Distributed Transactions

CDK 15

MDS E2014
Søren Debois

Meta

ME2

ME2

- Generally good!
- Time consumption low'ish:
  avg 3.5hr, min 0.1hr, max 7hr
- Beware! Very few did optimistic concurrency.
- A few groups must re-submit. Holger and Frederik will contact you.
- (Still a few submissions missing.)

Examination submission

Examination submission

- **Individual** submission.
- Deadline December 14.
- Submit the 3 mini-projects and 2 mandatory exercise sets you contributed to.
- Submit a .zip file containing sub-directories for mini-projects “mp1”, “mp2”, “mp3”; and for mandatory exercises “me1”, “me2”.
- “Exam assignment” to appear in learnit shortly.

Examination submission

- Q: What if I no longer have my submissions?
- A: Try learnit. Then try group members. Then Frederik & Holger.

Examination submission

- Examination will be guided by your submission.
- Good parts we *likely* won't ask so much about.
- Bad parts we *likely* will ask a lot about.
- We will *likely* also ask about curriculum not touched by the submission.
- NB! We can in general ask wherever in the curriculum we think answers will be most helpful to determine a grade.

  That might emphasise the submissions, and it might not.

Examination

Examination form

- Oral, no preparation, no aids.
- Free-form questions in the entire curriculum & mandatory submissions.
- You’ll be invited to pick a starting topic.
- We’ll leave it quickly.

How to prepare

- Make sure you’ve solved every exercise posted on the learnit page.
- Make sure you know all about your mini-projects.
- Then read the book. Suggested reading order: 2 (failure models only), 4, 5, 6, 10, 11, 14–17, 2 (rest), 3, 9, 1.

Summary

Coordination & Agreement

- Motivation: Fundamentals of agreeing.
- Distributed Mutual Exclusion
- Elections
- Consensus

Transactions

- Transactions
- Serially equivalent transactions
- Locking implementation
- Optimistic implementation

Figure 16.5
The lost update problem

Transaction *T* :		**Transaction *U*:**	
balance = b.getBalance(); *b.setBalance(balance*1.1);* *a.withdraw(balance/10)*		*balance = b.getBalance();* *b.setBalance(balance*1.1);* *c.withdraw(balance/10)*	
balance = b.getBalance();	$200		
		balance = b.getBalance();	$200
		*b.setBalance(balance*1.1);*	$220
*b.setBalance(balance*1.1);*	$220		
a.withdraw(balance/10)	$80		
		c.withdraw(balance/10)	$280

Figure 16.6
The inconsistent retrievals problem

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100) *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	\$100		
		total = a.getBalance()	\$100
		total = total+b.getBalance()	\$300
		total = total+c.getBalance()	
b.deposit(100)	\$300	:	

Figure 16.8
A serially equivalent interleaving of *V* and *W*

Transaction *V*:		**Transaction *W*:**	
a.withdraw(100); *b.deposit(100)*		*aBranch.branchTotal()*	
a.withdraw(100);	\$100		
b.deposit(100)	\$300		
		total = a.getBalance()	\$100
		total = total+b.getBalance()	\$400
		total = total+c.getBalance()	
		...	

Figure 16.10
A non-serially equivalent interleaving of operations of transactions *T* and *U*

Transaction *T*:	**Transaction *U*:**
$x = read(i)$	
$write(i, 10)$	
	$y = read(j)$
	$write(j, 30)$
$write(j, 20)$	
	$z = read\ (i)$

Distributed Transactions

Plan

- Recap of Consensus & Agreement, Transactions.
- Motivation/properties (ACID, failure model)
- Commit protocols
- Concurrency control
- Distributed deadlock detection

Like transactions, but distributed

- A distributed transaction accesses objects on different servers.
- The distributed transaction is either *successful* or *aborted*.
- A distributed transaction must have the ACID properties.

ACID

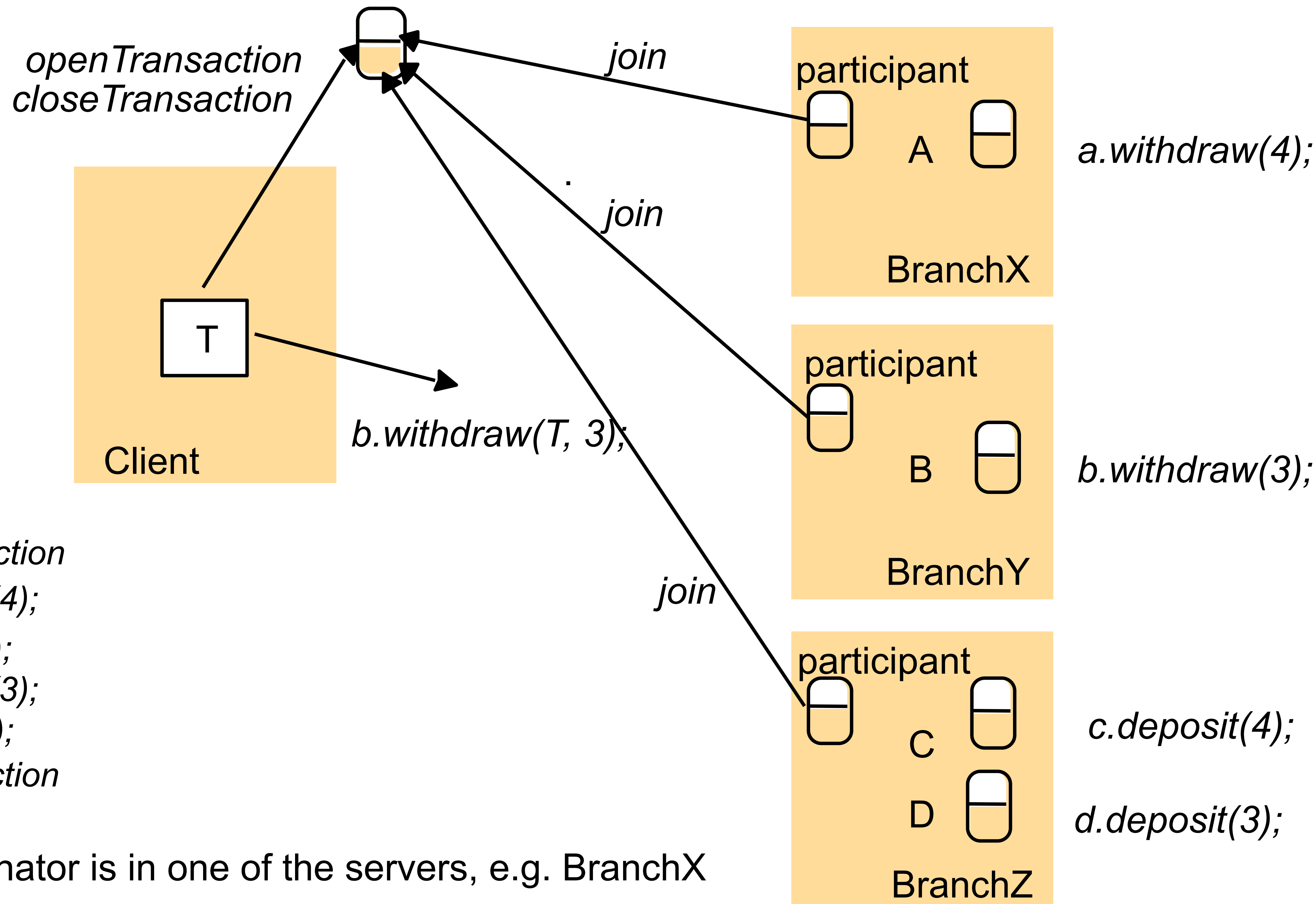
- Atomicity
  (Commits entirely or fails entirely)
- Consistency
- Isolation
  (Serial equivalence)
- Durability

Failure model

- Asynchronous system
- Crash failures
- Lost messages
- No byzantine failures, no corrupted messages
- *Failed processes are replaced, recover state.*

Commit protocols

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

Discuss:
Design a protocol for the commit operation.

One-phase commit

- Coordinator tells servers to commit, wait for acknowledgements.
- Consequence: Servers can’t unilaterally abort.

Two phase commit

- Phase 1. Servers vote on whether to commit.
(“Coordination & Agreement”: Agree on a value.)

- Phase 2. Carry out the decision.
(Beware crash- and channel failures.)

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):

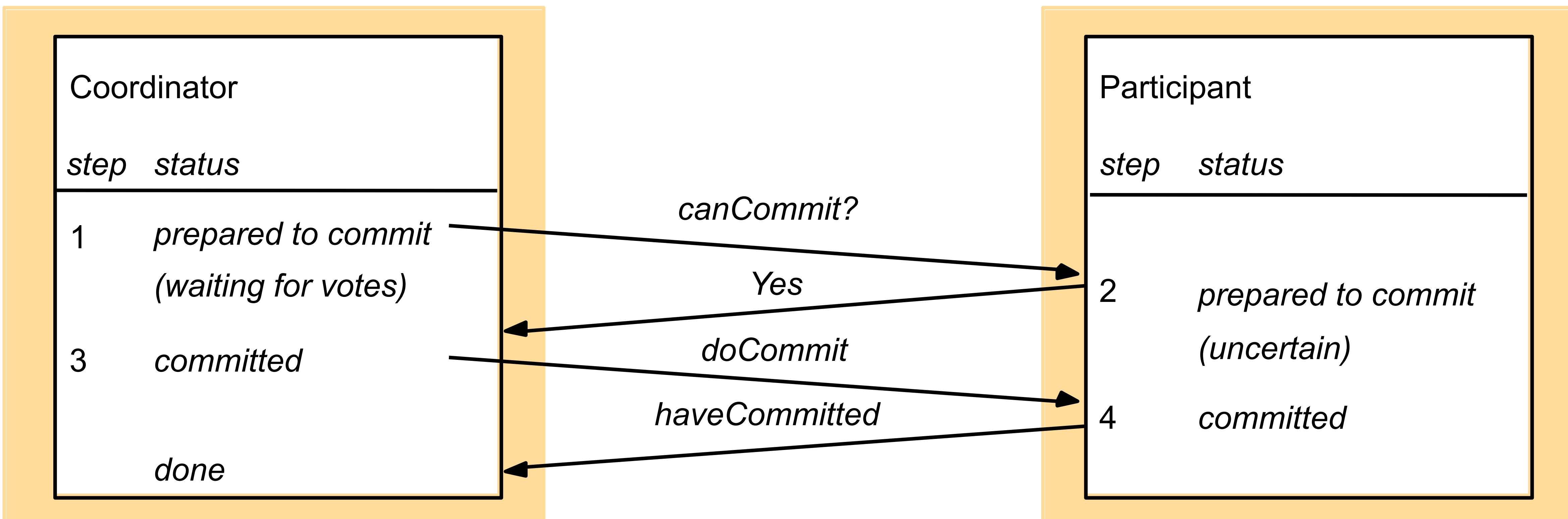
3. The coordinator collects the votes (including its own).

(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.

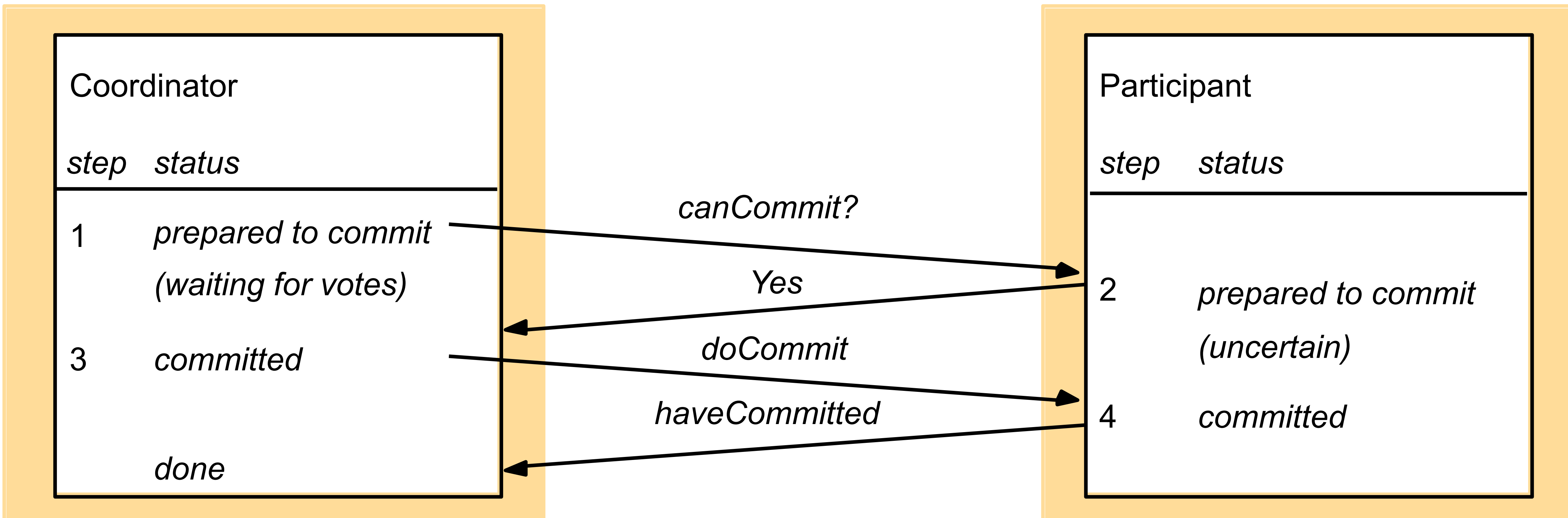
Figure 17.6
Communication in two-phase commit protocol



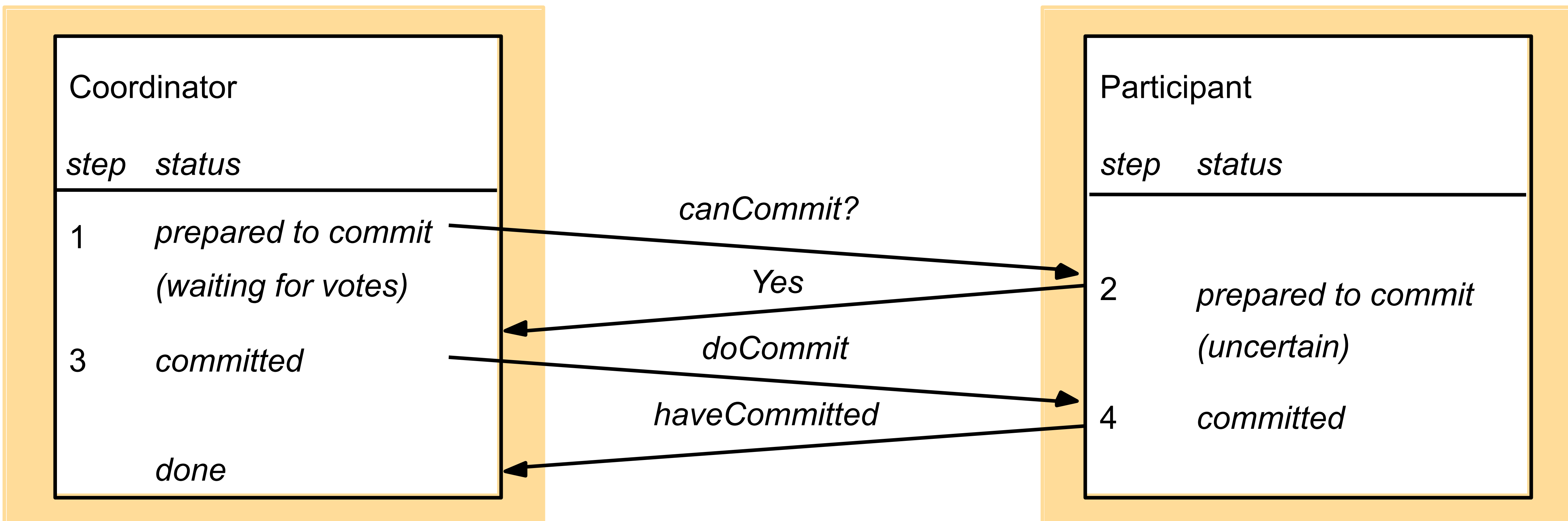
We cheat:

- We’re solving consensus in a an asynchronous system. That’s impossible!
- We cheat:
  Failed processes are replaced, recover state.

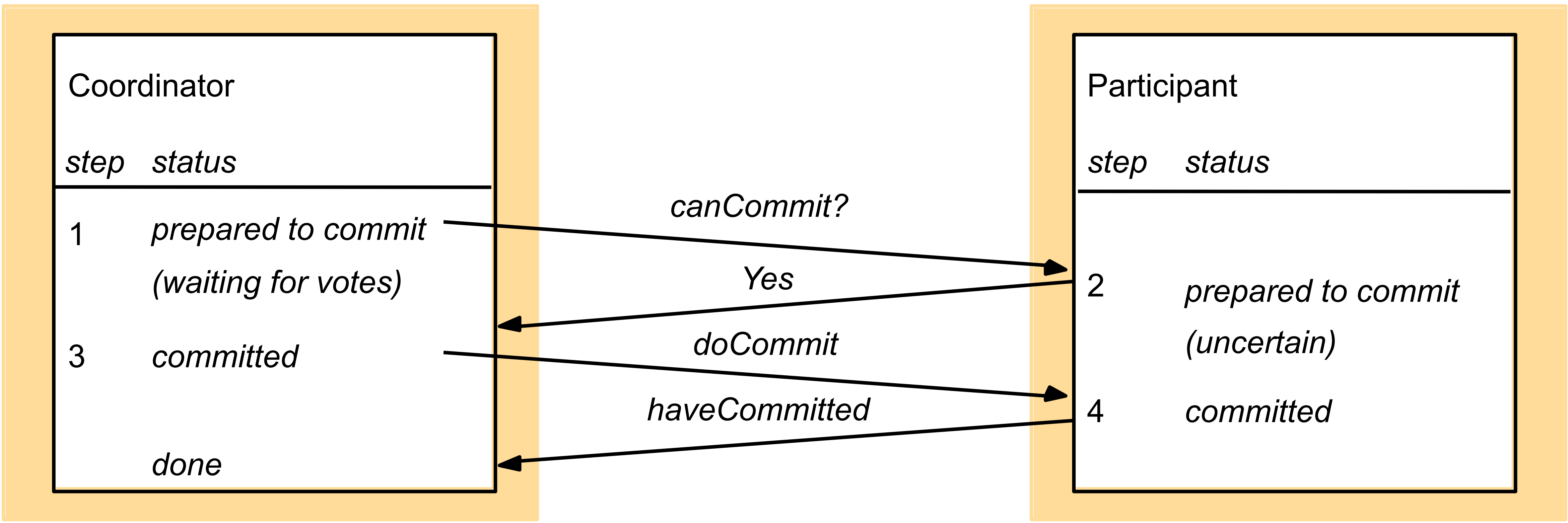
Figure 17.6
Communication in two-phase commit protocol



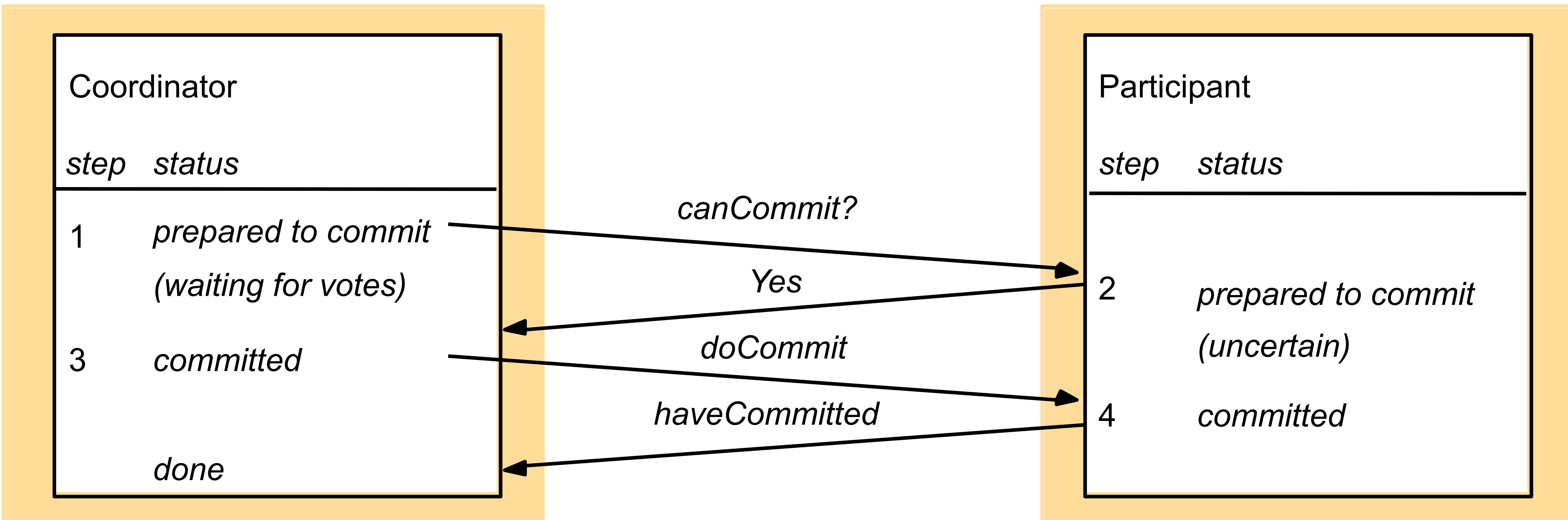
Performance



- N * (canCommit? + Yes/No)
- N * doCommit
- = 3N (messages)
  = 3 rounds (time)

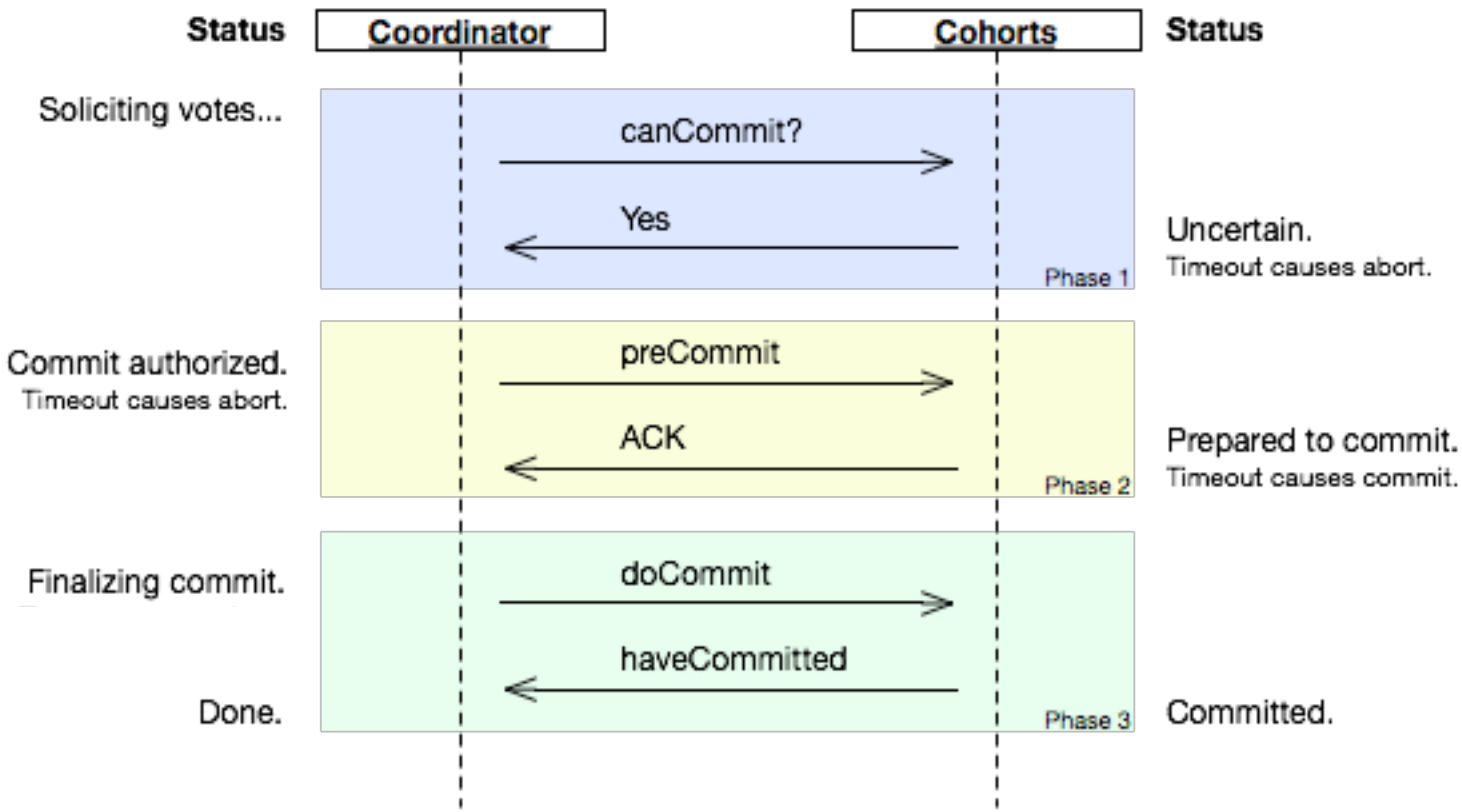


What if the coordinator fails while waiting for votes?



What if the coordinator and a server fails?

Three phase commit



Credit: Wikipedia

http://en.wikipedia.org/wiki/File:Three-phase_commit_diagram.png

Summary

- Motivation: How do we agree to commit/abort?
- One-phase commit
- Two-phase commit
- Three-phase commit

Concurrency control

Serial equivalence

- Same, but distributed:
- Transactions T, U happens as if either T first **at all servers,** or vice versa.

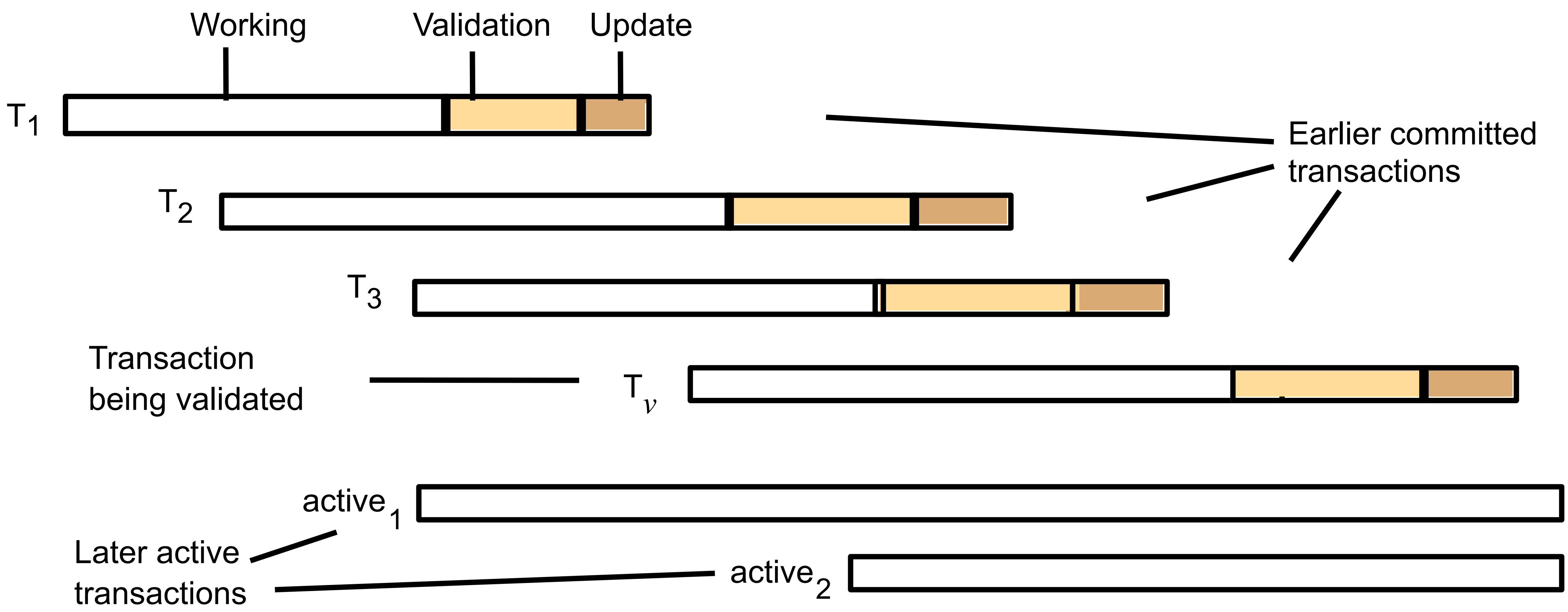
Locking

- Like with non-distributed transactions (Strict two-phase locking.)
- Distributed deadlocks

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*

Opimistic control

Figure 16.28
Validation of transactions



Opimistic control, timestamp ordering

- Coordinator issues globally unique timestamp.
- Serial equivalence?
- Local validation. May happen in distinct orders!

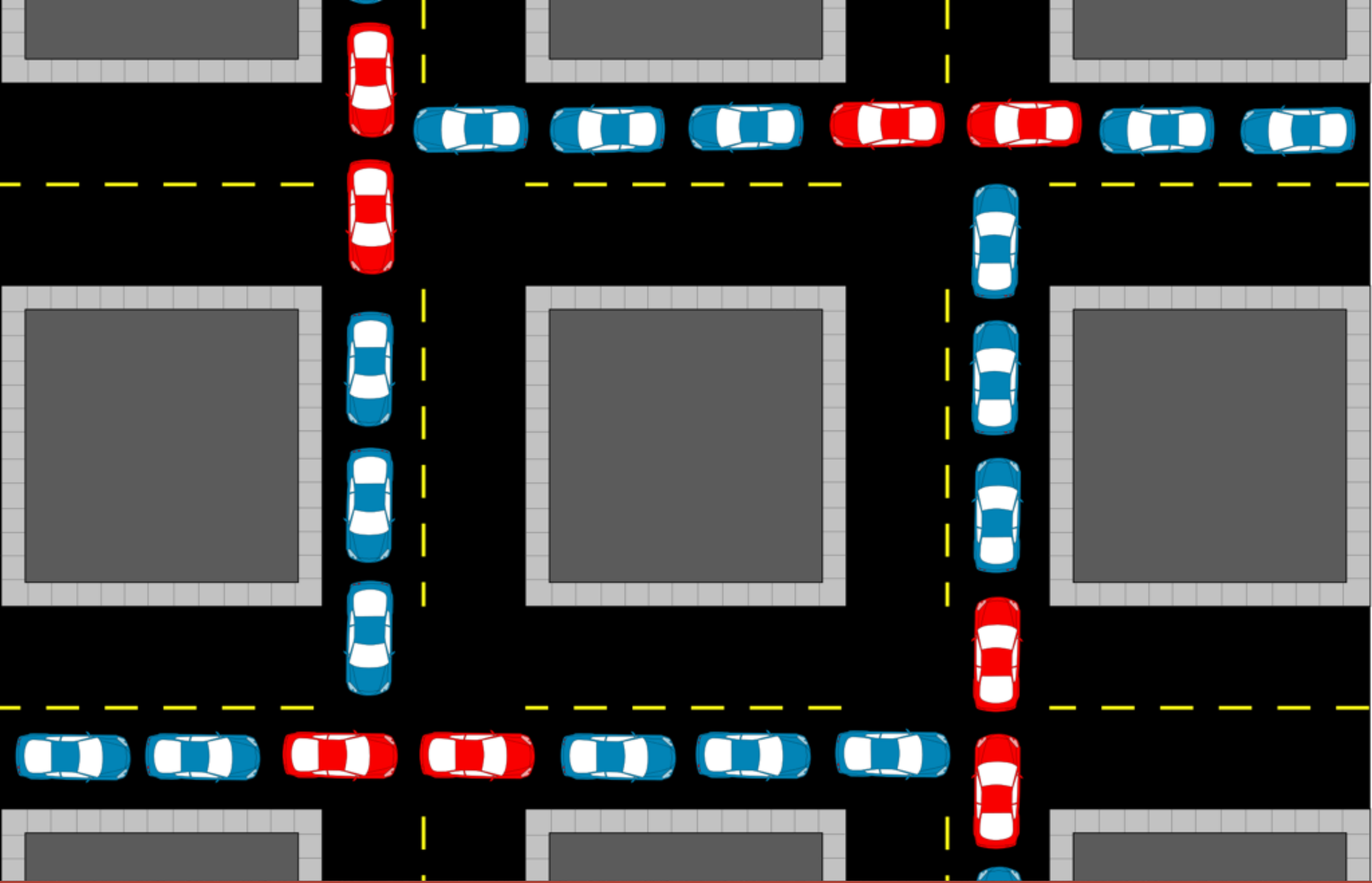
T		*U*	
read(A)	at *X*	*read(B)*	at *Y*
write(A)		*write(B)*	
read(B)	at *Y*	*read(A)*	at *X*
write(B)		*write(A)*	

Parallel validation

- Check also write set against earlier overlapping transactions in backwards validation.
- Servers still need to coordinate serialisation of validations.

Summary

- Concurrency control by strict two-phase locking, needs deadlock detection
- Concurrency control by optimism,
needs coordination of serialisation of validations.



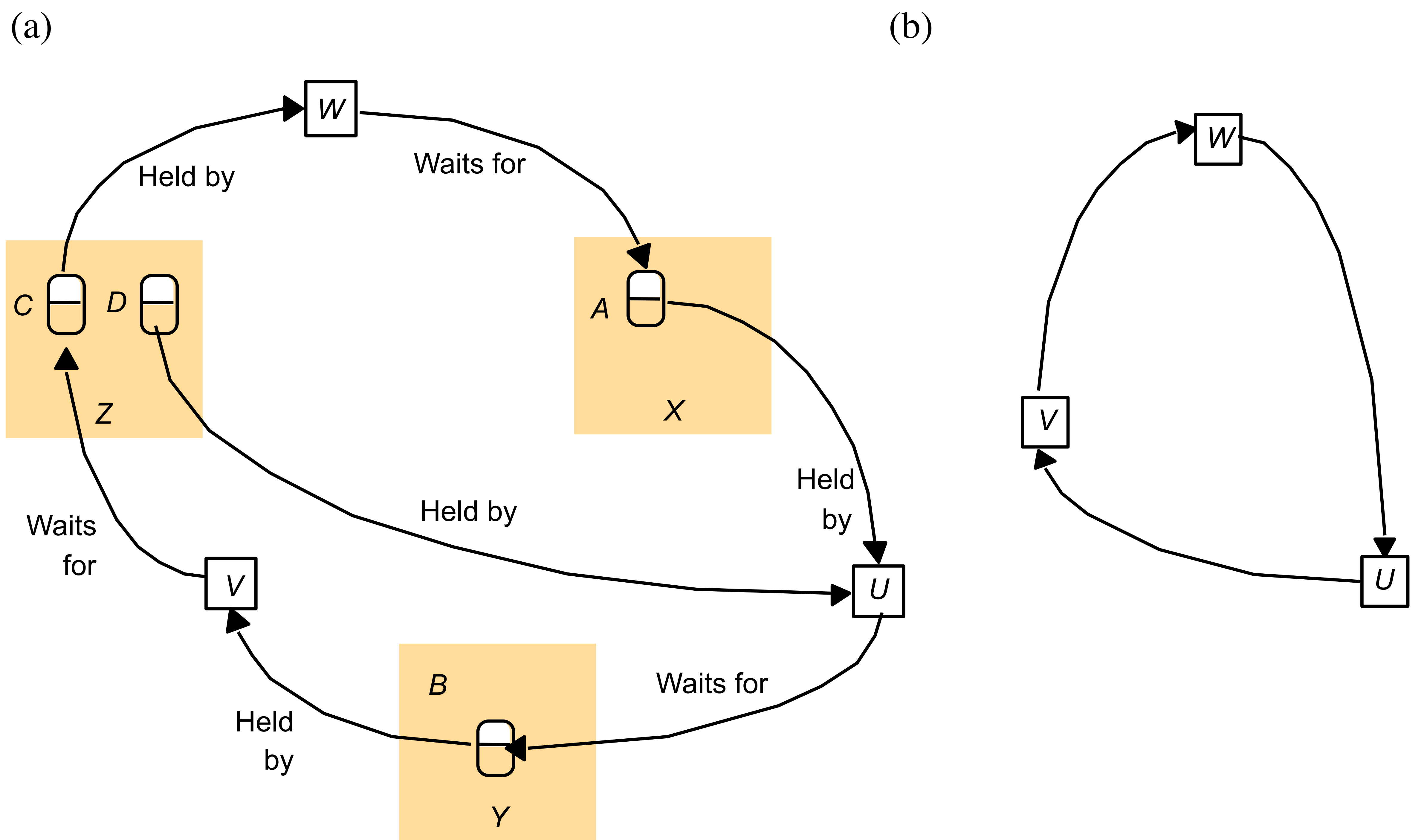
Distributed deadlock detection

Find the distributed waits-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*

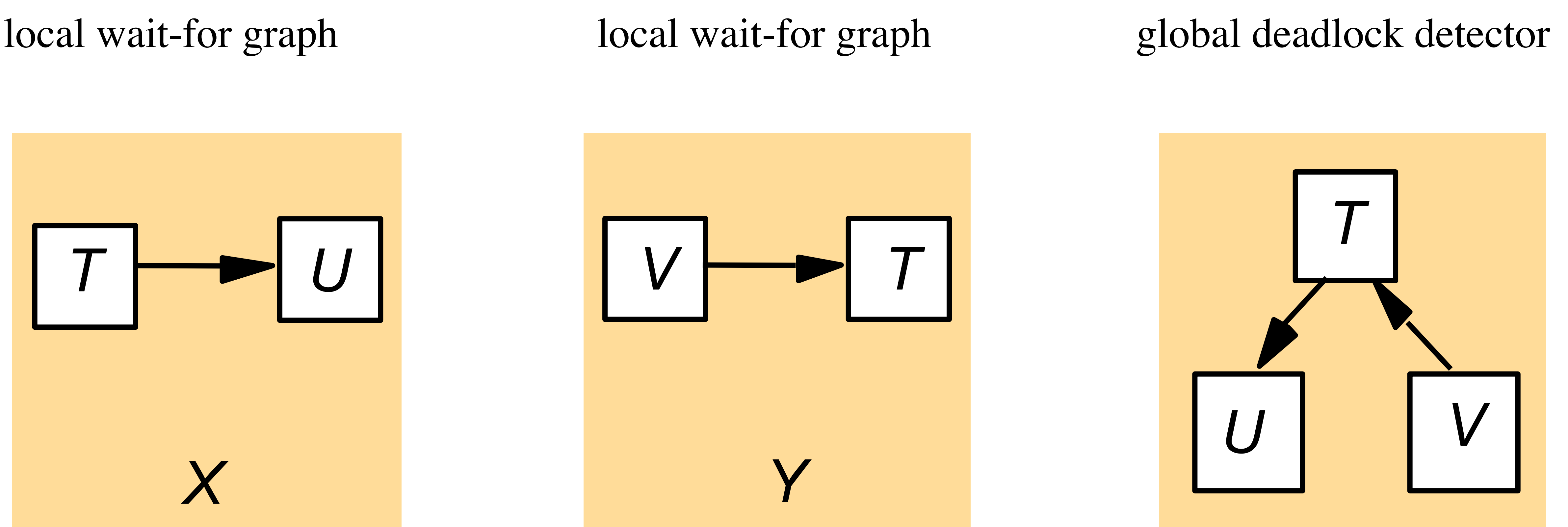
Figure 17.13
Distributed deadlock



Phantom deadlocks

- Because of concurrency, a lock may have been released when we think it’s still held
- so, no deadlock, even though we think so.

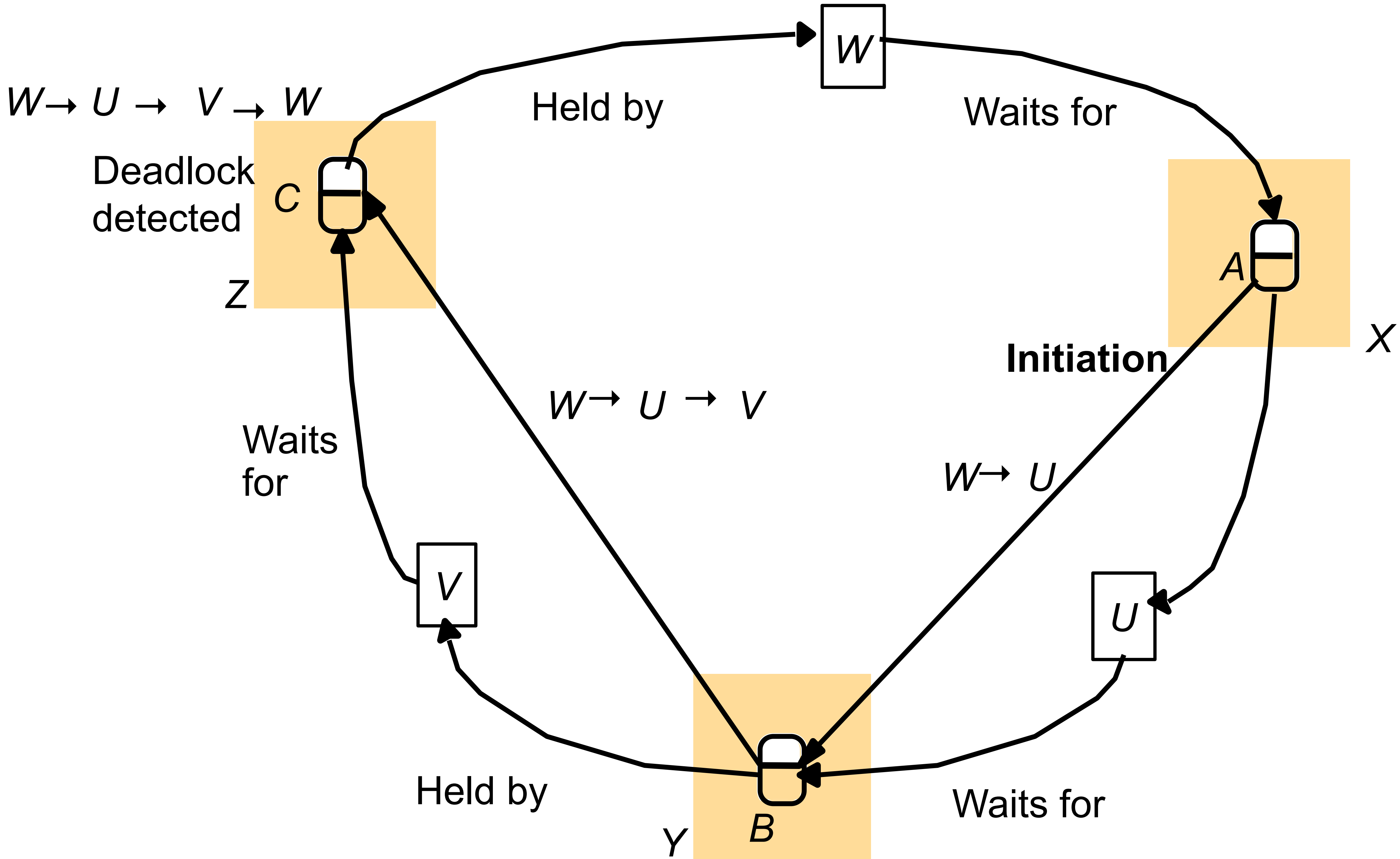
Figure 17.14
Local and global wait-for graphs



Edge-chasing

- Distributed depth-first search in the waits-for graph.
- Servers send out *probes* when
  a transaction requests a
  lock held by an already waiting transaction.
- 3 stages: initiaion, detection, resolution

Figure 17.15
Probes transmitted to detect deadlock



Performance of edge-chasing

- Assume cycle of N transactions.
- N-1 probes.
- N-1 questions for coordinator
- = 2(N-1) messages
- In practice, cycles tend to be very short.

What about concurrent detection?

- Coordinator assigns globally unique *transaction priorities*.
- Abort the least transaction
- Then it doesn't matter how you found the cycle.
  (i.e., 1 -> 2 -> 3 -> 1 breaks 1,
  2 -> 3 -> 1 -> 2 also breaks 1.)

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*

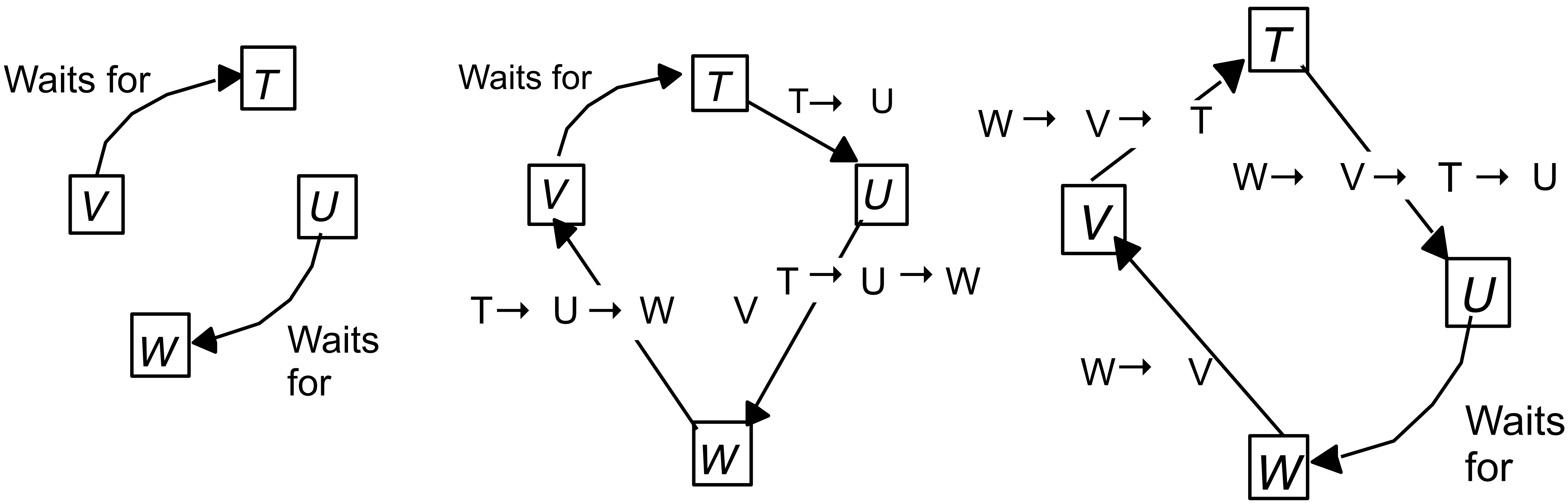
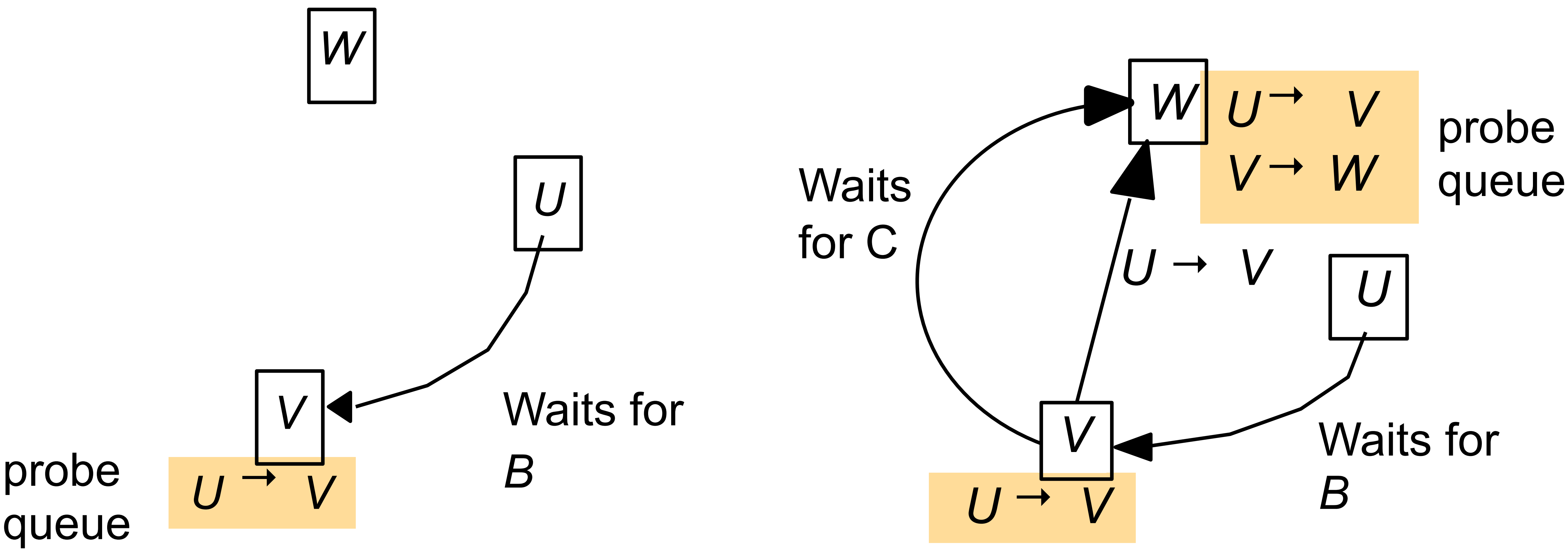


Figure 17.17
Probes travel downhill

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



Summary

- Locking needs deadlock detection.
- Phantom edges.
- Edge chasing, probing.

Summary

- Recap of Consensus & Agreement, Transactions.
- Motivation/properties (ACID, failure model)
- Commit protocols
- Concurrency control
- Distributed deadlock detection

Read on your own

- Probe queues
- Transaction recovery