CS 138: Distributed Transactions

Many of the slides through slide 21are figures from Coulouris, Dollimore, Kindberg, and Blair.

Transactions

- "ACID" property:
  - atomic
    - all or nothing
  - consistent
    - take system from one consistent state to another
  - isolated
    - have no effect on other transactions until committed
  - durable
    - persists

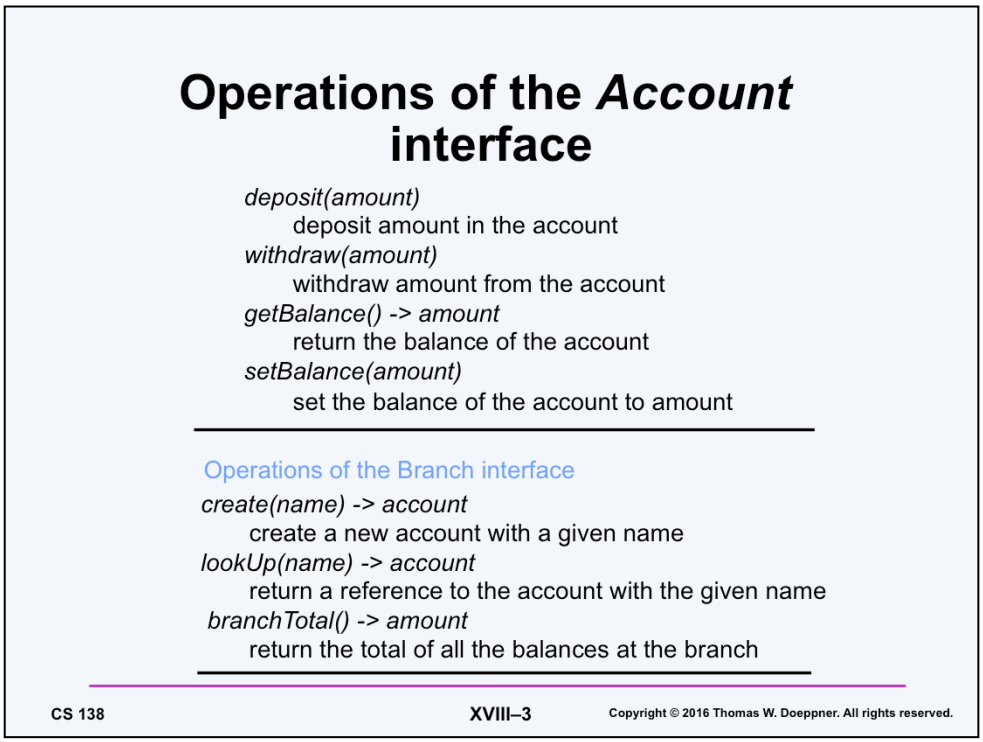


Figure 16.1. A pair of interfaces to be used in upcoming examples.

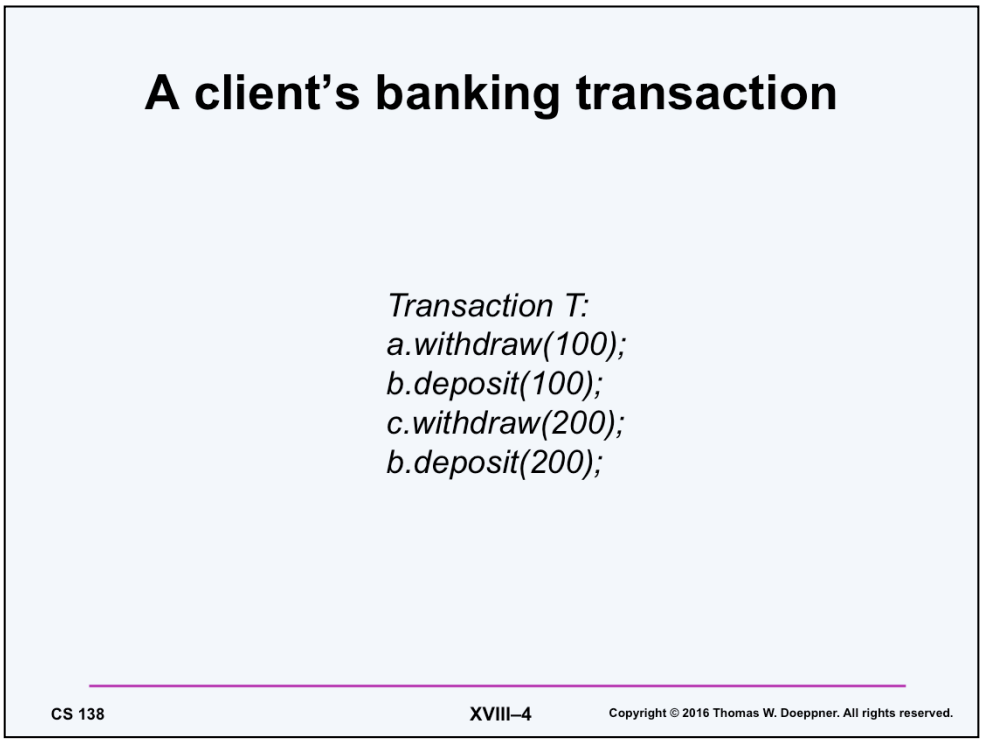


Figure 16.2: A sample banking transaction using the account interface.

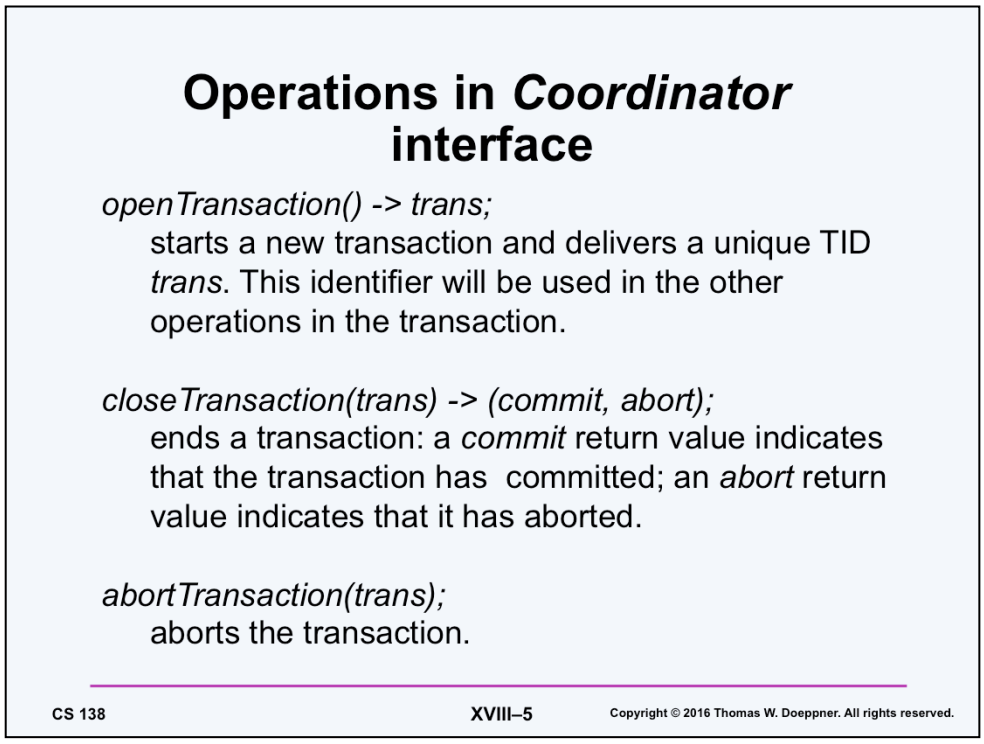

Operations in Coordinator interface
openTransaction() -> trans;
starts a new transaction and delivers a unique TID trans. This identifier will be used in the other operations in the transaction.
closeTransaction(trans) -> (commit, abort);
ends a transaction: a commit return value indicates that the transaction has committed; an abort return value indicates that it has aborted.
abortTransaction(trans);
aborts the transaction.
CS 138
XVIII–5
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.3

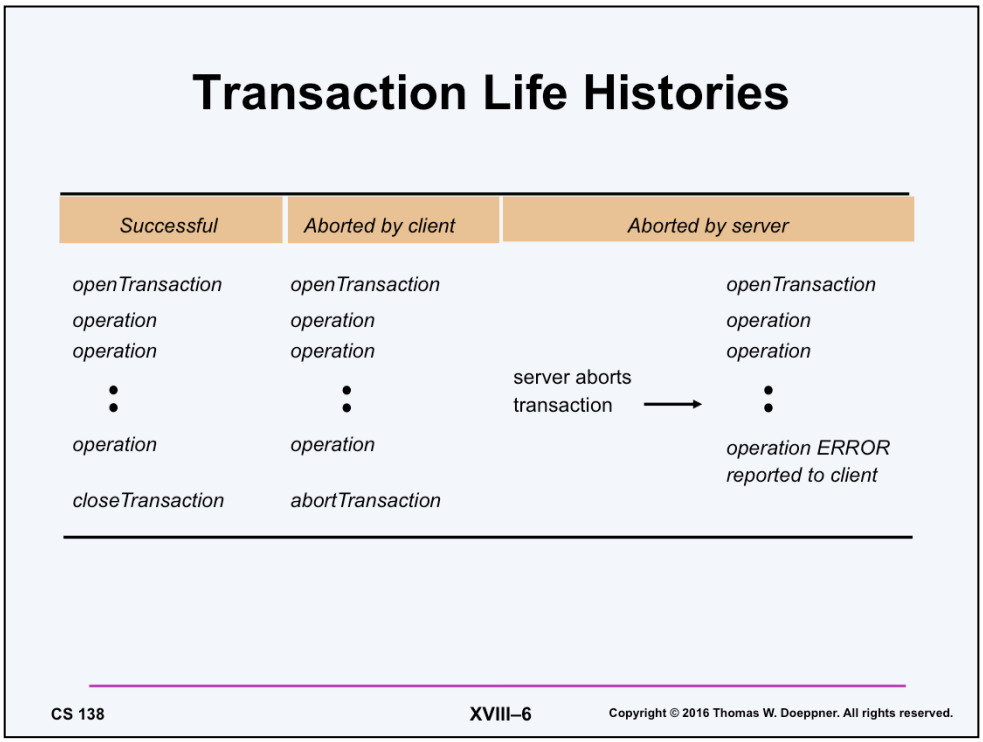

Transaction Life Histories
Successful
Aborted by client
Aborted by server
openTransaction
operation
operation
operation
closeTransaction
openTransaction
operation
operation
operation
abortTransaction
server aborts transaction
openTransaction
operation
operation
operation ERROR reported to client
CS 138
XVIII–6
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.4

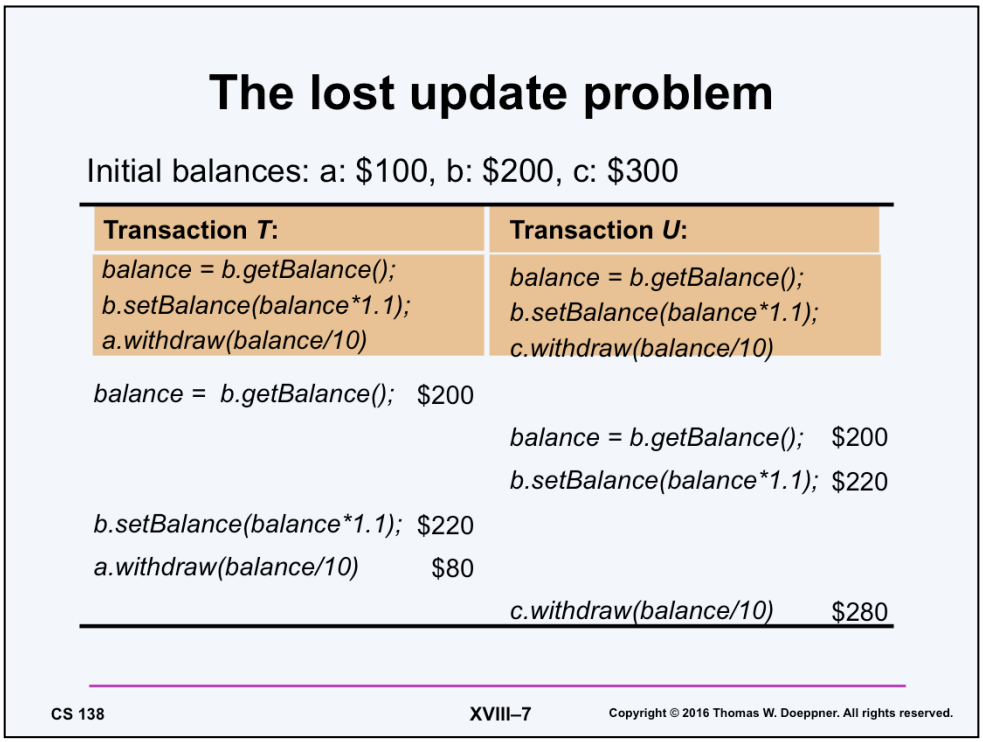

The lost update problem
Initial balances: a: $100, b: $200, c: $300
Transaction T:
Transaction U:
balance = b.getBalance();
b.setBalance(balance*1.1);
a.withdraw(balance/10)
balance = b.getBalance();
b.setBalance(balance*1.1);
c.withdraw(balance/10)
balance = b.getBalance(); $200
balance = b.getBalance(); $200
b.setBalance(balance*1.1); $220
b.setBalance(balance*1.1); $220
a.withdraw(balance/10) $80
c.withdraw(balance/10) $280
CS 138
XVIII–7
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.5

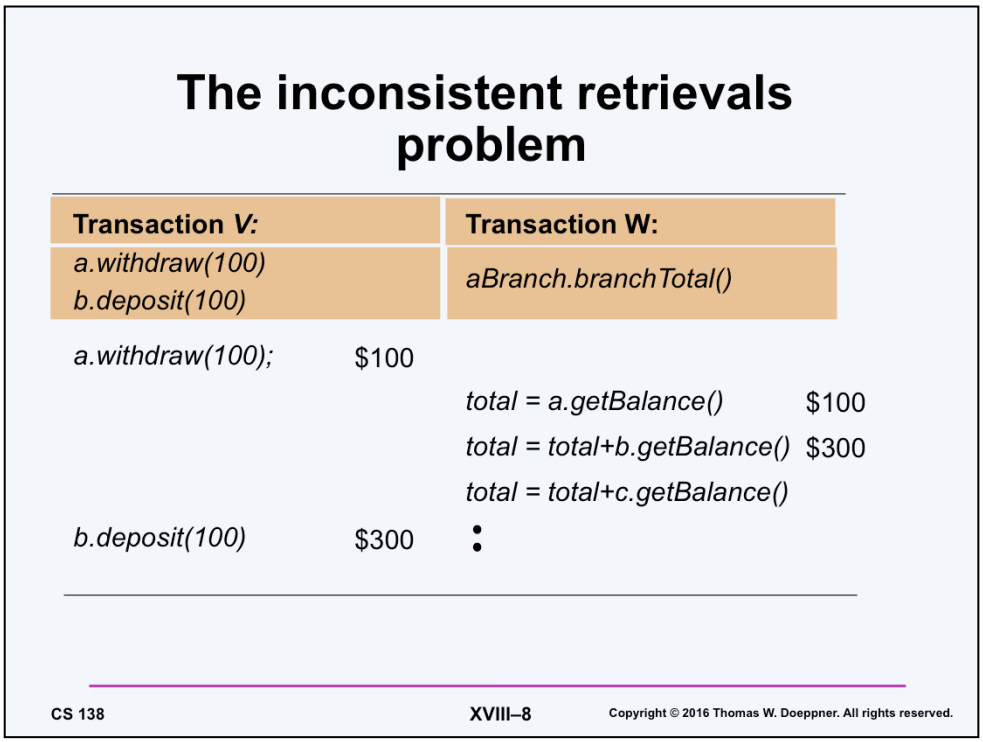

The inconsistent retrievals problem
Transaction V:
a.withdraw(100)
b.deposit(100)
Transaction W:
aBranch.branchTotal()
a.withdraw(100);
$100
b.deposit(100)
$300
total = a.getBalance()
$100
total = total+b.getBalance()
$300
total = total+c.getBalance()
:
CS 138
XVIII–8
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.6

Serial Equivalence

- Consider the effect of a concurrent execution of transactions A and B

  A || B

- What should our correctness criteria be?
- Intuitively, it should be equivalent to some serial execution:

  A;B

  B;A

- We say that A || B is *serially equivalent* if it has the same effect as either A;B or B;A

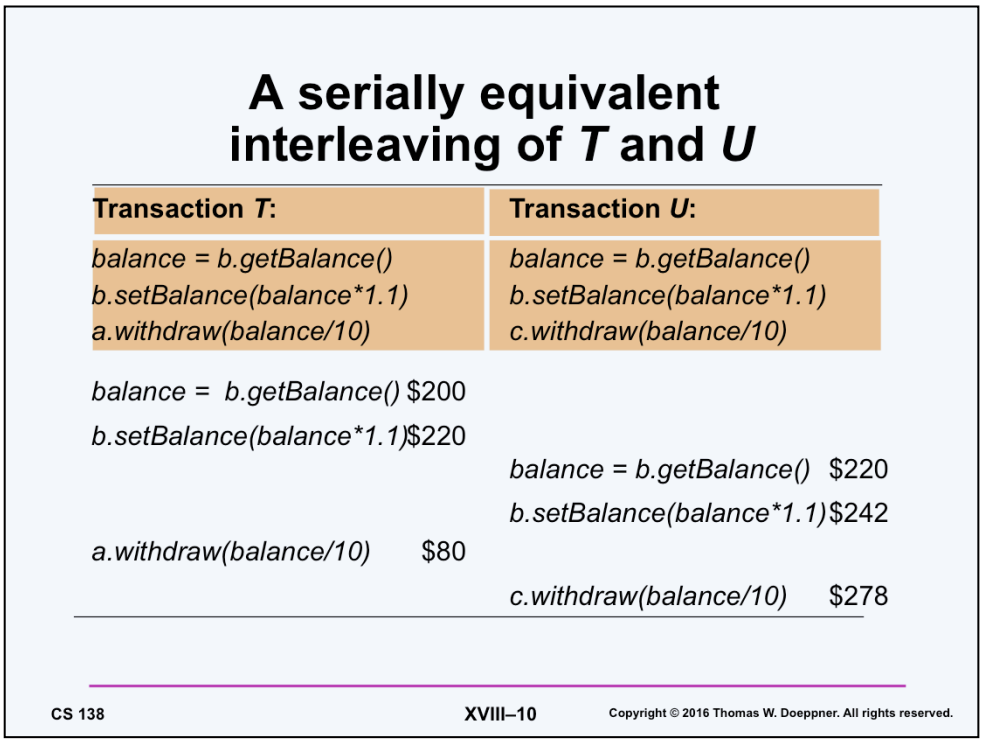

A serially equivalent interleaving of T and U
Transaction T:
Transaction U:
balance = b.getBalance()
b.setBalance(balance*1.1)
a.withdraw(balance/10)
balance = b.getBalance()
b.setBalance(balance*1.1)
c.withdraw(balance/10)
balance = b.getBalance() $200
b.setBalance(balance*1.1) $220
balance = b.getBalance() $220
b.setBalance(balance*1.1) $242
a.withdraw(balance/10) $80
c.withdraw(balance/10) $278
CS 138
XVIII–10
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.7

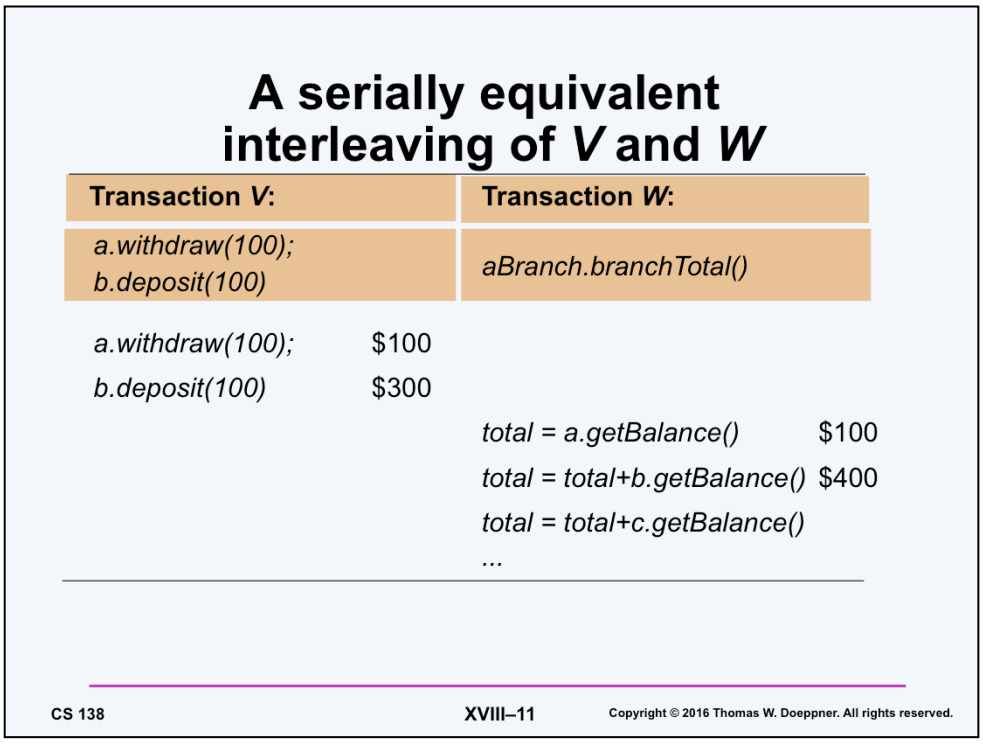

A serially equivalent interleaving of V and W
Transaction V:
Transaction W:
a.withdraw(100); b.deposit(100)
aBranch.branchTotal()
a.withdraw(100); $100
b.deposit(100) $300
total = a.getBalance() $100
total = total+b.getBalance() $400
total = total+c.getBalance()
...
CS 138
XVIII–11
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.8

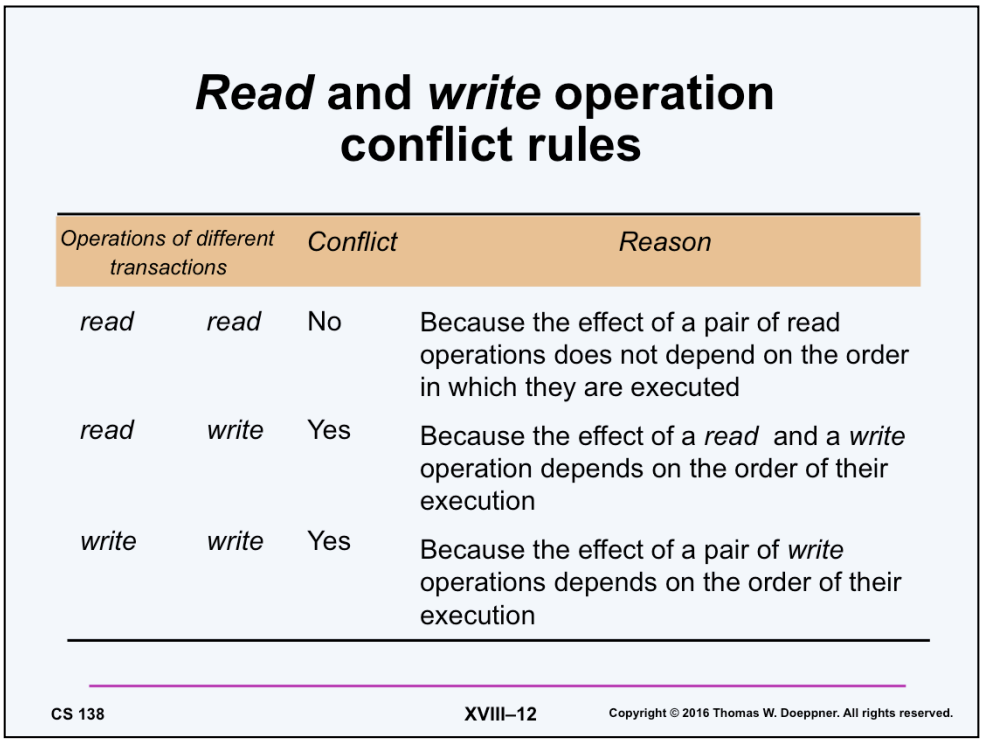

Read and write operation conflict rules
Operations of different transactions
Conflict
Reason
read
read
No
Because the effect of a pair of read operations does not depend on the order in which they are executed
read
write
Yes
Because the effect of a read and a write operation depends on the order of their execution
write
write
Yes
Because the effect of a pair of write operations depends on the order of their execution
CS 138
XVIII–12
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.9

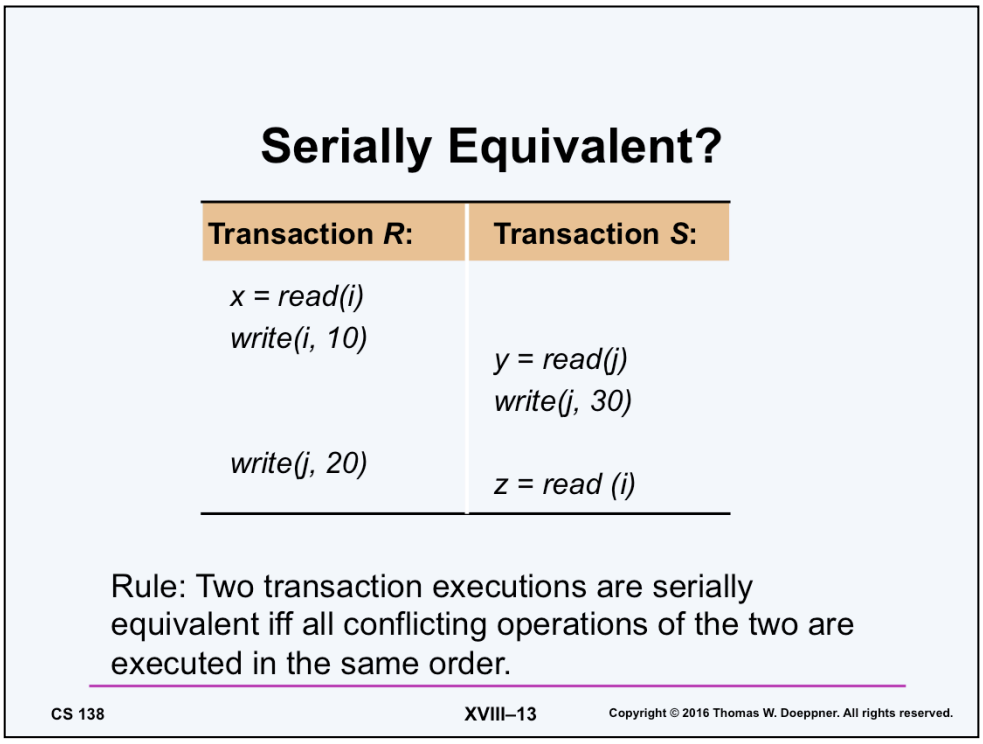


Figure 16.10: Note that all of R's accesses to i come before S's, and all of S's accesses to j come before R's. Nevertheless this is not a serially equivalent concurrent execution of R and S.

Concurrency Control

- How to ensure that transaction executions are serializable:
  - locking
  - optimism

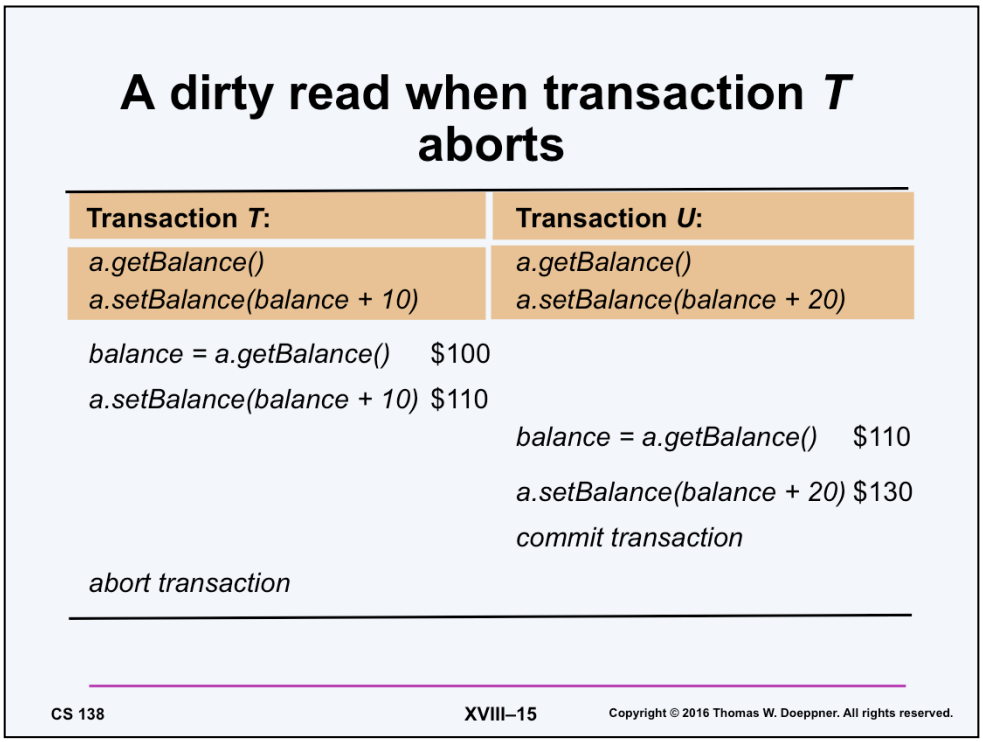


Figure 16.11: The isolation property requires that transactions do not see the uncommitted state of other transactions.

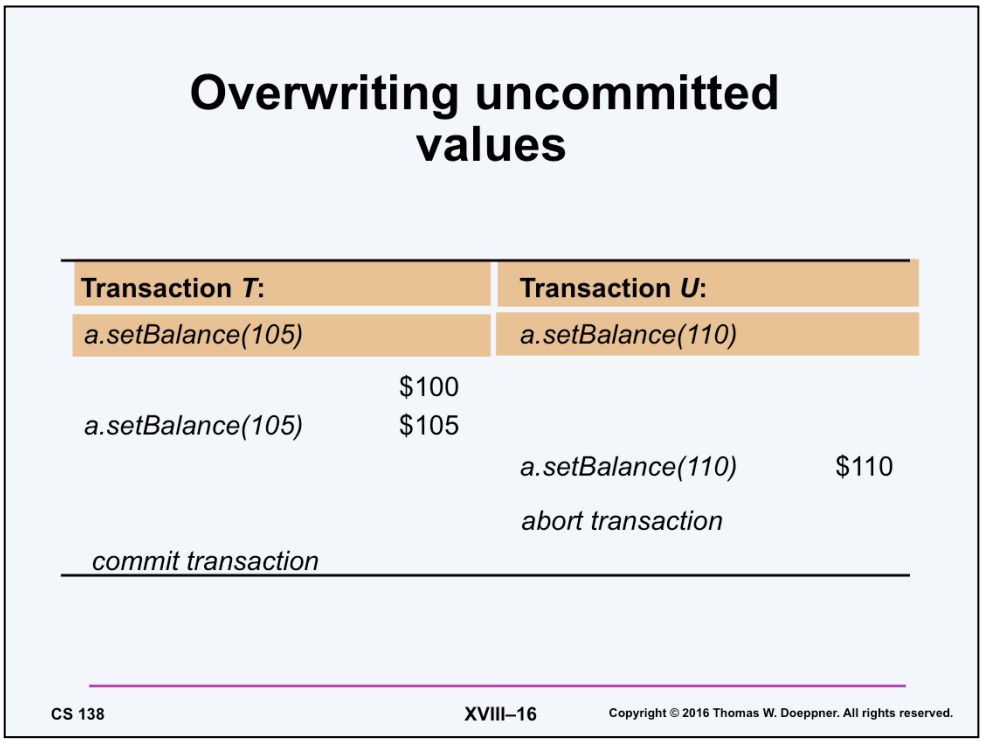


Figure 16.12: Suppose U aborts after overwriting A.

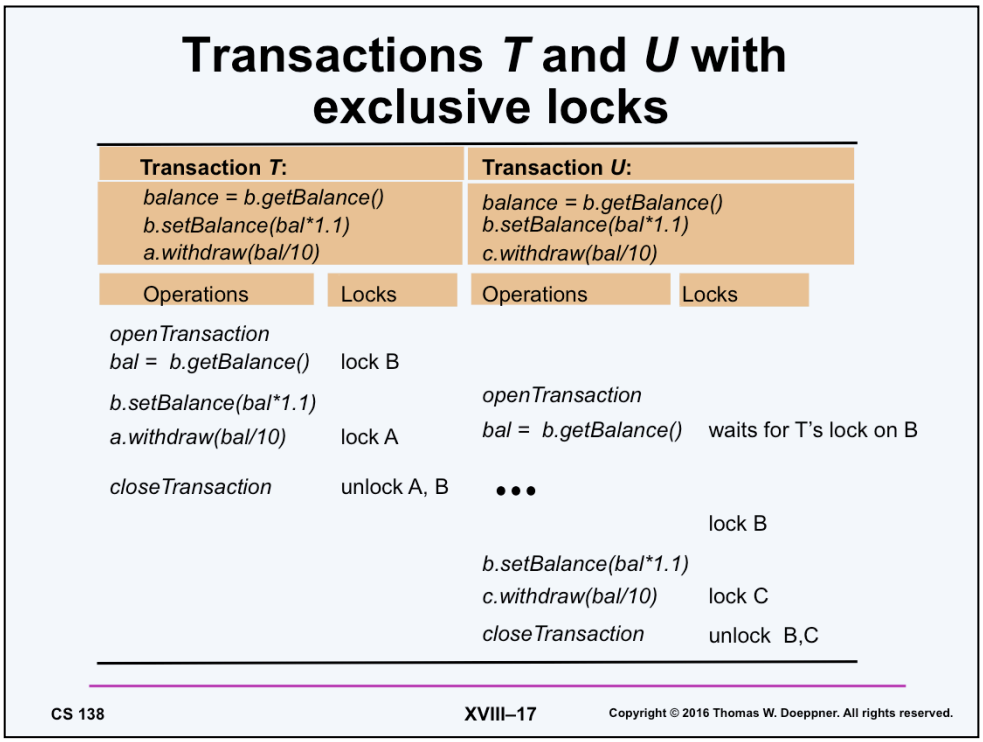
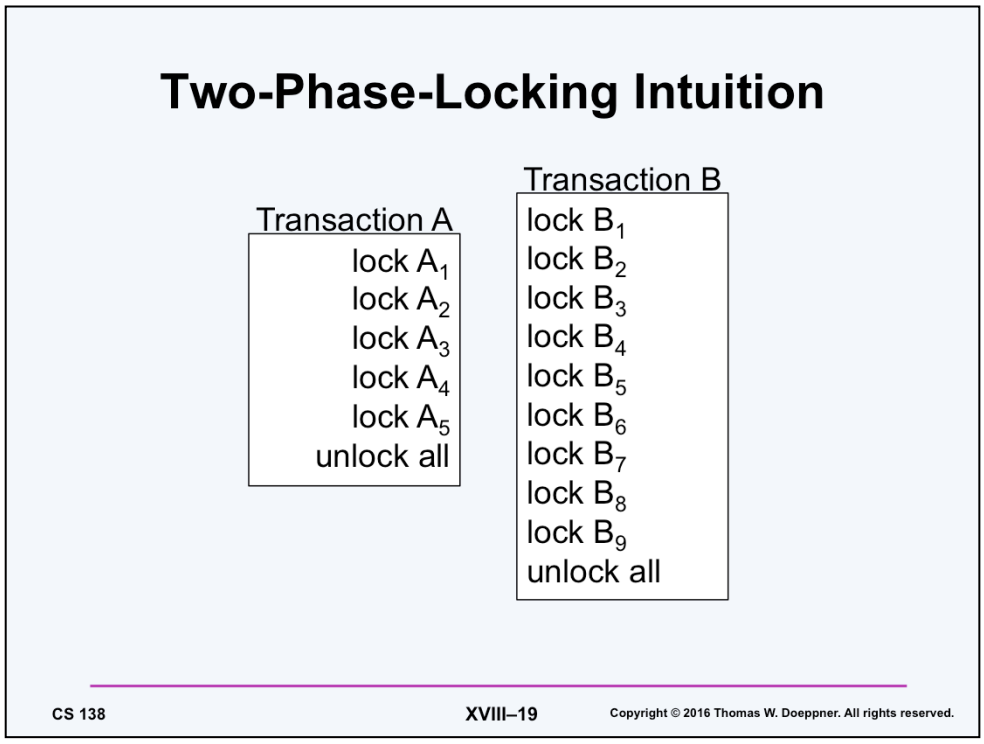

Transactions T and U with exclusive locks
Transaction T:
balance = b.getBalance()
b.setBalance(bal*1.1)
a.withdraw(bal/10)
Transaction U:
balance = b.getBalance()
b.setBalance(bal*1.1)
c.withdraw(bal/10)
Operations
Locks
Operations
Locks
openTransaction
bal = b.getBalance()
lock B
b.setBalance(bal*1.1)
openTransaction
a.withdraw(bal/10)
lock A
bal = b.getBalance()
waits for T's lock on B
closeTransaction
unlock A, B
lock B
b.setBalance(bal*1.1)
c.withdraw(bal/10)
lock C
closeTransaction
unlock B,C
CS 138
XVIII–17
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.14

Two-Phase Locking

1) Acquire locks
2) Release locks

- No more locks may be acquired after any lock is released
- Strict two-phase locking
  - no locks released until transaction commits



Transaction A completes before transaction B. Two-phase locking was used. Is it necessarily the case that their concurrent execution is equivalent to first executing all of A, then all of B?

Transaction Steps

- Accumulate changes
  - store as "tentative versions"
- Make sure everything is ok
- Commit or abort
  - move tentative versions to actual

  or
  - delete tentative versions

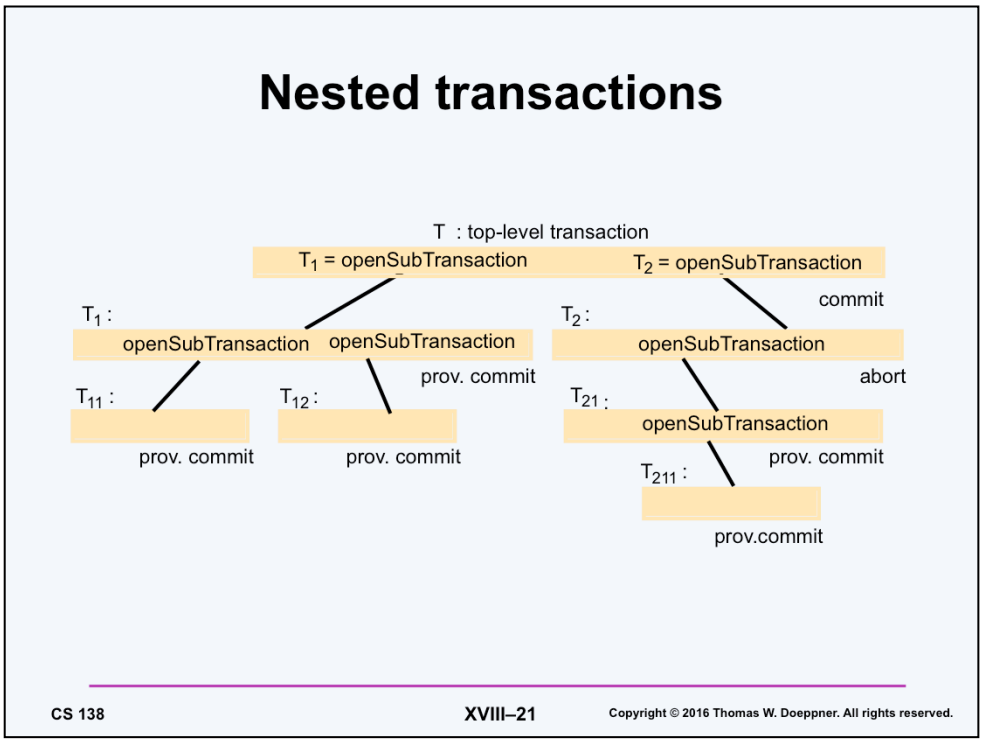
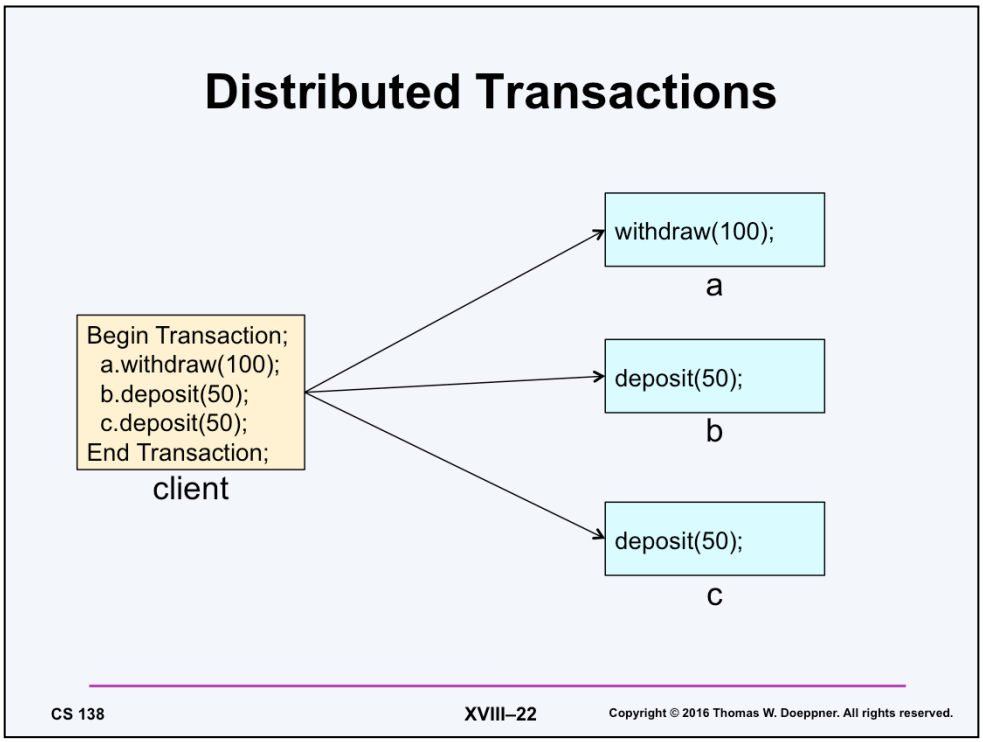
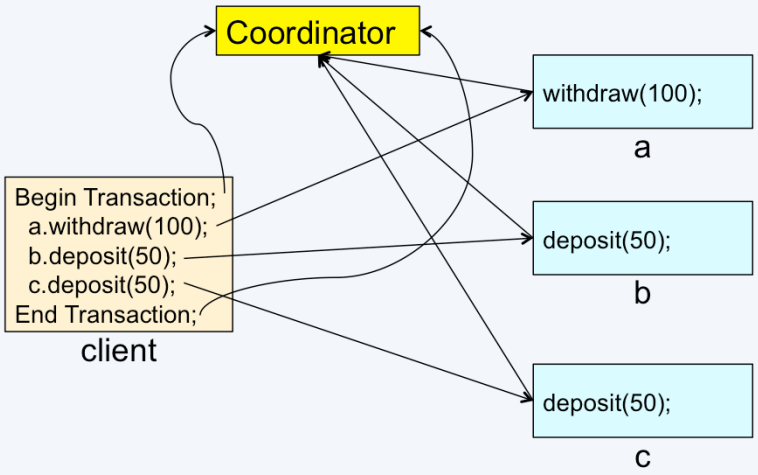

Nested transactions
T : top-level transaction
T1 = openSubTransaction
T2 = openSubTransaction
commit
T1 :
openSubTransaction
openSubTransaction
prov. commit
T2 :
openSubTransaction
abort
T11 :
T12 :
T21 :
openSubTransaction
prov. commit
prov. commit
prov. commit
T211 :
prov.commit
CS 138
XVIII–21
Copyright © 2016 Thomas W. Doeppner. All rights reserved.

Figure 16.13



Much of the remainder of this lecture is adapted from the textbook by Tanenbaum and Van Steen and from Chapter 7 of *Concurrency Control and Recovery in Database Systems*, by P. Bernstein, V. Hadzilacos, and N. Goodman, Addison-Wesley (1987). The latter text is available at http://research.microsoft.com/en-us/people/philbe/ccontrol.aspx.

Coordination



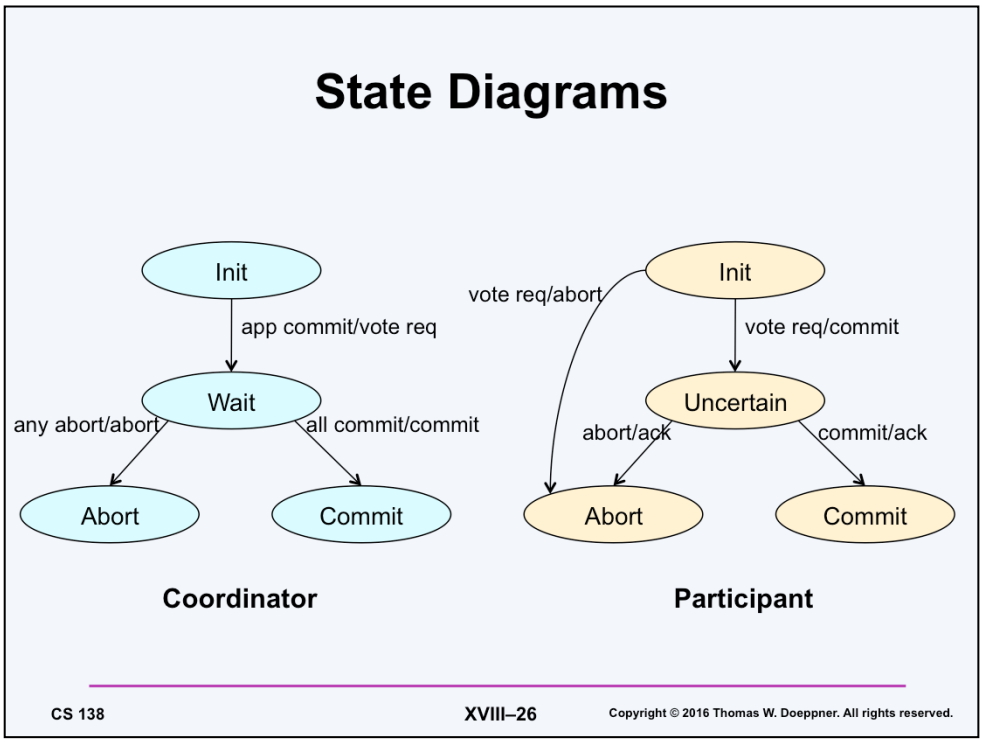
Atomic Commit

- AC1: All participants that reach a (commit/ abort) decision reach the same one
- AC2: A participant cannot reverse its decision
- AC3: The commit decision can be reached only if all participants agree
- AC4: If there are no failures and all participants vote yes, then decision will be commit
- AC5: For any execution, if all failures are repaired and no new failures occur for a sufficiently long interval, then all participants will reach a (commit/abort) decision

This is adapted from Bernstein et al.

Two-Phase Commit

- Phase 1
  - coordinator prepares to commit:
    - asks participants to vote either "commit" or "abort"
  - participants respond appropriately
- Phase 2
  - coordinator decides outcome:
    - if all participants vote commit, outcome is commit, otherwise outcome is abort
    - outcome sent to all participants
  - participants do what they're told

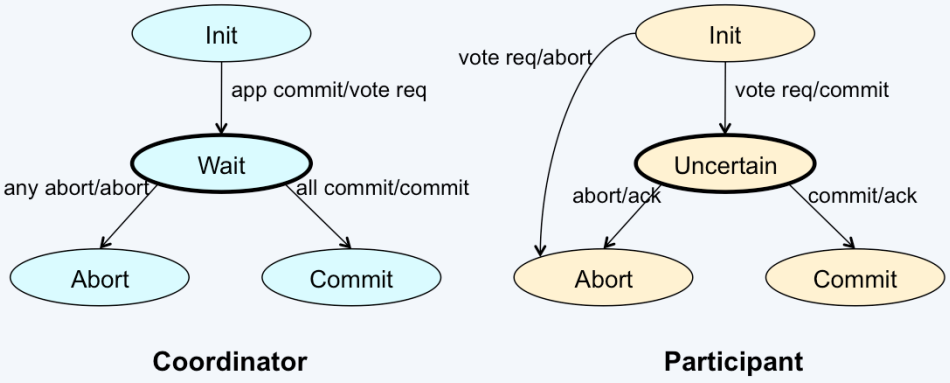


The labeling of the arcs (A/B) means that if "A" occurs, then perform action "B" and follow the arc to the next state.

Failures

- Coordinator or participants could crash
  - assume "fail-stop"
    - crash detected by time-out
    - no byzantine failures
  - crashed machines restart
    - recover their state

Crash Points

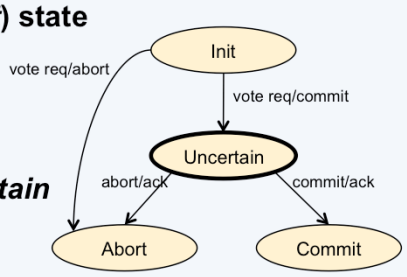


Dealing with Timeouts (1)

- Coordinator times out in *Wait* state
  - waiting for a participant to vote
  - takes no response to mean "abort"
  - sends abort to all other participants
- Participant times out in *Uncertain* state
  - waiting for coordinator to say "commit" or "abort"
  - can't assume either outcome
  - waits for coordinator to restart
  - contacts coordinator for final outcome

Dealing with Timeouts (2)

- Coordinator could take long time to restart
- Participants contact other participants
  - p contacts q (p is in *Uncertain* state)
  - q is in:
    - *commit* (or *abort*) state
      - p goes to *commit* (or *abort*) state
    - *init* state (hasn't yet voted)
      - both q and p abort
    - *uncertain* state
      - both p and q remain *uncertain*



This implies that the participants know one another's identities. They could be supplied by the coordinator in the initial vote request.

Improving on Two-Phase Commit

- It works fine in practice!
- But …
  - all participants could conceivably be in uncertain state and coordinator is down (for a long time)
- Can we make it so such blocking can't happen?

What Causes Blocking?

- Coordinator is down
- If all operational (not-failed) participants are in uncertain state, they are blocked
- If all participants are operational, they can elect new coordinator
- If any participant has crashed, the others don't know if it crashed before or after voting (to commit or abort)

Guaranteeing Non-Blocking

- **Non-blocking property (NB):**
  - ***if any operational process is in the Uncertain state, then no process (operational or failed) can have decided to commit***
- **If NB holds, then operational processes may elect new coordinator and decide to commit or abort**

Note that NB does not hold for two-phase commit!

Failures

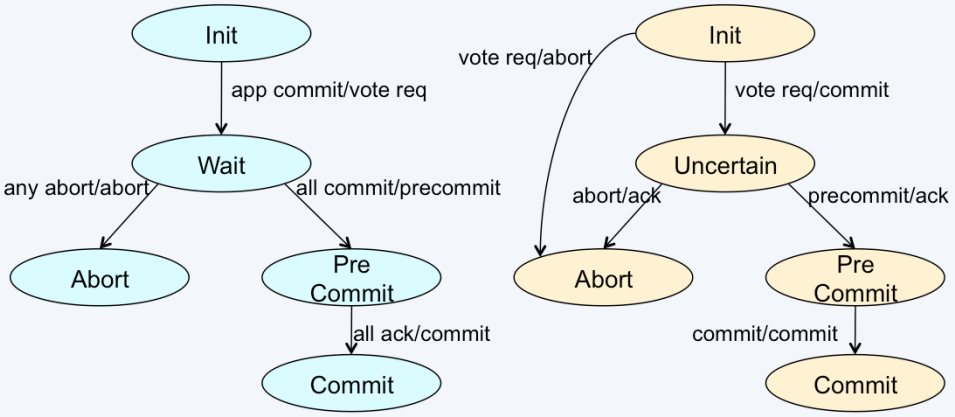
- Coordinator or participants could crash
  - *no communication failures*
  - assume "fail-stop"
    - crash detected by time-out
    - no byzantine failures
  - crashed machines restart
    - recover their state

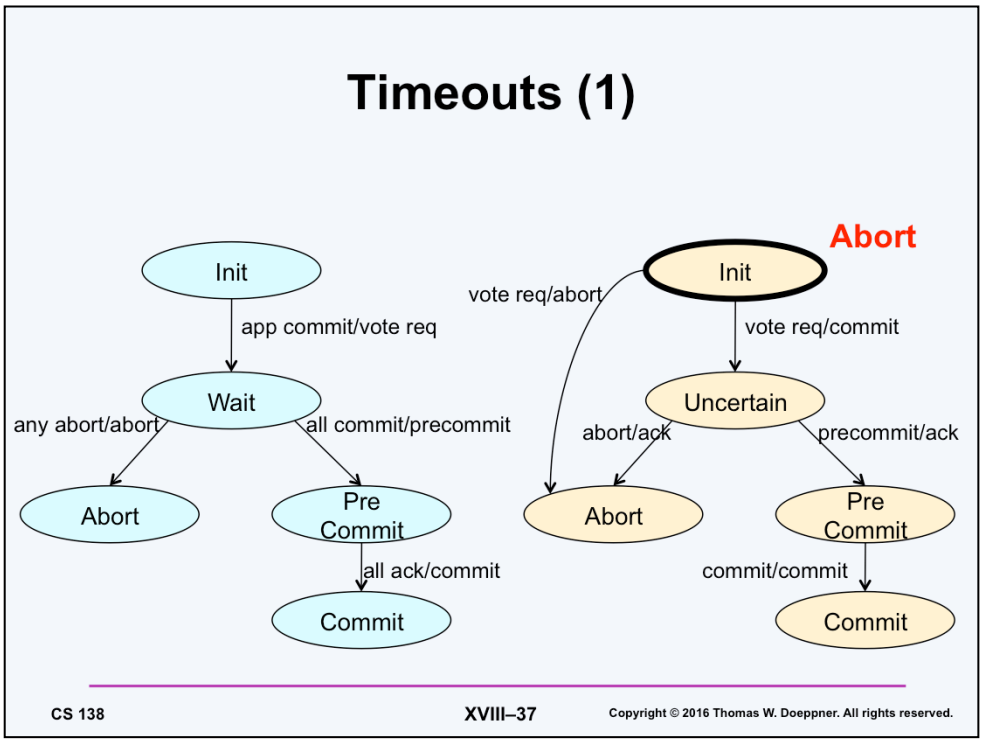
For our upcoming discussion of three-phase commit, we assume that the only sort of failure is that of a machine crashing (then recovering). In particular, communication failures do not happen.

Three-Phase Commit

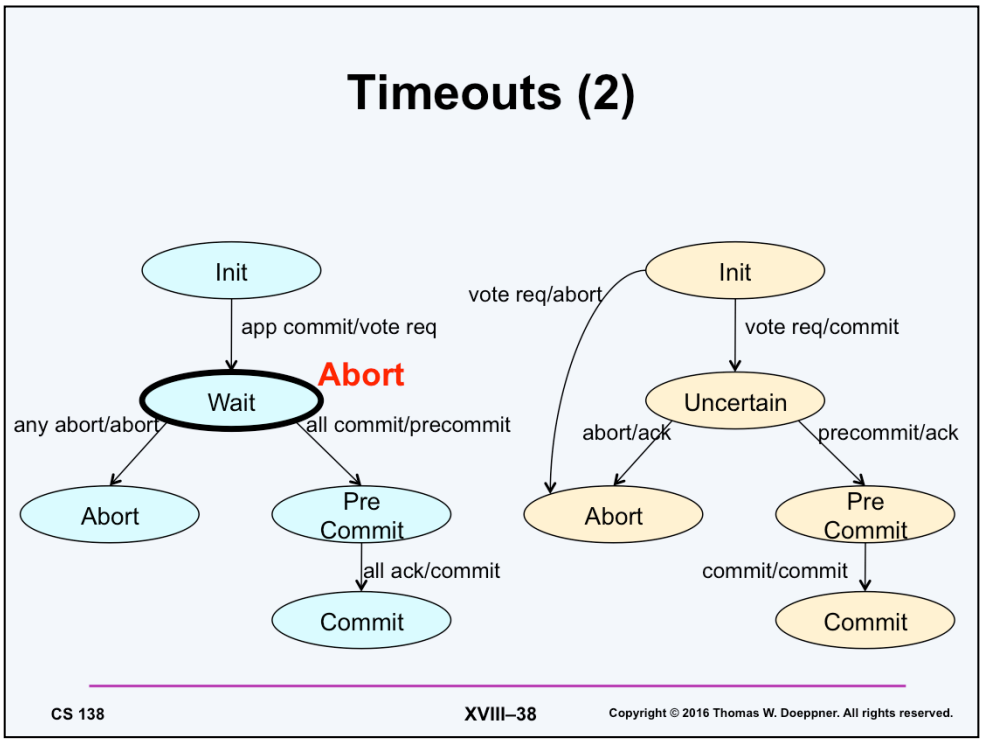
- Phase 1
  - coordinator prepares to commit:
    - asks participants to vote either "commit" or "abort"
  - participants respond appropriately
- Phase 2
  - coordinator counts votes:
    - if all participants vote commit, outcome is *pre-commit*, otherwise outcome is abort
    - outcome sent to all participants
  - participants ack and either abort or wait for commit
- Phase 3
  - coordinator waits for all acks
    - if committing, sends final commit to all participants
  - participants commit

Revised State Diagrams

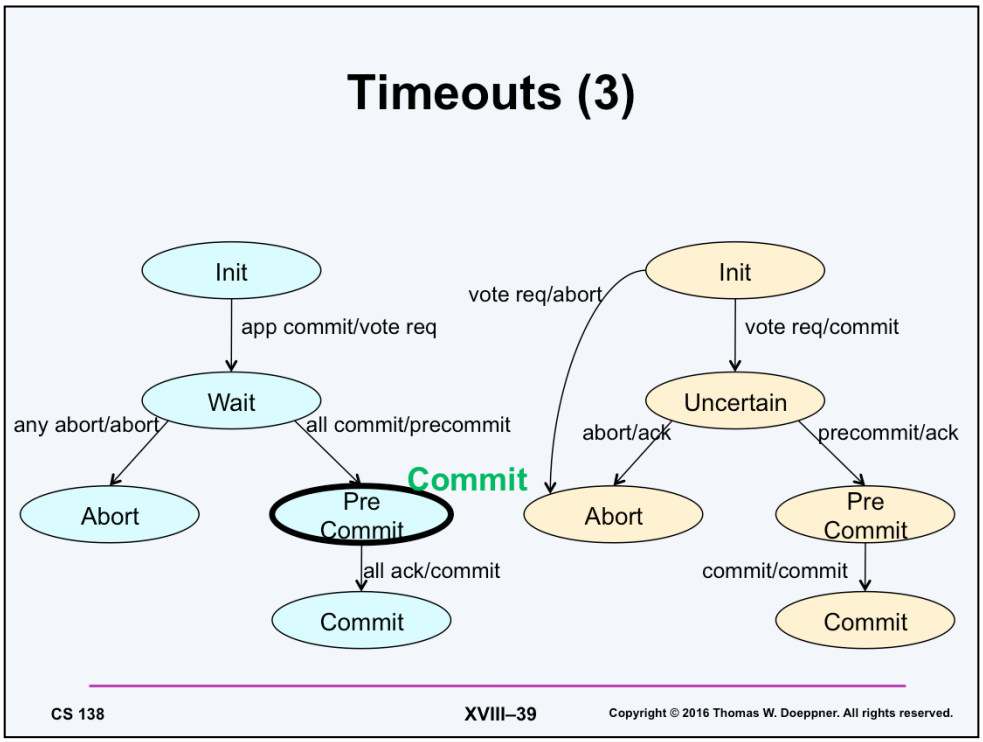




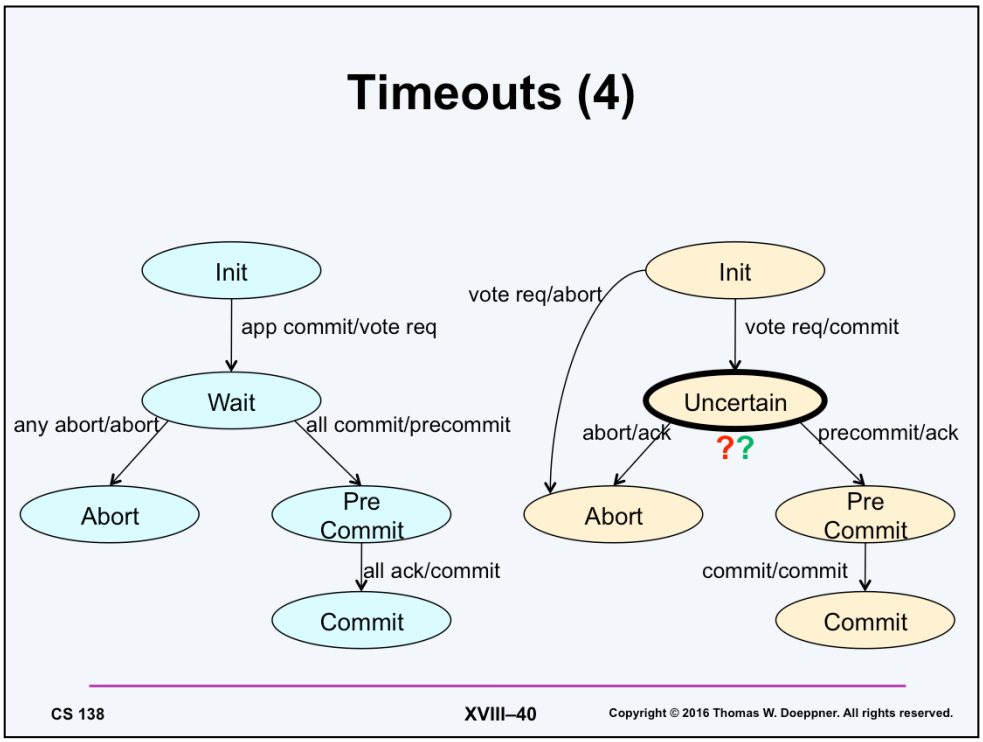
If a participant times out in its init state while waiting for a vote req from the coordinator, it may safely unilaterally abort.



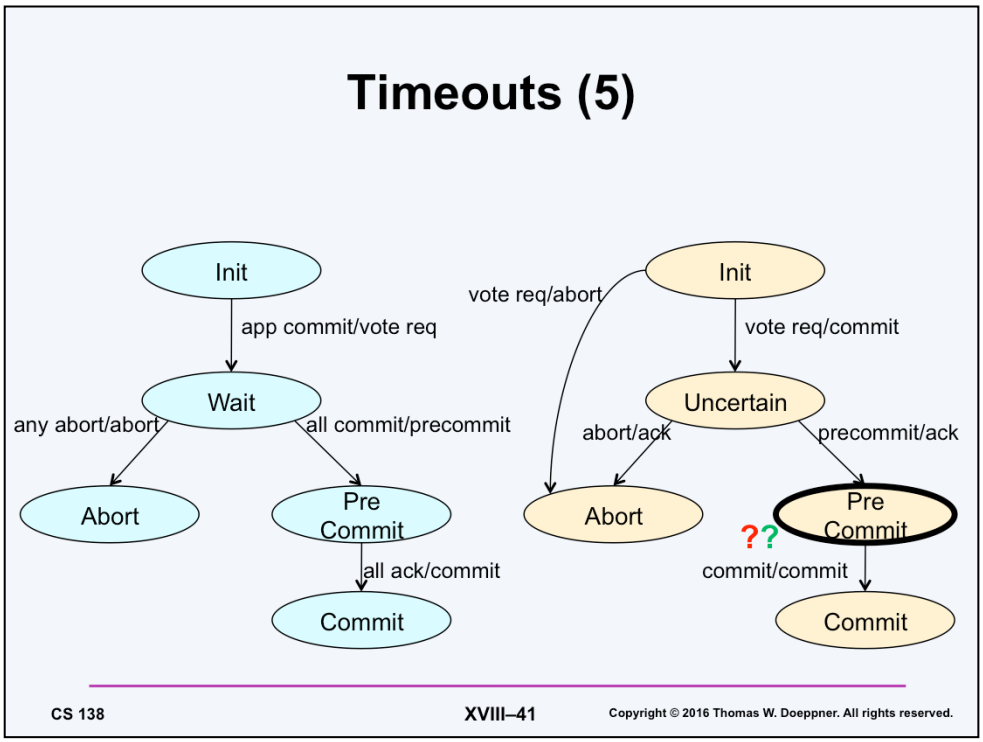
If the coordinator times out in its wait state while waiting to receive votes from participants, it should send aborts to all operational participants.



If the coordinator times out while in its precommit state, waiting to receive acks from the participants, it may safely commit, since it had received commit votes from all. The failed participants will learn about the commit when they reboot.



If a participant times out in its uncertain state waiting to hear from the coordinator, it must communicate with the other operational participants to determine if it should commit or abort. In particular, if any other participant has aborted, it should abort. But what if this is not the case? (Go on to the next slides ...)



If a participant times out in its precommit state, waiting to hear from the coordinator, shouldn't it simply assume it may commit? The answer is no, because that might violate NB: there may be some other participant that's still in the uncertain state.

The situation we're concerned about is that, after committing, the participant might fail, while some other participant (perhaps the only other participant) remains operational, but in the uncertain state. That participant, now not knowing anything about the states of the others, should be allowed to abort by virtue of NB.

Details (1)

- If original coordinator remains operational
  - participant crashes handled as in two-phase commit
- If participant times out in *Uncertain* or *PreCommit* states
  - if any other participant has aborted, it aborts (it must have been in Uncertain state)
  - otherwise, it starts an election for a new coordinator

Details (2)

- When newly elected coordinator starts up
  - sends state-request message to all operational participants
  - coordinator collects states and proceeds according to four termination rules (*termination protocol*):
    - TR1: if any participant is in *Abort* state, all are sent abort messages
    - TR2: if some participant is in *Commit* state, all are sent commit messages
    - TR3: if all participants are in *Uncertain* state, all are sent abort messages
    - TR4: if some participant is in *PreCommit* state, but none in *Commit* State, those in *Uncertain* state are sent PreCommit messages; once these are acked, all participants are sent commit messages

Note that the newly elected coordinator could fail. If so, a new one is elected. (Participants will time-out waiting for a message.)

Details (3)

- When failed participant comes back up
  - if it failed in *Init* state
    - it aborts
  - Otherwise it asks other participants for outcome
    - will eventually get either commit or abort
      - (could get abort even if it was in the *PreCommit* state when it crashed)

Correctness (1)

- Lemma 1: After a new coordinator starts up, exactly one of TR1 – TR4 will hold
- Theorem 1: In the absence of total failures, participants will never block
  - they clearly won't block if the coordinator never fails
  - if the coordinator fails, a new one is elected
  - one of TR1-TR4 will hold and a decision will be reached
  - if the new coordinator fails, a new one is elected; if it fails another is elected, etc. until there are no more participants

For details, see chapter 7 from Bernstein et al.

Correctness (2)

- Lemma 2: All participants that reach a decision on the same invocation of the termination protocol reach the same one
- Lemma 3: If NB holds before the termination protocol starts, it holds through the execution of the protocol
- Theorem 2: All operational participants reach the same decision
  - proof by induction on the invocations of the termination protocol

Total Failure

- What if coordinator and all participants fail?
- When they come back up, how do they decide?
  - if resurrected participant either didn't vote or voted abort, it may unilaterally abort
  - otherwise, must run termination protocol
  - but works only if last participant to fail has come back up

See Bernstein et al. for details.

Communication Failures

- Network could partition into multiple pieces
- Not sufficient to get agreement in a piece containing a quorum
  - consensus is required for commit!
- Scenario
  - all participants vote
  - coordinator collects results
  - network partitions before or after all results collected
  - if network reconnects: easy
  - network never fully reconnects, but each participant eventually can communicate (perhaps briefly) with all others

See Bernstein et al. for details.