

CS162
Operating Systems and
Systems Programming
Lecture 19
Transactions, Two Phase Locking (2PL),
Two Phase Commit (2PC)

November 5, 2012
 Ion Stoica
<http://inst.eecs.berkeley.edu/~cs162>

Goals of Today's Lecture

- Transaction scheduling
- Two phase locking (2PL) and strict 2PL
- Two-phase commit (2PC):

Note: Some slides and/or pictures in the following are adapted from lecture notes by Mike Franklin.

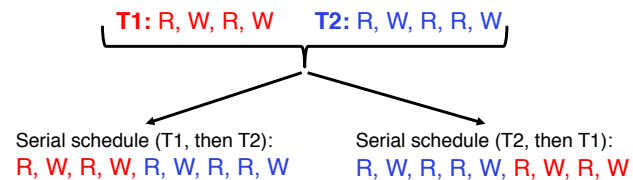
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.2

Goals of Transaction Scheduling

- Maximize system utilization, i.e., concurrency
 - Interleave operations from different transactions
- Preserve transaction semantics
 - Semantically equivalent to a serial schedule, i.e., one transaction runs at a time



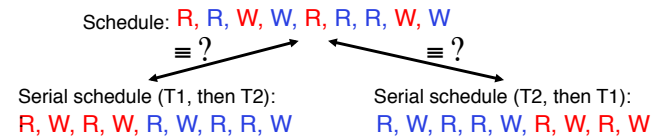
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.3

Two Key Questions

- 1) Is a given schedule equivalent to a serial execution of transactions?



- 2) How do you come up with a schedule equivalent to a serial schedule?

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.4

Transaction Scheduling

- **Serial schedule:** A schedule that **does not interleave** the operations of different transactions
 - Transactions run serially (one at a time)
- **Equivalent schedules:** For any storage/database state, the effect (on storage/database) and output of executing the first schedule is identical to the effect of executing the second schedule
- **Serializable schedule:** A schedule that is **equivalent** to some serial execution of the transactions
 - Intuitively: with a serializable schedule you only see things that could happen in situations where you were running transactions one-at-a-time

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.5

Anomalies with Interleaved Execution

- May violate transaction semantics, e.g., some data read by the transaction changes before committing
- Inconsistent database state, e.g., some updates are lost
- Anomalies always involves a “write”; Why?

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.6

Anomalies with Interleaved Execution

- Read-Write conflict (Unrepeatable reads)

T1 : R (A) ,	R (A) , W (A)
T2 :	R (A) , W (A)

- Violates transaction semantics
- Example: Mary and John want to buy a TV set on Amazon but there is only one left in stock
 - (T1) John logs first, but waits...
 - (T2) Mary logs second and buys the TV set right away
 - (T1) John decides to buy, but it is too late...

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.7

Anomalies with Interleaved Execution

- Write-read conflict (reading uncommitted data)

T1 : R (A) , W (A) ,	W (A)
T2 :	R (A) , ...

- Example:
 - (T1) A user updates value of A in two steps
 - (T2) Another user reads the intermediate value of A, which can be inconsistent
 - Violates transaction semantics since T2 is not supposed to see intermediate state of T1

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.8

Anomalies with Interleaved Execution

- Write-write conflict (overwriting uncommitted data)

T1 : W (A) ,	W (B)
T2 :	W (A) , W (B)

- Get T1's update of B and T2's update of A
- Violates transaction serializability
- If transactions were serial, you'd get either:
 - T1's updates of A and B
 - T2's updates of A and B

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.9

Conflict Serializable Schedules

- Two operations **conflict** if they
 - Belong to different transactions
 - Are on the same data
 - At least one of them is a write
- Two schedules are **conflict equivalent** iff:
 - Involve same operations of same transactions
 - Every pair of **conflicting** operations is ordered the same way
- Schedule S is **conflict serializable** if S is conflict equivalent to some serial schedule

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.10

Conflict Equivalence – Intuition

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**

- Example:

T1 : R (A) , W (A) ,	R (B) , W (B)
T2 :	R (A) , W (A) , R (B) , W (B)



T1 : R (A) , W (A) ,	R (B) , W (B)
T2 :	R (A) , W (A) , R (B) , W (B)



T1 : R (A) , W (A) , R (B) ,	W (B)
T2 :	R (A) , W (A) , R (B) , W (B)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.11

Conflict Equivalence – Intuition (cont' d)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**

- Example:

T1 : R (A) , W (A) , R (B) ,	W (B)
T2 :	R (A) , W (A) , R (B) , W (B)



T1 : R (A) , W (A) , R (B) ,	W (B)
T2 :	R (A) , W (A) , R (B) , W (B)



T1 : R (A) , W (A) , R (B) , W (B)	
T2 :	R (A) , W (A) , R (B) , W (B)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.12

Conflict Equivalence – Intuition (cont' d)

- If you can transform an interleaved schedule by swapping *consecutive non-conflicting* operations of *different transactions* into a serial schedule, then the original schedule is **conflict serializable**

- Is this schedule serializable?

T1: R(A),	W(A)
T2:	R(A), W(A),

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.13

Dependency Graph

• Dependency graph:

- Transactions represented as nodes
- Edge from T_i to T_j :
 - » an operation of T_i conflicts with an operation of T_j
 - » T_i appears earlier than T_j in the schedule

- **Theorem:** Schedule is conflict serializable if and only if its dependency graph is acyclic

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.14

Example

- Conflict serializable schedule:

T1: R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)



- No cycle!

11/5

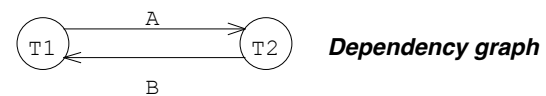
Ion Stoica CS162 ©UCB Fall 2012

Lec 19.15

Example

- Conflict that is *not* serializable:

T1: R(A), W(A),	R(B), W(B)
T2:	R(A), W(A), R(B), W(B)



- Cycle: The output of T1 depends on T2, and vice-versa

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.16

Notes on Conflict Serializability

- Conflict Serializability doesn't allow all schedules that you would consider correct
 - This is because it is strictly *syntactic* - it doesn't consider the meanings of the operations or the data
- In practice, Conflict Serializability is what gets used, because it can be done efficiently
 - Note: in order to allow more concurrency, some special cases do get implemented, such as for travel reservations, ...
- Two-phase locking (2PL) is how we implement it

11/5

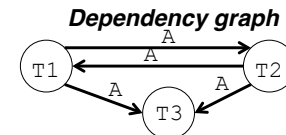
Ion Stoica CS162 ©UCB Fall 2012

Lec 19.17

Serializability \neq Conflict Serializability

- Following schedule is **not** conflict serializable

```
T1: R(A),      W(A),
T2:      W(A),
T3:              WA
```



- However, the schedule is serializable since its output is equivalent with the following serial schedule

```
T1: R(A), W(A),
T2:      W(A),
T3:              WA
```

- Note: deciding whether a schedule is serializable (not conflict-serializable) is NP-complete

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.18

Locks

- "Locks" to control access to data
- Two types of locks:
 - shared (S) lock – multiple concurrent transactions allowed to operate on data
 - exclusive (X) lock – only one transaction can operate on data at a time

Lock Compatibility Matrix

	S	X
S	✓	–
X	–	–

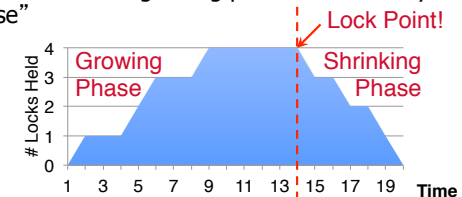
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.19

Two-Phase Locking (2PL)

- Each transaction must obtain:
 - S (*shared*) or X (*exclusive*) lock on data before reading,
 - X (*exclusive*) lock on data before writing
 - A transaction can not request additional locks once it releases any locks
- Thus, each transaction has a "growing phase" followed by a "shrinking phase"



11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.20

Two-Phase Locking (2PL)

- 2PL guarantees conflict serializability
- Doesn't allow dependency cycles. Why?
- Answer: a dependency cycle leads to deadlock
 - Assume there is a cycle between T_i and T_j
 - Edge from T_i to T_j : T_i acquires lock first and T_j needs to wait
 - Edge from T_j to T_i : T_j acquires lock first and T_i needs to wait
 - Thus, both T_i and T_j wait for each other
 - Since with 2PL neither T_i nor T_j release locks before acquiring all locks they need \rightarrow deadlock
- Schedule of conflicting transactions is conflict equivalent to a serial schedule ordered by "lock point"

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.21

Lock Management

- Lock Manager (LM) handles all lock and unlock requests
 - LM contains an entry for each currently held lock
- When lock request arrives see if anyone else holds a conflicting lock
 - If not, create an entry and grant the lock
 - Else, put the requestor on the wait queue
- Locking and unlocking are atomic operations
- Lock upgrade: share lock can be upgraded to exclusive lock

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.22

Example

- T_1 transfers \$50 from account A to account B

```
T1: Read(A), A:=A-50, Write(A), Read(B), B:=B+50, Write(B)
```

- T_2 outputs the total of accounts A and B

```
T2: Read(A), Read(B), PRINT(A+B)
```

- Initially, A = \$1000 and B = \$2000
- What are the possible output values?

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.23

Is this a 2PL Schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A:=A-50	
4	Write(A)	
5	Unlock(A)	<granted>
6		Read(A)
7		Unlock(A)
8		Lock_S(B) <granted>
9	Lock_X(B)	
10		Read(B)
11	<granted>	Unlock(B)
12		PRINT(A+B)
13	Read(B)	
14	B:=B+50	
15	Write(B)	
16	Unlock(B)	

No, and it is not serializable

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.24

Is this a 2PL Schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A := A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Unlock(A)	<granted>
7		Read(A)
8		Lock_S(B)
9	Read(B)	
10	B := B + 50	
11	Write(B)	
12	Unlock(B)	<granted>
13		Unlock(A)
14		Read(B)
15		Unlock(B)
16		PRINT(A+B)

Yes, so it is serializable

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.25

11/5

Cascading Aborts

- Example: T1 aborts
 - Note: this is a 2PL schedule

T1: R(A), W(A), R(B), W(B), Abort
T2: R(A), W(A)

- Rollback of T1 requires rollback of T2, since T2 reads a value written by T1
- Solution: **Strict Two-phase Locking (Strict 2PL)**: same as 2PL except
 - All locks held by a transaction are released only when the transaction completes

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.26

Strict 2PL (cont' d)

- All locks held by a transaction are released only when the transaction completes
- In effect, “shrinking phase” is delayed until:
 - Transaction has committed (commit log record on disk), or
 - Decision has been made to abort the transaction (then locks can be released after rollback).

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.27

Is this a Strict 2PL schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A := A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Unlock(A)	<granted>
7		Read(A)
8		Lock_S(B)
9	Read(B)	
10	B := B + 50	
11	Write(B)	
12	Unlock(B)	<granted>
13		Unlock(A)
14		Read(B)
15		Unlock(B)
16		PRINT(A+B)

No: Cascading Abort Possible

Lec 19.28

11/5

Is this a Strict 2PL schedule?

1	Lock_X(A) <granted>	
2	Read(A)	Lock_S(A)
3	A := A-50	
4	Write(A)	
5	Lock_X(B) <granted>	
6	Read(B)	
7	B := B + 50	
8	Write(B)	
9	Unlock(A)	
10	Unlock(B)	↓ <granted>
11		Read(A)
12		Lock_S(B) <granted>
13		Read(B)
14		PRINT(A+B)
15		Unlock(A)
16		Unlock(B)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.29

Quiz 19.1: Transactions

- Q1: True _ False _ It is possible for two read operations to conflict
- Q2: True _ False _ A strict 2PL schedule does not avoid cascading aborts
- Q3: True _ False _ 2PL leads to deadlock if schedule not conflict serializable
- Q4: True _ False _ A conflict serializable schedule is always serializable
- Q5: True _ False _ The following schedule is serializable

T1 : R (A) , W (A) ,	R (B) ,	W (B)
T2 :	R (A) ,	W (A) , R (B) , W (B)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.30

Quiz 19.1: Transactions

- Q1: True _ False X It is possible for two read operations to conflict
- Q2: True _ False X A strict 2PL schedule does not avoid cascading aborts
- Q3: True X False _ 2PL leads to deadlock if schedule not conflict serializable
- Q4: True X False _ A conflict serializable schedule is always serializable
- Q5: True X False _ The following schedule is serializable

T1 : R (A) , W (A) ,	R (B) ,	W (B)
T2 :	R (A) ,	W (A) , R (B) , W (B)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.31

Announcements

- Project 3 is due on **Tuesday, November 13, 11:59pm**
- Next lecture: Anthony Joseph
- Please remember that we have another "unannounced" quiz!

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.32

5min Break

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.33

Deadlock

- Recall: if a schedule is not conflict-serializable, 2PL leads to deadlock, i.e.,
 - Cycles of transactions waiting for each other to release locks
- Recall: two ways to deal with deadlocks
 - Deadlock prevention
 - Deadlock detection
- Many systems punt problem by using timeouts instead
 - Associate a timeout with each lock
 - If timeout expires release the lock
 - What is the problem with this solution?

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.34

Deadlock Prevention

- Prevent circular waiting
- Assign priorities based on timestamps. Assume T_i wants a lock that T_j holds. Two policies are possible:
 - Wait-Die: If T_i is older, T_i waits for T_j ; otherwise T_i aborts
 - Wound-wait: If T_i is older, T_j aborts; otherwise T_i waits
- If a transaction re-starts, make sure it gets its original timestamp
 - Why?

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.35

Deadlock Detection

- Allow deadlocks to happen but check for them and fix them if found
- Create a **wait-for graph**:
 - Nodes are transactions
 - There is an edge from T_i to T_j if T_i is waiting for T_j to release a lock
- Periodically check for cycles in the waits-for graph
- If cycle detected – find a transaction whose removal will break the cycle and kill it

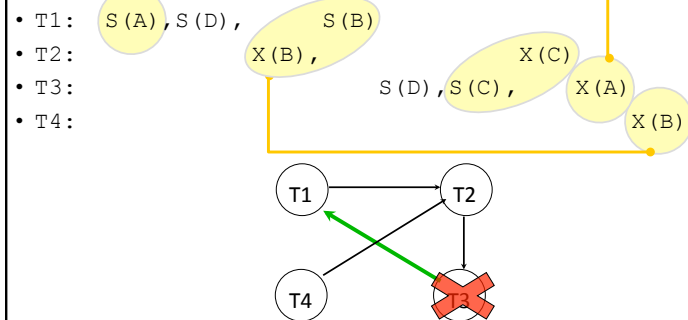
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.36

Deadlock Detection (Continued)

- Example:



11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.37

Durability and Atomicity

- How do you make sure transaction results persist in the face of failures (e.g., disk failures)?
- Replicate database
 - Commit transaction to each replica
- What happens if you have failures during a transaction commit?
 - Need to ensure atomicity: either transaction is committed on all replicas or none at all

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.38

Two Phase (2PC) Commit

- 2PC is a distributed protocol
- High-level problem statement
 - If no node fails and all nodes are ready to commit, then all nodes **COMMIT**
 - Otherwise **ABORT** at all nodes
- Developed by Turing award winner Jim Gray (first Berkeley CS PhD, 1969)

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.39

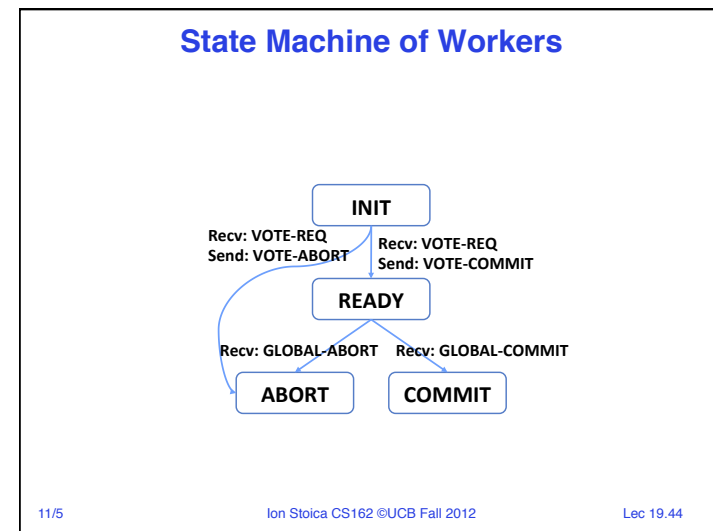
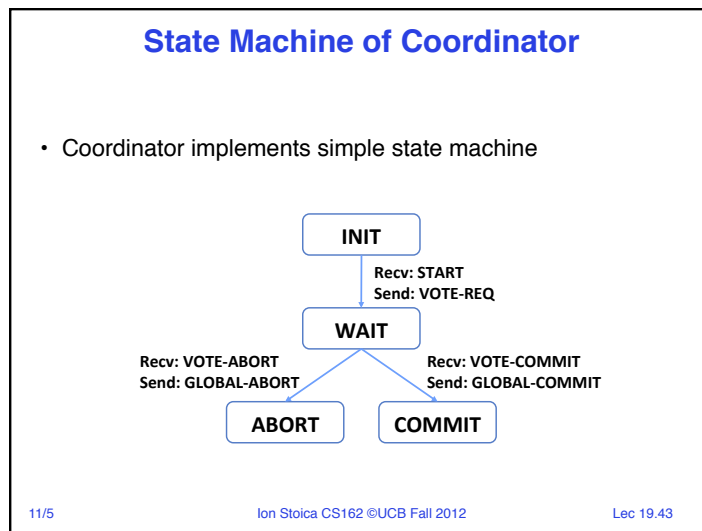
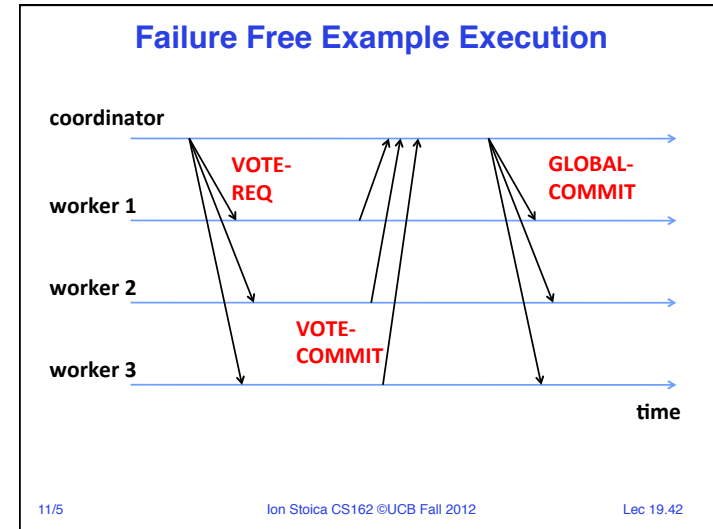
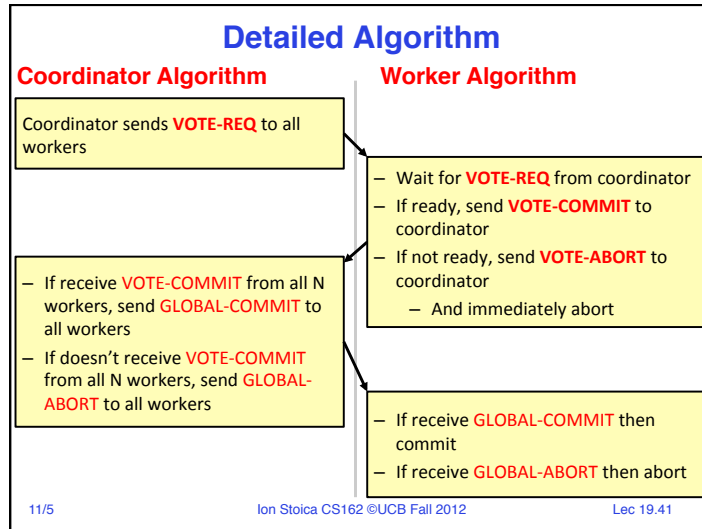
2PC Algorithm

- One coordinator
- N workers (replicas)
- High level algorithm description
 - Coordinator asks all workers if they can commit
 - If all workers reply "**VOTE-COMMIT**", then coordinator broadcasts "**GLOBAL-COMMIT**",
Otherwise coordinator broadcasts "**GLOBAL-ABORT**"
 - Workers obey the **GLOBAL** messages

11/5

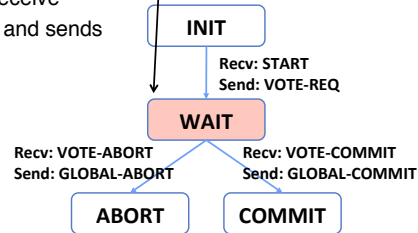
Ion Stoica CS162 ©UCB Fall 2012

Lec 19.40



Dealing with Worker Failures

- How to deal with worker failures?
 - Failure only affects states in which the node is waiting for messages
 - Coordinator only waits for votes in "WAIT" state
 - In WAIT, if doesn't receive N votes, it times out and sends GLOBAL-ABORT

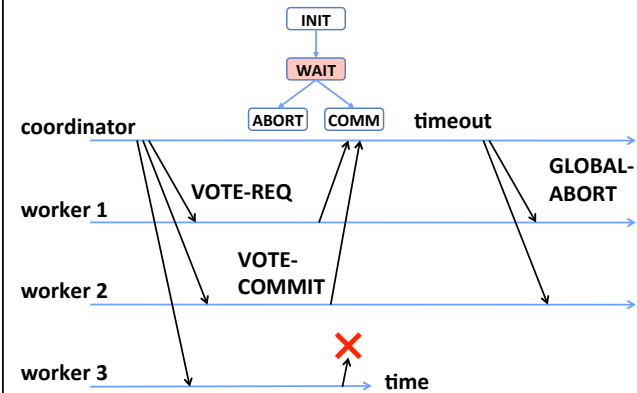


11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.45

Example of Worker Failure



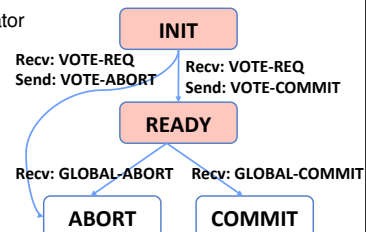
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.46

Dealing with Coordinator Failure

- How to deal with coordinator failures?
 - worker waits for VOTE-REQ in INIT
 - Worker can time out and abort (coordinator handles it)
 - worker waits for GLOBAL-* message in READY
 - If coordinator fails, workers must **BLOCK** waiting for coordinator to recover and send GLOBAL_* message

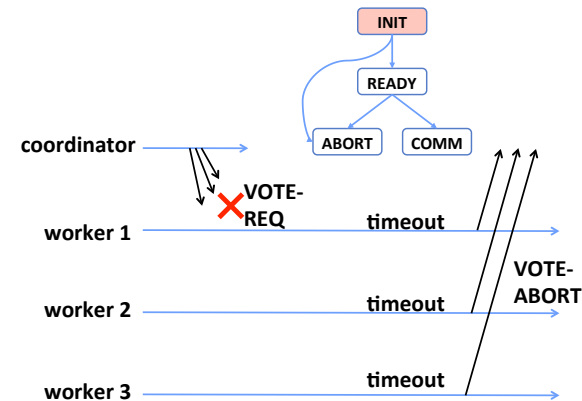


11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.47

Example of Coordinator Failure #1

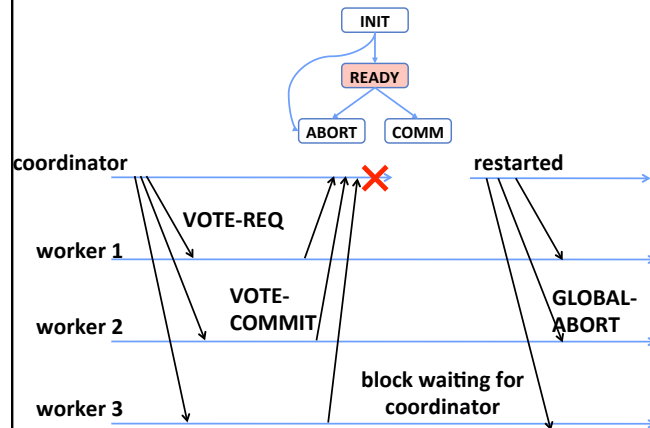


11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.48

Example of Coordinator Failure #2



11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.49

Remembering Where We Were

- All nodes use stable storage to store which state they were in
- Upon recovery, it can restore state and resume:
 - Coordinator aborts in INIT, WAIT, or ABORT
 - Coordinator commits in COMMIT
 - Worker aborts in INIT, READY, ABORT
 - Worker commits in COMMIT

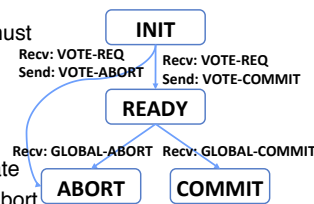
11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.50

Blocking for Coordinator to Recover

- A worker waiting for global decision can ask fellow workers about their state
 - If another worker is in ABORT or COMMIT state then coordinator must have sent GLOBAL-*
 - Thus, worker can safely abort or commit, respectively
 - If another worker is still in INIT state then both workers can decide to abort
 - If all workers are in ready, need to **BLOCK** (don't know if coordinator wanted to abort or commit)



11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.51

Summary

- Correctness criterion for transactions is “serializability”.
 - In practice, we use “conflict serializability”, which is somewhat more restrictive but easy to enforce
- Two phase locking (2PL) and strict 2PL
 - Ensure conflict-serializability for R/W operations
 - If scheduler not conflict-serializable deadlocks
 - Deadlocks can be either detected or prevented
- Two-phase commit (2PC):
 - Ensure atomicity and durability: a transaction is committed/aborted either by all replicas or by none of them

11/5

Ion Stoica CS162 ©UCB Fall 2012

Lec 19.52