COMP28112 Lecture 13

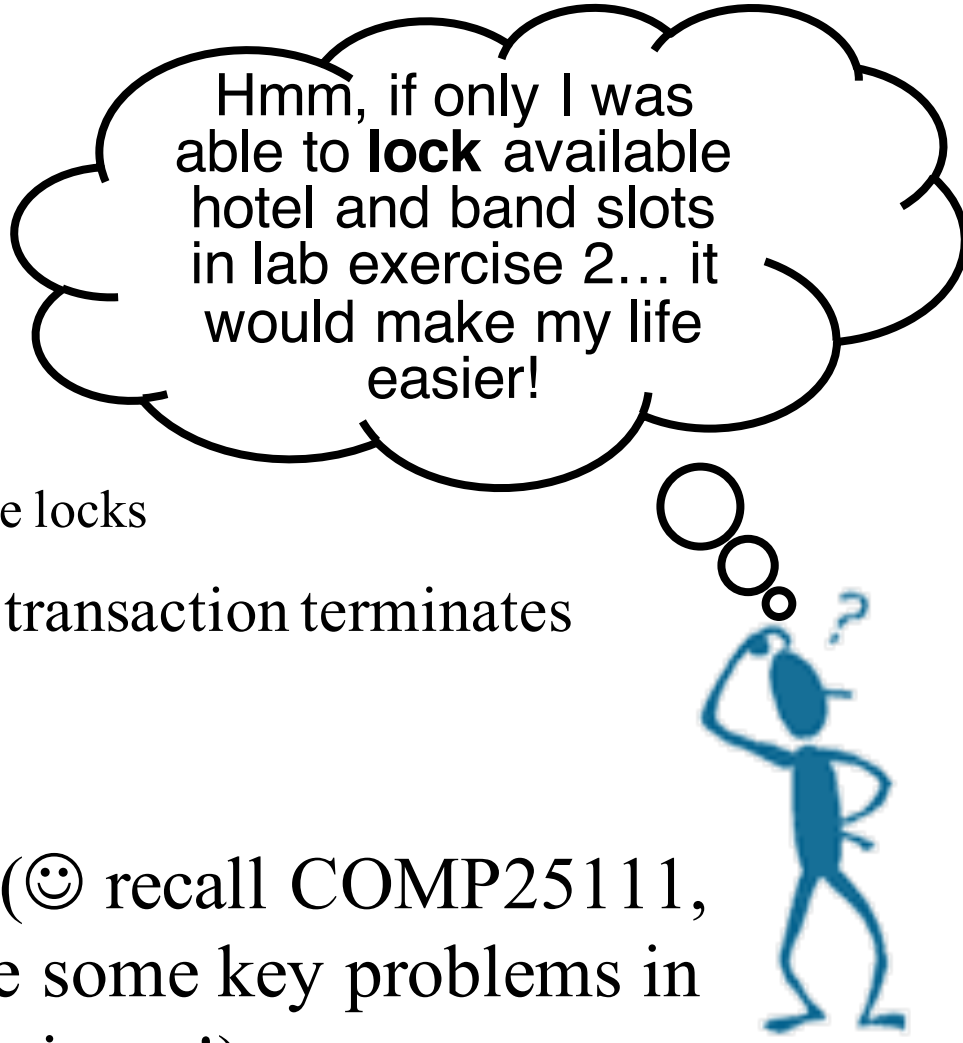
Distributed Transactions

Transactions (recap from last time)

- A set of operations that is either fully committed or aborted as a whole (i.e., the set is treated as an atomic operation); if aborted, no operation in the set is executed:
  - This guarantees that data will not be left in a corrupted state as a result of (unforeseen) server crashes or other concurrent transactions (cf. client-server example with bank transfer from last lecture).
- Need to provide:
  - Concurrency control
  - Recovery mechanisms

Concurrency Control

- Two-phase locking
  - "**Acquire locks**" phase
    - Get a read lock before reading
    - Get a write lock before writing
    - Read locks conflict with write locks
    - Write locks conflict with read and write locks
  - "**Release locks**" phase when the transaction terminates (commit or abort)



What does all this remind you of? (☺ recall COMP25111, thread synchronization: there are some key problems in core Computer Science!)

Using locks…

…concurrent transactions are serialised.

Now, think about part 3 of lab exercise 2. Assume that the server allowed you to submit (locked) transactions:

```
Begin transaction
      get hotel_free_slots;
      get band_free_slots;
      slot=find_earliest_common_slot;
      book_hotel(slot);
      book_band(slot);
End transaction
```

What would the problem be?

Disadvantages of locks

- Reduce significantly the potential for concurrency even though they are really needed in extreme cases.
- May result in a deadlock!
- Improvements:
  - Optimistic concurrency control: transactions are allowed to proceed as normal and everything is checked at the 'commit transaction' phase. If there is a problem, transactions are then aborted.
  - Timestamp ordering: each operation in a transaction is validated when it is carried out. If it cannot be validated, the transaction is aborted immediately.

Recovery

When a transaction needs to be aborted:

- **Backward recovery**: bring the system from its present (erroneous) state into a previously correct state. To do so, the system's state from time to time is recorded; each time this happens, a ***checkpoint*** is said to be made.
- **Forward recovery**: try to bring the system to a correct new state from which it can continue to execute. It must know in advance which errors may occur (so that it is possible to correct them!)

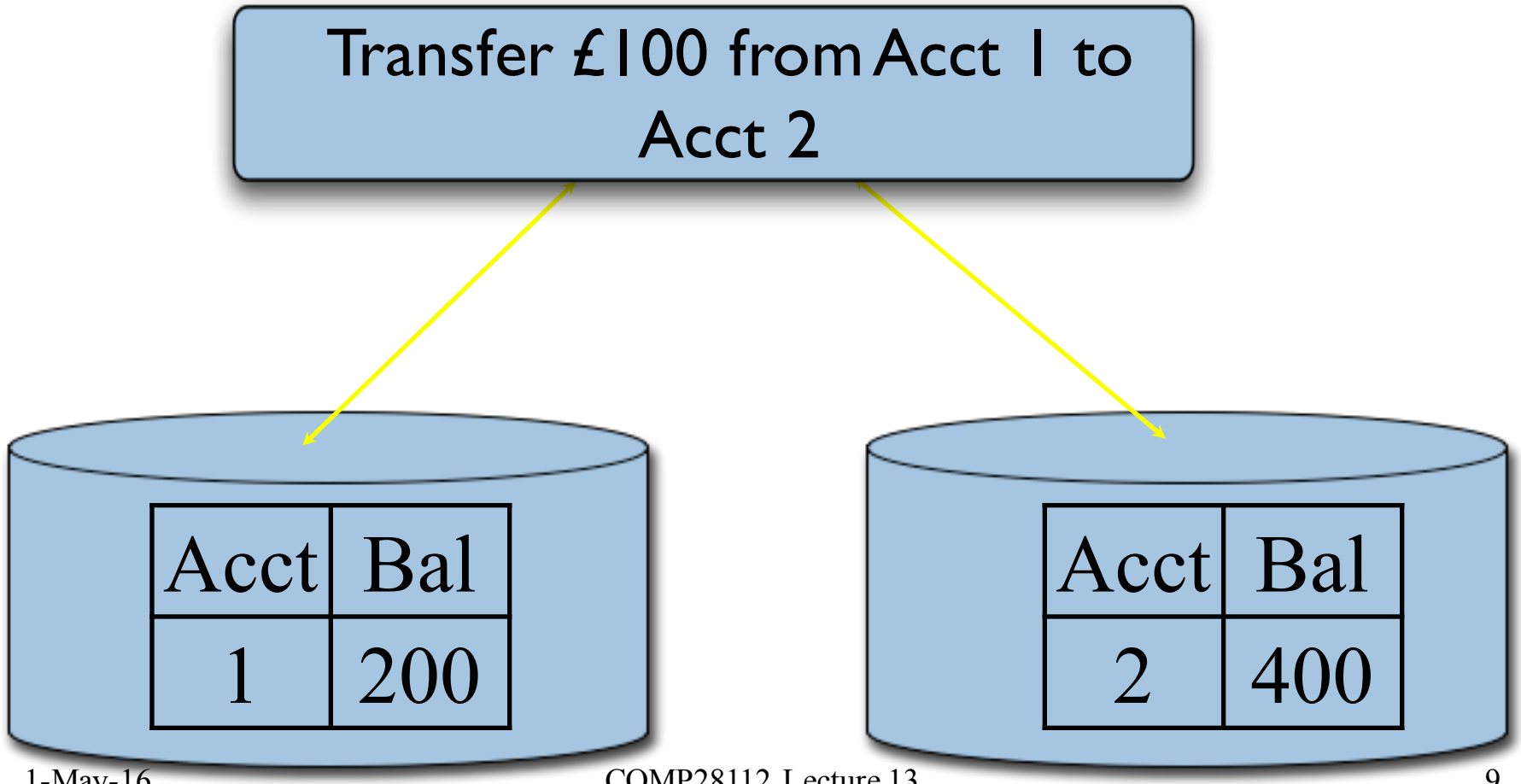
Recall our Simple Application Example
(a client communicating with a remote server)

Transfer £100 from account 1 to account 2

```
x = read_balance(1);
y = read_balance(2);
write_balance(1, x - 100);
write_balance(2, y + 100);
```

What if the accounts are held in two databases?

Transfer Funds Across Databases

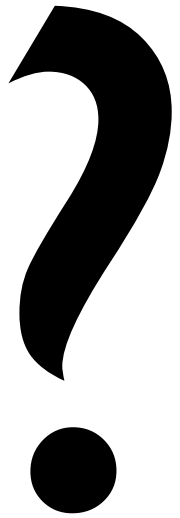


The Joys of Distributed Computing

- More problems to worry about:
  - One or both databases can fail at anytime or be slow to respond
  - Slow or faulty network
  - How does your distributed application handle these failures?

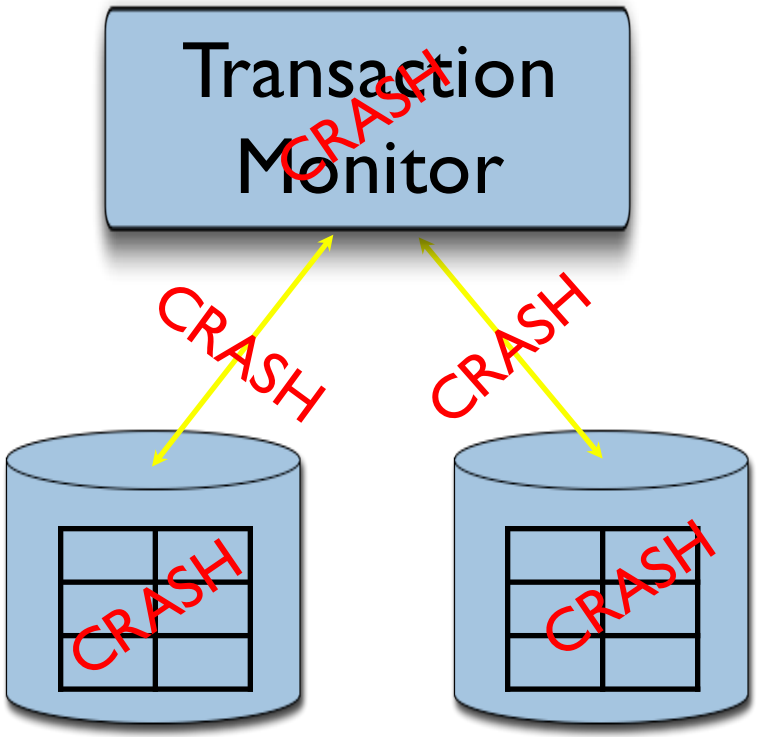
Distributed transactions to the rescue

(transactions where more than one server is involved)



Distributed Transactions

```
begin_transaction
        x = read_balance(1);
        y = read_balance(2);
        write_balance(1, x - 100);
        write_balance(2, y + 100);
commit;
```



All-or-Nothing

- ALWAYS

  either

  — ALL databases commit the transaction

  or

  — ALL databases abort the transaction

- Example of a consensus problem

  — Everyone MUST agree on a single outcome

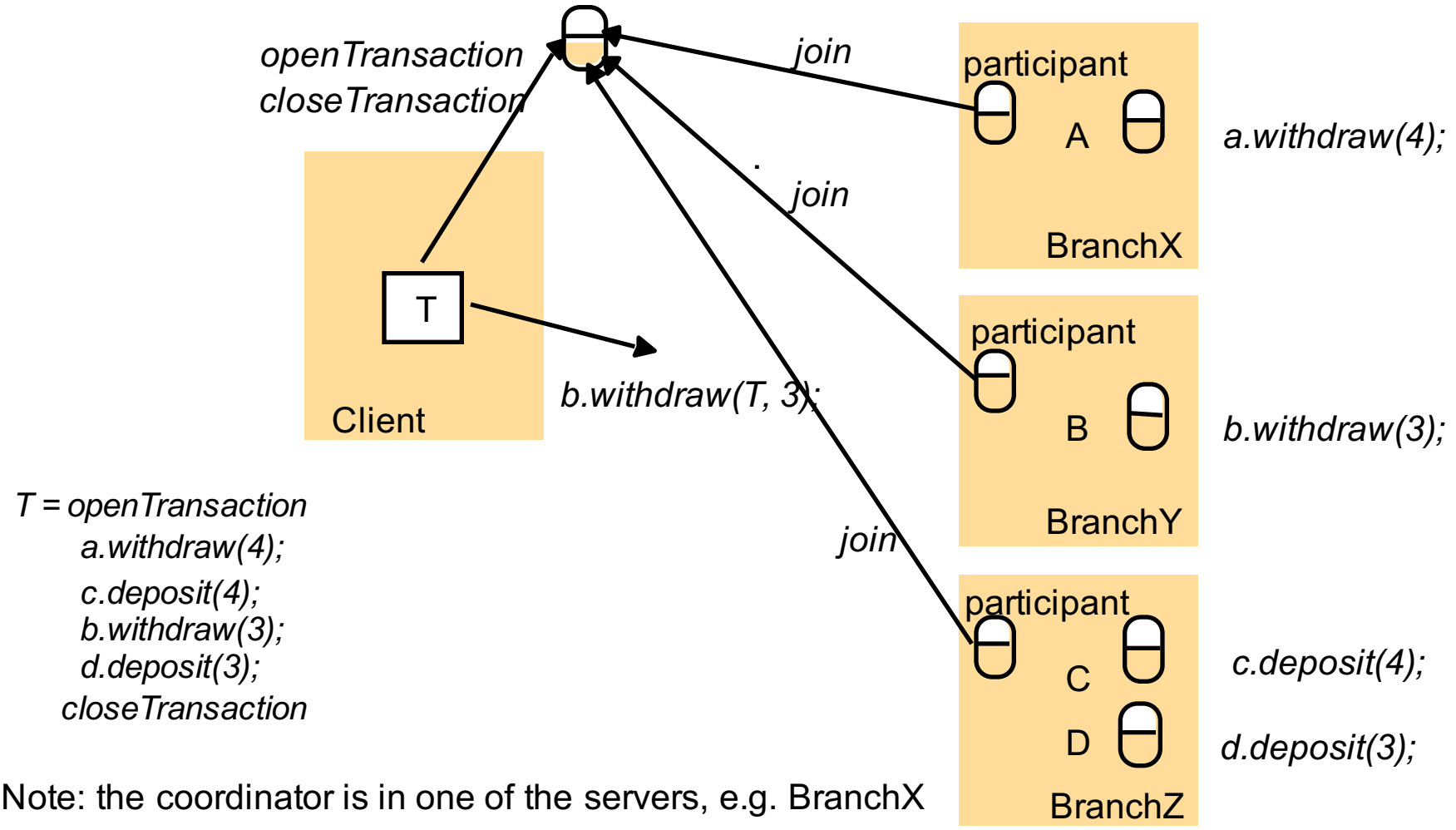
- More generally:

  — The **distributed commit** problem: an operation is performed by each member of a process group or none at all.

What protocol do we need to support distributed transactions?

(**protocol** = standard rules regarding the messages exchanged between the servers)

- Step 1: A coordinator is chosen (figure 14.3 in CDK)

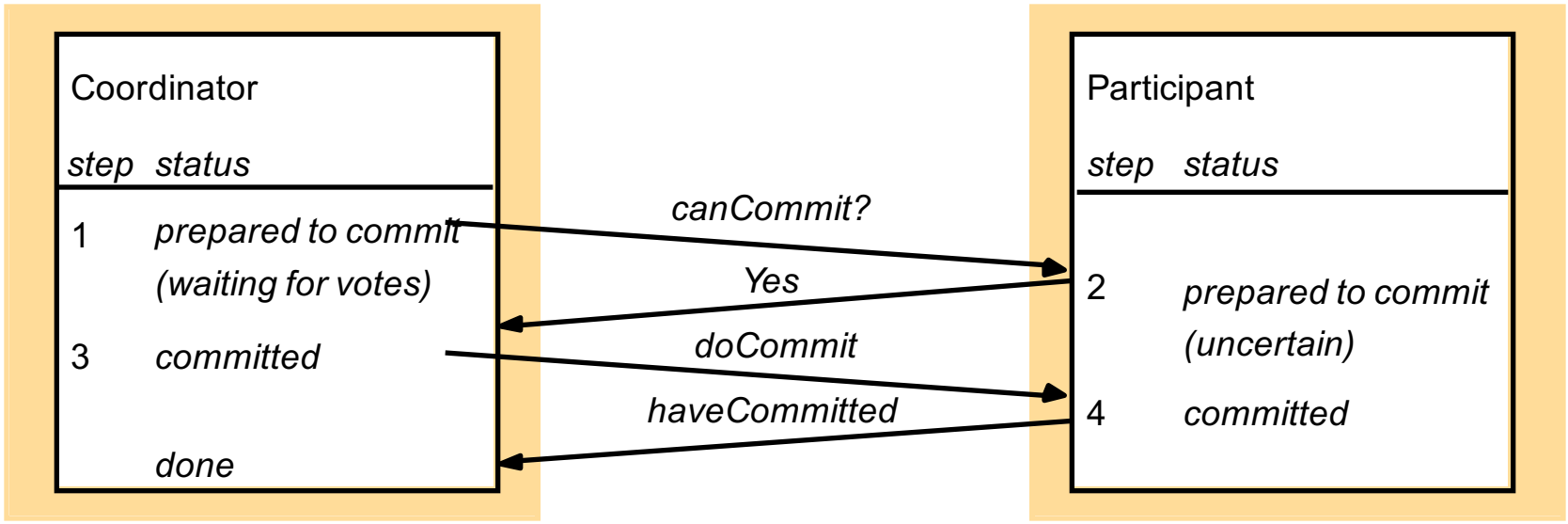


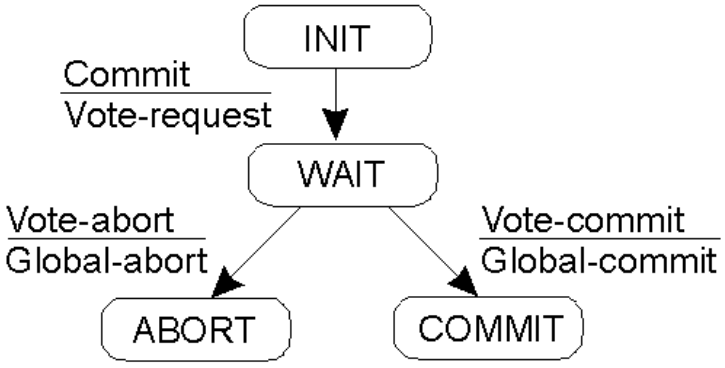
Note: the coordinator is in one of the servers, e.g. BranchX

One-phase atomic commit

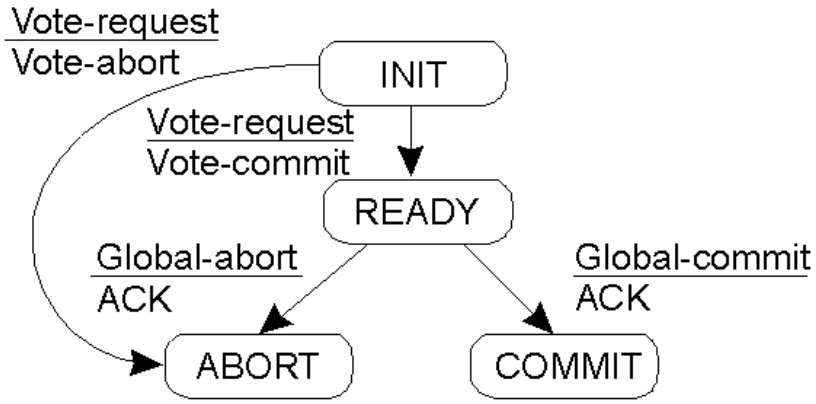
- Client tells the coordinator to commit or abort a transaction
- The coordinator communicates the commit (or abort) to all participants.
- (obvious) problem: if one of the participants cannot actually perform the operation it cannot tell the coordinator.

Two-Phase Commit (see Fig. 14.6 in CDK and Fig.8.18 in TvS)





(a) coordinator



(b) participant

Drawbacks of Two-Phase-Commit

- What if the coordinator has failed?
  - Three-phase commit protocol
  - Multicast to all other participants
- Participants need to trust the coordinator
- Transactions should be short in duration
- Distributed deadlocks may occur!

Conclusion

- A distributed transaction is a transaction whose activity involves several different servers.
- Nested transactions may be used for additional concurrency.
- Atomicity requires that all servers participating in a transaction either all commit it or all abort it.
- Reading: Coulouris4, Chapter 14; Coulouris5, Chapter 17; Tanenbaum, Sections 8.4-8.6 (too detailed in parts and not transaction-focused)

Some additional information on deadlocks

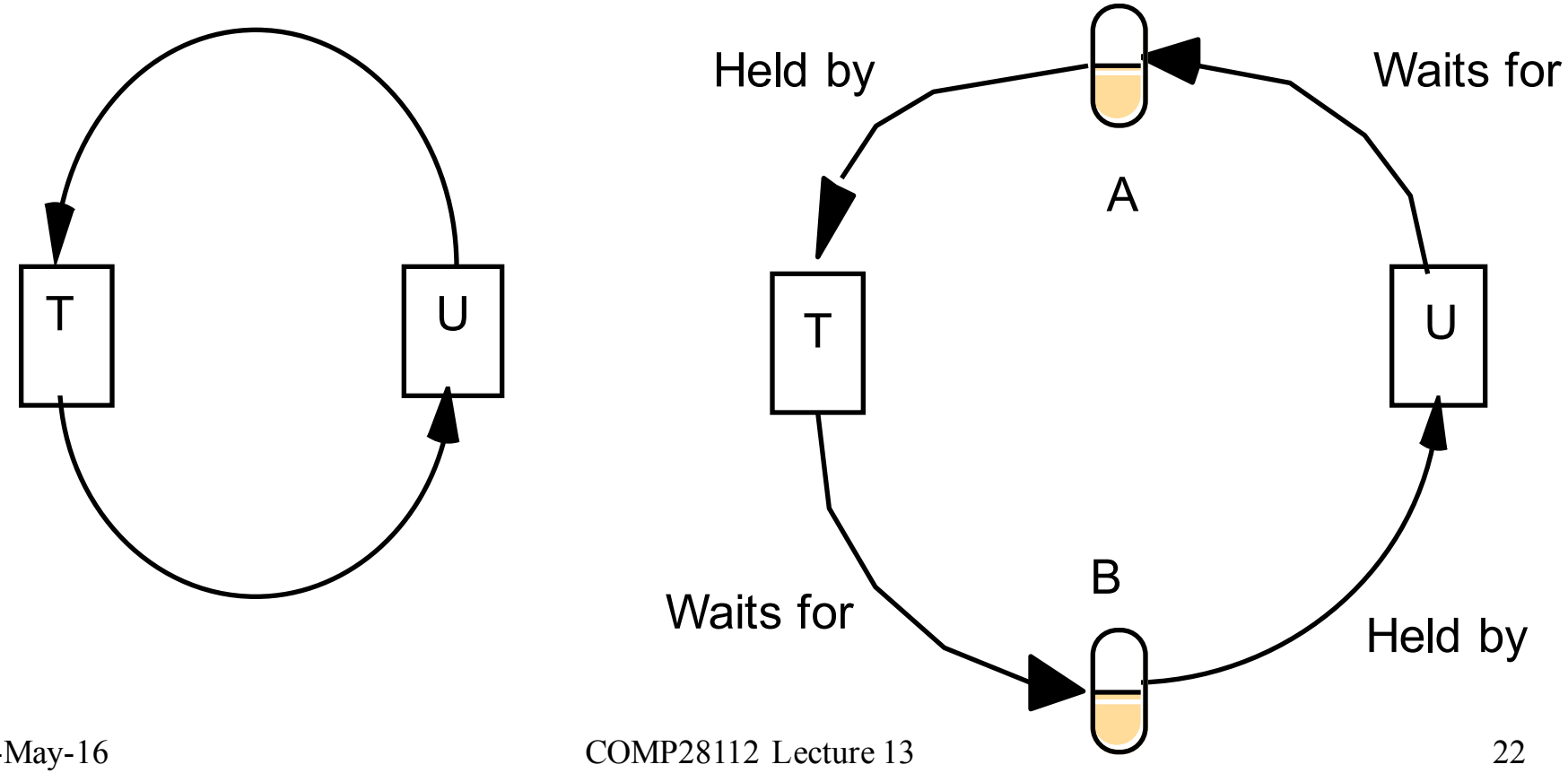
Deadlocks

- If we use locking to implement concurrency control in transactions, we can get deadlocks (even within a single server)
- So we need to discuss:
  - Deadlock detection within a single system
  - Distributed deadlock

Deadlock detection

- A deadlock occurs when there is a cycle in the *wait-for* graph of transactions for locks
- There may be more than one
- Resolve the deadlock by aborting one of the transactions ….
- E.g. the youngest, or the one involved in more than one cycle, or can even use "priority" ….

CDK Figure 13.20
A cycle in a wait-for graph



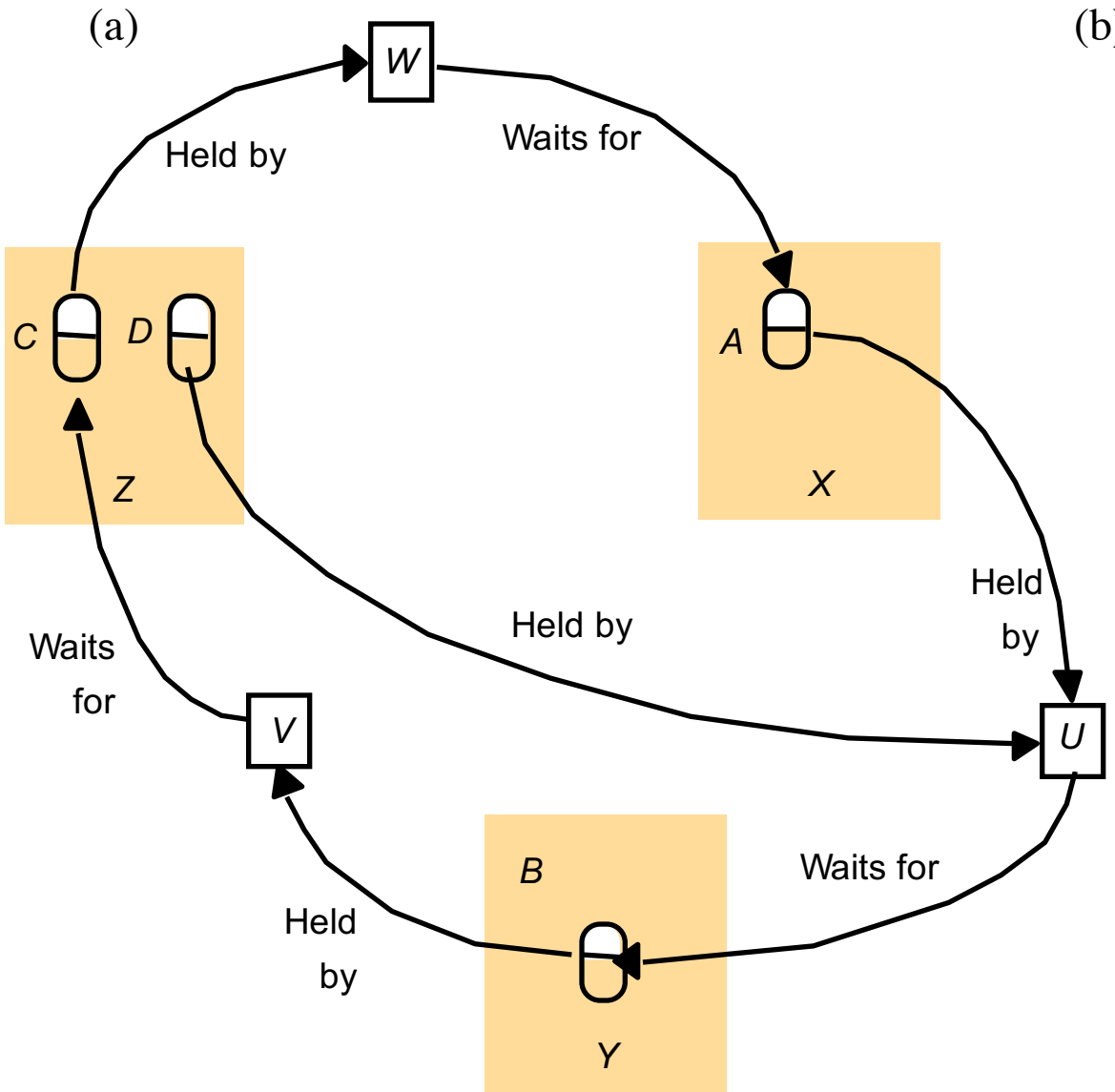
Distributed Deadlock

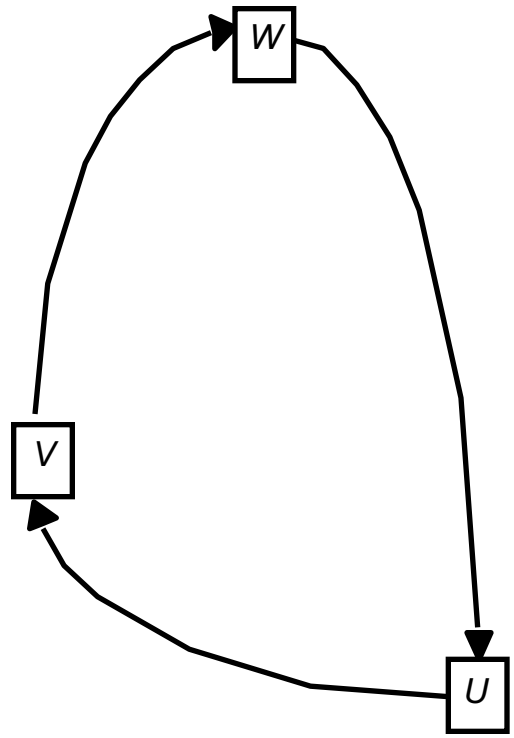
- Within a single server, allocating and releasing locks can be done so as to maintain a wait-for graph which can be periodically checked.
- With distributed transactions locks are held in different servers – and the loop in the entire wait-for graph will not be apparent to any one server

Distributed deadlock (2)

- 1 solution is to have a coordinator to which each server forwards its wait-for graph
- But centralised coordination is not ideal in a distributed system
- Have problems of *phantom deadlocks*

CDK Figure 14.14
Distributed deadlock





Phantom deadlocks

- Information gathered at central coordinator is likely to be out-of-date
- So a transaction may have released a lock (by aborting) but the global wait-for graph shows it as still holding it!
- Thus a deadlock might be detected which never existed!

An Alternative: Edge Chasing

- An alternative to a centralised deadlock checker is “Edge Chasing” or “Path Pushing”.
- Send a message (probe) containing “T->U” to the server at which U is blocked
- This message gets forwarded (and added to) if the lock U is waiting for is held by V …
- If a transaction repeats in the message, the deadlock is detected …
- E.g, T -> U -> V -> W -> X -> T