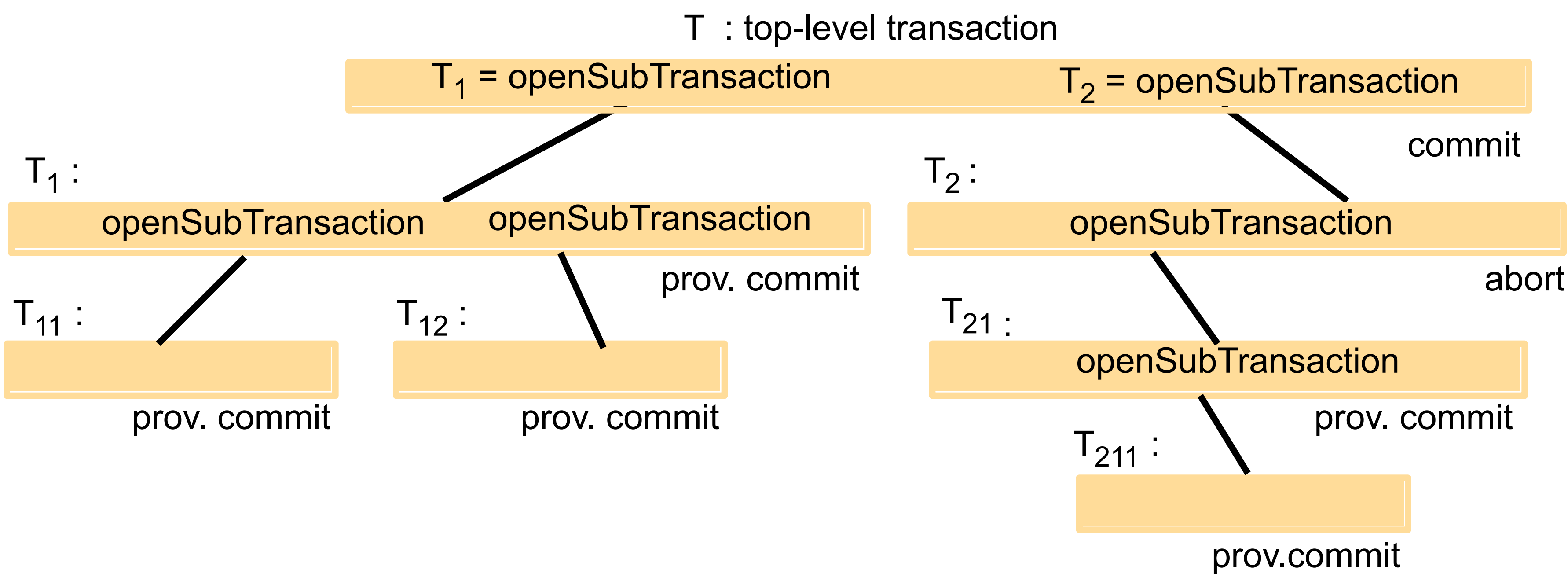
Transaction *T*:		**Transaction *U*:**	
a.getBalance() *a.setBalance(balance + 10)*		*a.getBalance()* *a.setBalance(balance + 20)*	
balance = a.getBalance()	\$100		
a.setBalance(balance + 10)	\$110		
		balance = a.getBalance()	\$110
		a.setBalance(balance + 20)	\$130
		commit transaction	
abort transaction			

Figure 16.12
Overwriting uncommitted values

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

Figure 16.13

Nested transactions



Nested Transactions

- Subtransactions at one level may run concurrently with other subtransactions at the same level
- Subtransactions can commit or abort independently

Nested transaction commit rules

1. A transaction may commit or abort only after its children transactions have completed
2. When a subtransaction completes, it decides independently to provisionally commit or abort
3. When a parent aborts, all its subtransactions abort
4. When a subtransaction aborts, the parent decides whether to abort or not
5. When the top-level transaction commits, all of the sub transaction that have provisionally committed can commit too

How to guarantee serial equivalence?

1. Locks (seen in Operating Systems)
2. Optimistic concurrency control
3. Timestamping

Figure 16.14
Transactions *T* and *U* with exclusive locks

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*

Figure 16.15
Lock compatibility

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

Figure 16.16
Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:
   - (a) If the object is not already locked, it is locked and the operation proceeds.
   - (b) If the object has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   - (c) If the object has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   - (d) If the object has already been locked in the same transaction, the lock will be promoted if necessary and the operation proceeds. (Where promotion is prevented by a conflicting lock, rule (b) is used.)
2. When a transaction is committed or aborted, the server unlocks all objects it locked for the transaction.

Figure 16.19
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*
		• • •	
• • •		• • •	
• • •		• • •	

Figure 16.20
The wait-for graph for Figure 16.19

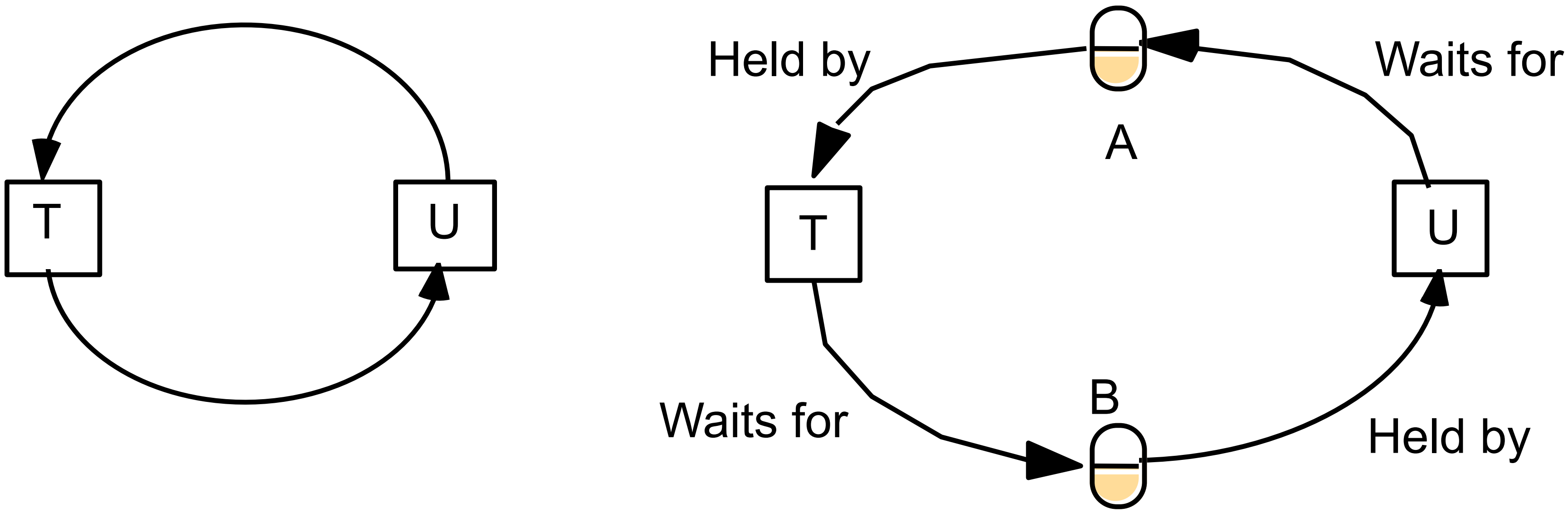


Figure 16.21
A cycle in a wait-for graph

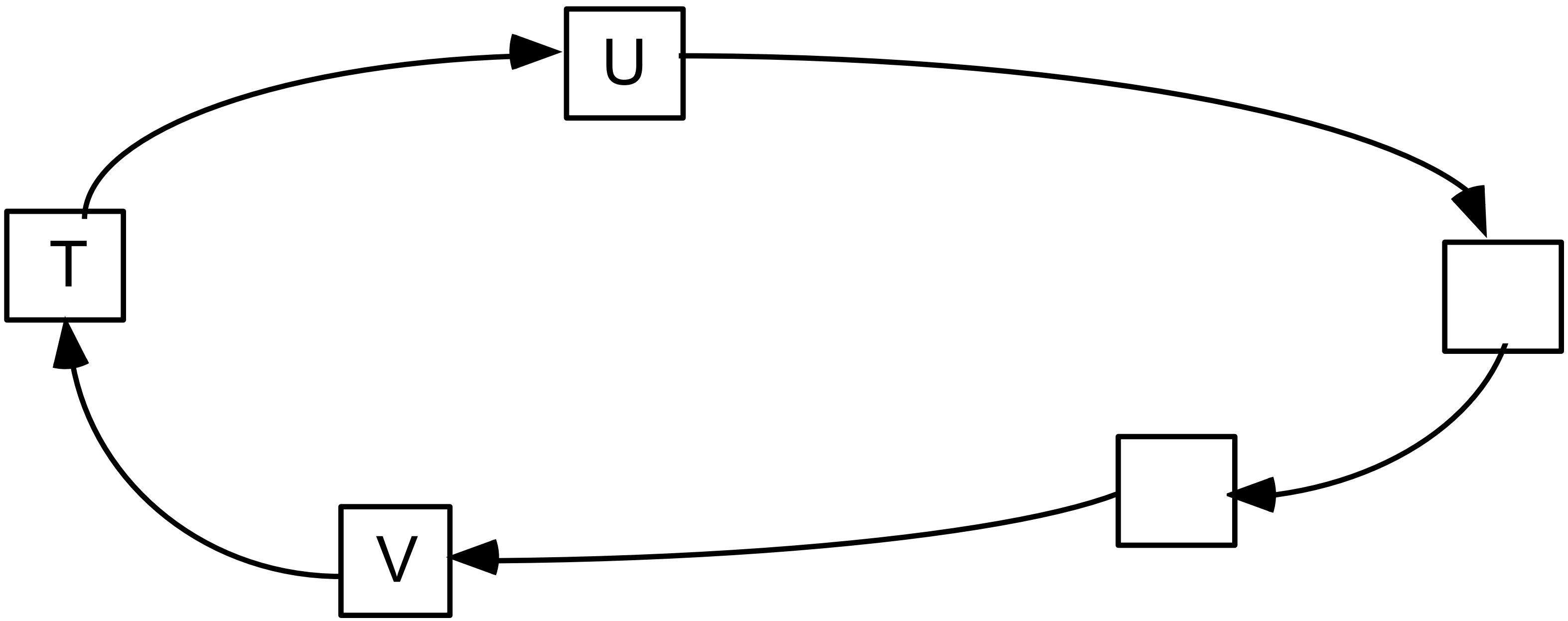


Figure 16.22
Another wait-for graph

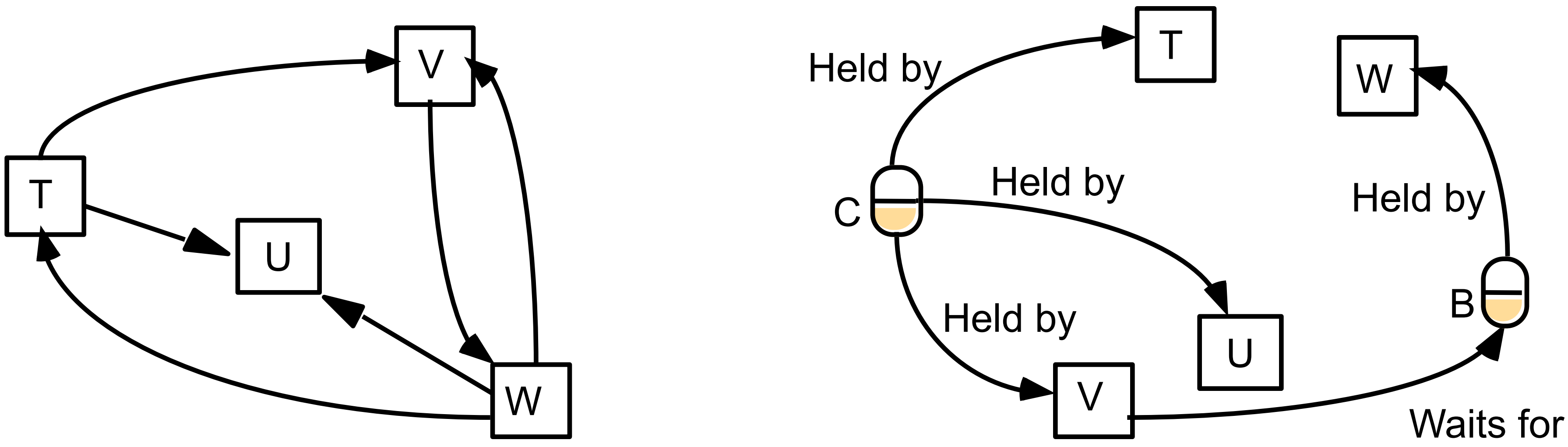


Figure 16.23
Resolution of the deadlock in Figure 15.19

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*

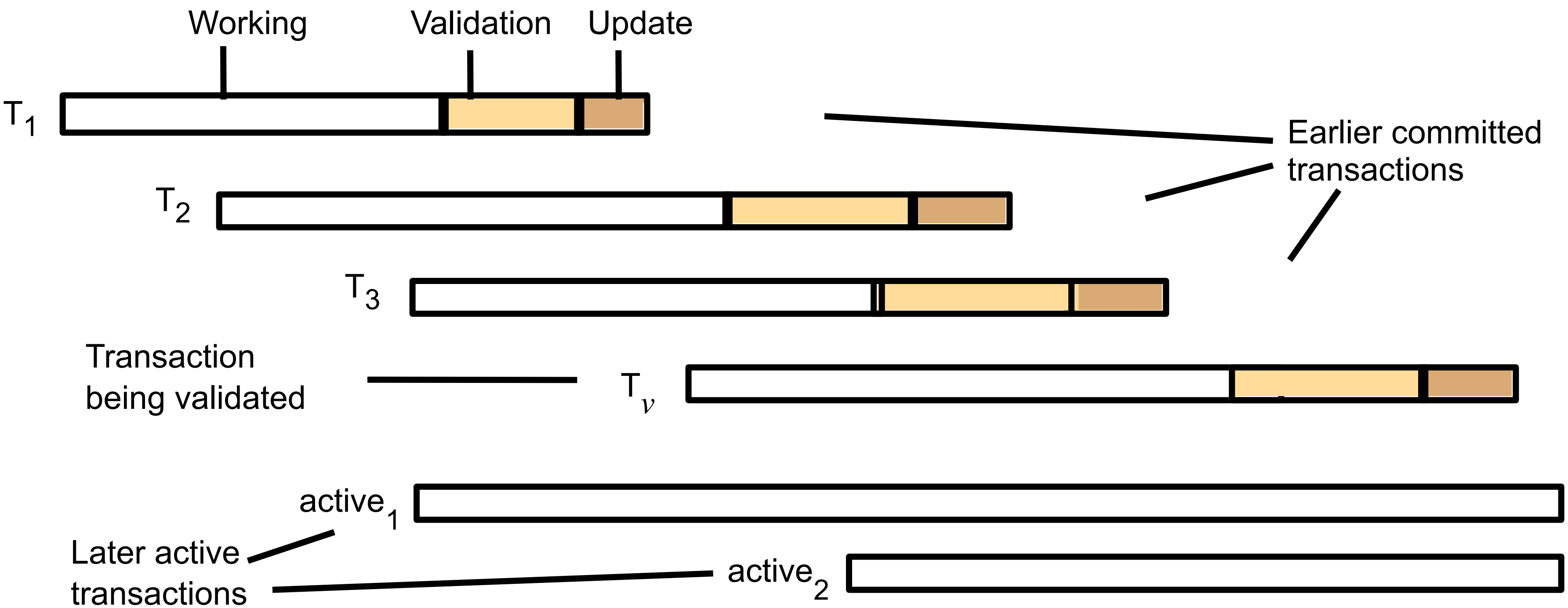
Optimistic concurrency control

- Locking has drawbacks: overheads, deadlocks, limited concurrency to avoid cascading aborts
- Optimistic: transactions are allowed to proceed until ready to commit. If a conflict occurs, then abort.
- How? In phases:
  - *Working phase:* execute operations. If concurrent transactions, many values of an object may coexist. Keep *write set* and *read set* for each transaction.
  - *Validation phase:* either no conflict or call conflict resolution policy.
  - *Update phase:* if validated, make changes permanent

Table on page 708
Serializability of transaction *T* with respect to transaction T_i

T_v	T_i	*Rule*	
write	*read*	1.	T_i must not read objects written by T_v
read	*write*	2.	T_v must not read objects written by T_i
write	*write*	3.	T_i must not write objects written by T_v and T_v must not write objects written by T_i

Figure 16.28
Validation of transactions



Validation of Transactions

Backward validation of transaction T_v

boolean valid = true;

for (int T_i = $startTn$+1; T_i <= $finishTn$; T_i++){

if (read set of T_v intersects write set of T_i) valid = false;

}

- the read set of T_v must be compared with the write sets of T_2 T_3

Forward validation of transaction T_v

boolean valid = true;

for (int T_{id} = $active1$; T_{id} <= $activeN$; T_{id}++){

if (write set of T_v intersects read set of T_{id}) valid = false;

}

- the write set of T_v must be compared with the read sets of $active_{1,2}$

starvation?

Optimistic concurrency control

- Alternatives to conflict resolution:
  - defer validation until all conflicting transactions have finished
  - abort all the conflicting transactions except the one being validated
  - abort the transaction being validated

Timestamp ordering

- A transaction's request to <u>write</u> an object is valid only if that object was last read and written by earlier transactions.
- A transaction's request to <u>read</u> an object is valid only if that object was last written by an earlier transaction.
- I.e., tentative version of each object are committed in the order determined by the timestamps of the transactions.

Figure 16.29
Operation conflicts for timestamp ordering

Rule	T_c	T_i	
1.	*write*	*read*	T_c must not *write* an object that has been *read* by any T_i where $T_i > T_c$ this requires that $T_c \geq$ the maximum read timestamp of the object.
2.	*write*	*write*	T_c must not *write* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.
3.	*read*	*write*	T_c must not *read* an object that has been *written* by any T_i where $T_i > T_c$ this requires that $T_c >$ write timestamp of the committed object.

Examples

- **Dropbox:** uses an optimistic form of concurrency control, keeps track of conflicts, granularity of files
- **Google apps:** finer granularity, user resolve conflict manually
- **Wikipedia:** editing concurrency is optimistic, manual conflict resolution
- **Amazon Dynamo:** key value storage. No isolation guarantee, optimistic concurrency, conflict resolution via application logic or timestamping (last write wins)

Distributed transactions

- What if more servers hold the objects accessed by the operations of a transaction? *Distributed transactions.*
- *Coordinator:* ensuring the same outcome at all of the servers

Figure 17.1
Distributed transactions

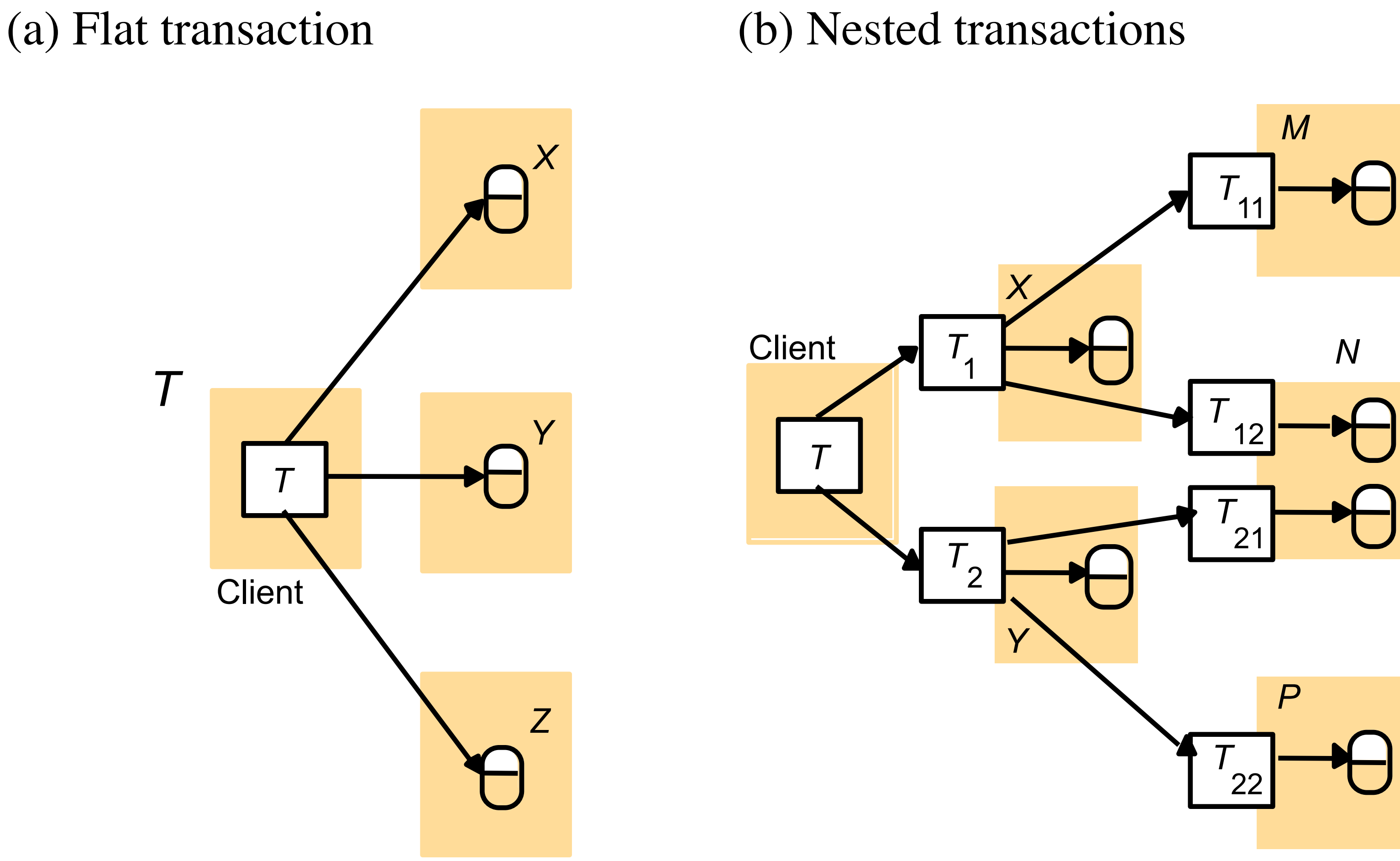
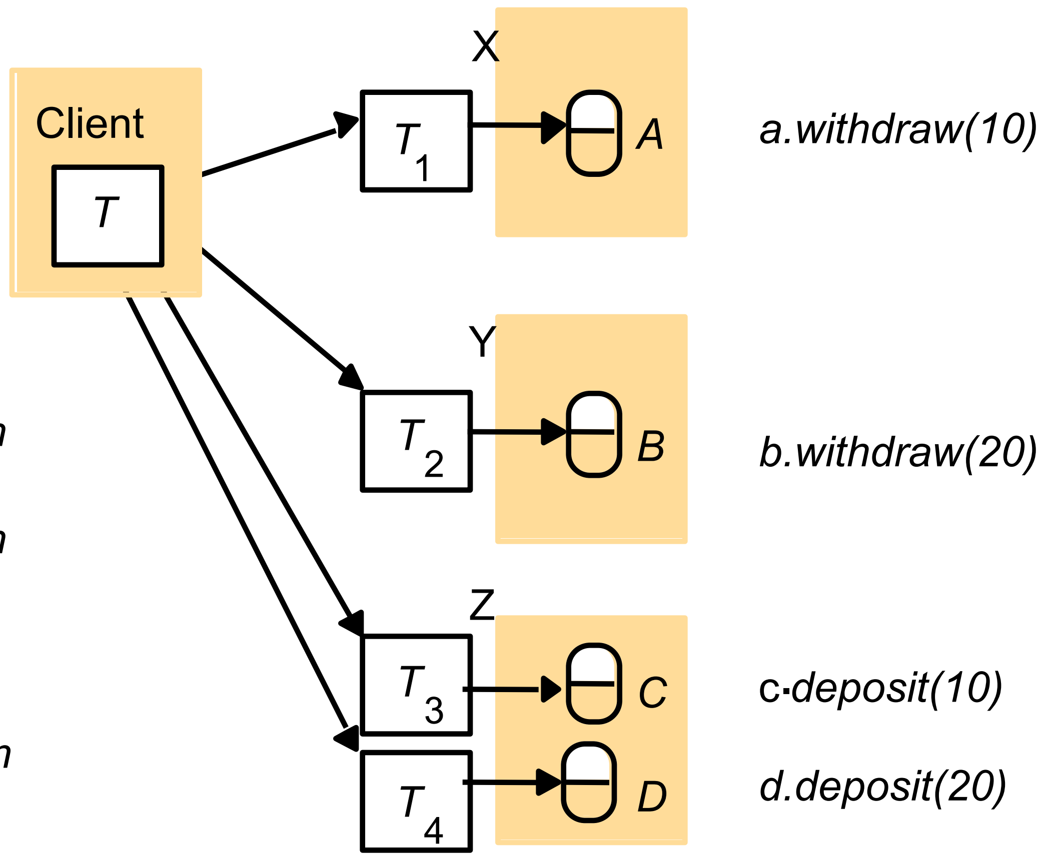


Figure 17.2
Nested banking transaction

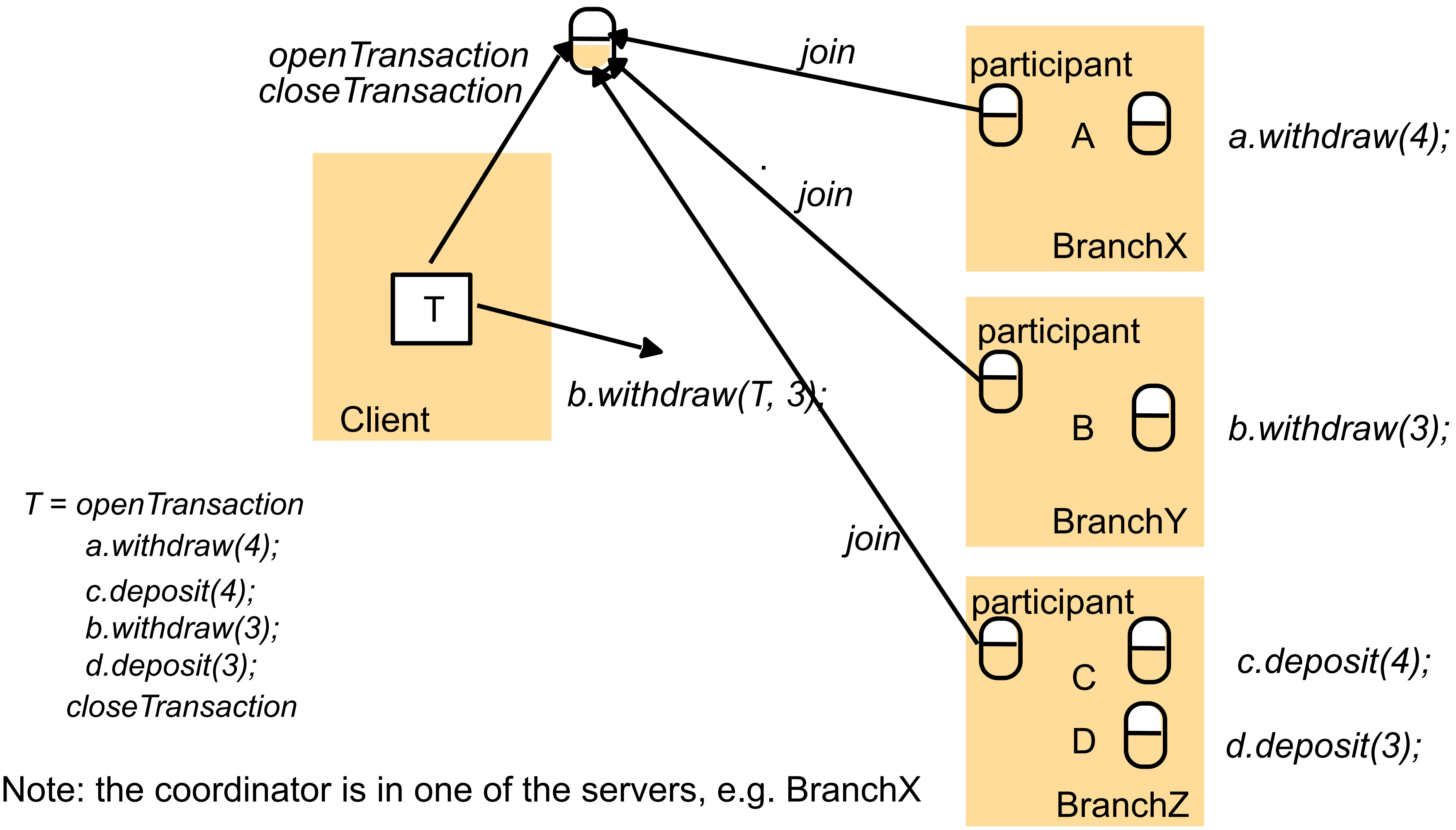


T = openTransaction

openSubTransaction
a.withdraw(10);
openSubTransaction
b.withdraw(20);
openSubTransaction
c.deposit(10);
openSubTransaction
d.deposit(20);

closeTransaction

Figure 17.3
A distributed banking transaction



Note: the coordinator is in one of the servers, e.g. BranchX

Figure 17.4
Operations for two-phase commit protocol

canCommit?(trans)-> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote.

doCommit(trans)

Call from coordinator to participant to tell participant to commit its part of a transaction.

doAbort(trans)

Call from coordinator to participant to tell participant to abort its part of a transaction.

haveCommitted(trans, participant)

Call from participant to coordinator to confirm that it has committed the transaction.

getDecision(trans) -> Yes / No

Call from participant to coordinator to ask for the decision on a transaction after it has voted *Yes* but has still had no reply after some delay. Used to recover from server crash or delayed messages.

Figure 17.5
The two-phase commit protocol

Phase 1 (voting phase):

1. The coordinator sends a *canCommit*? request to each of the participants in the transaction.
2. When a participant receives a *canCommit*? request it replies with its vote (*Yes* or *No*) to the coordinator. Before voting *Yes*, it prepares to commit by saving objects in permanent storage. If the vote is *No* the participant aborts immediately.

Phase 2 (completion according to outcome of vote):

3. The coordinator collects the votes (including its own).

(a) If there are no failures and all the votes are *Yes* the coordinator decides to commit the transaction and sends a *doCommit* request to each of the participants.

(b) Otherwise the coordinator decides to abort the transaction and sends *doAbort* requests to all participants that voted *Yes*.

4. Participants that voted *Yes* are waiting for a *doCommit* or *doAbort* request from the coordinator. When a participant receives one of these messages it acts accordingly and in the case of commit, makes a *haveCommitted* call as confirmation to the coordinator.

Figure 17.6
Communication in two-phase commit protocol

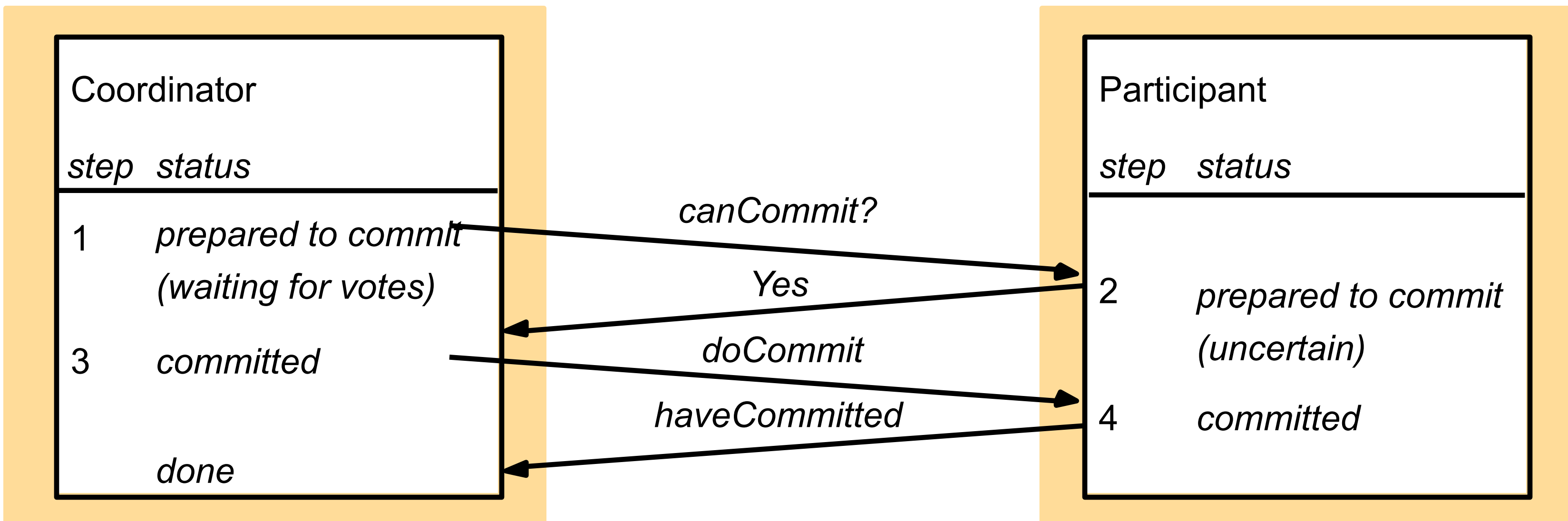


Figure 17.7
Operations in coordinator for nested transactions

openSubTransaction(trans) -> subTrans

Opens a new subtransaction whose parent is *trans* and returns a unique subtransaction identifier.

getStatus(trans)-> committed, aborted, provisional

Asks the coordinator to report on the status of the transaction *trans*. Returns values representing one of the following: *committed*, *aborted*, *provisional*.

Complexity

- In case of no failures
  - Cost in messages is proportional to 3N (with N servers)
  - Cost in time: 3 rounds of messages
- In case of failures (server replacement)
  - guaranteed to complete eventually, but with no bound on the complexity
- Three phase commit protocol exist to avoid waiting for uncertain outcome

Figure 17.8
Transaction *T* decides whether to commit

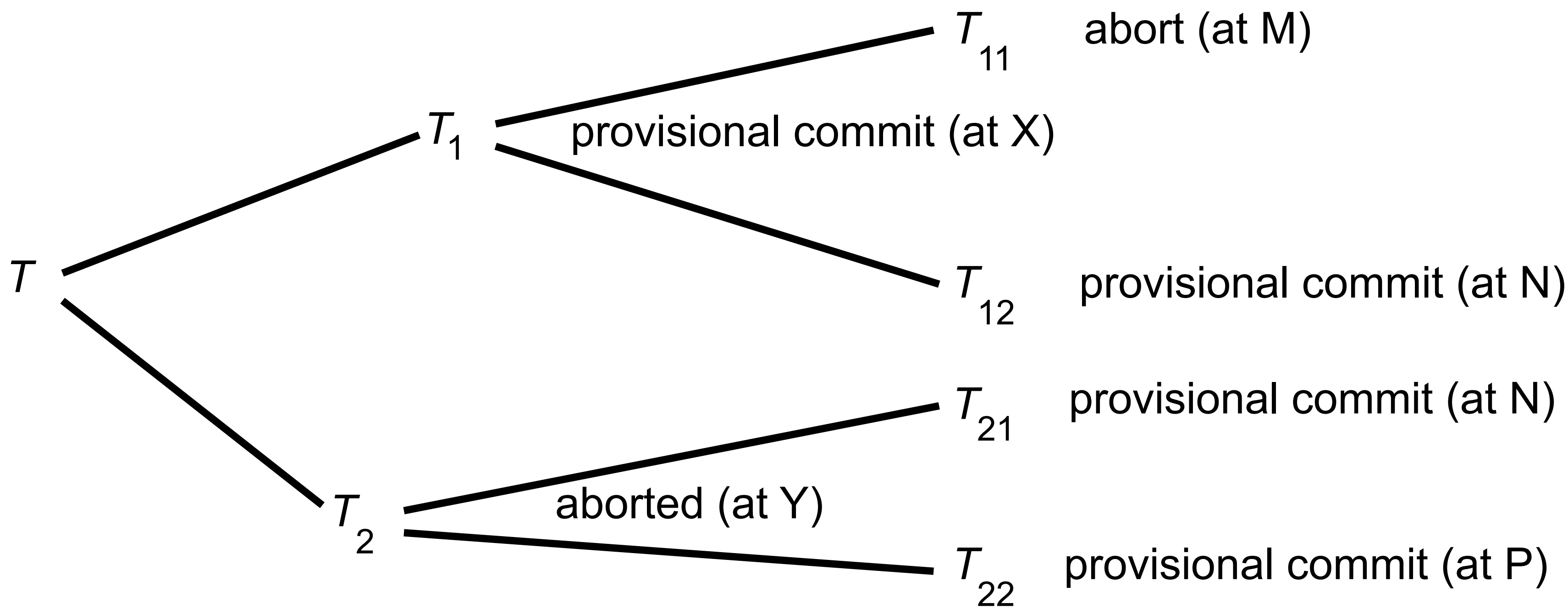


Figure 17.9
Information held by coordinators of nested transactions

Coordinator of transaction	*Child transactions*	*Participant*	*Provisional commit list*	*Abort list*
T	T_1, T_2	yes	T_1, T_{12}	T_{11}, T_2
T_1	T_{11}, T_{12}	yes	T_1, T_{12}	T_{11}
T_2	T_{21}, T_{22}	no (aborted)		T_2
T_{11}		no (aborted)		T_{11}
T_{12}, T_{21}		T_{12} but not T_{21} *	T_{21}, T_{12}	
T_{22}		no (parent aborted)	T_{22}	

*T_{21}'s parent has aborted

Figure 17.10
canCommit? for hierarchic two-phase commit protocol

canCommit?(trans, subTrans) -> Yes / No

Call a coordinator to ask coordinator of child subtransaction whether it can commit a subtransaction *subTrans*. The first argument *trans* is the transaction identifier of top-level transaction. Participant replies with its vote *Yes / No*.

Figure 17.11
canCommit? for flat two-phase commit protocol

canCommit?(trans, abortList) -> Yes / No

Call from coordinator to participant to ask whether it can commit a transaction. Participant replies with its vote *Yes* / *No*.

Other transaction models

- **ACID** is the traditional model (consistency)
- **BASE** often with NoSQL databases (availability, scalability)
  - **B**asically **A**vailable: via replication
  - **S**oft state: with possibly inconsistent values
  - **E**ventually consistent: with reads possible before consistency

CAP Theorem

NoSQL (not only SQL) databases:
- key value, wide column, graph
- horizontal scaling
- partial replication