17.4 Concurrency control in distributed transactions

- 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
  - therefore 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*

17.4.1 Locking

- In a distributed transaction, the locks on an object are held by the server that manages it.
  - The local lock manager decides whether to grant a lock or make the requesting transaction wait.
  - it cannot release any locks until it knows that the transaction has been committed or aborted at all the servers involved in the transaction.
  - the objects remain locked and are unavailable for other transactions during the atomic commit protocol
    - an aborted transaction releases its locks after phase 1 of the protocol.

Interleaving of transactions T and U at servers X and Y

- in the example on page 741, we have
  - *T* before *U* at server X and *U* before *T* at server Y
- different orderings lead to cyclic dependencies and distributed deadlock
  - detection and resolution of distributed deadlock in next section

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

17.4.2 Timestamp ordering concurrency control

- Single server transactions
  - coordinator issues a unique timestamp to each transaction when it starts
  - serial equivalence ensured by committing objects in order of timestamps
- Distributed transactions
  - the first coordinator accessed by a transaction issues a globally unique timestamp
  - as before the timestamp is passed with each object access
  - the servers are jointly responsible for ensuring serial equivalence
    - that is if T access an object before U, then T is before U at all objects
  - coordinators agree on timestamp ordering
    - a timestamp consists of a pair <*local timestamp*, *server-id*>.
    - the agreed ordering of pairs of timestamps is based on a comparison in which the server-id part is less significant – they should relate to time

Can the same ordering be achieved at all servers without clock synchronization?

Why is it better to have roughly synchronized clocks?

- The same ordering can be achieved at all servers even if their clocks are not synchronized
  - for efficiency it is better if local clocks are roughly synchronized
  - then the ordering of transactions corresponds roughly to the real time order in which they were started
- Timestamp ordering
  - conflicts are resolved as each operation is performed
  - if this leads to an abort, the coordinator will be informed
    - it will abort the transaction at the participants
  - any transaction that reaches the client request to commit should always be able to do so
    - participant will normally vote *yes*
    - unless it has crashed and recovered during the transaction

Answer

Problems when local orderings far from real time, e.g. S1 has 10 and S2 has 100 then transactions at S1 are always too late.

- suppose that transactions T and U are started at S1 and S2 with timestamps <S1, 10> and <S2, 100>
- we have <S2, 100> > <S1, 10> , similarly <S2, 100> > <S1, 11> etc
- so transactions such as T at S1 will find that transactions such as U at S2 have timestamp < timestamps set by U when reading and writing objects. so it will be hard for T to succeed

Use backward validation

Optimistic concurrency co

1. write/read, 2. read/write, 3. write/write

- each transaction is validated before it is allowed to commit
  - transaction numbers assigned at start of validation
  - transactions serialized according to transaction numbers
  - validation takes place in phase 1 of 2PC protocol
- consider the following interleavings of *T* and *U*
  - *T* before *U* at *X* and *U* bef

Suppose T & U start validation at about the same time

T		*U*	
Read(*A*)	at *X*	*Read*(*B*)	at *Y*
Write(*A*)		*Write*(*B*)	
Read(*B*)	at *Y*	*Read*(*A*)	at *X*

X validates T first
Y validates U first

each server only validates one transaction at a time, so each server will be unable to validate the other transaction until the first has completed. Thus, commitment deadlock

Commitment deadlock in optimistic concurrency control

- servers of distributed transactions do parallel validation
  - therefore rule 3 must be validated as well as rule 2
    - the write set of T_v is checked for overlaps with write sets of earlier transactions
  - this prevents commitment deadlock
  - it also avoids delaying the 2PC protocol
- another problem - independent servers may schedule transactions in different orders
  - e.g. T before U at X and U before T at Y
  - this must be prevented - some hints as to how on page 743

- Global validation after local ones
- Use of globally unique transaction numbers with agreed orderings

14.5 Distributed deadlocks

- Single server transactions can experience deadlocks
  - prevent or detect and resolve
  - use of timeouts is clumsy, detection is preferable.
    - it uses wait-for graphs.
- Distributed transactions lead to distributed deadlocks
  - in theory can construct global wait-for graph from local ones
  - a cycle in a global wait-for graph that is not in local ones is a distributed deadlock

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

- objects *A*, *B* managed by *X* and *Y* ; *C* and *D* by *Z*
  - next slide has global wait-for graph

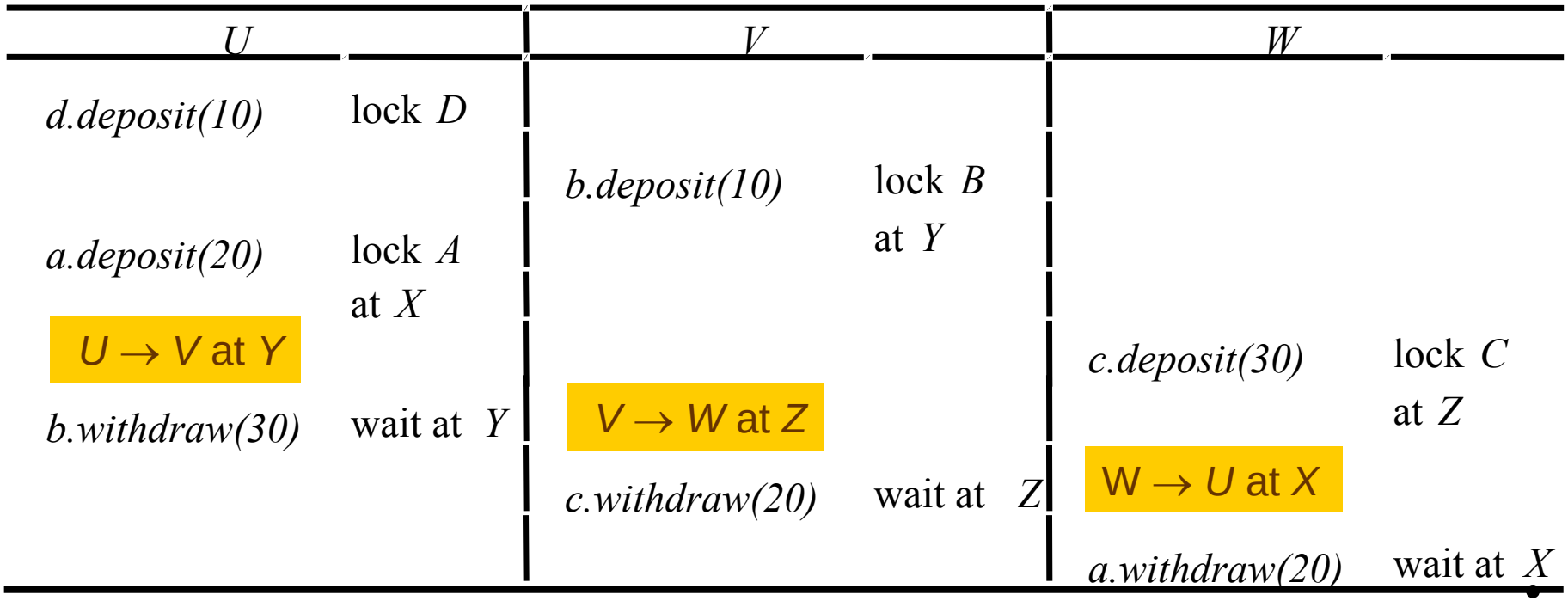
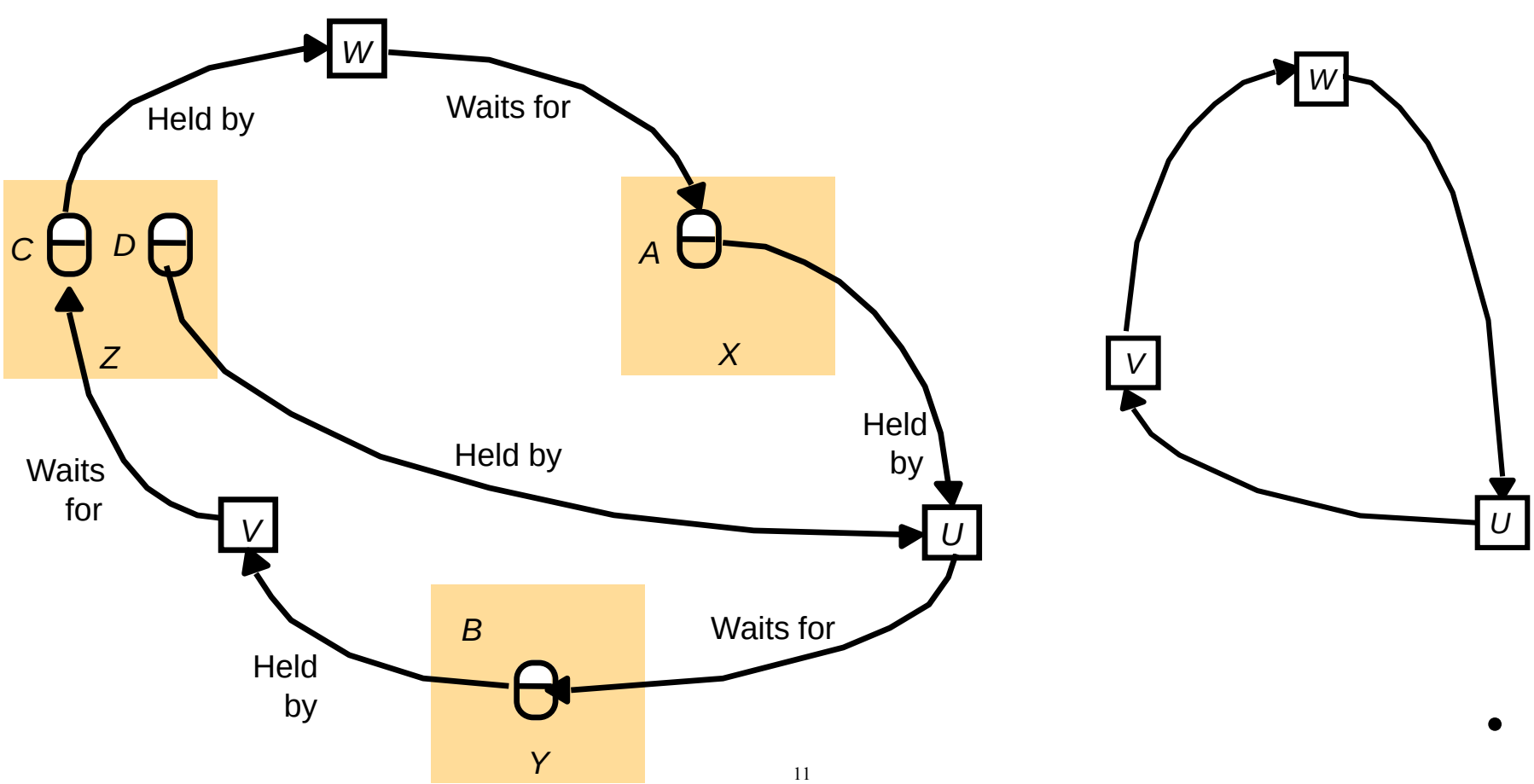


Figure 17.13
Distributed deadlock

- a deadlock cycle has alternate edges showing wait-for and held-by
- wait-for added in order: $U \rightarrow V$ at Y; $V \rightarrow W$ at Z and $W \rightarrow U$ at X

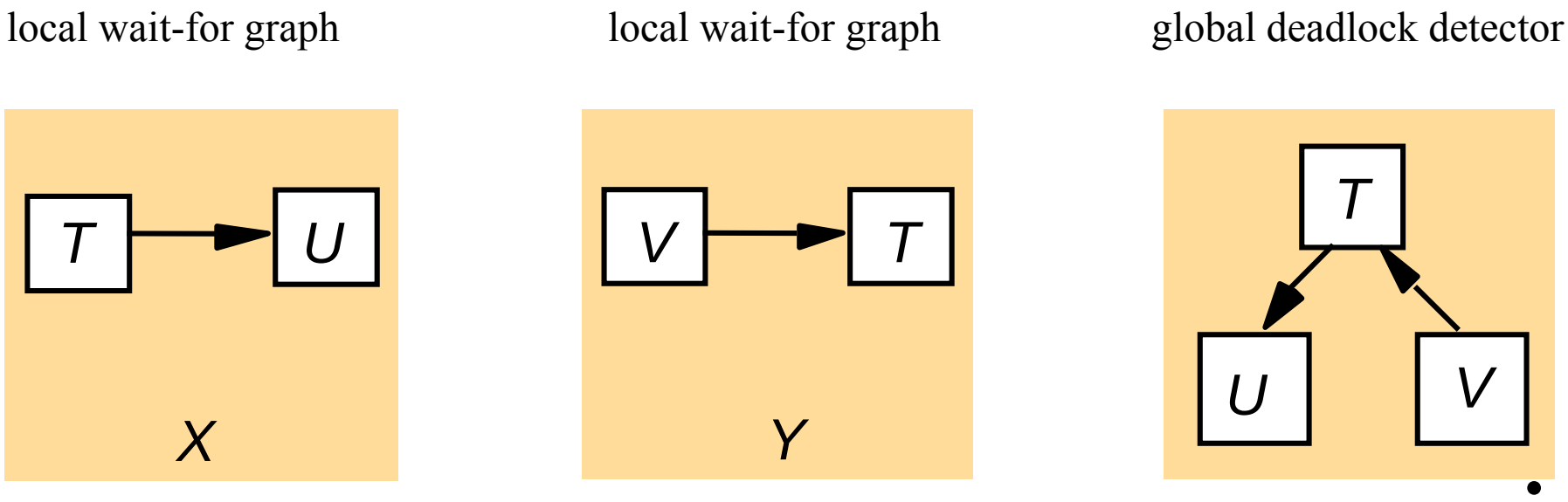


Deadlock detection - local wait-for graphs

- Local wait-for graphs can be built, e.g.
  - 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)
- to find a global cycle, communication between the servers is needed
- centralized deadlock detection
  - one server takes on role of global deadlock detector
  - the other servers send it their local graphs from time to time
  - it detects deadlocks, makes decisions about which transactions to abort and informs the other servers
  - usual problems of a centralized service - poor availability, lack of fault tolerance and no ability to scale. Besides, it is costly.

Figure 17.14
Local and global wait-for graphs

- Phantom deadlocks
  - a ‘deadlock’ that is detected, but is not really one
  - happens when there appears to be a cycle, but one of the transactions has released a lock, due to time lags in distributing graphs
  - in the figure suppose U releases the object at X then waits for V at Y
    - and the global detector gets Y’s graph before X’s (T → U → V → T)



Edge chasing - a distributed approach to deadlock detection

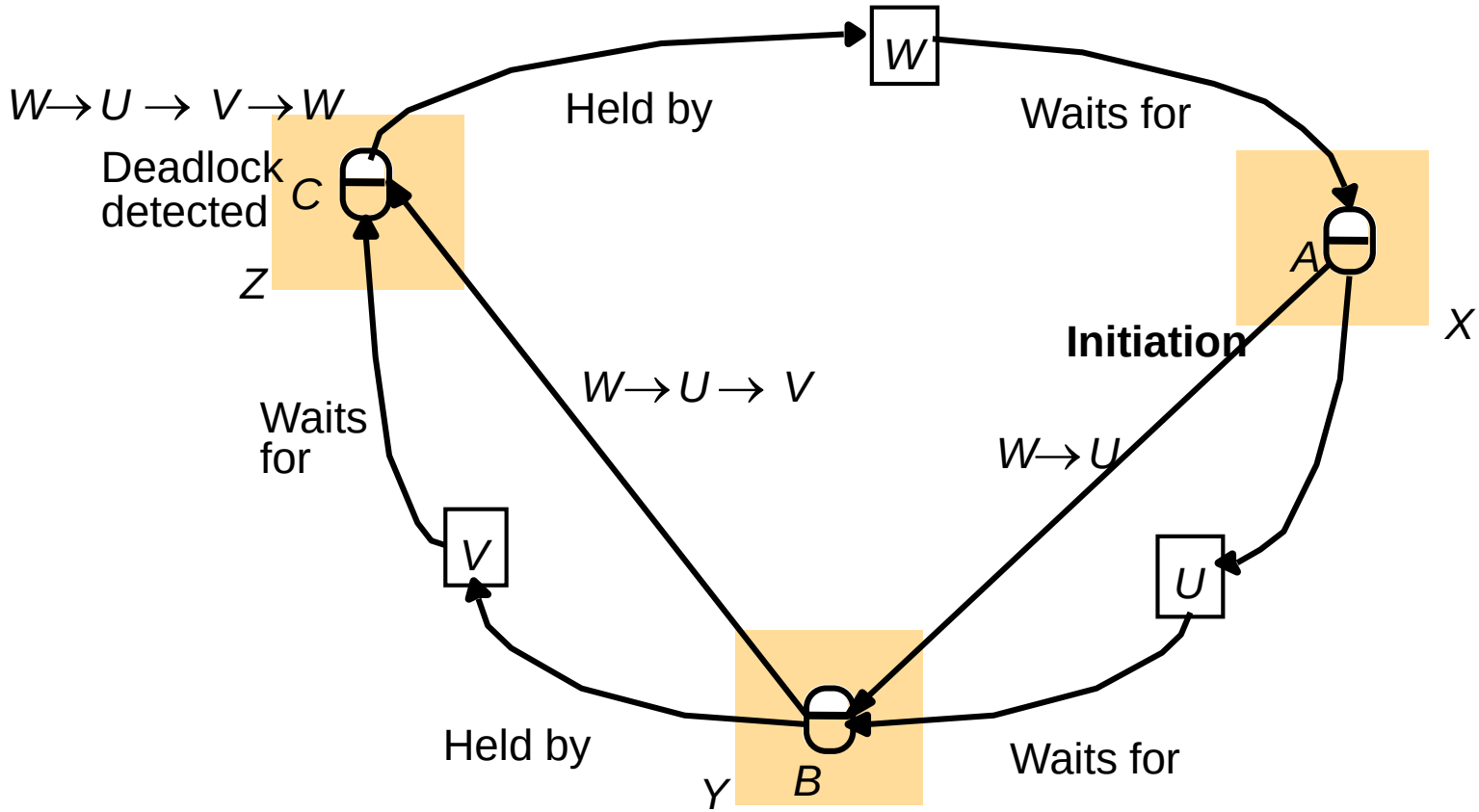
- a global graph is not constructed, but each server knows about some of the edges
  - servers try to find cycles by sending *probes* which follow the edges of the graph through the distributed system
  - when should a server send a probe (go back to Fig 17.13)
  - edges were added in order $U \rightarrow V$ at Y; $V \rightarrow W$ at Z and $W \rightarrow U$ at X
    - when $W \rightarrow U$ at X was added, U was waiting, but
    - when $V \rightarrow W$ at Z, W was not waiting
  - send a probe when an edge $T1 \rightarrow T2$ when $T2$ is waiting
  - each coordinator records whether its transactions are active or waiting
    - the local lock manager tells coordinators if transactions start/stop waiting
    - when a transaction is aborted to break a deadlock, the coordinator tells the participants, locks are removed and edges taken from wait-for graphs

Edge-chasing algorithms

- Three steps
  - Initiation:
    - When a server notes that T starts waiting for U, where U is waiting at another server, it initiates detection by sending a probe containing the edge < T $\rightarrow$ U > to the server where U is blocked.
    - If U is sharing a lock, probes are sent to all the holders of the lock.
  - Detection:
    - Detection consists of receiving probes and deciding whether deadlock has occurred and whether to forward the probes.
      - e.g. when server receives probe < T $\rightarrow$ U > it checks if U is waiting, e.g. U $\rightarrow$ V, if so it forwards < T $\rightarrow$ U $\rightarrow$ V > to server where V waits
      - when a server adds a new edge, it checks whether a cycle is there
  - Resolution:
    - When a cycle is detected, a transaction in the cycle is aborted to break the deadlock.

Figure 17.15
Probes transmitted to detect deadlock

- example of edge chasing starts with *X* sending <$W \rightarrow U$>, then *Y* sends <$W \rightarrow U \rightarrow V$ >, then *Z* sends <$W \rightarrow U \rightarrow V \rightarrow W$>



Edge chasing conclusion

- probe to detect a cycle with *N* transactions will require 2(*N*-1) messages.
  - Studies of databases show that the average deadlock involves 2 transactions.
- the above algorithm detects deadlock provided that
  - waiting transactions do not abort
  - no process crashes, no lost messages
  - to be realistic it would need to allow for the above failures
- refinements of the algorithm
  - to avoid more than one transaction causing detection to start and then more than one being aborted
  - not time to study these now

Summary of concurrency control for distributed transactions

- each server is responsible for the serializability of transactions that access its own objects.
- additional protocols are required to ensure that transactions are serializable globally.
  - timestamp ordering requires a globally agreed timestamp ordering
  - optimistic concurrency control requires global validation or a means of forcing a global ordering on transactions.
  - two-phase locking can lead to distributed deadlocks.
    - distributed deadlock detection looks for cycles in the global wait-for graph.
    - edge chasing is a non-centralized approach to the detection of distributed deadlocks

17.6 Transaction recovery

Atomicity property of transactions

durability and failure atomicity

durability requires that objects are saved in permanent storage and will be available indefinitely

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.

for simplicity we assume that when a server is running, all of its objects are in volatile memory and all of its committed objects are in a *recovery file* in permanent storage

recovery consists of restoring the server with the latest committed versions of all of its objects from its recovery file

Recovery manager

- The task of the Recovery Manager (RM) is:
  - 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).
- media failures
  - i.e. disk failures affecting the recovery file
  - need another copy of the recovery file on an independent disk. e.g. implemented as stable storage or using mirrored disks
- we deal with recovery of 2PC separately (at the end)
  - we study logging (17.6.1) but not shadow versions (17.6.2)

Recovery - intentions lists

- Each server records an intentions list for each of its currently active transactions
  - an intentions list contains a list of the object references and the values of all the objects that are altered by a transaction
  - when a transaction commits, the intentions list is used to identify the objects affected
    - the committed version of each object is replaced by the tentative one
    - the new value is written to the server's recovery file
  - in 2PC, when a participant says it is ready to commit, its RM must record its intentions list and its objects in the recovery file
    - it will be able to commit later on even if it crashes
    - when a client has been told a transaction has committed, the recovery files of all participating servers must show that the transaction is committed,
      - even if they crash between *prepare* to commit and *commit*

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 <identifier of object>, <position in recovery file of value of object>.

Figure 17.18

Object state flattened to bytes

first entry says *prepared*

- For distributed transactions we need information relating to the 2PC as well as object values, that is:
  - transaction status (committed, prepared or aborted)
  - intentions list

Note that the objects need not be next to one another in the recovery file

Logging - a technique for the recovery file

- the recovery file represents a log of the history of all the transactions at a server
  - it includes objects, intentions lists and transaction status
  - in the order that transactions prepared, committed and aborted
  - a recent snapshot + a history of transactions after the snapshot
  - during normal operation the RM is called whenever a transaction prepares, commits or aborts
    - prepare - RM appends to recovery file all the objects in the intentions list followed by status (prepared) and the intentions list
    - commit/abort - RM appends to recovery file the corresponding status
    - assume *append* operation is atomic, if server fails only the last write will be incomplete
    - to make efficient use of disk, buffer *writes*. Note: sequential *writes* are more efficient than those to random locations
    - committed status is forced to the log - in case server crashes

Log for banking service

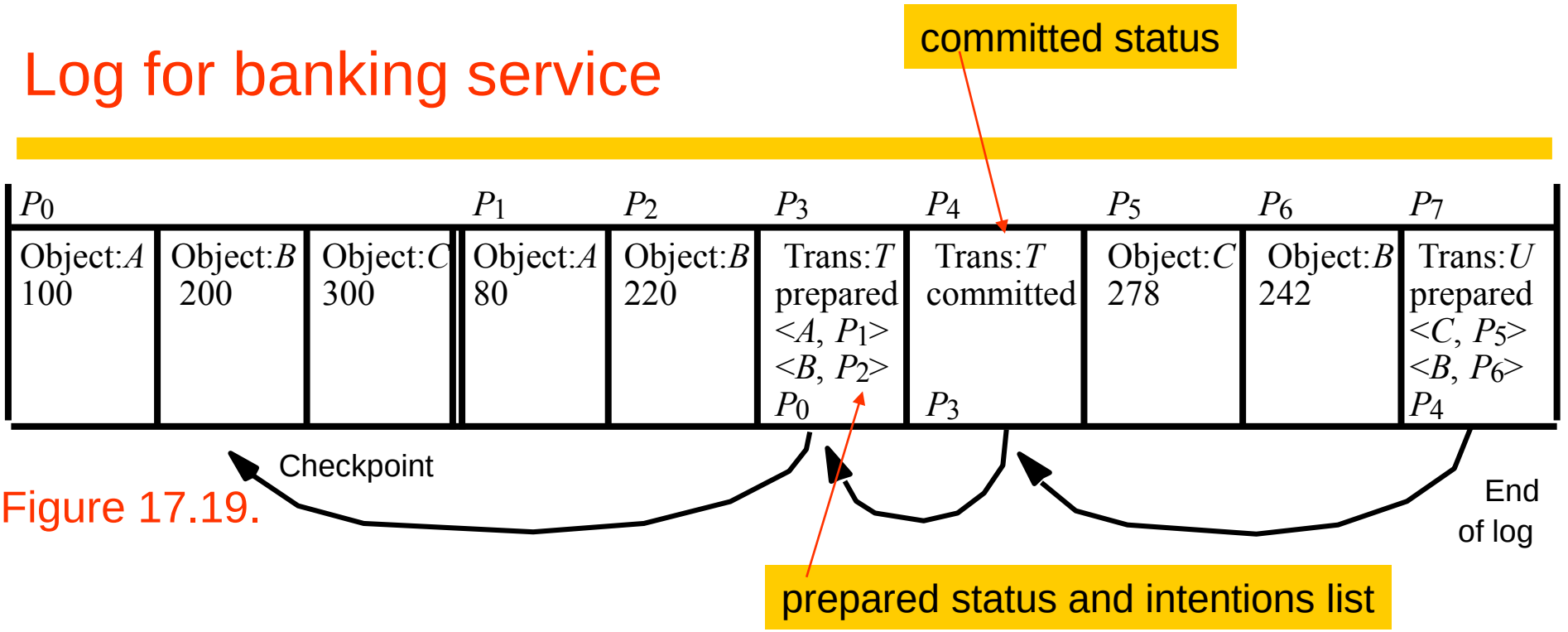


Figure 17.19.

- Logging mechanism for Fig 17.7 (there would really be other objects in log file)
  - initial balances of *A*, *B* and *C* $100, $200, $300
  - *T* sets *A* and *B* to $80 and $220. *U* sets *B* and *C* to $242 and $278
  - entries to left of line represent a snapshot (checkpoint) of values of *A*, *B* and *C* before *T* started. *T* has committed, but *U* is prepared.
  - the RM gives each object a unique identifier (*A*, *B*, *C* in diagram)
  - each status entry contains a pointer to the previous status entry, then the checkpoint can follow transactions backwards through the file

Recovery of objects - with logging

- When a server is replaced after a crash
  - it first sets default initial values for its objects
  - and then hands over to its recovery manager.
- The RM restores the server's objects to include
  - all the effects of all the committed transactions in the correct order and
  - none of the effects of incomplete or aborted transactions
  - it 'reads the recovery file backwards' (by following the pointers)
    - restores values of objects with values from committed transactions
    - continuing until all of the objects have been restored
  - if it started at the beginning, there would generally be more work to do
  - to recover the effects of a transaction use the intentions list to find the value of the objects
    - e.g. look at previous slide (assuming the server crashed before T committed)
  - the recovery procedure must be idempotent

Logging - reorganising the recovery file

- RM is responsible for reorganizing its recovery file
  - so as to make the process of recovery faster and
  - to reduce its use of space
- checkpointing
  - the process of writing the following to a new recovery file
    - the current committed values of a server's objects,
    - transaction status entries and intentions lists of transactions that have not yet been fully resolved
    - including information related to the two-phase commit protocol
  - checkpointing makes recovery faster and saves disk space
    - done after recovery and from time to time
    - can use old recovery file until new one is ready, add a 'mark' to old file
    - do as above and then copy items after the mark to new recovery file
    - replace old recovery file by new recovery file

Summary of transaction recovery

- Transaction-based applications have strong requirements for the long life and integrity of the information stored.
- Transactions are made durable by performing checkpoints and logging in a recovery file, which is used for recovery when a server is replaced after a crash.
- Users of a transaction service would experience some delay during recovery.
- It is assumed that the servers of distributed transactions exhibit crash failures and run in an asynchronous system,
  - but they can reach consensus about the outcome of transactions because crashed servers are replaced with new processes that can acquire all the relevant information from permanent storage or from other servers

Assignment3 (Chapter 17)

- 17.1
- 17.3