Figure 13.1 Transactions T and U with exclusive locks.

Transaction T: *Bank$Withdraw(A, 4)* *Bank$Deposit(B, 4)*		**Transaction U:** *Bank$Withdraw(C, 3)* *Bank$Deposit(B, 3)*	
Operations	Locks	Operations	Locks
OpenTransaction			
balance := A.Read( )	locks A		
A.Write(balance – 4)			
		OpenTransaction	
		balance := C.Read( )	locks *C*
		C.Write(balance – 3)	
balance := B.Read( )	locks B		
		balance := B.Read()	waits for T’s lock on B
B.Write(balance + 4)		•	
CloseTransaction	unlocks A, B	•	
		•	locks B
		B.Write(balance + 3)	
		CloseTransaction	unlocks B, C

Figure 13.2 *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 13.3 Lock compatibility.

For one data item		*Lock requested*	
		Read	*Write*
Lock already set	*None*	OK	OK
	Read	OK	Wait
	Write	Wait	Wait

Figure 13.4 Use of locks in strict two-phase locking.

1. When an operation accesses a data item within a transaction:
   a) If the data item is not already locked, the server locks it and the operation proceeds.
   b) If the data item has a conflicting lock set by another transaction, the transaction must wait until it is unlocked.
   c) If the data item has a non-conflicting lock set by another transaction, the lock is shared and the operation proceeds.
   d) If the data item 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 data items it locked for the transaction.

Figure 13.5 Lock manager functions.

Lock (Trans, DataItem, LockType)

if there is a conflicting lock, that is, if there is an entry in the table belonging to another transaction that conflicts with *DataItem*, *Wait* on the condition variable associated with the entry.

if (immediately or after a *Wait*) there are no conflicting locks:

- if there is no entry for *DataItem*, add an entry to the table of locks
- else if there is an entry for *DataItem* belonging to a different transaction, add *Trans* to the entry (share the lock)
- else if there is an entry for *DataItem* belonging to *Trans* and *LockType* is more exclusive than the type in the entry, change entry to *LockType* (promote lock).

UnLock (Trans)

if there are any entries in the table belonging to transaction *Trans*, for each entry:

- if there is only one holder (*Trans*) in the entry, remove the entry
- else (a shared lock) remove *Trans* from the entry and *Signal* the associated condition variable.

Figure 13.6 Deadlock with read and write locks.

Transaction T		**Transaction U**	
Operations	Locks	Operations	Locks
balance:= A.Read()	read locks A		
		balance:= C.Read()	read locks C
		C.Write(balance – 3)	write locks C
A.Write(balance – 4)	write locks A		
•••			
balance := B.Read()	read locks B		
		balance := B.Read()	shares read lock on B
B.Write(balance + 4)	waits for U		
•••		*B.Write(balance + 3)*	waits for T
•••		•••	

Figure 13.7 An illustration of Violet showing the union of some diaries.

View: {Smith.qmw, Jones.qmw}

January 1988

25 Monday	26 Tuesday	27 Wednesday	28 Thursday	29 Friday
9:00–10:00 Jones unavailable	10:00–12:00 Jones unavailable	9:00–10:00 Jones unavailable	9:00–12:00 Jones Smith unavailable	
13:00–14:00 Jones Smith unavailable	11:00-–12:00 Jones unavailable	14:00–15:00 Jones Smith unavailable		

View: Meetings.qmw

January 1988

25 Monday	26 Tuesday	27 Wednesday	28 Thursday	29 Friday
	10:00–12:00 hardware research		9:00–12:00 Equipment planning	
13:00–14:00 Dept. meeting		14:00–15:00 Dr. Visitor Interesting facts		

Figure 13.8 The wait-for graph for Figure 13.6.

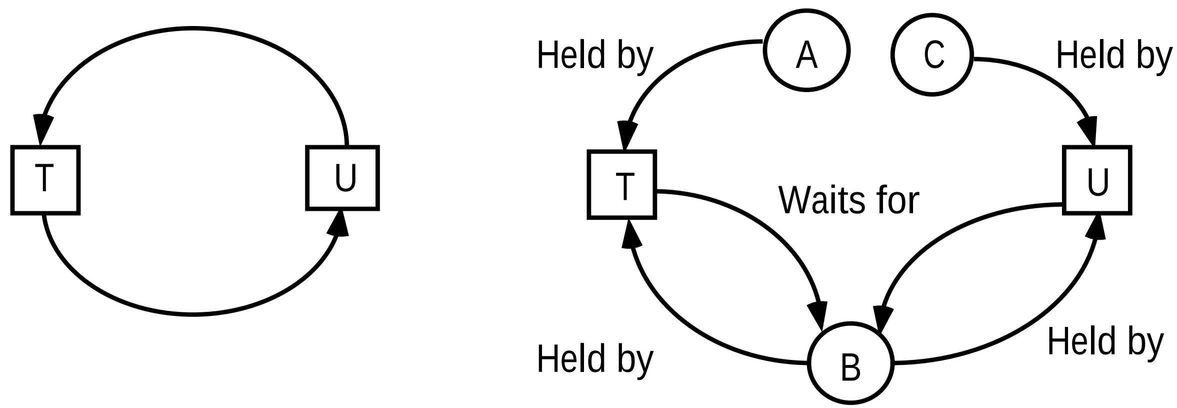


Figure 13.9 A cycle in a wait-for graph.

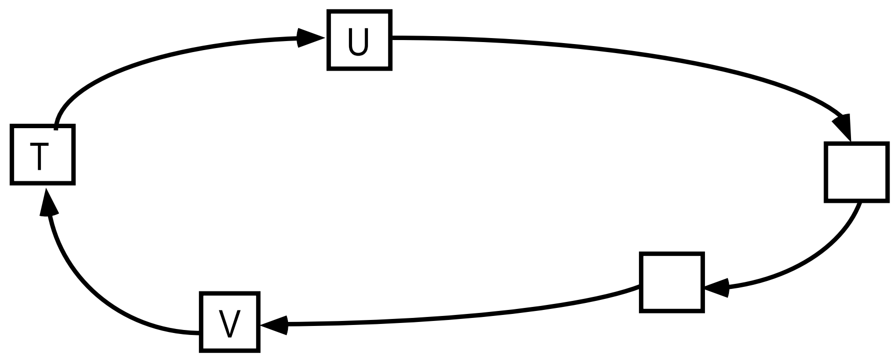


Figure 13.10 Another wait-for graph.

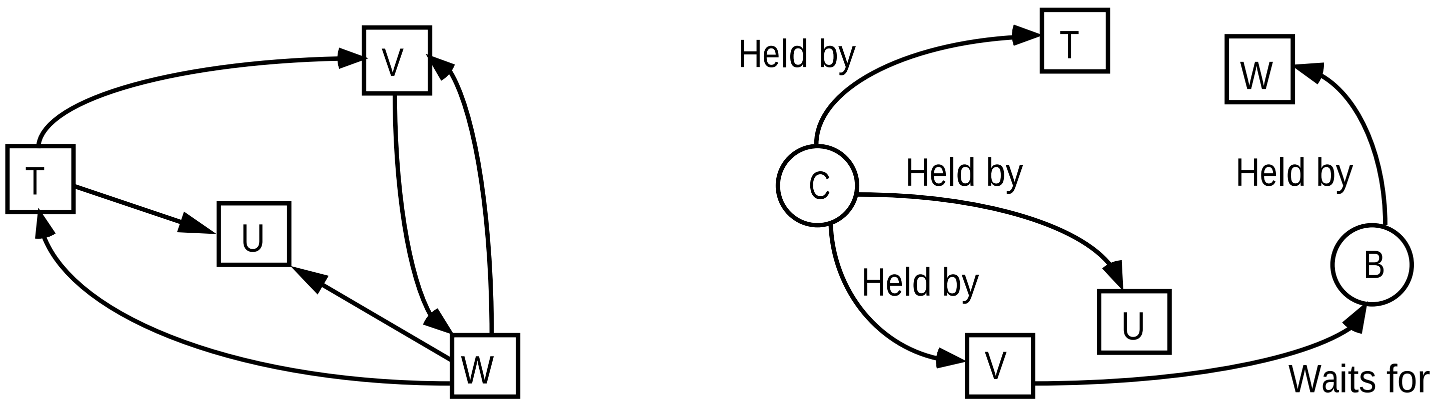


Figure 13.11 Resolution of the deadlock in Figure 13.6.

Transaction T Operations	Locks	**Transaction U** Operations	Locks
balance:= A.Read()	read locks *A*		
		balance:= C.Read()	read locks *C*
		C.Write(balance – 3)	write locks *C*
A.Write(balance – 4)	write locks *A*		
•••		•••	
balance := B.Read()	read locks *B*		
		balance := B.Read()	shares read lock on *B*
B.Write(balance + 4)	waits on U's read lock on*B*		
•••		*B.Write(balance + 3)*	waits on T's read lock on *B*
(timeout elapses) T's lock on *B* becomes vulnerable, unlock *B*, abort T		•••	
		B.Write(balance +3)	write locks *B*
			unlock *B* and *C*

Figure 13.12 Lock compatibility (*read*, *write* and *commit* locks).

For one data item		*Lock to be set*		
		Read	*Write*	*Commit*
Lock already set	*None*	OK	OK	OK
	Read	OK	OK	Wait
	Write	OK	Wait	Wait
	Commit	Wait	Wait	Wait

Figure 13.13 Lock hierarchy for the Bank Server.

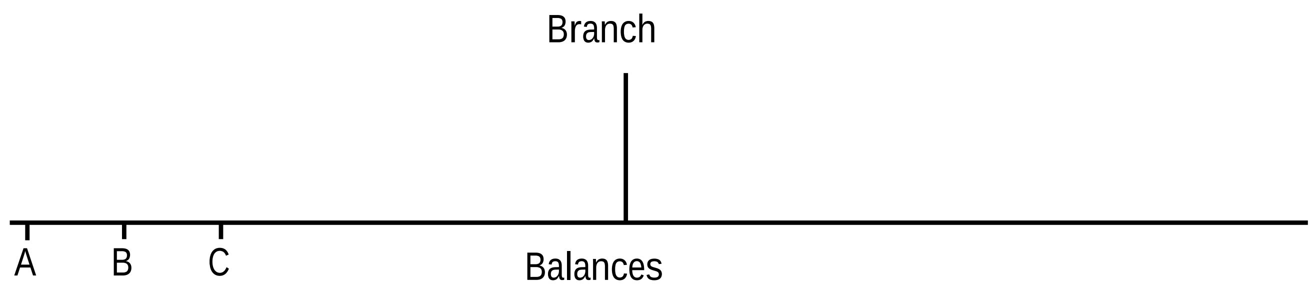


Figure 13.14 Lock hierarchy for Violet.

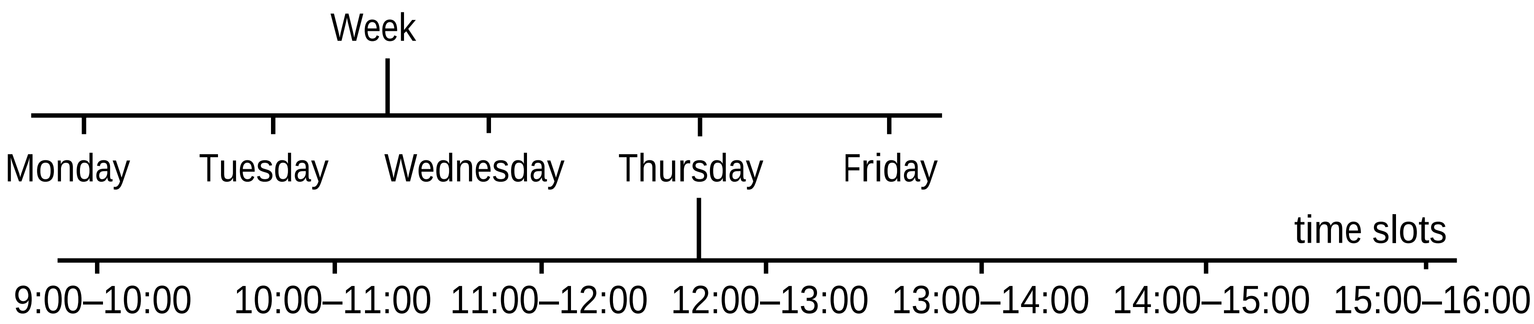


Figure 13.15 Lock compatibility table for hierarchic locks.

For one data item		*Lock to be set*			
		Read	*Write*	*I-Read*	*I-Write*
Lock already set	*Read*	OK	Wait	OK	Wait
	Write	Wait	Wait	Wait	Wait
	I-Read	OK	Wait	OK	OK
	I-Write	Wait	Wait	OK	OK

T_i	T_j	Rule	
Read	*Write*	1.	T_i must not read data items written by T_j
Write	*Read*	2.	T_j must not read data items written by T_i
Write	*Write*	3.	T_i must not write data items written by T_j and T_j must not write data items written by T_i.

Figure 13.16 Validation of transactions.

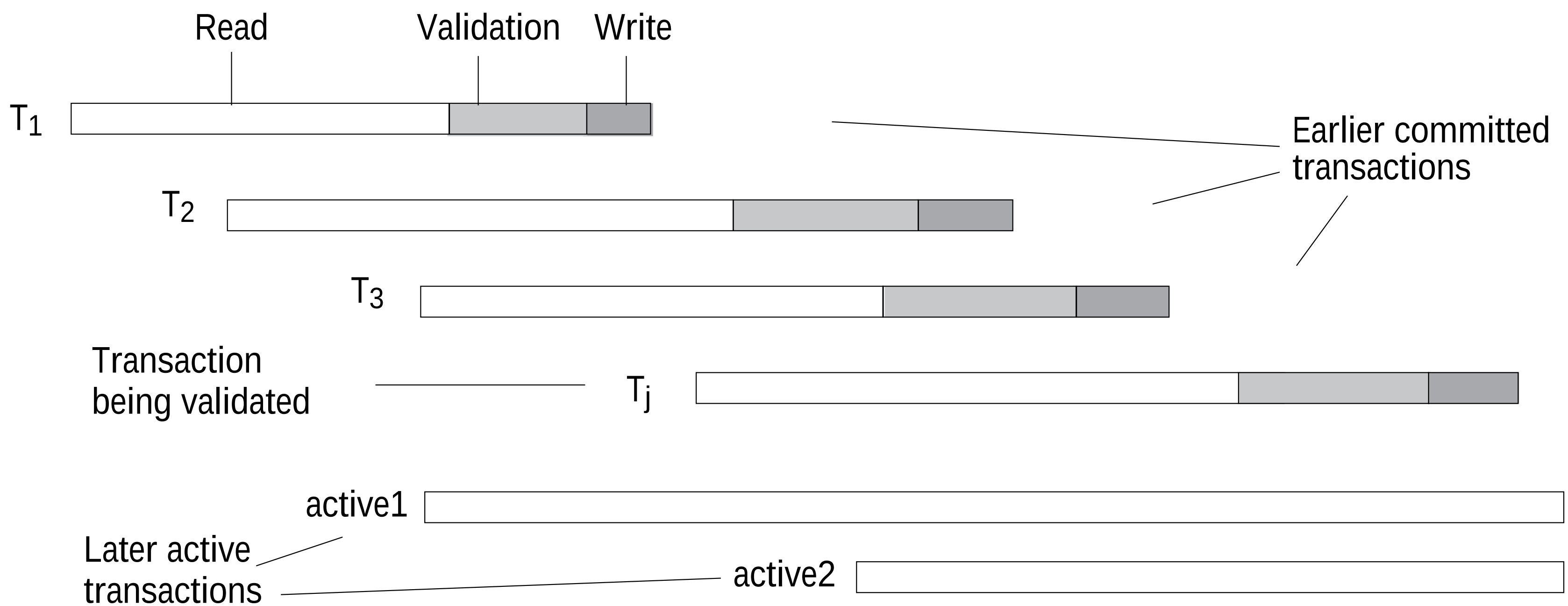


Figure 13.17 Transaction conflicts for timestamp ordering.

Rule	T_j	
1.	*Write*	T_j must not write a data item that has been read by any T_i where $T_i > T_j$ this requires that $T_j >$ the maximum read timestamp of the data item
2.	*Write*	T_j must not write a data item that has been written by any T_i where $T_i > T_j$ this requires that $T_j >$ the maximum write timestamp of the data item
3.	*Read*	T_j must not read a data item that has been written by any T_i where $T_i > T_j$ this implies that T_j cannot read if $T_j <$ write timestamp of the committed version of the data item

Figure 13.18 Write operations and timestamps.

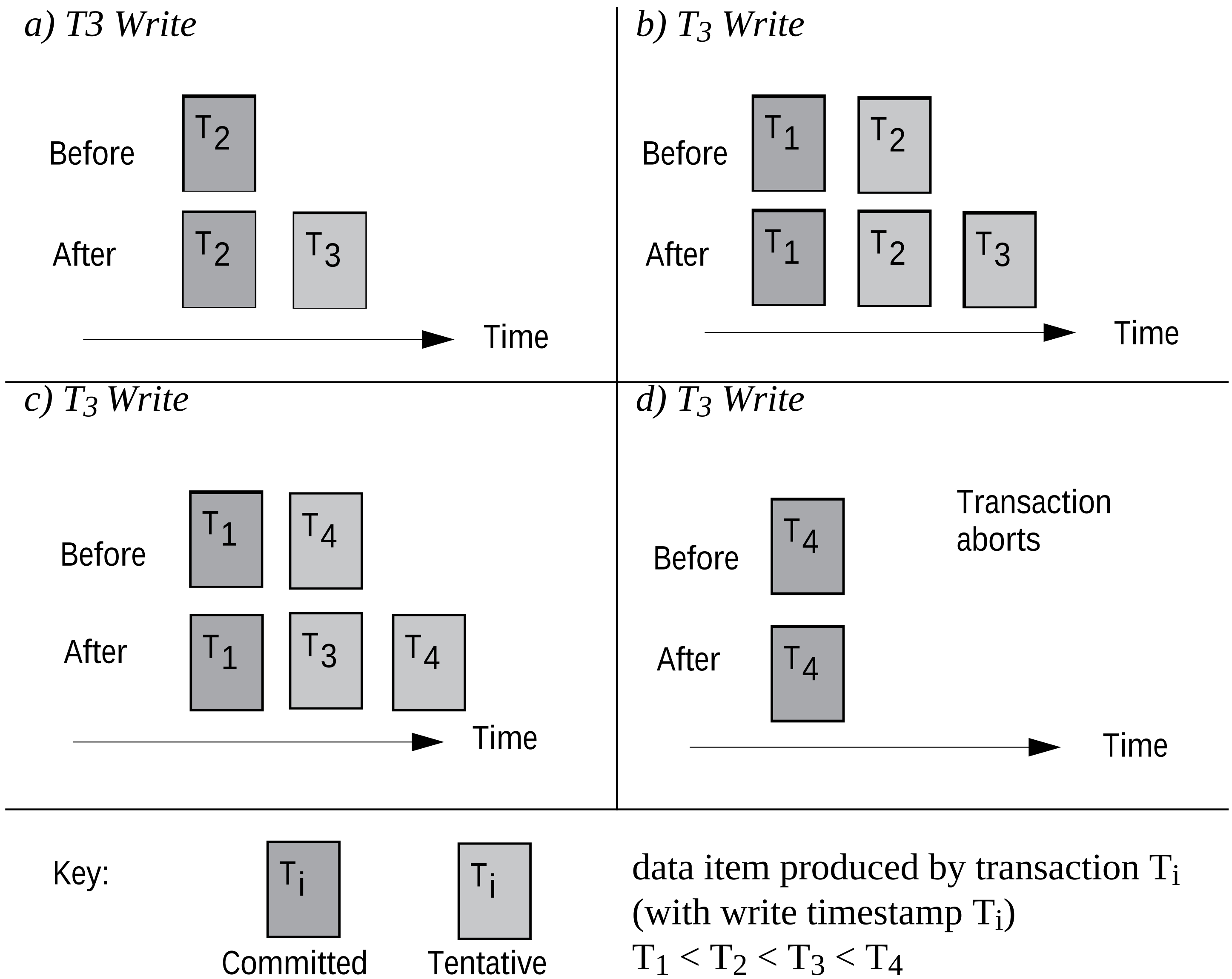


Figure 13.19 *Read* operations and timestamps.

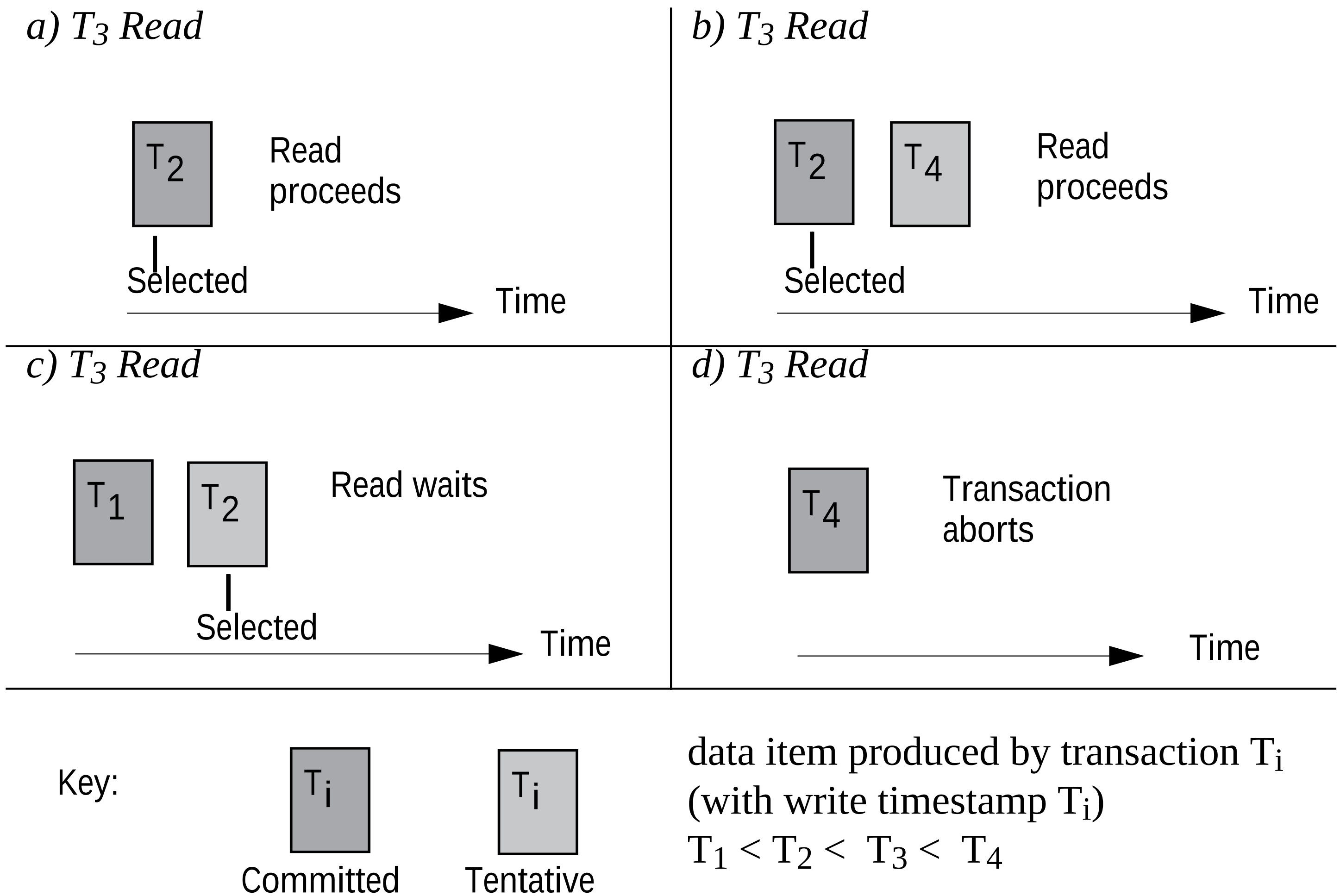


Figure 13.20 Timestamps in transactions T and U.

T	U	*Timestamps and versions of data items* A		B		C	
		RTS	*WTS*	*RTS*	*WTS*	*RTS*	*WTS*
		{}	**S**	{}	**S**	{}	**S**
OpenTransaction							
bal := A.Read ()		{T}					
	OpenTransaction						
	bal := C.Read()					{U}	
A.Write (bal – 4)			**S**, T				
bal:= B.Read ()				{T}			
	C.Write(bal – 3)						**S**,U
	bal := B.Read()			{U}			
B.Write(bal + 4)							
Aborts							
	B.Write(bal + 3)				**S**,U		

Figure 13.21 Late *Write* operation would invalidate a *Read*.

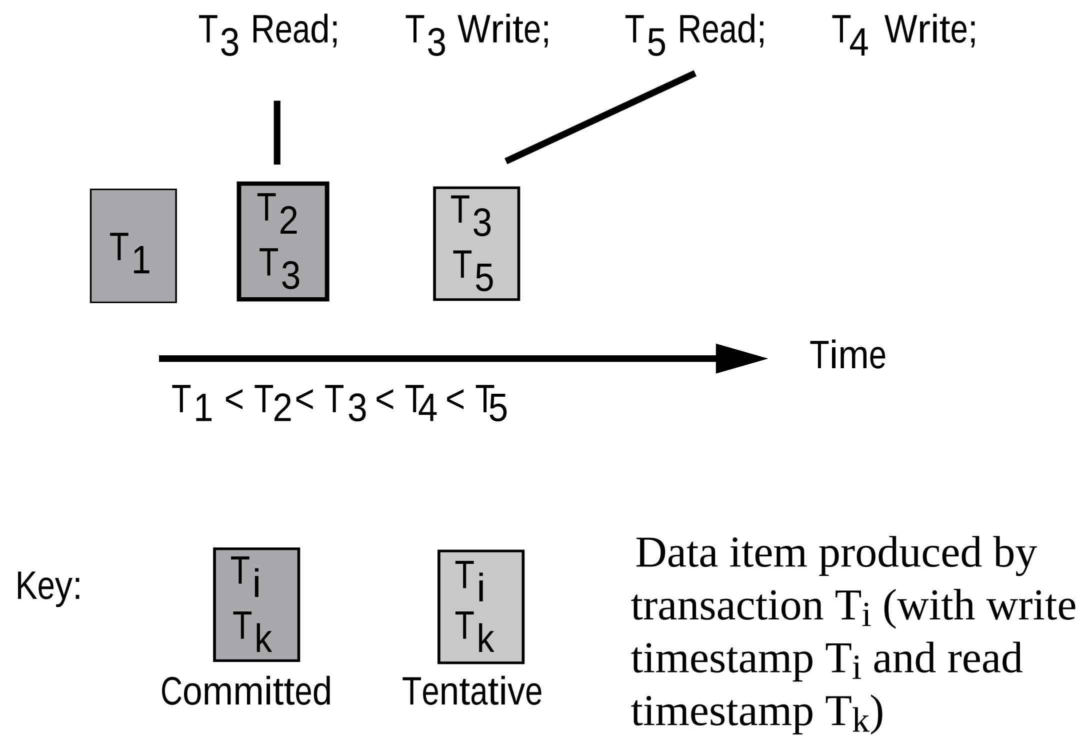


Figure 14.1 Distributed transactions.

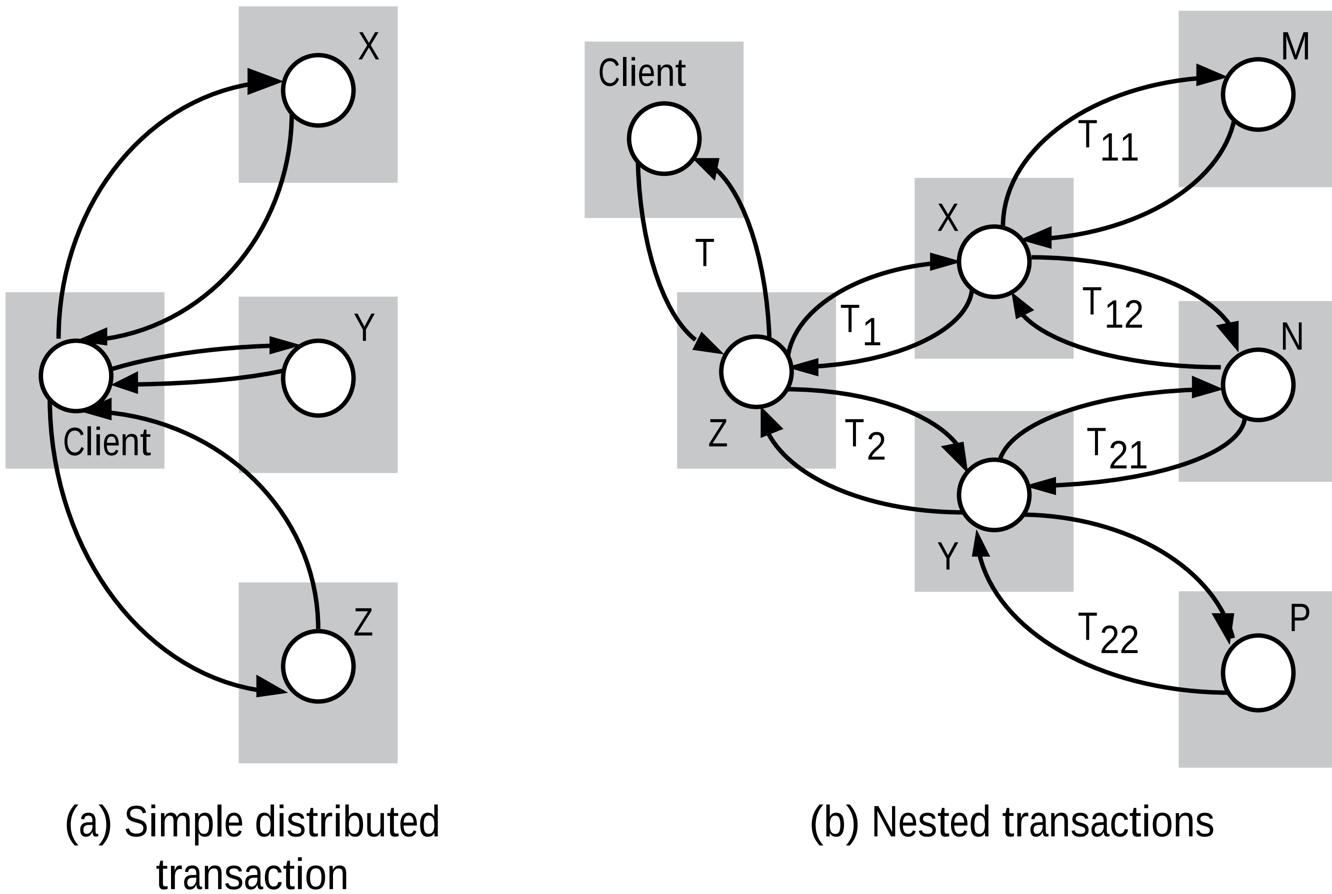


Figure 14.2 Nested banking transaction.

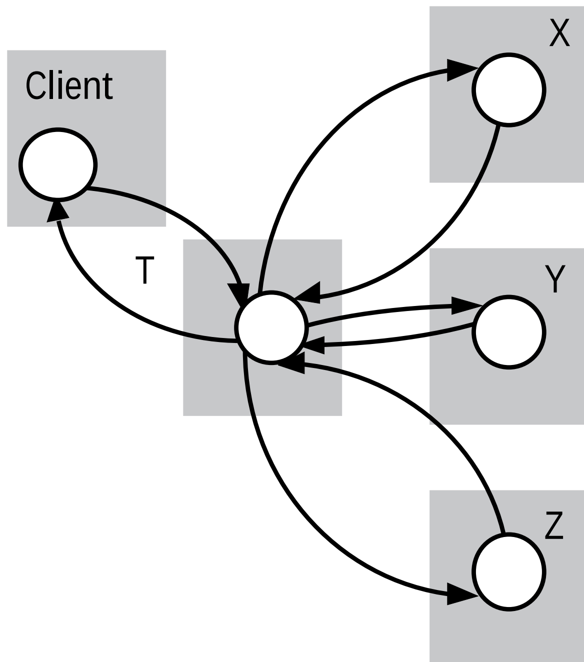


Figure 14.3 A distributed banking transaction.

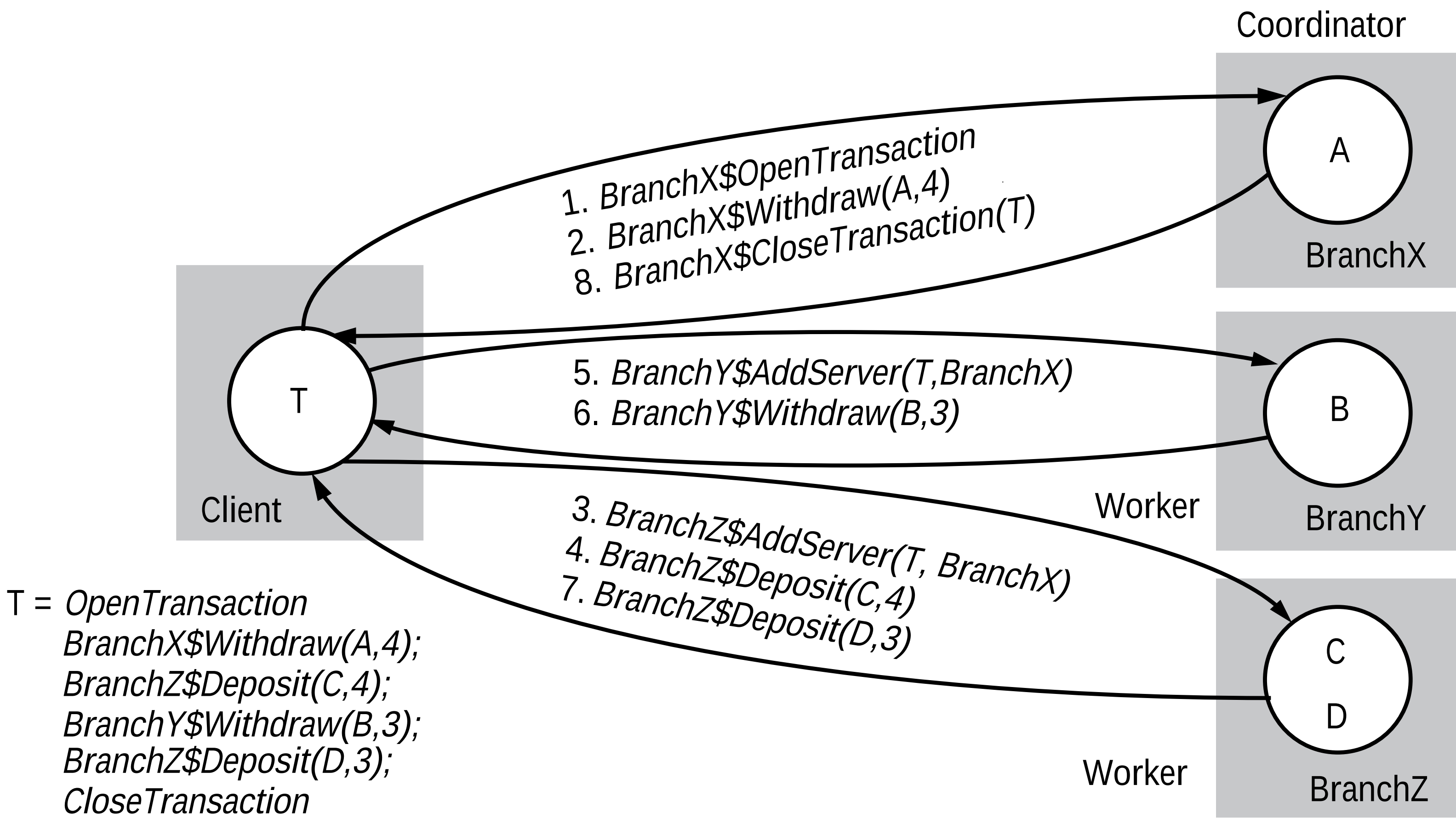


Figure 14.4 Operations for two-phase commit protocol.

CanCommit?(Trans) $\rightarrow$ *Yes / No*

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

DoCommit(Trans)

Call from coordinator to worker to tell worker to commit its transaction.

HaveCommitted(Trans)

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

GetDecision(Trans) $\rightarrow$ *Yes / No*

Call from worker 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 failure or time out.

Figure 14.5 The two-phase commit protocol.

Phase 1 (voting phase):

1. The coordinator sends a *CanCommit?* request to each of the workers in the transaction;
2. When a worker receives a *CanCommit?* request it replies with its vote (*Yes* or *No*) to the coordinator. If the vote is *No* the worker 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 workers;
   b) Otherwise the coordinator decides to abort the transaction and sends *AbortTransaction* requests to all workers that voted *Yes*;
4. Workers that voted *Yes* are waiting for a DoCommit or AbortTransaction request from the coordinator. When a worker receives one of these messages it acts accordingly and in the case of commit, makes a HaveCommitted call as confirmation to the coordinator.

Figure 14.6 Communication in two-phase commit protocol.

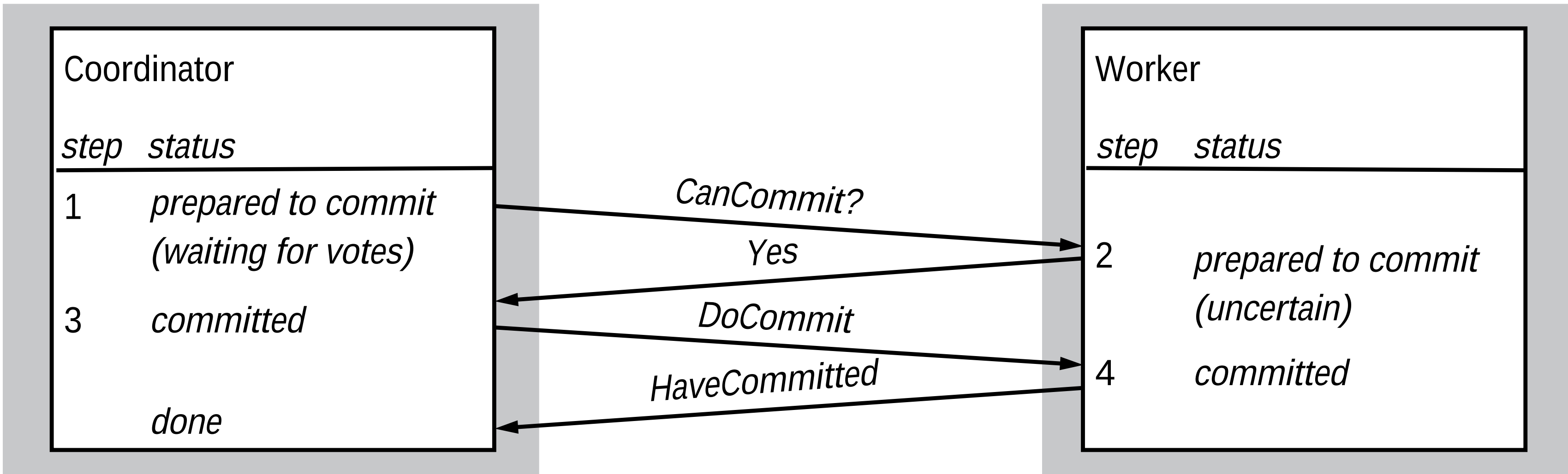


Figure 14.7 Operations in service for nested transactions.

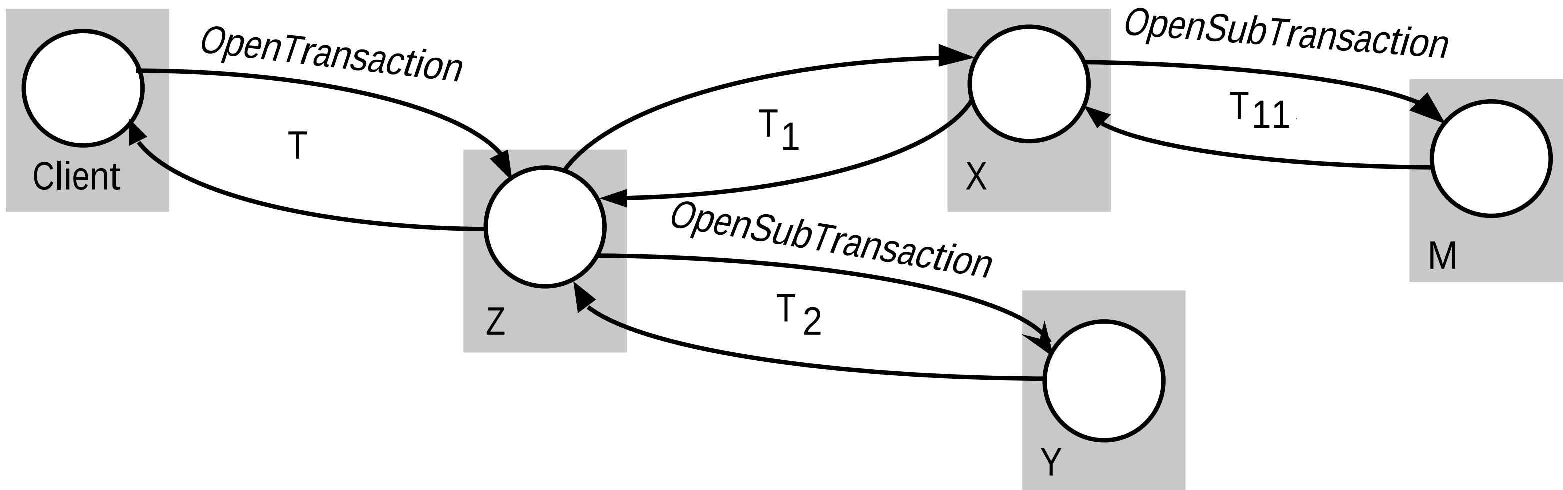
OpenSubTransaction(Trans) $\rightarrow$ *NewTrans*

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

GetStatus(Trans) $\rightarrow$ *committed, aborted, tentative*

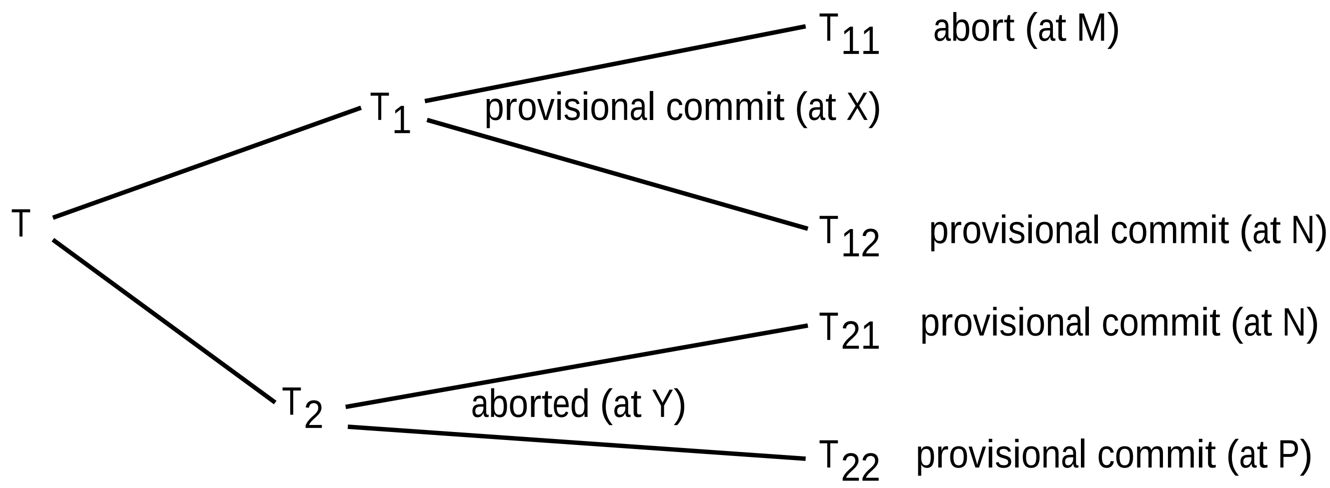
Asks transaction *Trans* to report on its status.

Figure 14.8 Nested transactions.



TID in example	T	T_1	T_{11}	T_2
actual TID	Z, n_Z	Z, n_Z:X, n_X	Z, n_Z:X, n_X:M, n_m	Z, n_Z:Y, n_Y

Figure 14.9 Transaction T decides whether to commit.



The information held by each server in the example shown in Figure 14.9 is as follows:

Server	*Transaction*	*Child transactions*	*Provisional Commit list*	*Abort List*
Z	T	T_1, T_2	T_1@X, T_{12}@N	T_{11}, T_2
X	T_1	T_{11}, T_{12}	T_1, T_{12}@N	T_{11}
Y	T_2	T_{21}, T_{22}	~~(T_{21}@N, T_{22}@P)~~	T_2
M	T_{11}			T_{11}
N	T_{12}, T_{21}		T_{21}, T_{12}	
P	T_{22}		T_{22}	

Figure 14.10 *CanCommit?* operation of nested transaction service.

CanCommit?(Trans, AbortList) $\rightarrow$ *Yes / No*

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

Figure 14.11 Interleavings of transactions U, V and W.

U		V		W	
Deposit(D)	lock D				
		Deposit(B)	lock B		
Deposit(A)	lock A				
				Deposit(C)	lock C
Withdraw(B)	wait				
		Withdraw(C)	wait		
				Withdraw(A)	wait

Figure 14.12 Distributed deadlock.

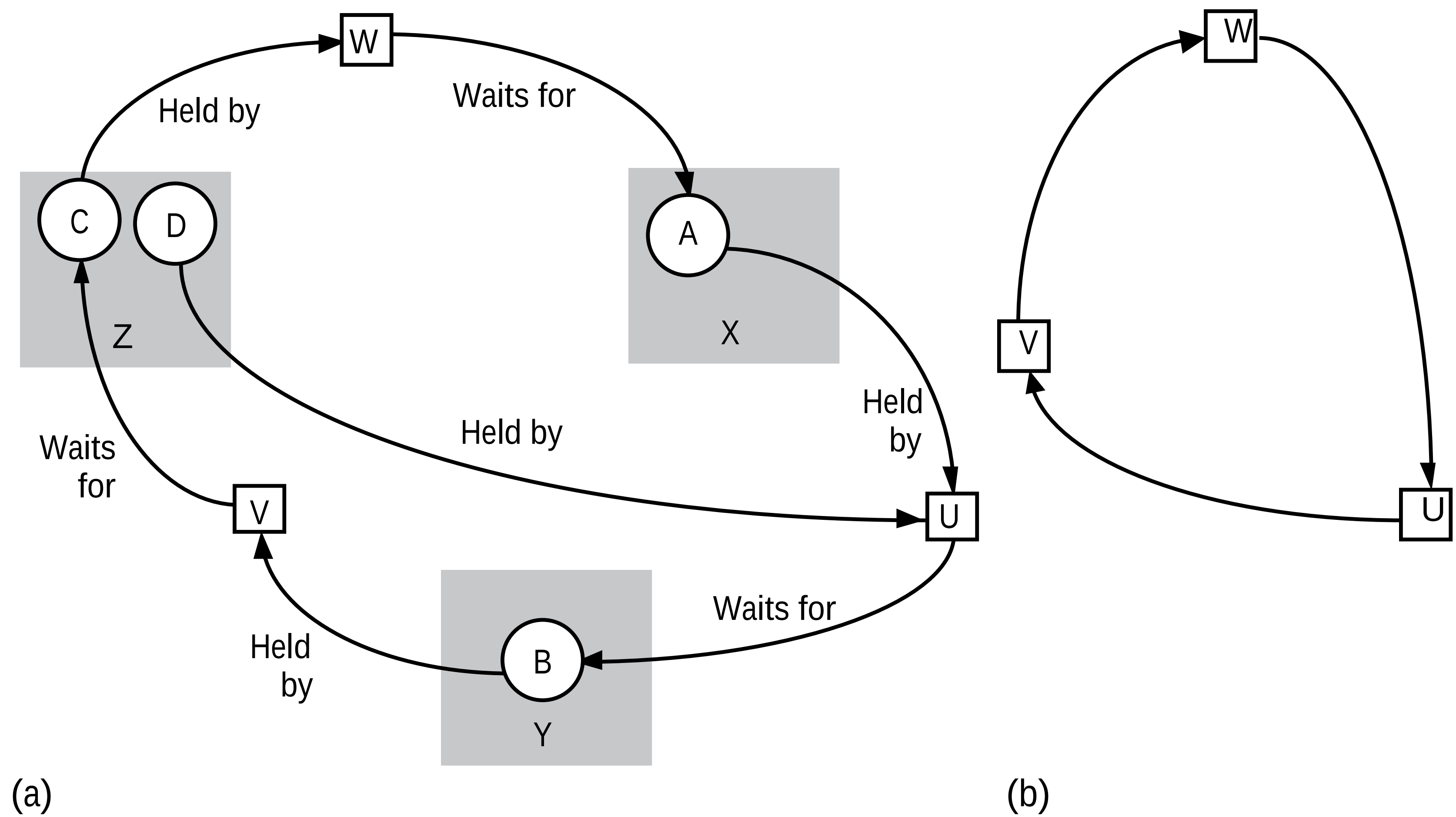


Figure 14.13 Local and global wait-for graphs.

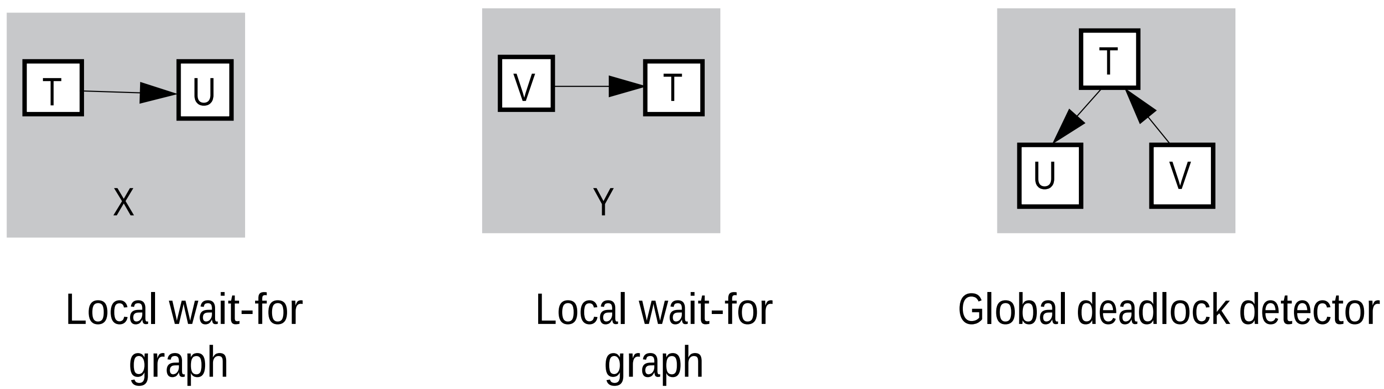


Figure 14.14 Probes transmitted to detect deadlock.

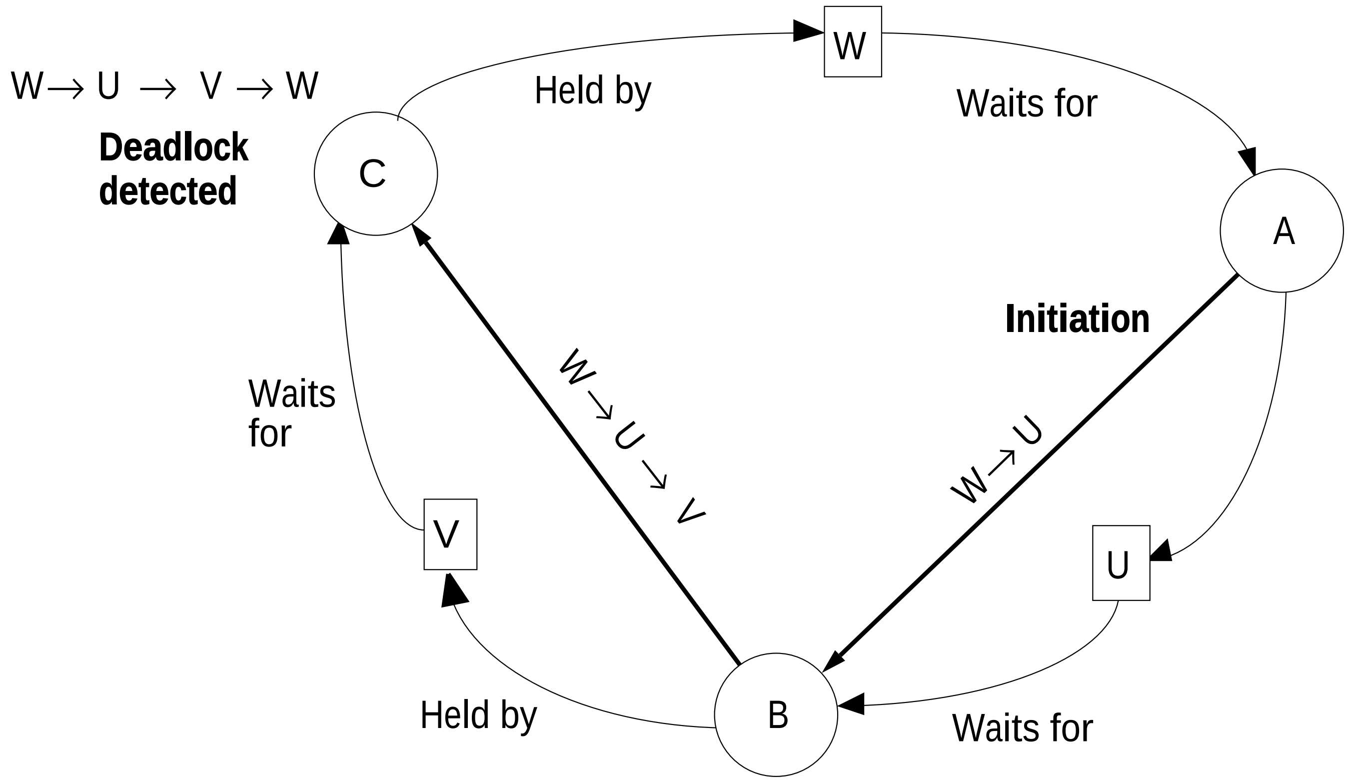
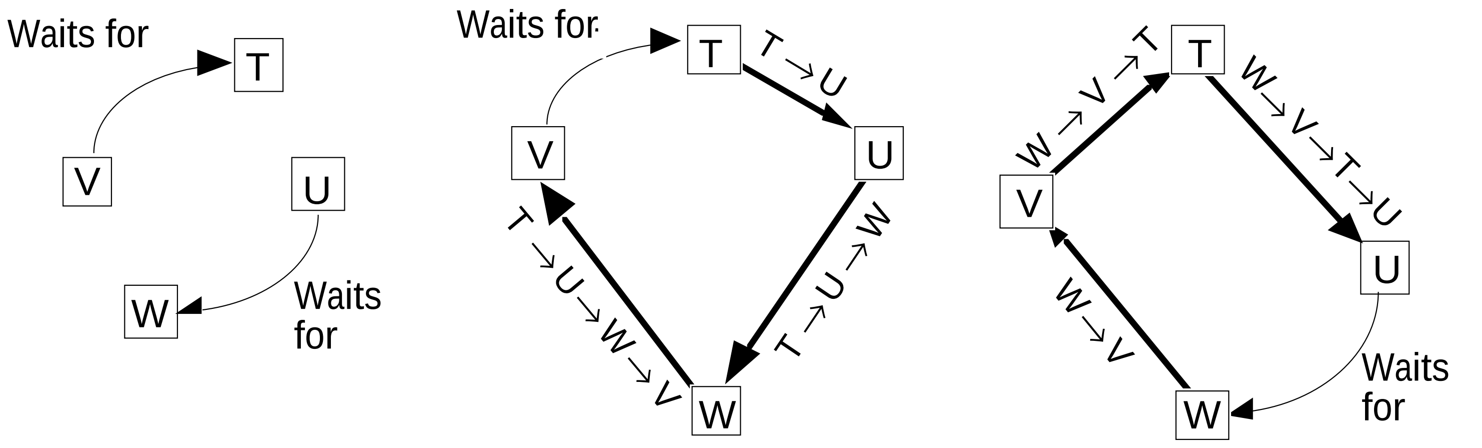


Figure 14.15 Two probes initiated.

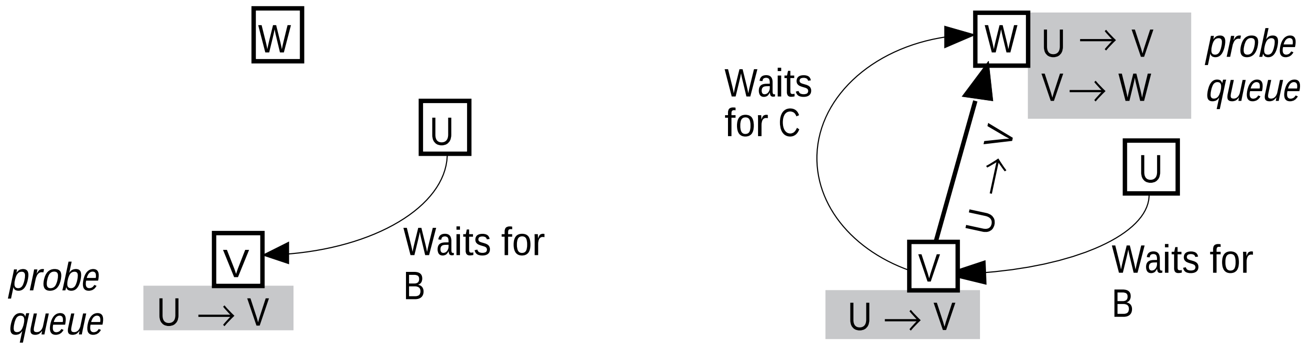


(a) Initial situation

(b) Detection initiated at data item requested by T

(c) Detection initiated at data item requested by W

Figure 14.16 Probes travel downhill.



(a) V stores probe when U starts waiting (b) Probe is forwarded when V starts waiting

Figure 14.17 Cycles and shared locks.

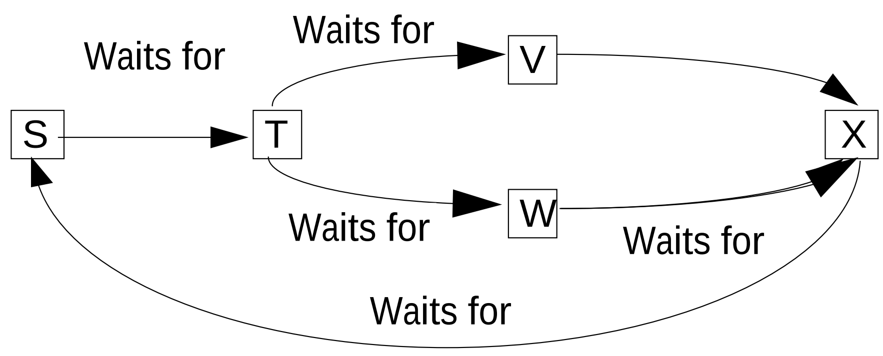


Figure 14.18 Replicated transactional service.

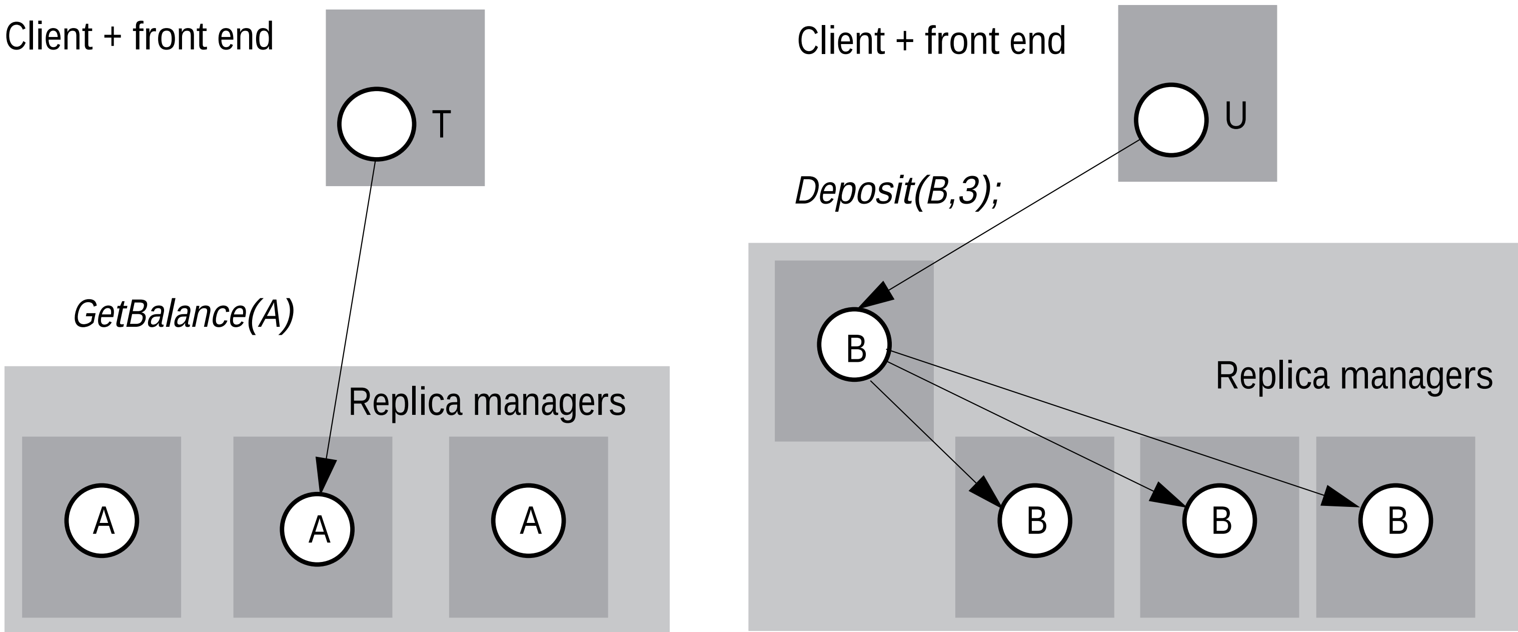


Figure 14.19 Available copies.

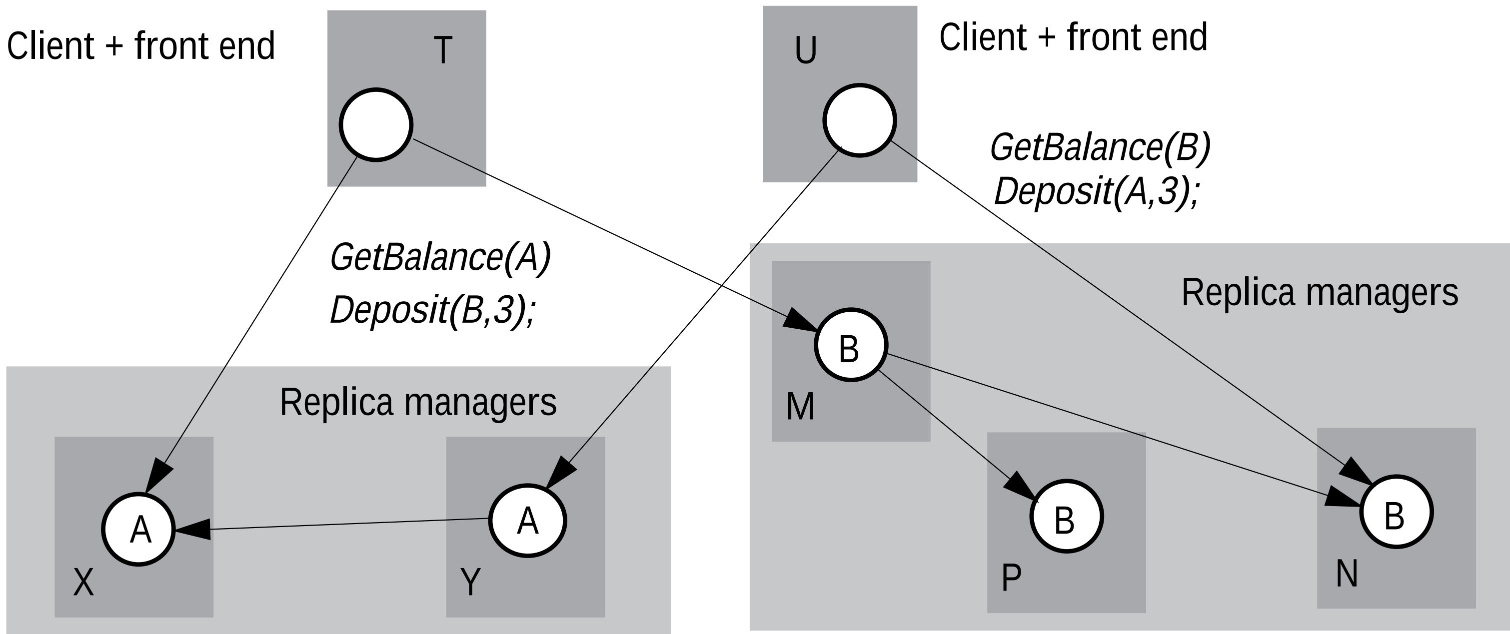


Figure 14.20 Network partition.

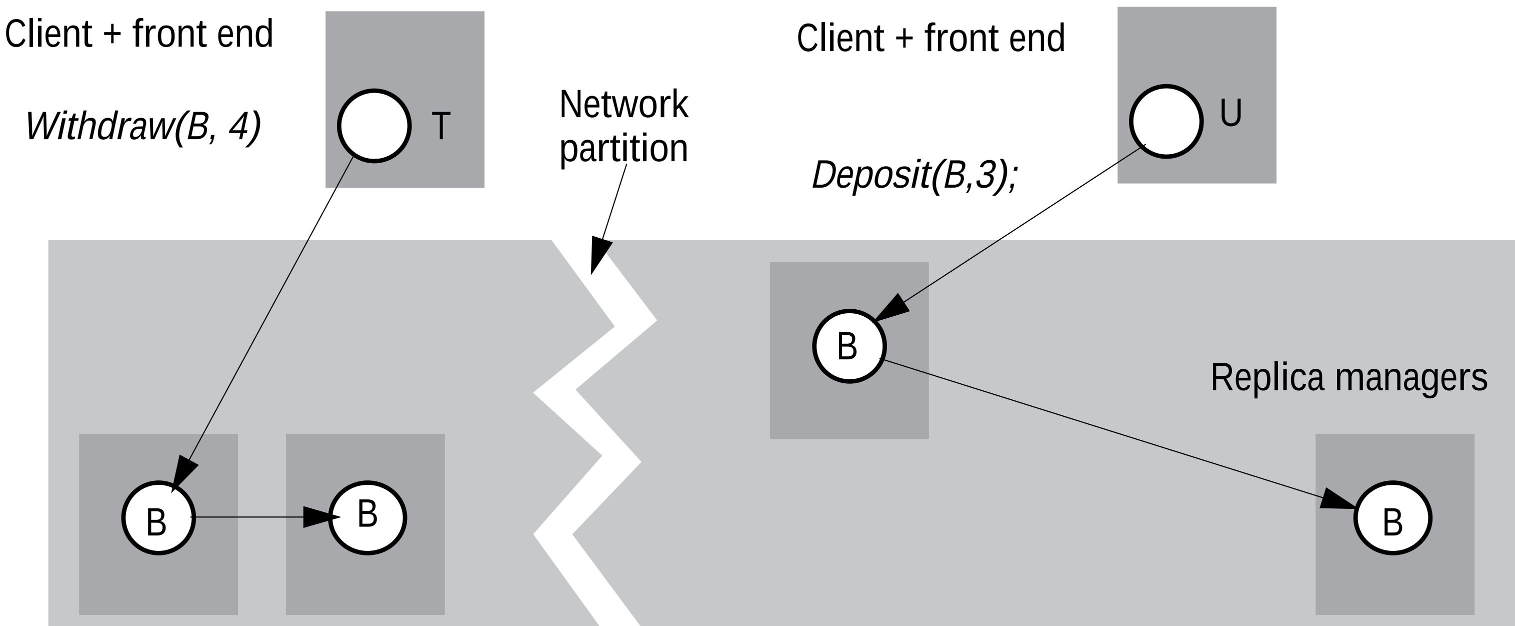


Figure 14.21 Two network partitions.

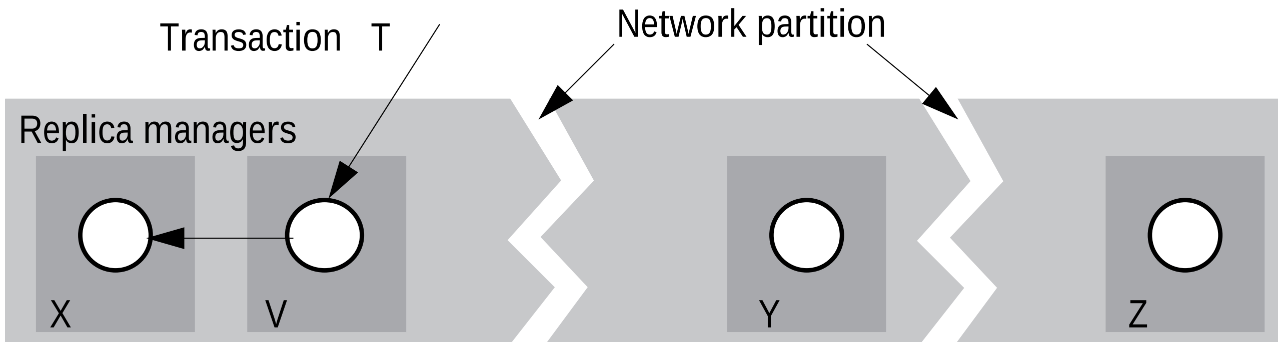


Figure 14.22 Virtual partition.

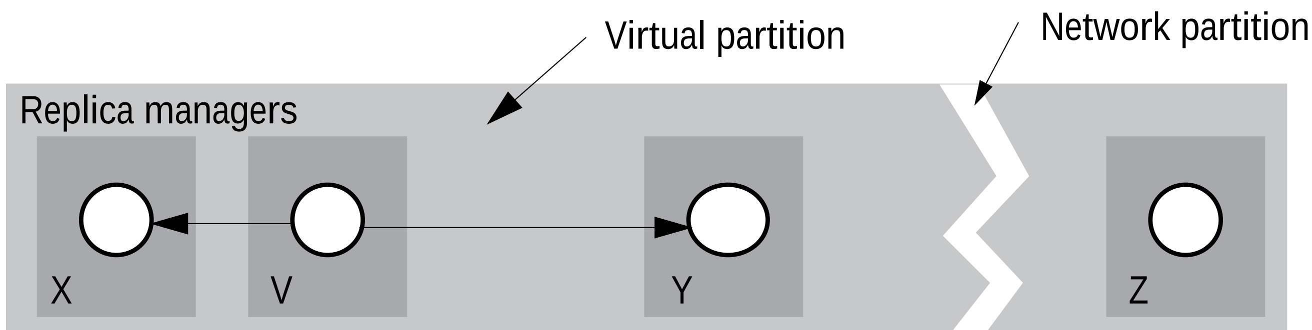


Figure 14.23 Two overlapping virtual partitions.

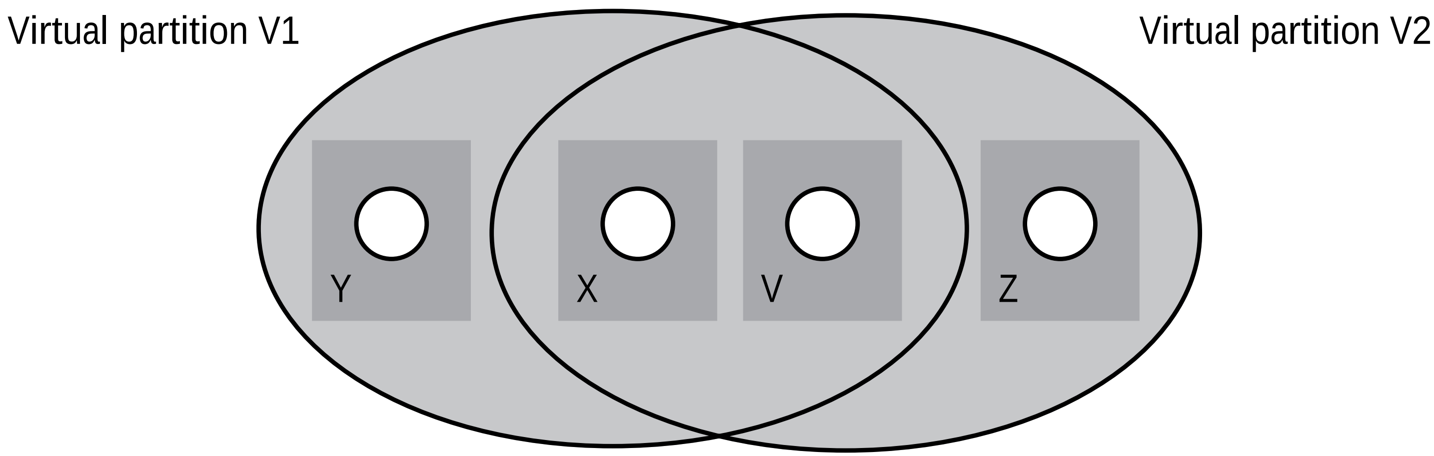


Figure 14.24 Creating a virtual partition.

Phase 1:

- The initiator sends a *Join* request to each potential member. The argument of *Join* is a proposed logical timestamp for the new virtual partition;
- When a replica manager receives a *Join* request it compares the proposed logical timestamp with that of its current virtual partition;
  - If the proposed logical timestamp is greater it agrees to join and replies *Yes;*
  - If it is less, it refuses to join and replies *No;*

Phase 2:

- If the initiator has received sufficient *Yes* replies to have *Read* and *Write* quora, it may complete the creation of the new virtual partition by sending a *Confirmation* message to the sites that agreed to join. The creation timestamp and list of actual members are sent as arguments;
- Replica managers receiving the *Confirmation* message join the new virtual partition and record its creation timestamp and list of actual members.
