Slides for Chapter 17: Distributed transactions

From **Coulouris, Dollimore, Kindberg and Blair**

Distributed Systems: Concepts and Design

Edition 5, © Addison-Wesley 2012

*** Modified by Prof. Rivalino Matias, Jr.**

Outline

- Introduction
- COMMIT protocols
- Concurrency control in DS
- Distributed deadlocks
- Transaction recovery

Introduction

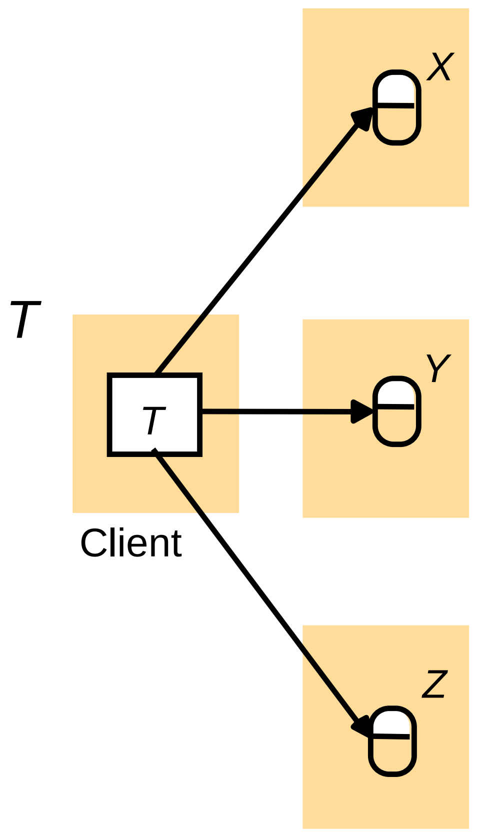
- We use the term distributed transaction to refer to a flat or nested transaction that accesses objects managed by multiple servers (distributed processes).
  - When a distributed transaction comes to an end, the atomicity property of transactions requires that either all of the servers involved commit the transaction or all of them abort the transaction.
  - To achieve this, one of the servers takes on a **coordinator role**, which involves ensuring the same outcome at all of the servers.
  - The manner in which the coordinator achieves this depends on the protocol chosen.
    - A protocol known as the ‘two-phase commit protocol’ is the most commonly used.
    - This protocol allows the servers to communicate with one another to reach a joint decision as to whether to commit or abort.

Introduction

- **Flat and nested distributed transactions**
  - A client transaction becomes distributed if it invokes operations in several different servers.
  - There are two different ways that distributed transactions can be structured:
    - **Flat transactions**:
      - ✓ In a flat transaction, a client makes requests to more than one server. A flat client transaction completes each of its requests before going on to the next one. Therefore, each transaction accesses servers' objects sequentially. When servers use locking, a transaction can only be waiting for one object at a time.
    - **Nested transactions**:
      - ✓ In a nested transaction, the top-level transaction can open subtransactions, and each subtransaction can open further subtransactions down to any depth of nesting. Subtransactions at the same level can run concurrently.

Figure 17.1
Distributed transactions

(a) Flat transaction



(b) Nested transactions

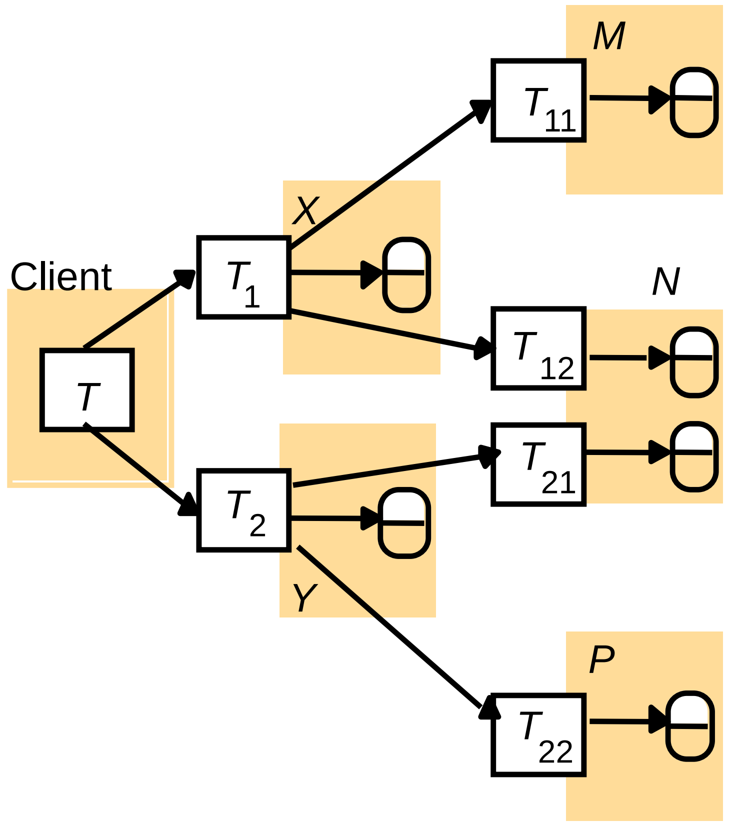


Figure 17.2
Nested banking transaction

T = *openTransaction*
    openSubTransaction
        a.withdraw(10);
    openSubTransaction
        b.withdraw(20);
    openSubTransaction
        c.deposit(10);
    openSubTransaction
        d.deposit(20);
    closeTransaction

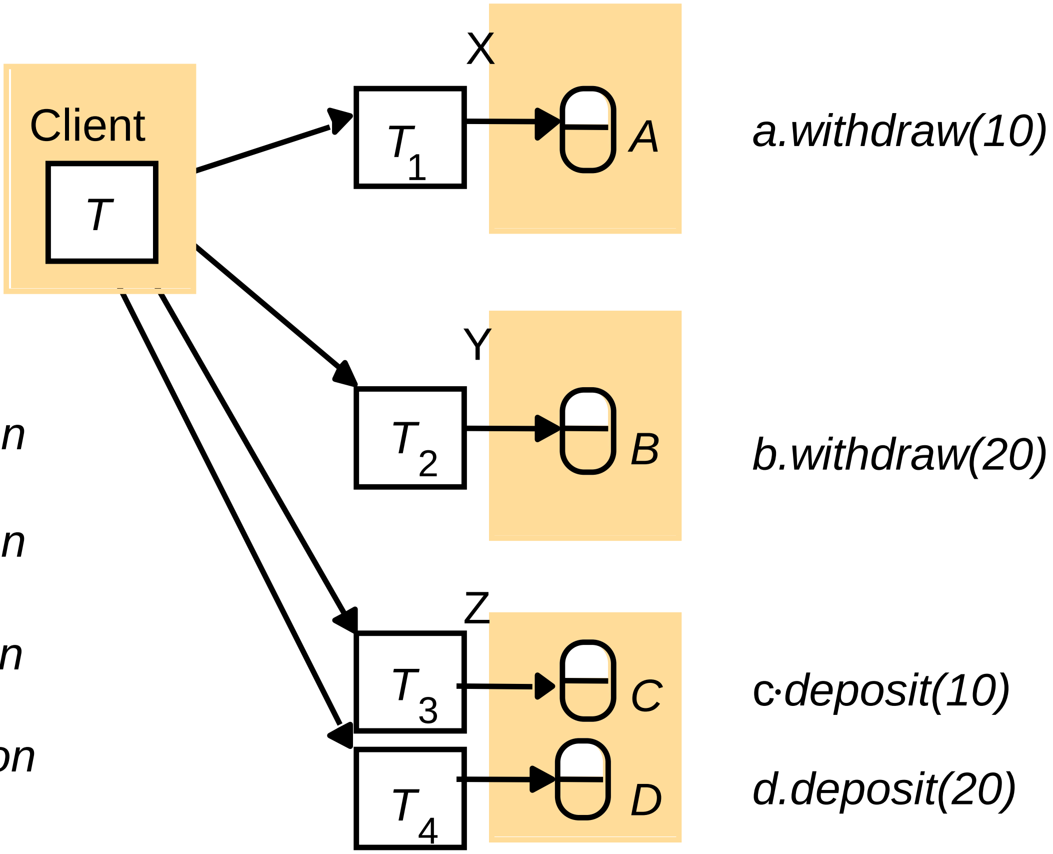
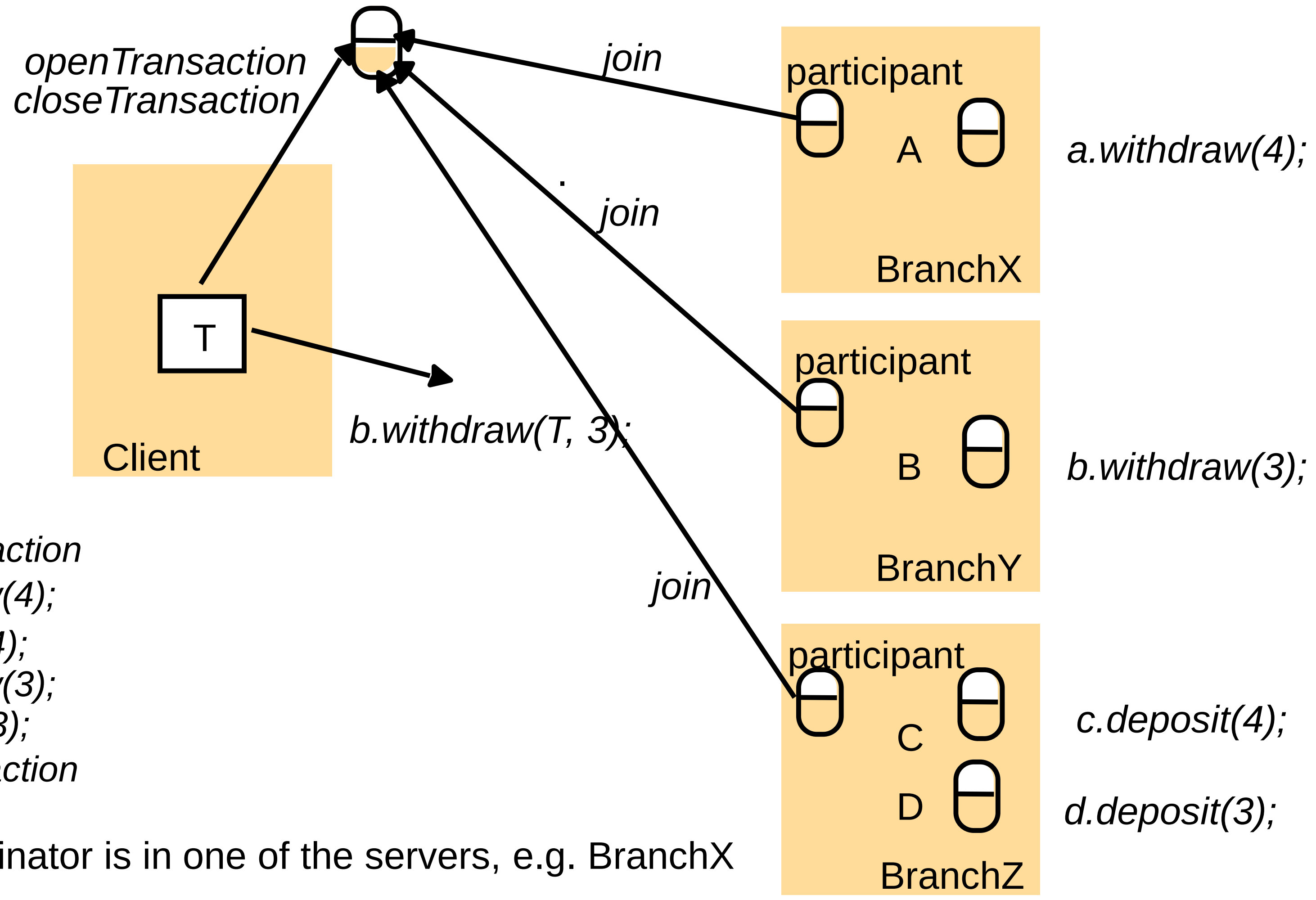


Figure 17.3
A distributed banking transaction



T = openTransaction
a.withdraw(4);
c.deposit(4);
b.withdraw(3);
d.deposit(3);
closeTransaction

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

COMMIT Protocols

- **Atomic commit**
  - Transaction commit protocols were devised in the early 1970s, and the two-phase commit protocol appeared in Gray [1978].
  - The atomicity property of transactions requires that when a distributed transaction comes to an end, <u>either all of its operations are carried out or none of them</u>.
  - In the case of a distributed transaction, the client has requested operations at more than one server.
    - A transaction comes to an end when the client requests that it be committed or aborted.
      - A simple way to complete the transaction in an atomic manner is for the coordinator to communicate the <u>commit</u> or <u>abort</u> request to all the participants in the transaction and to keep on repeating the request until all of them have acknowledged that they have carried it out.
    - This is an example of a one-phase atomic commit protocol.

COMMIT Protocols

- **Atomic commit (one-phase commit)**
  - This simple one-phase atomic commit protocol is inadequate, because it does not allow a server to make a unilateral decision to abort a transaction when the client requests a commit.
  - Reasons that prevent a server from being able to commit its part of a transaction generally relate to issues of concurrency control.
    - For example, if locking is in use, the resolution of a deadlock can lead to the aborting of a transaction without the client being aware unless it makes another request to the server.
    - Also, if optimistic concurrency control is in use, the failure of validation at a server would cause it to decide to abort the transaction.
    - Finally, the coordinator may not know if a server has crashed and been replaced during the progress of a distributed transaction – such a server will need to abort the transaction.

COMMIT Protocols

- **Atomic commit (two-phase commit)**
  - The two-phase commit protocol is designed to allow any participant to abort its part of a transaction.
    - Due to the requirement for atomicity, if one part of a transaction is aborted, then the whole transaction must be aborted.
  - The protocol is composed of two phases.

COMMIT Protocols

- **Atomic commit (two-phase commit)**
  - In the **first phase:**
    - Each participant votes for the transaction to be committed or aborted.
    - Once a participant has voted to commit a transaction, it is not allowed to abort it.
      - ✓ Therefore, before a participant votes to commit a transaction, it must ensure that it will eventually be able to carry out its part of the commit protocol, even if it fails and is replaced in the interim.
    - A participant in a transaction is said to be in a **prepared state** for a transaction if it will eventually be able to commit it.
    - To make sure of this, each participant saves in permanent storage all of the objects that it has altered in the transaction, together with its **status – prepared**.

COMMIT Protocols

- **Atomic commit (two-phase commit)**
  - In the **second phase:**
    - Every participant in the transaction carries out the joint decision.
    - If any one participant votes to abort, then the decision must be to abort the transaction.
    - If all the participants vote to commit, then the decision is to commit the transaction.
    - The problem is to ensure that all of the participants vote and that they all reach the same decision.
      - ✓ This is simple if no errors occur, but the protocol must work correctly even when some of the servers fail, messages are lost, or servers are temporarily unable to communicate with one another.

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

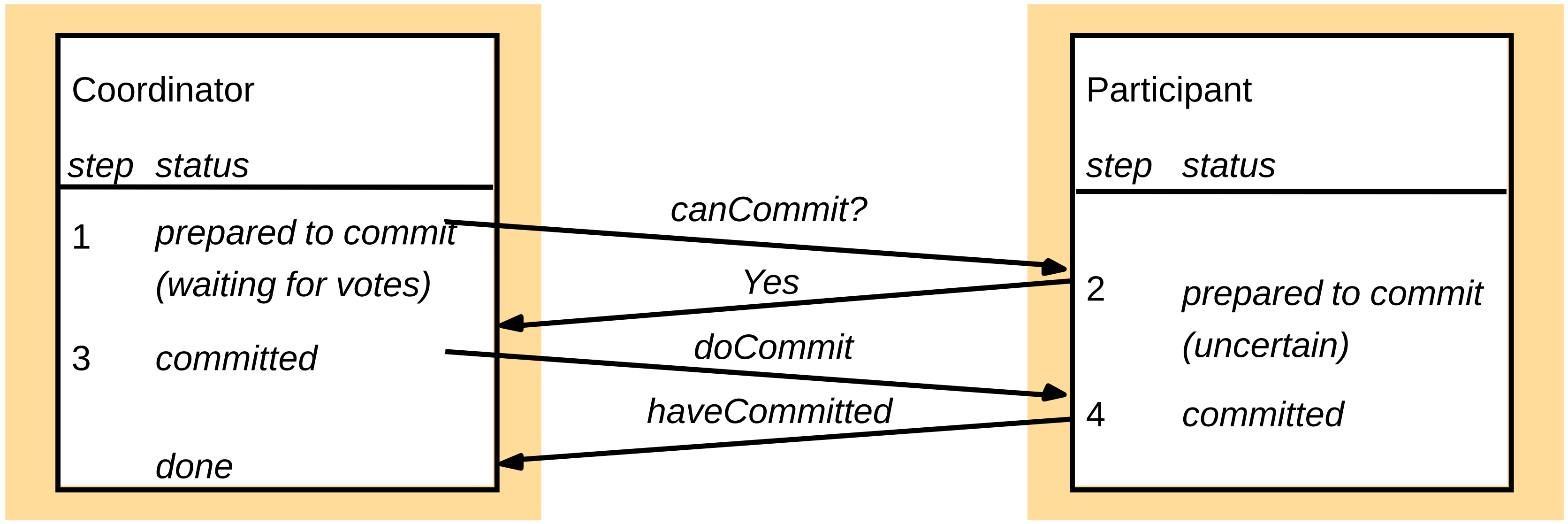


Figure 17.7
Operations in coordinator for nested transactions

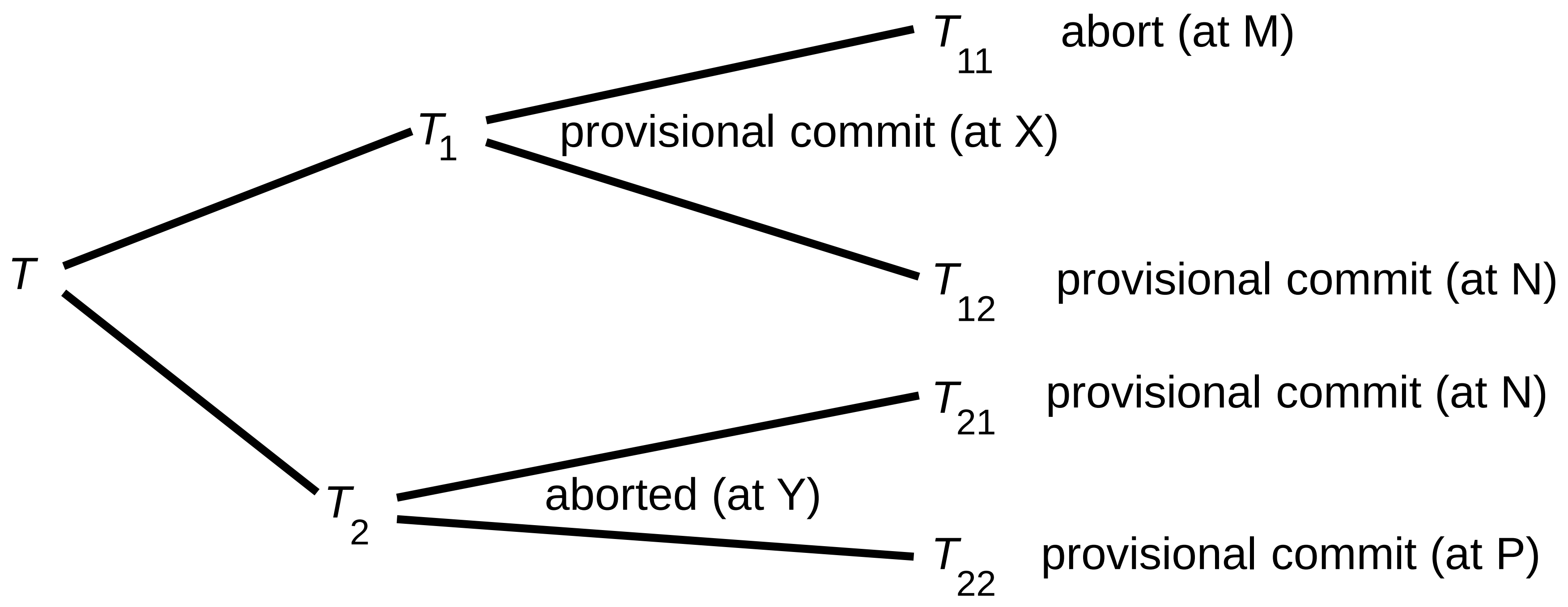
openSubTransaction(trans) -> subTrans

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



T_{11} abort (at M)
T_1 provisional commit (at X)
T
T_{12} provisional commit (at N)
T_{21} provisional commit (at N)
T_2 aborted (at Y)
T_{22} provisional commit (at P)

(b) Nested transactions

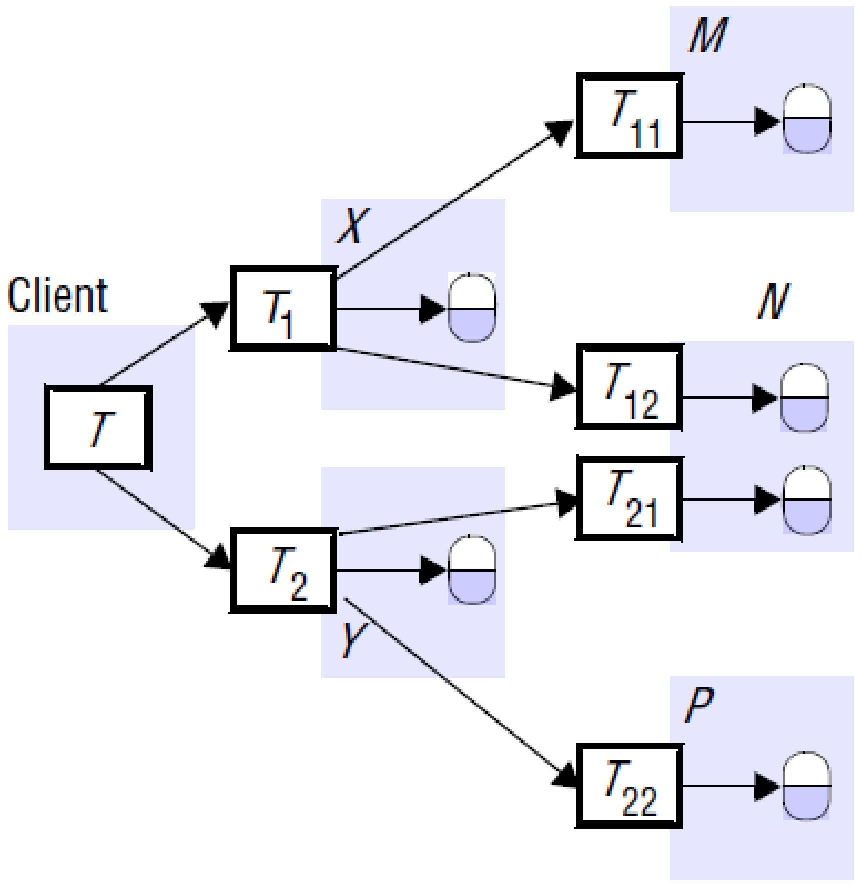


Figure 17.9
Information held by coordinators of nested transactions

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

© Pearson Education 2012

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*.

Concurrency control

- **Overview**:
  - Each server manages a set of objects and is responsible for ensuring that they remain consistent when accessed by concurrent transactions.
    - Therefore, each server is responsible for applying concurrency control to its own objects.
  - The members of a collection of servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner.
    - This implies that if transaction **T** is before transaction **U** in their conflicting access to objects at one of the servers, then they must be in that order at all of the servers whose objects are accessed in a conflicting manner by both **T** and **U**.

Concurrency control

- **Locking**:
  - In a distributed transaction, the *locks* on an object are held locally (in the same server).
  - The local lock manager can decide whether to grant a lock or make the requesting transaction wait.
  - However, it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
  - When locking is used for concurrency control, the objects remain locked and are unavailable for other transactions during the atomic commit protocol,
    - although an aborted transaction releases its locks after phase 1 of the protocol.
  - As lock managers in different servers set their locks independently of one another, it is possible that different servers may impose different orderings on transactions.

Concurrency control

- Consider the following interleaving of transactions T and U at servers X and Y:

T			*U*		
write(A)	at *X*	locks *A*			
			write(B)	at *Y*	locks *B*
read(B)	at *Y*	waits for *U*			
			read(A)	at *X*	waits for *T*

Instructor's Guide for Coulouris, Dollimore, Kindberg and Blair, Distributed Systems: Concepts and Design Edn. 5
© Pearson Education 2012

Concurrency control

- **Locking**:
  - The transaction **T** locks object A at server X, and then transaction **U** locks object B at server Y.
  - After that, **T** tries to access B at server Y and waits for **U**'s lock.
  - Similarly, transaction **U** tries to access A at server X and has to wait for **T**'s lock.
  - Therefore, we have **T** before **U** in one server and **U** before **T** in the other.
  - These different orderings can lead to cyclic dependencies between transactions, giving rise to a distributed deadlock situation.
  - When a deadlock is detected, a transaction is aborted to resolve the deadlock.
  - In this case, the coordinator will be informed and will abort the transaction at the participants involved in the transaction.

Concurrency control

- **Timestamp ordering concurrency control**:
  - In a **single server transaction**, the coordinator issues a unique timestamp to each transaction when it starts.
    - Serial equivalence is enforced by committing the versions of objects in the order of the timestamps of transactions that accessed them.
  - In **distributed transactions**, it is required that each coordinator issue globally unique timestamps.
    - A globally unique transaction timestamp is issued to the client by the first coordinator accessed by a transaction.
    - The transaction timestamp is passed to the coordinator at each server whose objects perform an operation in the transaction.

Concurrency control

- **Timestamp ordering concurrency control**:
  - In **distributed transactions**:
    - The servers of distributed transactions are jointly responsible for ensuring that they are performed in a serially equivalent manner.
      - ✓ For example, if the version of an object accessed by transaction **U** commits after the version accessed by **T** at one server, if **T** and **U** access the same object as one another at other servers they must commit them in the same order.
    - To achieve the same ordering at all the servers, the coordinators must agree as to the ordering of their timestamps.
    - A timestamp consists of a **<local timestamp, server-id>** pair.
    - The agreed ordering of pairs of timestamps is based on a comparison in which the server-id part is less significant.

Concurrency control

- **Timestamp ordering concurrency control**:
  - In **distributed transactions**:
    - The same ordering of transactions can be achieved at all the servers even if their local clocks are not synchronized.
    - However, for reasons of efficiency it is required that the timestamps issued by one coordinator be roughly synchronized with those issued by the other coordinators.
    - When this is the case, the ordering of transactions generally corresponds to the order in which they are started in real time.
    - Timestamps can be kept roughly synchronized by the use of synchronized local physical clocks.

Distributed Deadlocks

- **Characterization**
  - Deadlocks can arise within a <u>single server</u> when locking is used for concurrency control.
    - Servers must either prevent or detect and resolve deadlocks.
    - Using timeouts to resolve possible deadlocks is a clumsy approach – it is difficult to choose an appropriate timeout interval, and transactions may be aborted unnecessarily.
    - With deadlock detection schemes, a transaction is aborted only when it is involved in a deadlock.
    - Most deadlock detection schemes operate by finding cycles in the transaction wait-for graph.

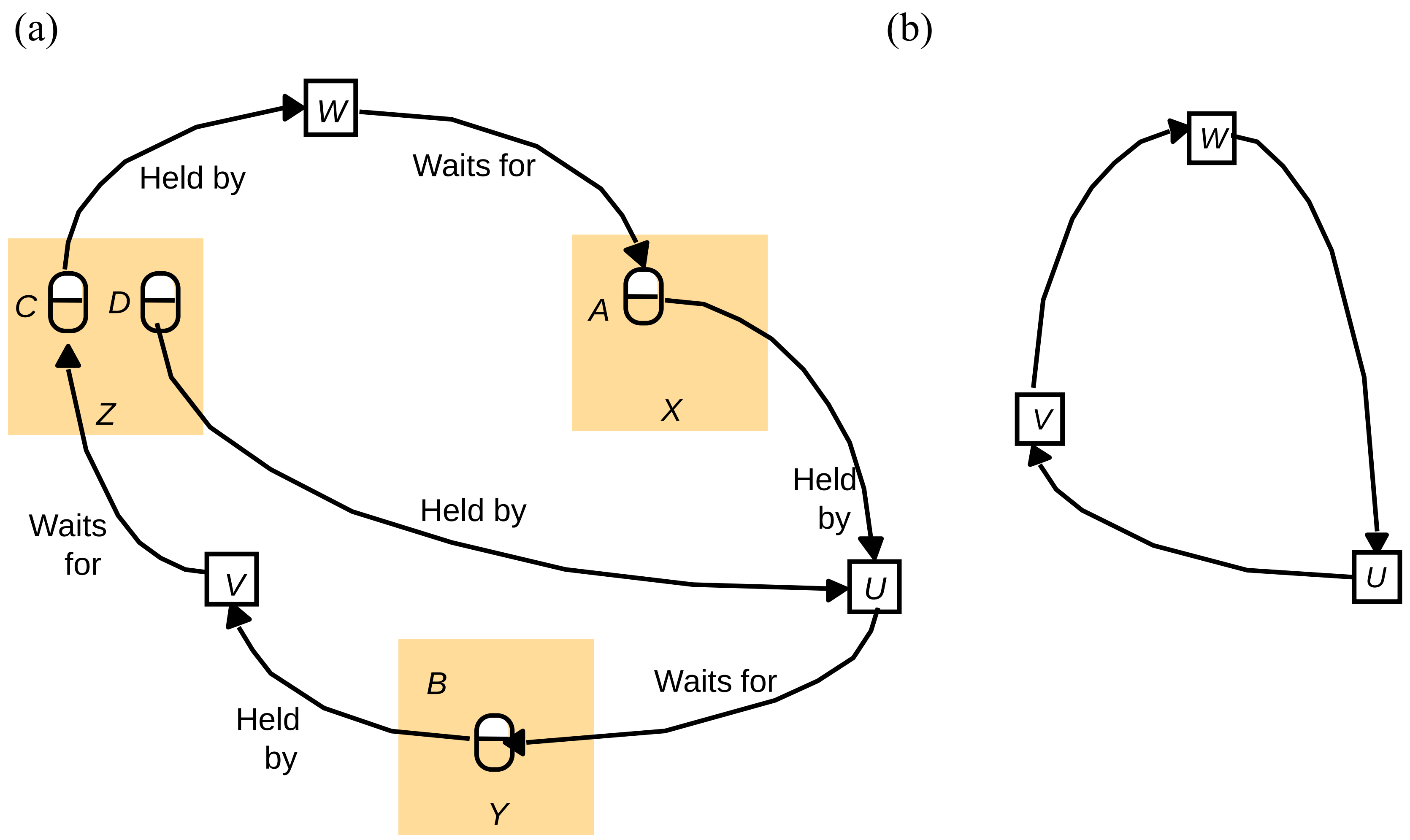
Distributed Deadlocks

- **Characterization**
  - In a distributed system involving multiple servers being accessed by multiple transactions, a global wait-for graph can be constructed from the local ones.
  - There can be a cycle in the *global wait-for graph* that is not in any single local one – that is, there can be a **distributed deadlock**.
    - Recall that the *wait-for graph* is a directed graph in which nodes represent transactions and objects, and edges represent either an object held by a transaction or a transaction waiting for an object.
    - There is a deadlock if and only if there is a cycle in the wait-for graph.

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



Distributed Deadlocks

- **Detection**
  - Detection of a distributed deadlock requires a cycle to be found in the global transaction wait-for graph that is distributed among the servers that were involved in the transactions.
  - Local wait-for graphs can be built by the lock manager at each server.
  - In the previous example, the local wait-for graphs of the servers are:

    server Y: $U \rightarrow V$ (added when U requests *b.withdraw*(30))
    server Z: $V \rightarrow W$ (added when V requests *c.withdraw*(20))
    server X: $W \rightarrow U$ (added when W requests *a.withdraw*(20))

  - As the global wait-for graph is held in part by each of the several servers involved, communication between these servers is required to find cycles in the graph.

Distributed Deadlocks

- **Detection**
  - A simple solution is to use centralized deadlock detection, in which one server takes on the role of global deadlock detector.
    - From time to time, each server sends the latest copy of its local wait-for graph to the global deadlock detector, which fuses the information in the local graphs to construct a global wait-for graph.
    - The global deadlock detector checks for cycles in the global wait-for graph.
    - When it finds a cycle, it makes a decision on how to resolve the deadlock and tells the servers which transaction to abort.
  - Centralized deadlock detection is not that good, because it depends on a single server to carry it out.
    - It suffers from the usual problems associated with centralized solutions in distributed systems – poor availability, lack of fault tolerance and no ability to scale.

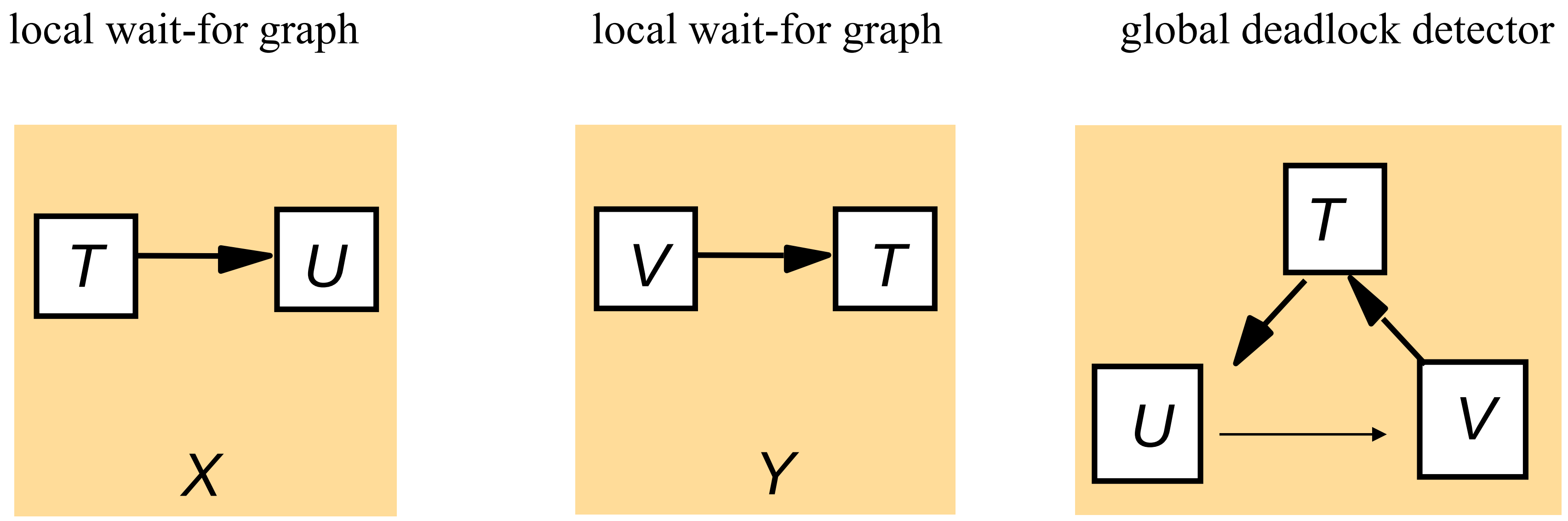
Distributed Deadlocks

- **Phantom deadlocks**
  - A deadlock that is ‘detected’ but is not really a deadlock is called a phantom deadlock.
  - In distributed deadlock detection, information about wait-for relationships between transactions is transmitted from one server to another.
    - If there is a deadlock, the necessary information will eventually be collected in one place and a cycle will be detected.
    - As this procedure will take some time, there is a chance that one of the transactions that holds a lock will meanwhile have released it, in which case the deadlock will no longer exist.

Distributed Deadlocks

- **Phantom deadlocks**
  - Consider the case of a global deadlock detector that receives local wait-for graphs from servers X and Y (next slide).
    - Suppose that transaction **U** then releases an object at server X and requests the one held by **V** at server Y.
    - Suppose also that the global detector receives server Y's local graph before server X's.
    - In this case, it would detect a cycle **T** → **U** → **V** → **T**, although the edge **T** → **U** no longer exists.
    - This is an example of a phantom deadlock.

Figure 17.14
Local and global wait-for graphs



Distributed Deadlocks

- **Edge chasing**
  - A distributed approach to deadlock detection uses a technique called **edge chasing** or **path pushing**.
  - In this approach, the global wait-for graph is not constructed, but each of the servers involved has knowledge about some of its edges.
  - The <u>servers attempt to find cycles</u> by forwarding messages called **probes**, which follow the edges of the graph throughout the distributed system.
  - A **probe** message consists of transaction wait-for relationships representing a path in the global wait-for graph.

Distributed Deadlocks

- **Edge chasing**
  - Each distributed transaction starts at a server (<u>coordinator of the transaction</u>) and moves to several other servers (<u>participants in the transaction</u>), which can communicate with the coordinator.
  - At any point in time, a transaction can be either <u>active</u> or <u>waiting</u> at just one of these servers.
    - The coordinator is responsible for recording whether the transaction is active or is waiting for a particular object, and participants can get this information from their coordinator.
  - <u>Lock managers</u> inform coordinators when transactions start *waiting* for objects and when transactions acquire objects and become *active* again.
    - When a transaction is aborted to break a deadlock, its coordinator will inform the participants and all its locks will be removed, with the effect that all edges involving that transaction will be removed from the local wait-for graphs.

Distributed Deadlocks

- **Edge chasing algorithms have three steps:**
  - Initiation
  - Detection
  - Resolution

Distributed Deadlocks

- **Edge chasing algorithms have three steps:**
  - Initiation
    - When a server notes that a transaction **T** starts waiting for another transaction **U**, where **U** is waiting to access an object at another server, it initiates detection by sending a probe containing the edge **< T → U >** to the server of the object at which transaction U is blocked.
    - If **U** is sharing a lock, probes are sent to all the holders of the lock.
    - Sometimes further transactions may start sharing the lock later on, in which case probes can be sent to them too.

Distributed Deadlocks

- **Edge chasing algorithms have three steps:**
  - Detection
    - Detection consists of receiving probes and deciding whether a deadlock has occurred and whether to forward the probes.
    - For example:
      - ✓ When a server of an object receives a probe **< T → U >** (indicating that **T** is waiting for a transaction **U** that holds a local object), it checks to see whether **U** is also waiting.
      - ✓ If it is, the transaction it waits for (for example, **V**) is added to the probe (making it **< T → U → V >** and if the new transaction (**V**) is waiting for another object elsewhere, the probe is forwarded.
      - ✓ Before forwarding a probe, the server checks to see whether the transaction (e.g., **T**) it has just added has caused the probe to contain a cycle (for example, **< T → U → V → T >**).
      - ✓ If this is the case, it has found a cycle in the graph and a deadlock has been detected.

Distributed Deadlocks

- **Edge chasing algorithms have three steps:**
  - Resolution
    - When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.
    - The transaction to be aborted can be chosen according to transaction priorities.

Figure 17.15
Probes transmitted to detect deadlock

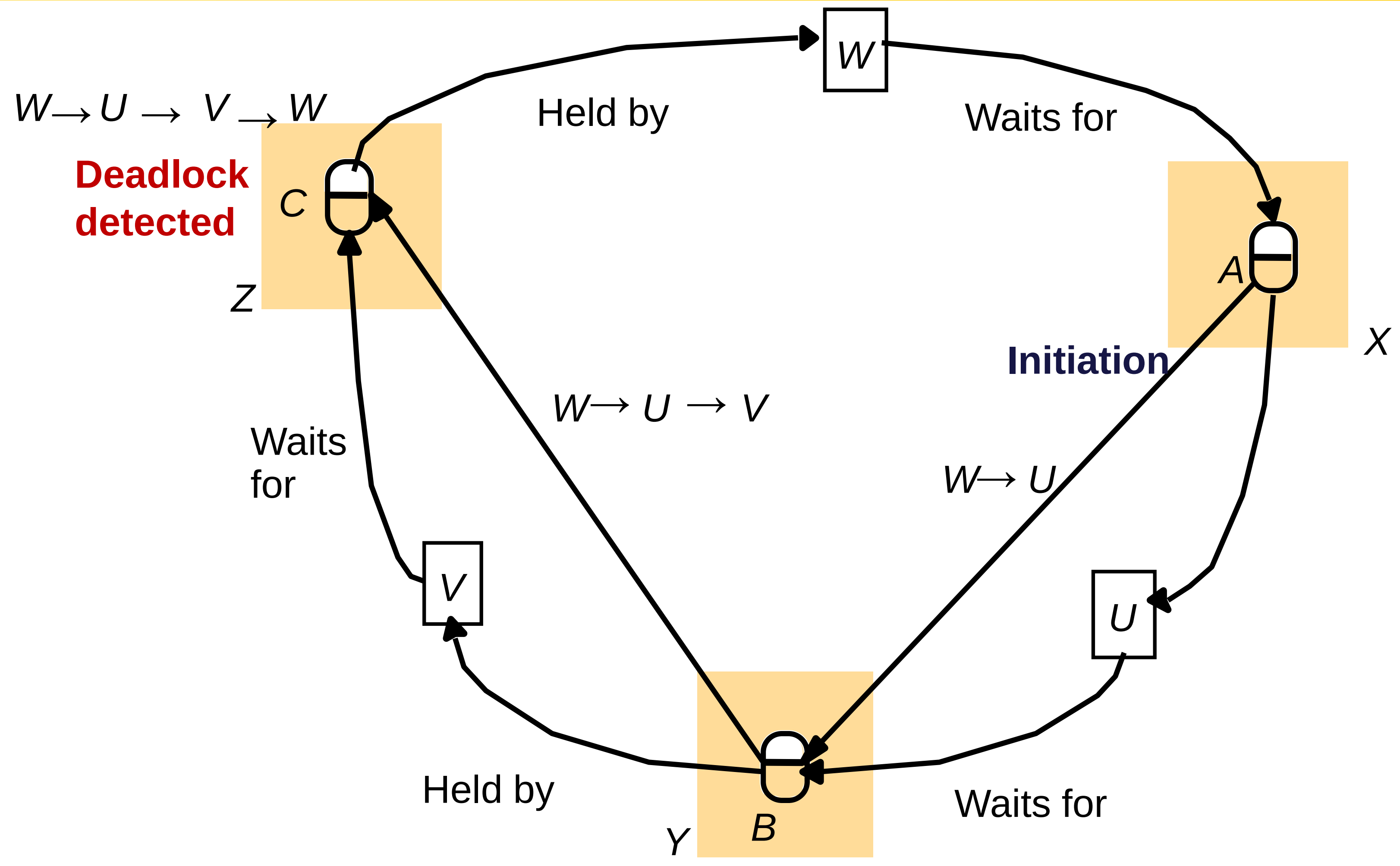
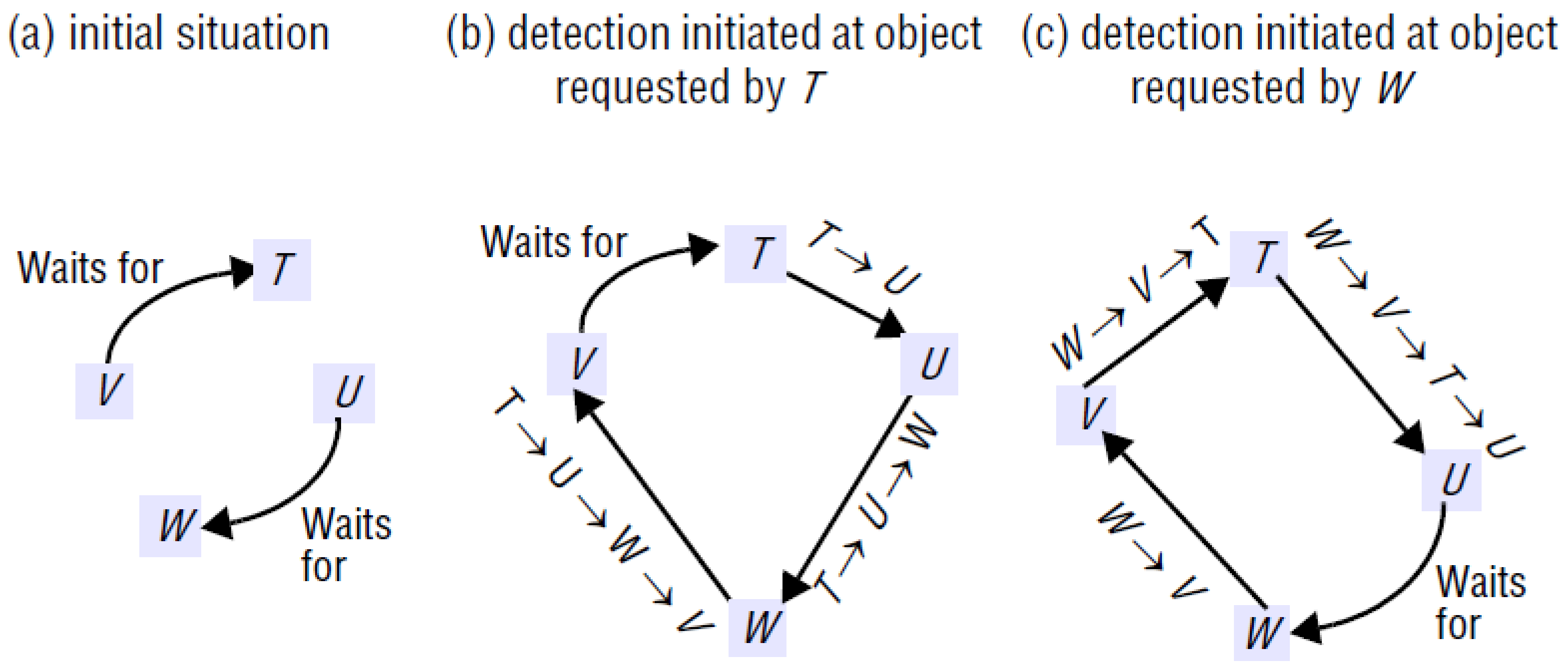


Figure 17.16
Two probes initiated



Transaction Recovery

- **Properties**
  - The atomic property of transactions requires that all the effects of committed transactions and none of the effects of incomplete or aborted transactions are reflected in the objects they accessed.
  - This property can be described in terms of two aspects:
    - **Durability**: requires that objects are saved in permanent storage and will be available indefinitely thereafter. Therefore, an acknowledgement of a client's commit request implies that all the effects of the transaction have been recorded in permanent storage as well as in the server's (volatile) objects.
    - **Failure atomicity**: requires that effects of transactions are atomic even when the server crashes. Recovery is concerned with ensuring that a server's objects are durable and that the service provides failure atomicity.

Transaction Recovery

- **Recovery file**
  - Although file servers and database servers maintain data in permanent storage, other kinds of servers of recoverable objects need not do so except for recovery purposes.
  - It is assumed that when a server is running it keeps all its objects in its volatile memory and records its committed objects in a <u>recovery file</u> or files.
  - Therefore, recovery consists of restoring the server with the latest committed versions of its objects from permanent storage.
  - Databases need to deal with large volumes of data.
    - They generally hold the objects in stable storage on disk with a cache in volatile memory.

Transaction Recovery

- **Recovery manager**
  - The requirements for durability and failure atomicity are not independent of one another and can be dealt with by a single mechanism – the **recovery manager**.
  - The tasks of a **recovery manager** are:
    - to save objects in permanent storage (in a recovery file) for committed transactions.
    - to restore the server's objects after a crash.
    - to reorganize the recovery file to improve the performance of recovery.
    - to reclaim storage space (in the recovery file).

Transaction Recovery

- **Recovery manager**
  - It is required the **recovery manager** to be resilient to media failures.
    - Corruption during a crash, random decay or a permanent failure can lead to failures of the recovery file, which can result in some of the data on the disk being lost.
    - In such cases we need another copy of the recovery file.
    - Stable storage, which is implemented to be very unlikely to fail by using mirrored disks or copies at a different location may be used for this purpose.

Transaction Recovery

- **Intentions list**
  - Any server that provides transactions needs to keep track of the objects accessed by clients' transactions.
  - At each server, an **intentions list** is recorded for all its currently active transactions.
    - An **intentions list** of a transaction contains a list of the references and values of all the objects that are altered by that transaction.
    - When a transaction is committed, that transaction's intentions list is used to identify the objects it affected.
    - The committed version of each object is replaced by the tentative version made by that transaction, and the new value is written to the server's **recovery file**.
    - When a transaction aborts, the server uses the intentions list to delete all the tentative versions of objects made by that transaction.

Transaction Recovery

- **Intentions list**
  - Recall that a distributed transaction must carry out an atomic commit protocol before it can be committed or aborted.
  - If the participants cannot agree to commit, they must abort the transaction.
  - At the point when a participant says it is prepared to commit a transaction, its **recovery manager** must have
    - saved both its <u>intentions list</u> for that transaction and the objects in that intentions list in its <u>recovery file</u>, so that it will be able to carry out the commitment later, even if it crashes in the interim.

Transaction Recovery

- **Intentions list**
  - When all the participants involved in a transaction agree to commit it, the coordinator informs the client and then sends messages to the participants to commit their part of the transaction.
  - Once the client has been informed that a transaction has committed, the **recovery files** of the participating servers must contain sufficient information to ensure that the transaction is committed by all of the servers, even if some of them crash between preparing to commit and committing.

Transaction Recovery

- **Recovery files**
  - To deal with recovery of a server that can be involved in distributed transactions, further information in addition to the values of the objects is stored in the recovery file.
  - This information concerns the status of each transaction – whether it is **committed**, **aborted** or **prepared** to commit.
  - In addition, each object in the recovery file is associated with a particular transaction by saving the intentions list in the recovery file.

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.

Transaction Recovery

- **Logging**
  - In the logging technique, the recovery file represents a log containing the history of all the transactions performed by a server.
  - The history consists of values of objects, transaction status entries and transaction intentions lists.
  - The order of the entries in the log reflects the order in which transactions have prepared, committed and aborted at that server.
  - In practice, the recovery file will contain a recent snapshot of the values of all the objects in the server followed by a history of transactions postdating the snapshot.

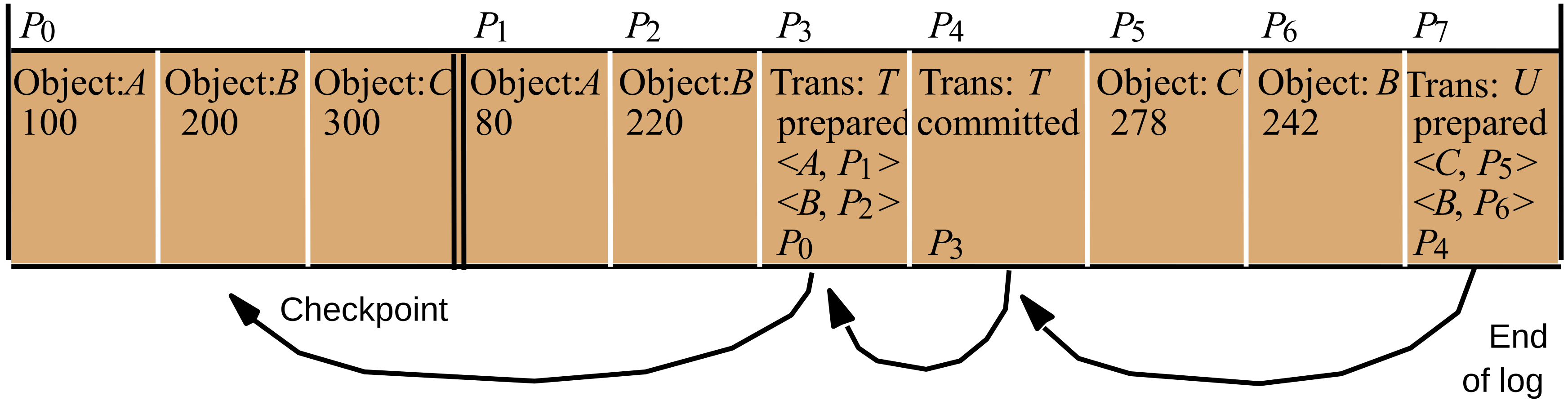
Transaction Recovery

- **Logging**
  - During the normal operation of a server, its <u>recovery manager</u> is called whenever a transaction prepares to commit, commits or aborts a transaction.
  - When the server is prepared to commit a transaction, the <u>recovery manager</u> appends all the objects in its intentions list to the recovery file, followed by the current status of that transaction (prepared) together with its intentions list.
  - When a transaction is eventually committed or aborted, the recovery manager appends the corresponding status of the transaction to its recovery file.

Transaction Recovery

- **Logging**
  - It is assumed that the append operation is atomic in the sense that it writes one or more complete entries to the recovery file.
    - If the server fails, only the last write can be incomplete.
  - To make efficient use of the disk, several subsequent writes can be buffered and then written to disk as a single write.
    - An additional advantage of the logging technique is that sequential writes to disk are faster than writes to random locations.
  - After a crash, any transaction that does not have a committed status in the log is aborted.
    - Therefore, when a transaction commits, its committed status entry must be forced to the log.

Figure 17.19
Log for banking service



© Pearson Education 2012

Transaction Recovery

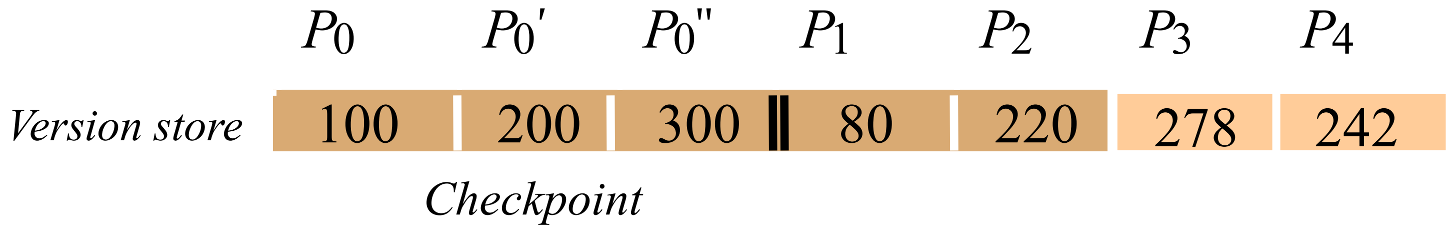
- **Shadow versions**
  - The logging technique records transaction status entries, intentions lists and objects all in the same file – the log.
  - The **shadow versions** technique is an alternative way to organize a recovery file.
    - It uses a map to locate versions of the server’s objects in a file called a version store.
    - The map associates the identifiers of the server’s objects with the positions of their current versions in the version store.
    - The versions written by each transaction are ‘shadows’ of the previous committed versions.
    - The transaction status entries and intentions lists are stored separately.

Transaction Recovery

- **Shadow versions**
  - When a transaction is prepared to commit, any of the objects changed by the transaction are appended to the version store, leaving the corresponding committed versions unchanged.
    - These new as-yet-tentative versions are called shadow versions.
  - When a transaction commits, a new map is made by copying the old map and entering the positions of the shadow versions.
    - To complete the commit process, the new map replaces the old map.
  - To restore the objects when a server is replaced after a crash, its recovery manager reads the map and uses the information in the map to locate the objects in the version store.

Figure 17.20
Shadow versions

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



Transaction Recovery

- **Recovery of the two-phase commit protocol**
  - The recovery management must deal with any transactions that are performing the two-phase commit protocol at the time when a server fails.
  - The recovery managers use two new status values for this purpose: **done** and **uncertain**.
  - A coordinator uses committed to indicate that the outcome of the vote is **Yes** and done to indicate that the two-phase commit protocol is **complete**.
  - A participant uses uncertain to indicate that it has voted **Yes** but does not yet know the outcome of the vote.

Transaction Recovery

- **Recovery of the two-phase commit protocol**
  - **In phase 1 of the protocol**:
    - When the coordinator is prepared to commit (and has already added a *prepared* status entry to its recovery file), its recovery manager adds a coordinator entry to its recovery file.
    - Before a participant can vote Yes, it must have already prepared to commit (and must have already added a *prepared* status entry to its recovery file).
    - When it votes Yes, its recovery manager records a participant entry and adds an *uncertain* transaction status to its recovery file as a forced write.
    - When a participant votes No, it adds an *abort* transaction status to its recovery file.

Transaction Recovery

- **Recovery of the two-phase commit protocol**
  - **In phase 2 of the protocol**:
    - The recovery manager of the coordinator adds either a *committed* or an *aborted* transaction status to its recovery file, according to the decision.
      - ✓ This must be a forced write (that is, it is written immediately to the recovery file).
    - Recovery managers of participants add a *commit* or *abort* transaction status to their recovery files according to the message received from the coordinator.
    - When a coordinator has received a confirmation from all of its participants, its recovery manager adds a *done* transaction status to its recovery file – this need not be forced.
    - The *done* status entry is not part of the protocol but is used when the recovery file is reorganized.

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

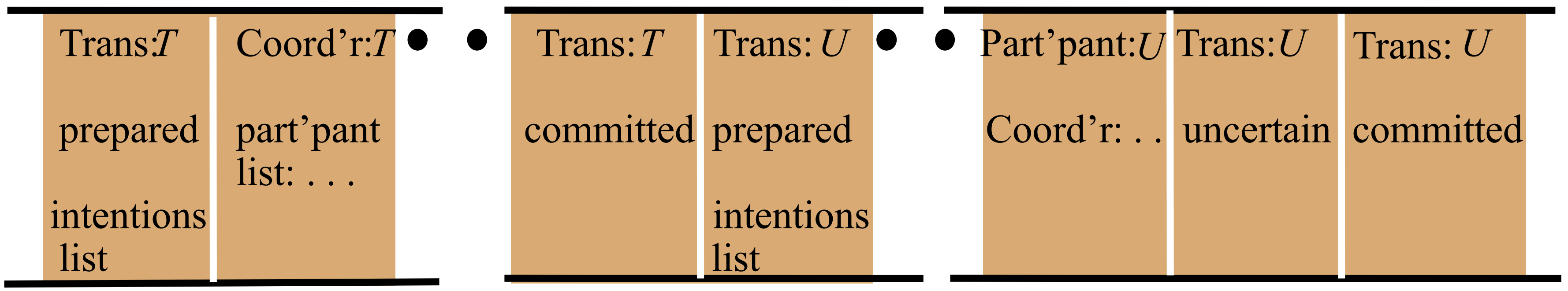


Figure 17.22
Recovery of the two-phase commit protocol

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 13.5).
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.