Slides for Chapter 17: Distributed transactions

Figure 17.1
Distributed transactions

(a) Flat transaction

(b) Nested transactions

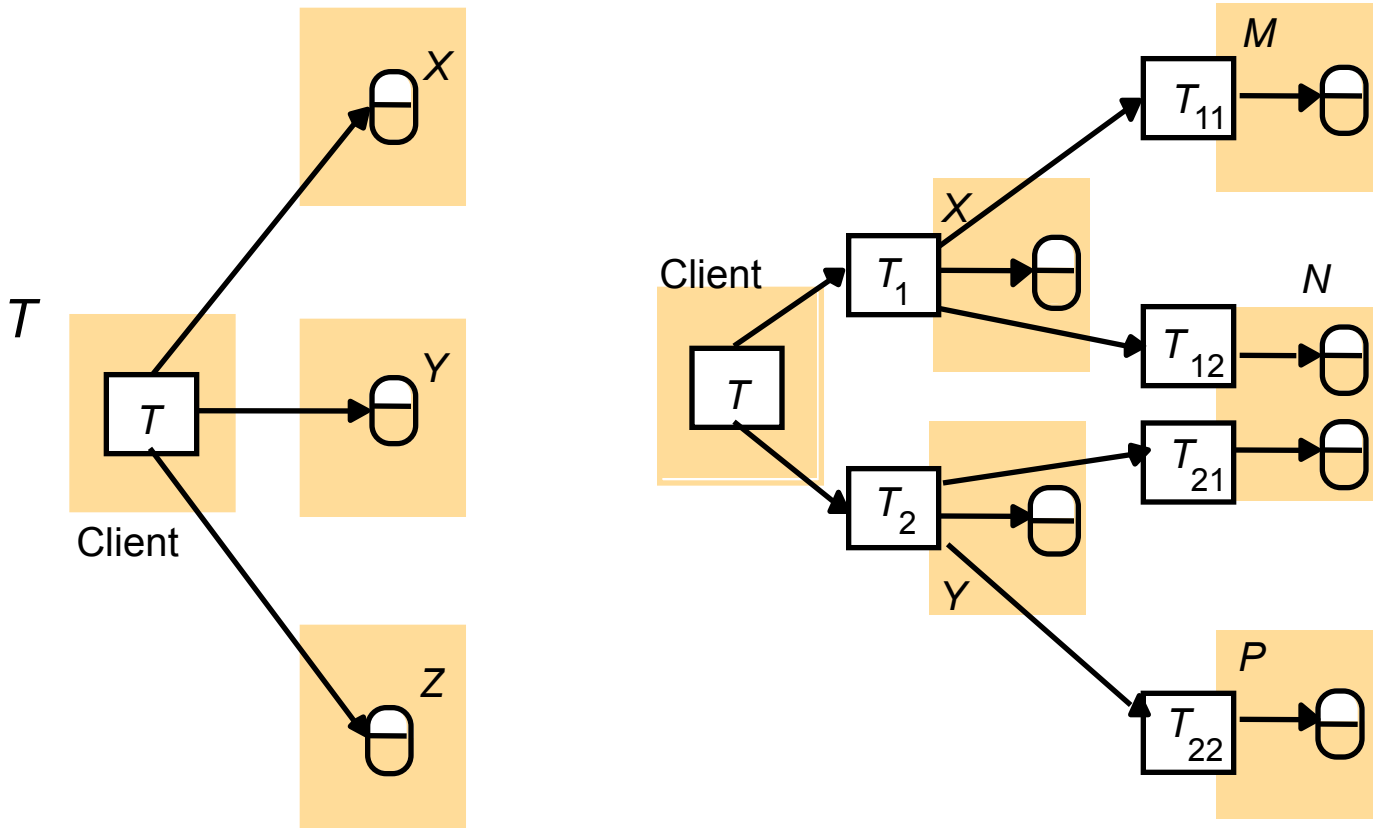


Figure 17.2
Nested banking transaction

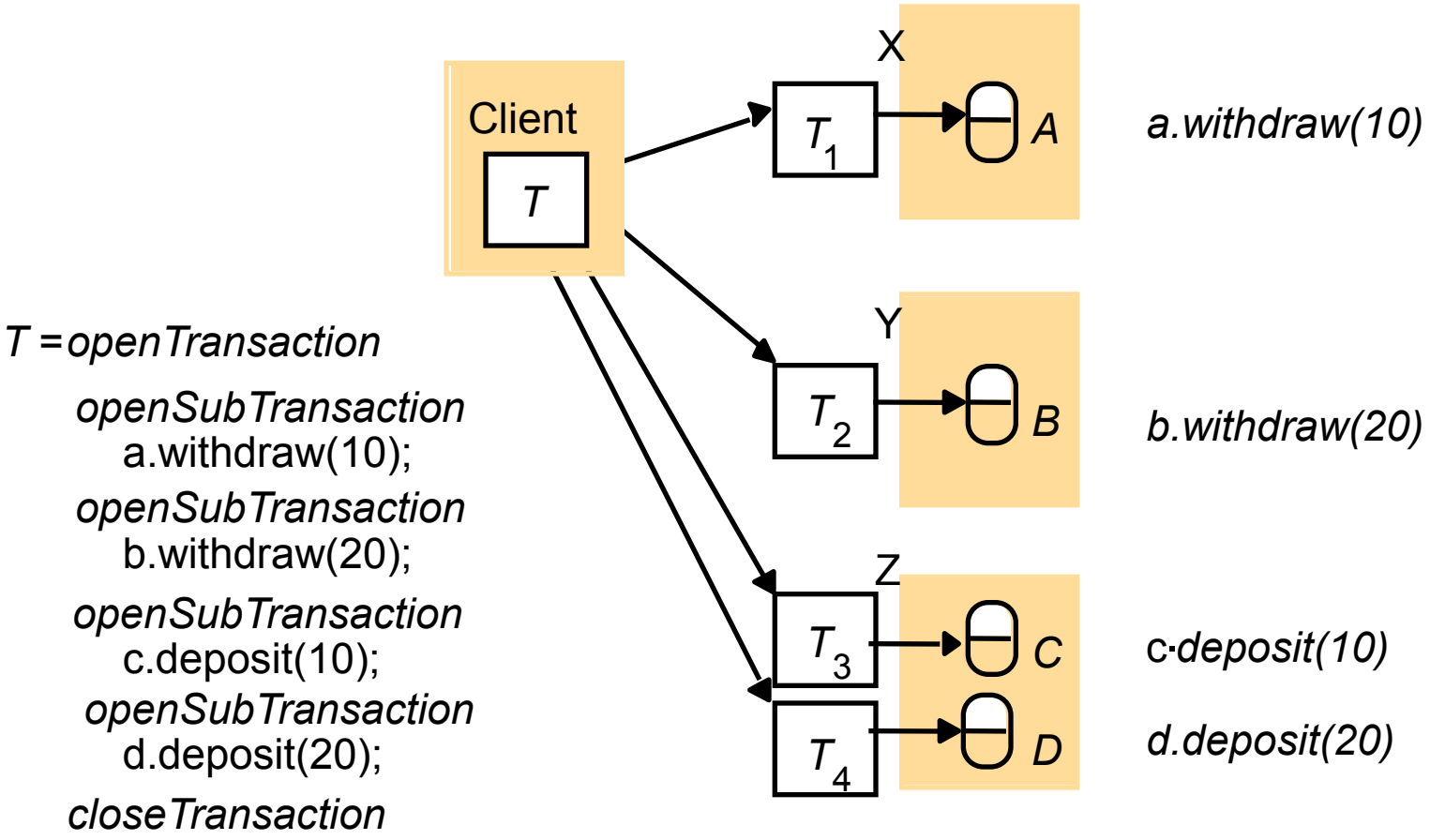
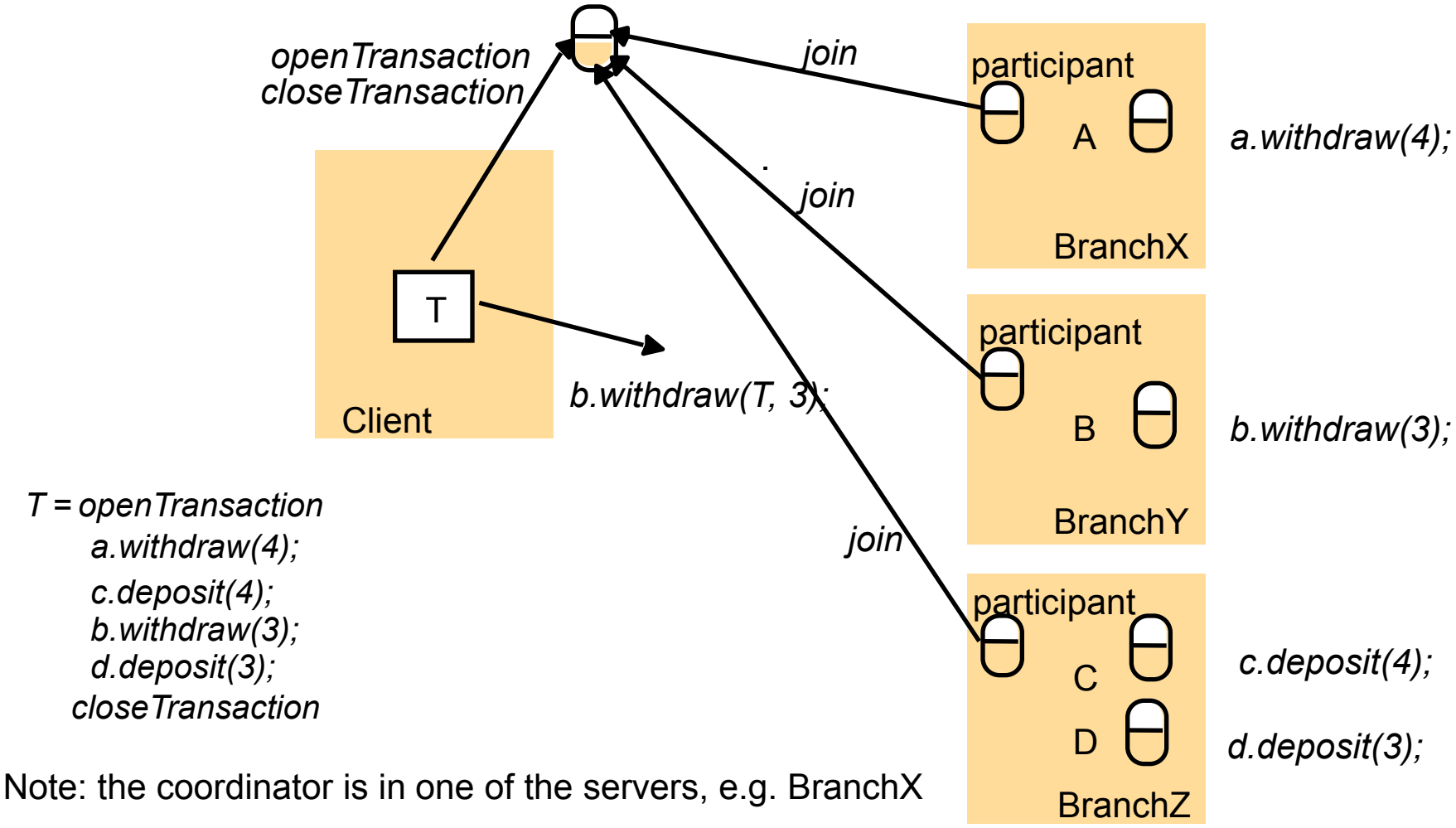


Figure 17.3
A distributed banking transaction



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

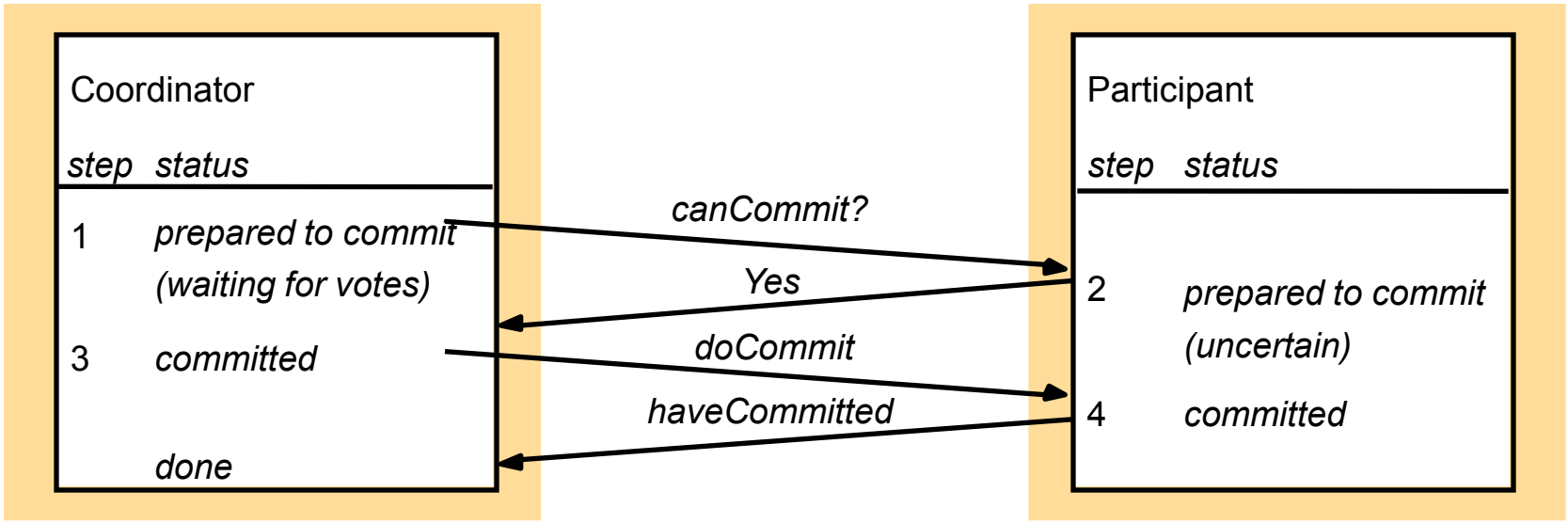


Figure 17.7
Operations in coordinator for nested transactions

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*.

Figure 17.8
Transaction *T* decides whether to commit

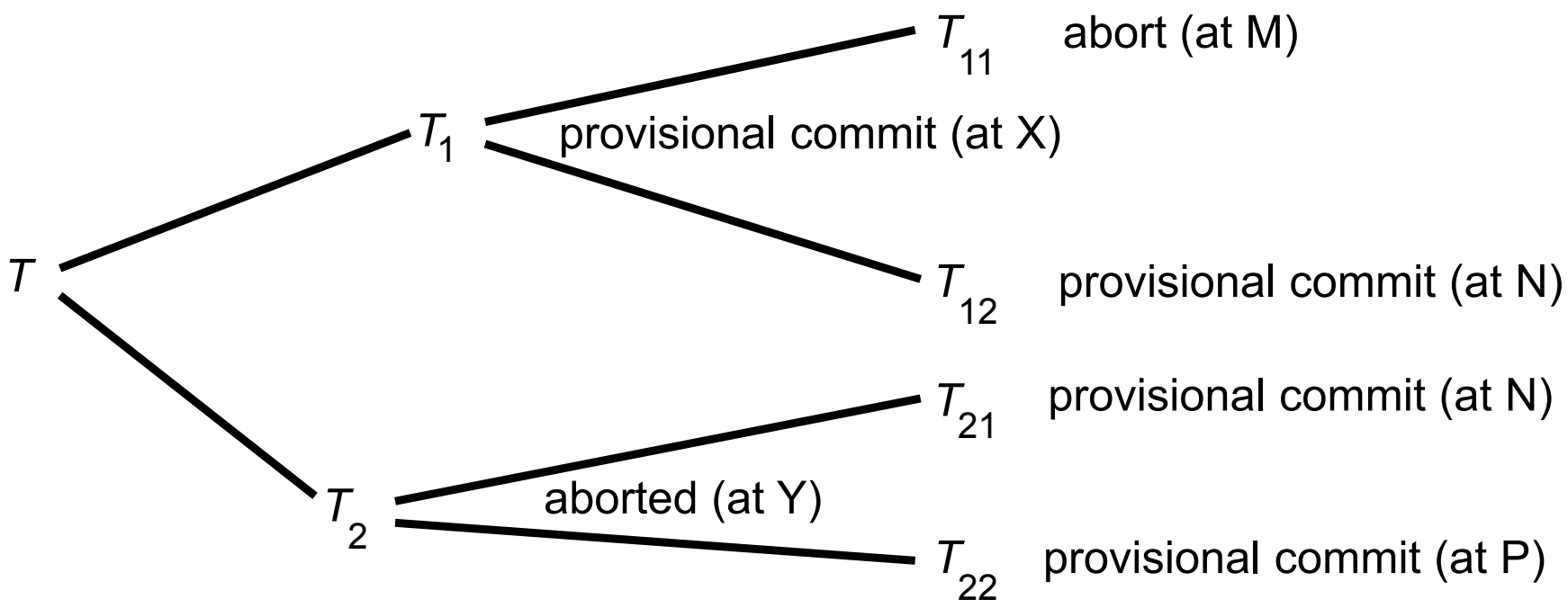


Figure 17.9
Information held by coordinators of nested transactions

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 protoco

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*.

Figure 17.12
Interleavings of transactions *U*, *V* and *W*

U		*V*		*W*	
d.deposit(10)	lock *D*				
		b.deposit(10)	lock *B* at *Y*		
a.deposit(20)	lock *A* at *X*				
				c.deposit(30)	lock *C* at *Z*
b.withdraw(30)	wait at *Y*				
		c.withdraw(20)	wait at *Z*		
				a.withdraw(20)	wait at *X*

Figure 17.13
Distributed deadlock

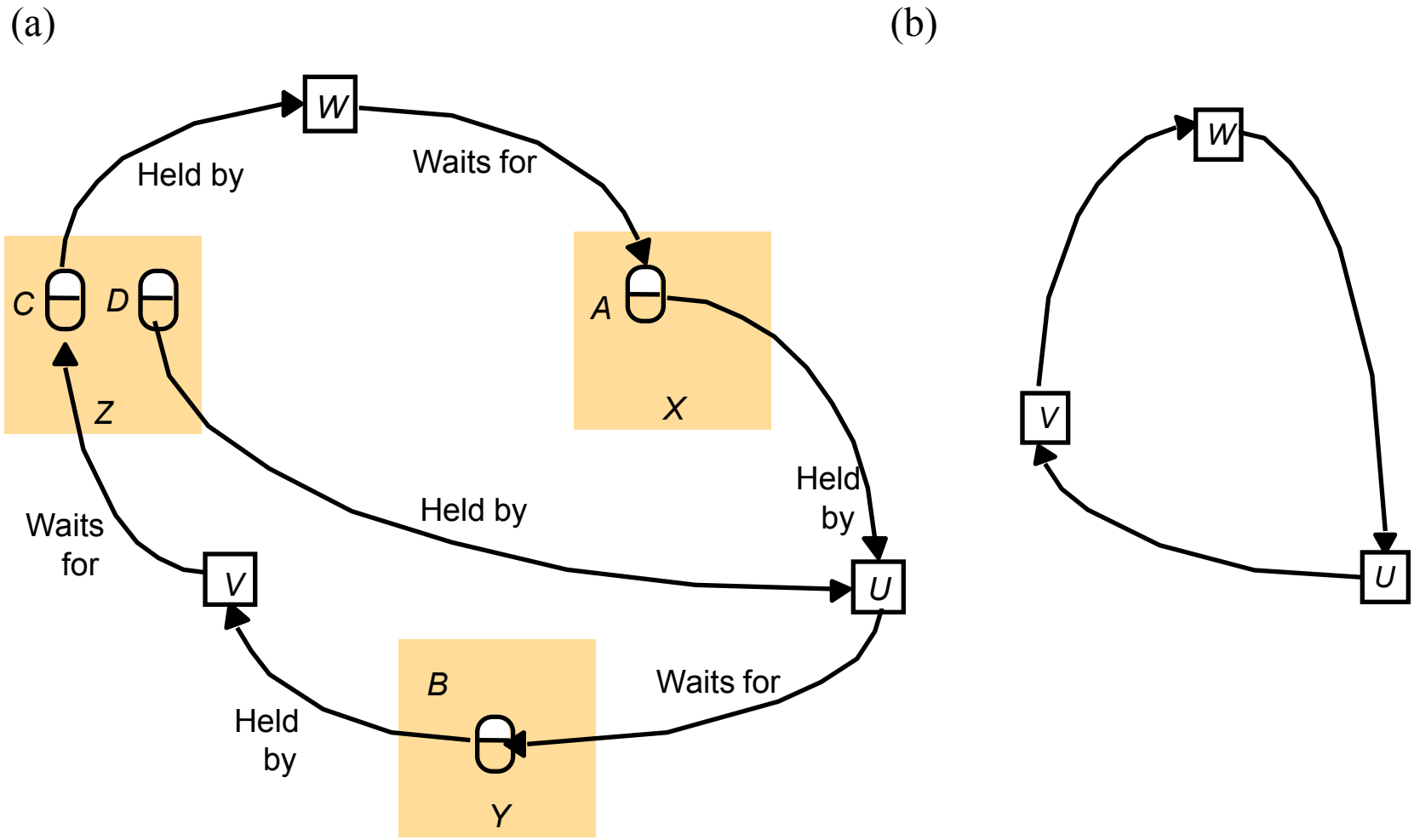


Figure 17.14
Local and global wait-for graphs

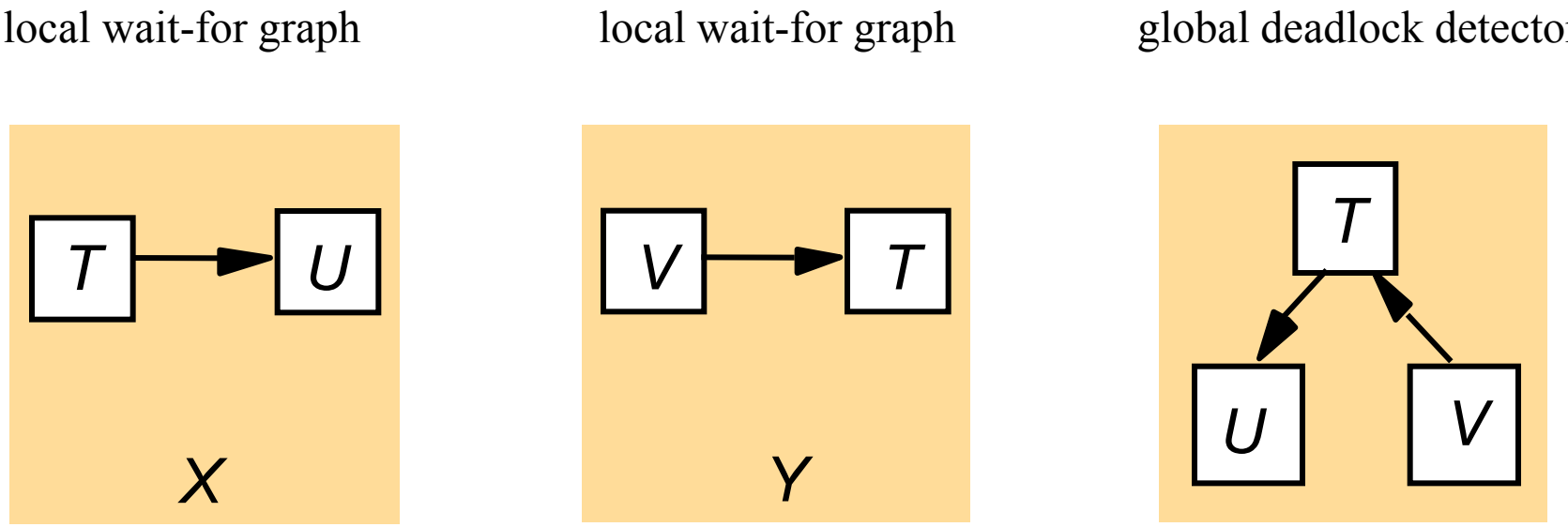


Figure 17.15
Probes transmitted to detect deadlock

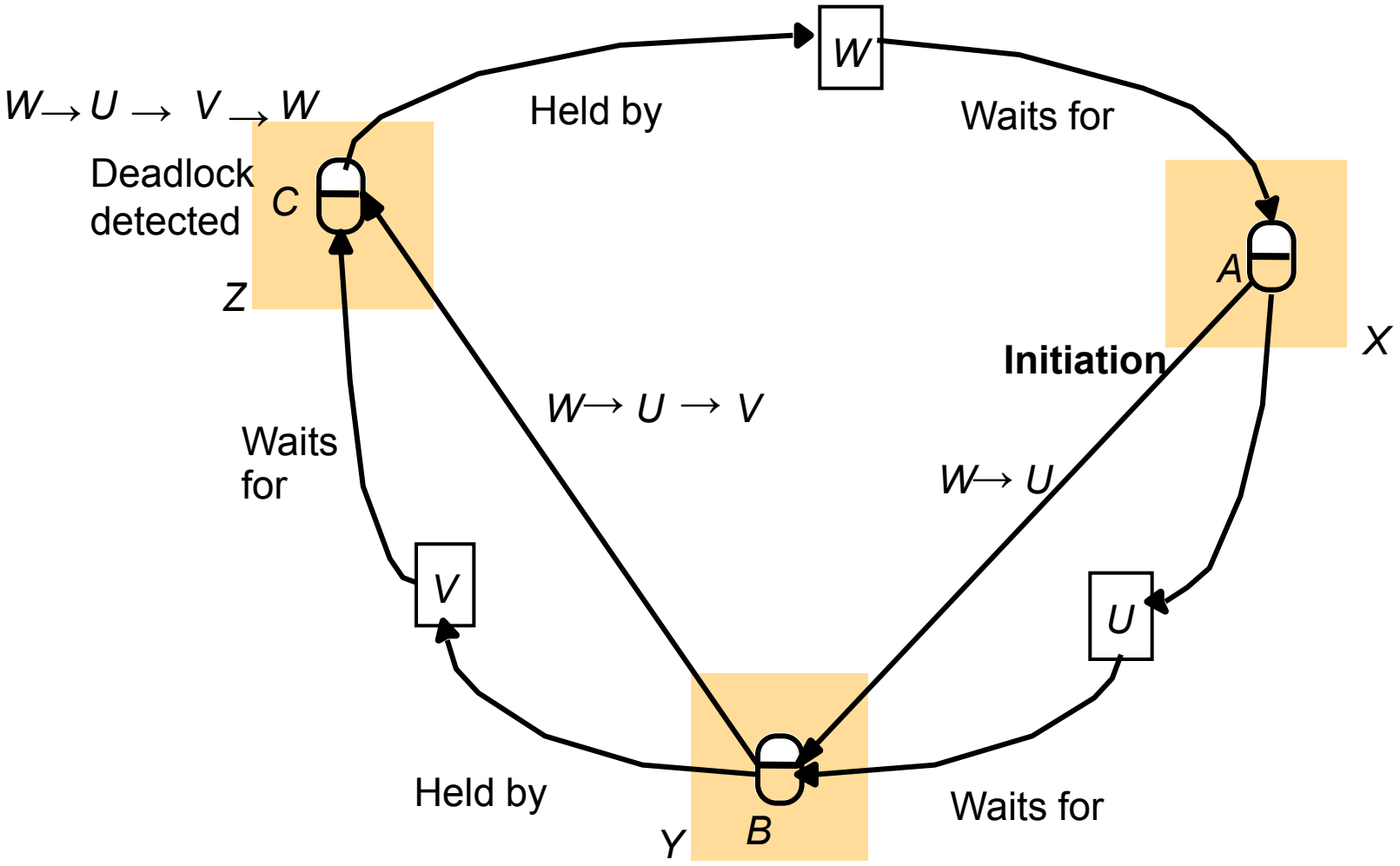


Figure 17.16
Two probes initiated

(a) initial situation

(b) detection initiated at object requested by *T*

(c) detection initiated at object requested by *W*

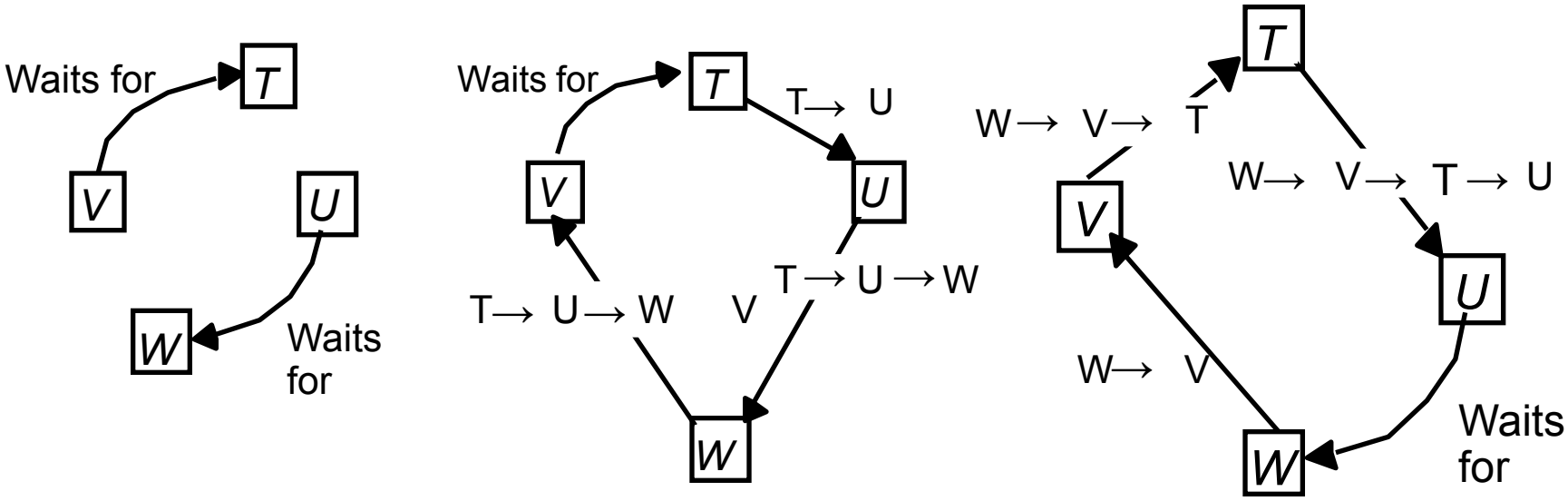


Figure 17.17
Probes travel downhill

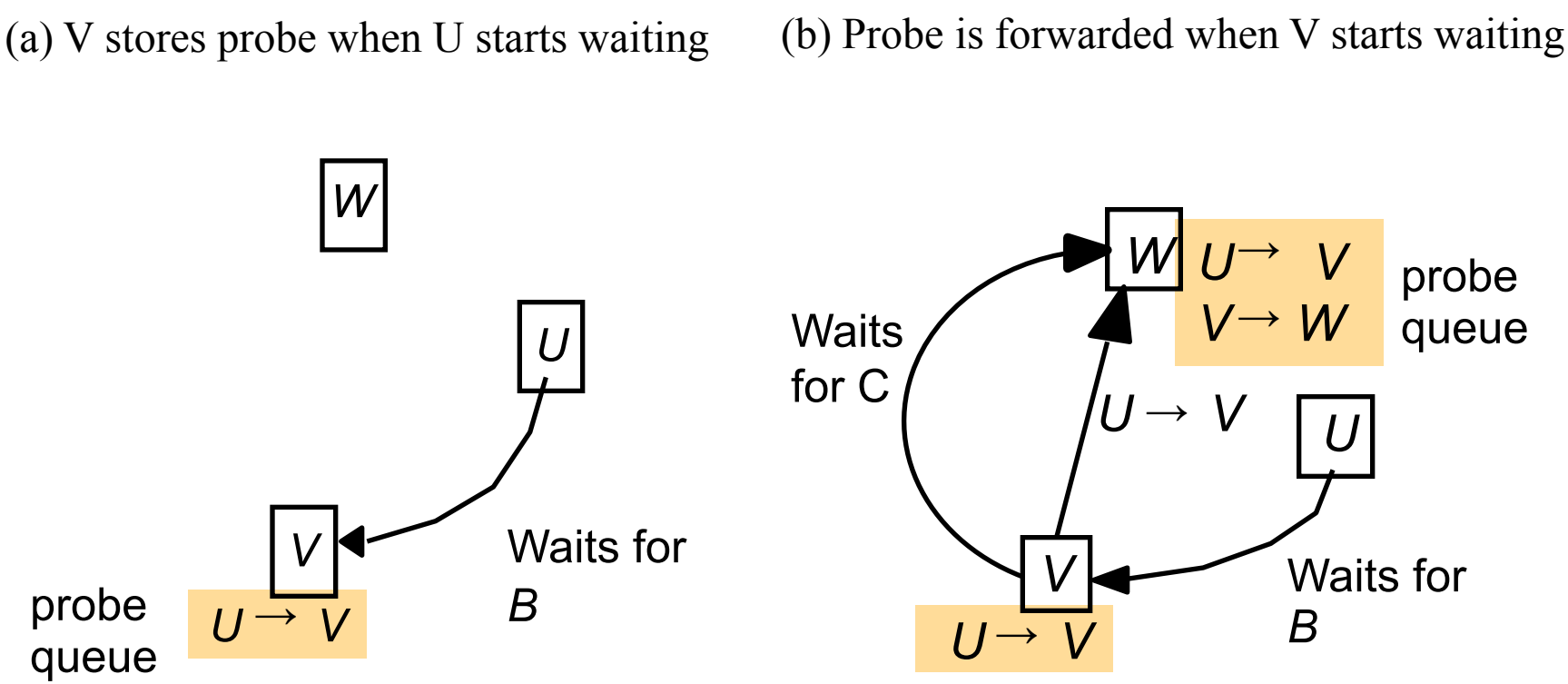


Figure 17.18
Types of entry in a recovery file

Type of entry	*Description of contents of entry*
Object	A value of an object.
Transaction status	Transaction identifier, transaction status (*prepared*, *committed* *aborted*) and other status values used for the two-phase commit protocol.
Intentions list	Transaction identifier and a sequence of intentions, each of which consists of <*objectID, Pi*>, where *Pi* is the position in the recovery file of the value of the object.

Figure 17.19
Log for banking service

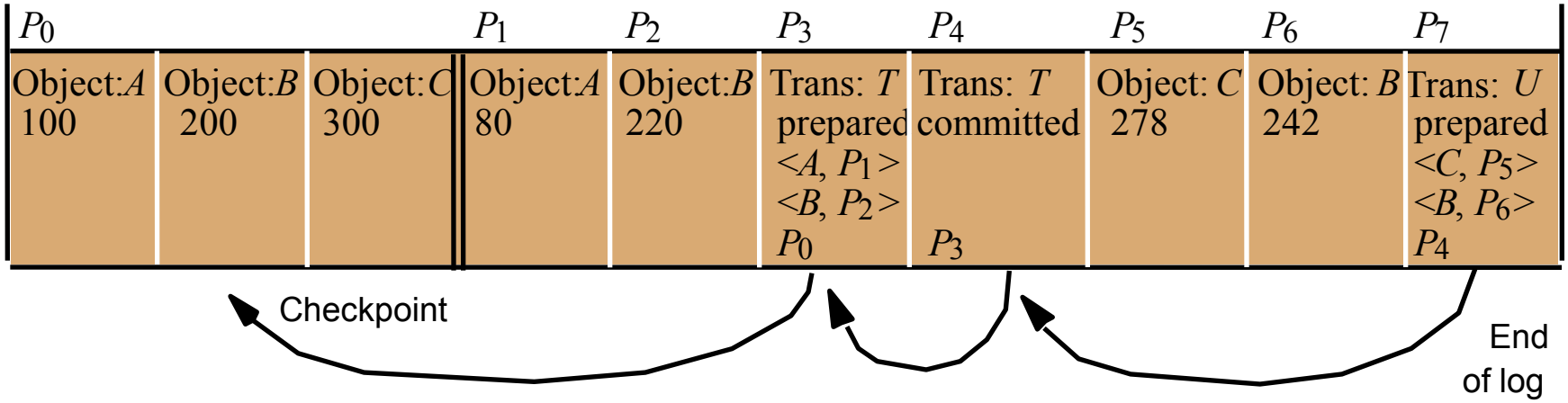


Figure 17.20
Shadow versions

Map at start	*Map when T commits*
$A \rightarrow P_0$	$A \rightarrow P_1$
$B \rightarrow P_0'$	$B \rightarrow P_2$
$C \rightarrow P_0''$	$C \rightarrow P_0''$

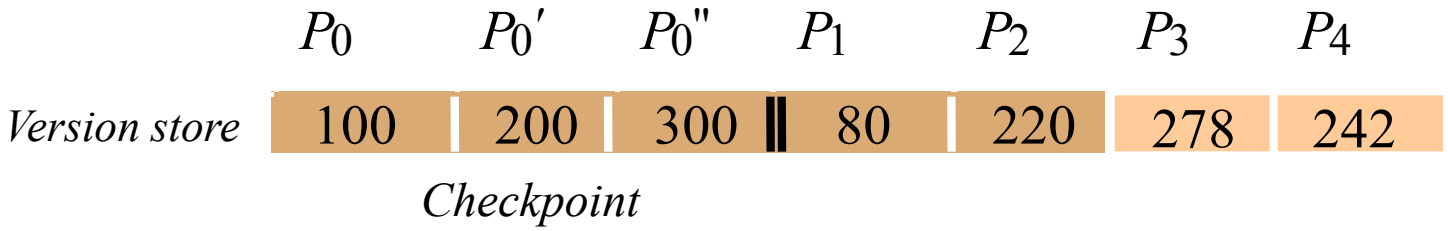


Figure 17.21
Log with entries relating to two-phase commit protocol

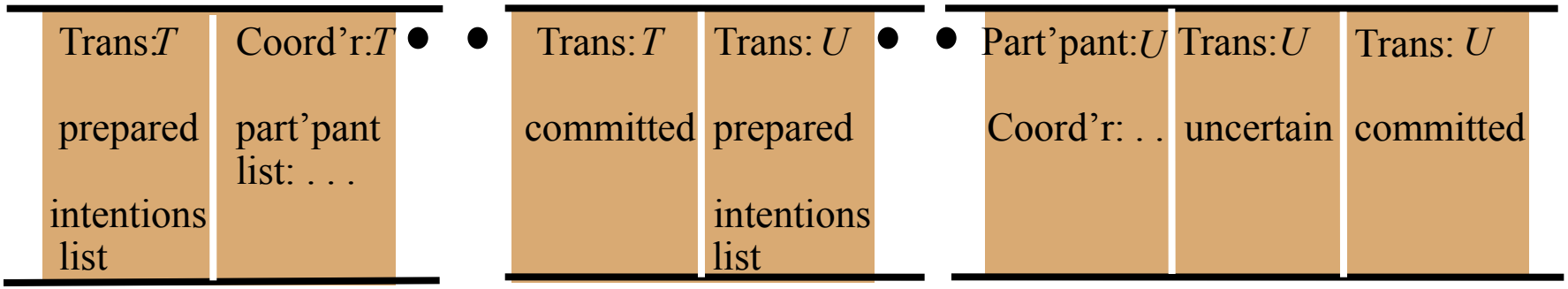


Figure 17.22
Recovery of the two-phase commit protocol

Role	*Status*	*Action of recovery manager*
Coordinator	*prepared*	No decision had been reached before the server failed. It sends *abortTransaction* to all the servers in the participant list and adds the transaction status *aborted* in its recovery file. Same action for state *aborted*. If there is no participant list, the participants will eventually timeout and abort the transaction.
Coordinator	*committed*	A decision to commit had been reached before the server failed. It sends a *doCommit* to all the participants in its participant list (in case it had not done so before) and resumes the two-phase protocol at step 4 (Fig 17.5).
Participant	*committed*	The participant sends a *haveCommitted* message to the coordinator (in case this was not done before it failed). This will allow the coordinator to discard information about this transaction at the next checkpoint.
Participant	*uncertain*	The participant failed before it knew the outcome of the transaction. It cannot determine the status of the transaction until the coordinator informs it of the decision. It will send a *getDecision* to the coordinator to determine the status of the transaction. When it receives the reply it will commit or abort accordingly.
Participant	*prepared*	The participant has not yet voted and can abort the transaction.
Coordinator	*done*	No action is required.

Figure 17.23
Nested transactions