Distributed Systems

Lec 15: Crashes and Recovery:
Write-ahead Logging

Slide acks: Dave Andersen

(http://www.cs.cmu.edu/~dga/15-440/F10/lectures/Write-ahead-Logging.pdf)

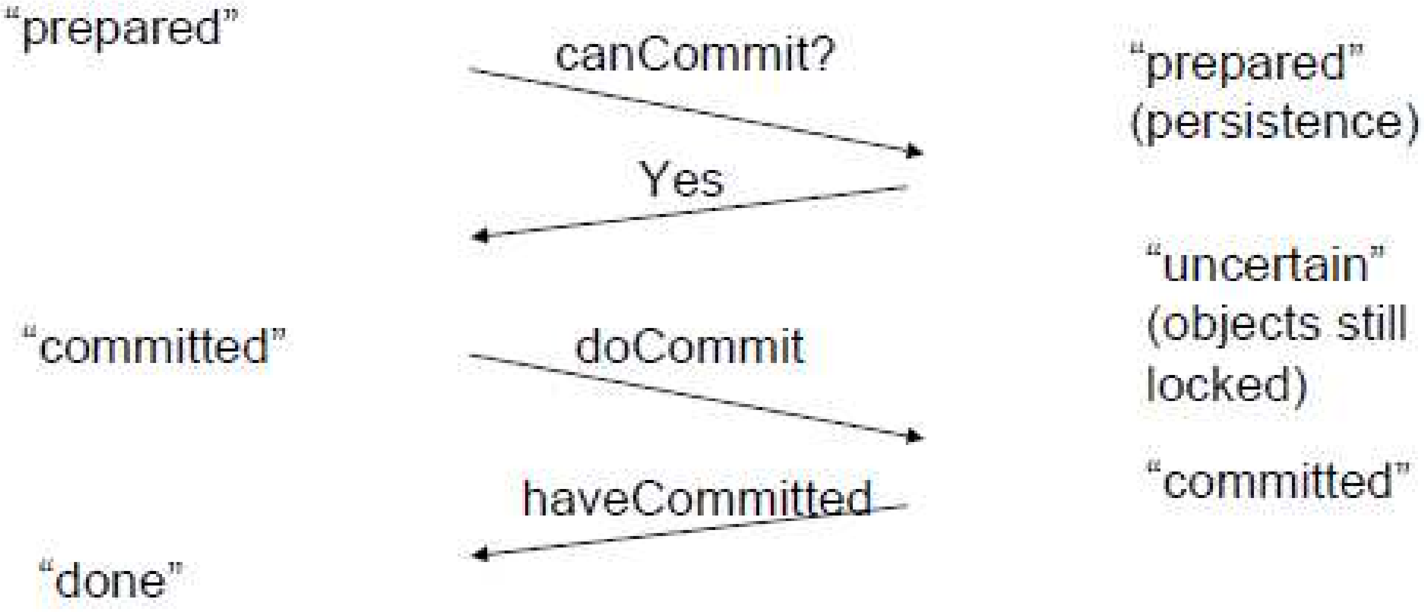
Last Few Times (Reminder)

- Single-operation consistency
  - Strict, sequential, causal, and eventual consistency
- Multi-operation transactions
  - ACID properties: atomicity, consistency, isolation, durability
- Isolation: two-phase locking (2PL)
  - Grab locks for all touched objects, then release all locks
  - Detect or avoid deadlocks by timing out and reverting
- Atomicity: two-phase commit (2PC)
  - Two phases: prepare and commit

Two-Phase Commit (Reminder)

Transaction
Coordinator (TC)
-- just one --

Transaction
Participant (TP)
-- one or more --



"prepared"

canCommit?

"prepared"
(persistence)

Yes

"uncertain"
(objects still
locked)

"committed"

doCommit

"committed"

haveCommitted

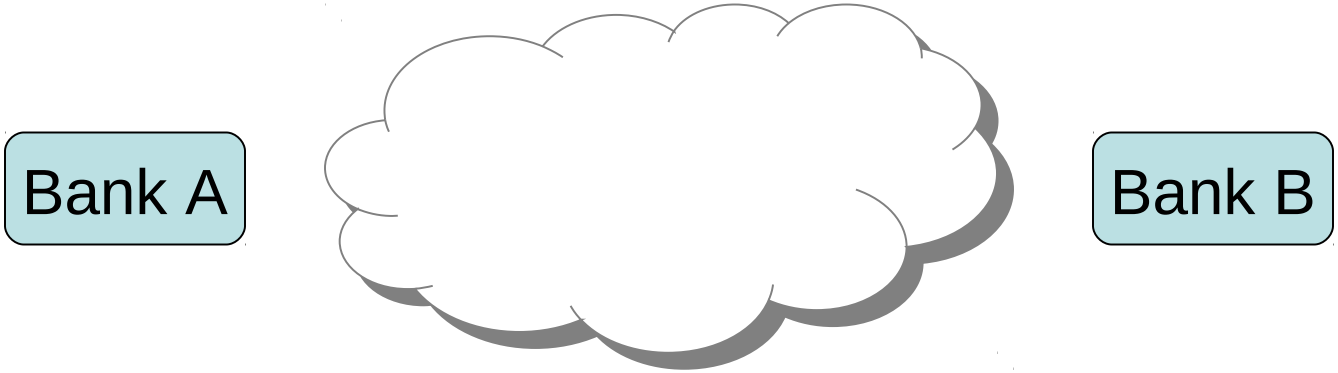
"done"

TP not allowed to Abort after it's agreed to Commit

Example

transfer (X@bank A, Y@bank B, $20)

Suppose initially: X.bal = $100
Y.bal = $3



- Clients desire:
  1. Atomicity: transfer either happens or not at all
  2. Concurrency control: maintain serializability

Example

transfer (X@bank A, Y@bank B, $20)

Suppose initially: X.bal = $100
Y.bal = $3

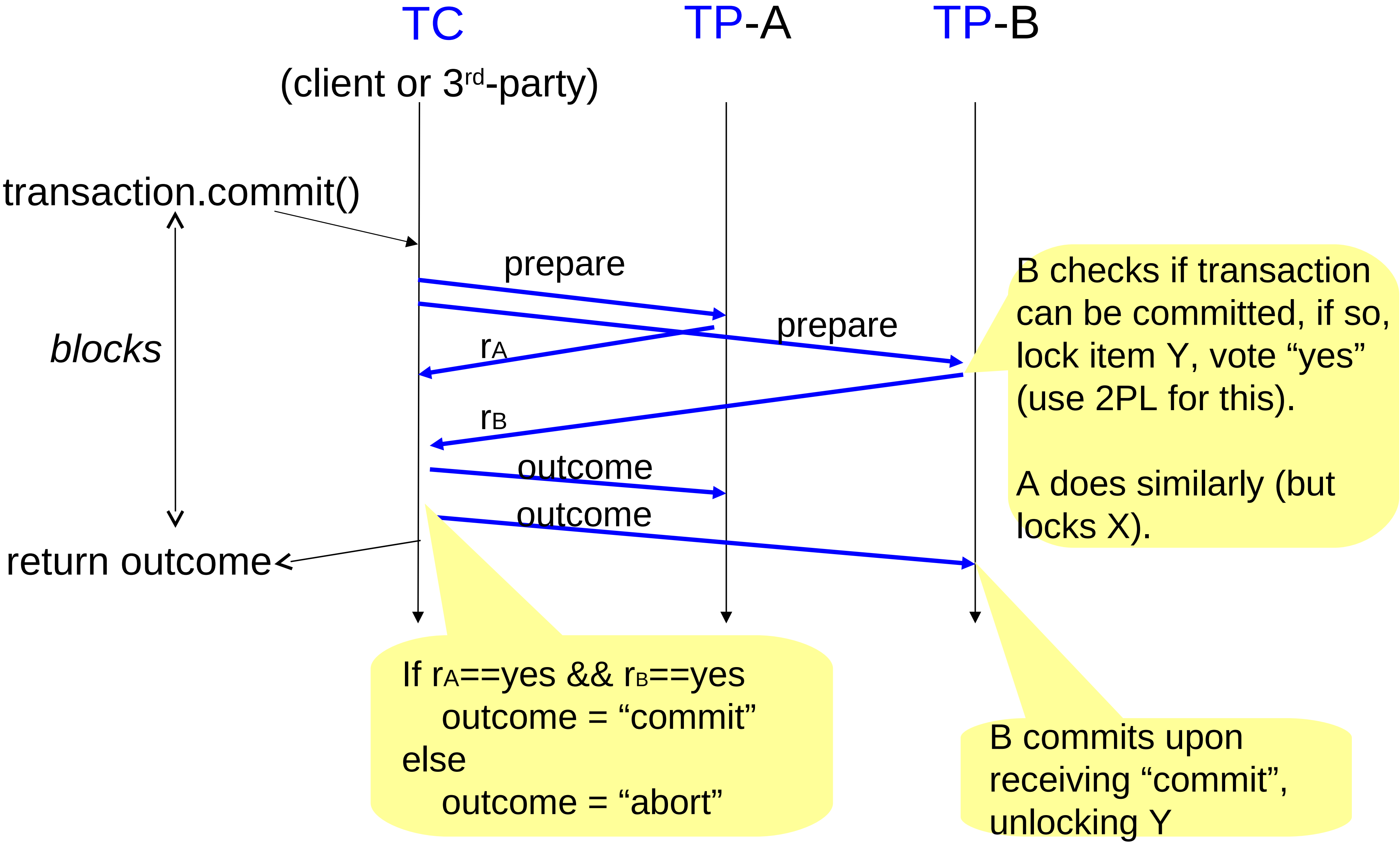
```
int transfer(src, dst, amt) {
    transaction = begin();
    if (src.bal > amt) {
        src.bal -= amt;
        dst.bal += amt;
        return transaction.commit();
    } else {
        transaction.abort();
        return ABORT;
    }
}
```

For simplicity, assume the client code looks like this:

```
int transfer(src, dst, amt) {
    transaction = begin();
    src.bal -= amt;
    dst.bal += amt;
    return transaction.commit();
}
```

The banks can unilaterally decide to COMMIT or ABORT transaction

Example



B checks if transaction can be committed, if so, lock item Y, vote "yes" (use 2PL for this).

A does similarly (but locks X).

```
If rA==yes && rB==yes
    outcome = "commit"
else
    outcome = "abort"
```

B commits upon receiving "commit", unlocking Y

Failure Modes

- Network can fail or be very slow
  - B times out waiting for the outcome
  - TC times out waiting for A/B’s votes
  - How are they supposed to proceed?
- Machines can crash
  - Assume: disks cannot fail
  - Assume: failures are not hard (reboot fixes them)
  - Example crashes: software bug, power loss cause reboot

Today: Fault Recovery

- Goal: Recover state after crash / network failures
- Two requirements for recovery:
  - Correctness:
    - Committed transactions are not lost (durability)
    - Non-committed transactions either continued or aborted
  - Performance:
    - Low overheads
    - Remember that disks are slow (particularly random writes)
- Our plan:
  - Consider first recovery of local system
    - I.e., assume a local transaction (TC=A=B)
  - Then consider recovery in distributed 2PC setting

Local Recovery:
Write-Ahead Logging (a.k.a. Journaling)

Write-Ahead Logging

- In addition to evolving the state in RAM and on disk, keep a separate, on-disk log of all operations
  - Transaction begin, commit, abort
  - All updates (e.g., X = X- $20; Y = Y + $20)
- A transaction’s operations are provisional until “commit” outcome is logged to disk
  - The result of these operations will not be revealed to other clients in meantime (i.e., new value of X will only be revealed after transaction is committed)
- Observation:
  - Disk writes of single pages/blocks are atomic, but disk writes across pages may not be

`begin`/`commit`/`abort` records

- Log Sequence Number (LSN)
  - Usually implicit, the address of the first-byte of the log entry
- LSN of previous record for transaction
  - Linked list of log records for each transaction
- Transaction ID
- Operation type

`update` records

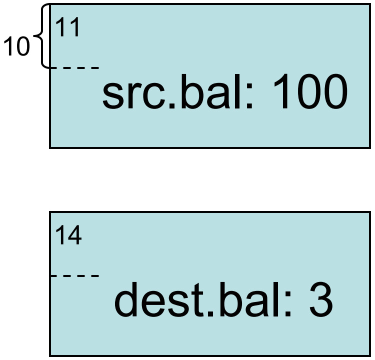
- Need all information to undo and redo the update
  - prevLSN + xID + opType as before
  - The update itself, e.g.:
    - the update location (usually pageID, offset, length)
    - old-value
    - new-value

```
xId = begin();   // suppose xId <- 42
src.bal -= 20;
dest.bal += 20;
commit(xId);
```

Log:

Disk:

Page cache:



10

11

src.bal: 100

14

dest.bal: 3

Transaction table:

Dirty page table:

```
→ xId = begin();   // suppose xId <- 42
  src.bal -= 20;
  dest.bal += 20;
  commit(xId);
```

Log:

780

prevLSN: 0
xId: 42
type: begin

Disk:

10
11

src.bal: 100

14

dest.bal: 3

Page cache:

Transaction table:

42: prevLSN = 780

Dirty page table:

```
   xId = begin();   // suppose xId <- 42
→  src.bal -= 20;
   dest.bal += 20;
   commit(xId);
```

Log:

780
prevLSN: 0
xId: 42
type: begin

⋮

860
prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
src.bal
old-val: 100
new-val: 80

Disk:

10
11
src.bal: 100

14
dest.bal: 3

Page cache:

11
src.bal: 80

Transaction table:

42: prevLSN = 860

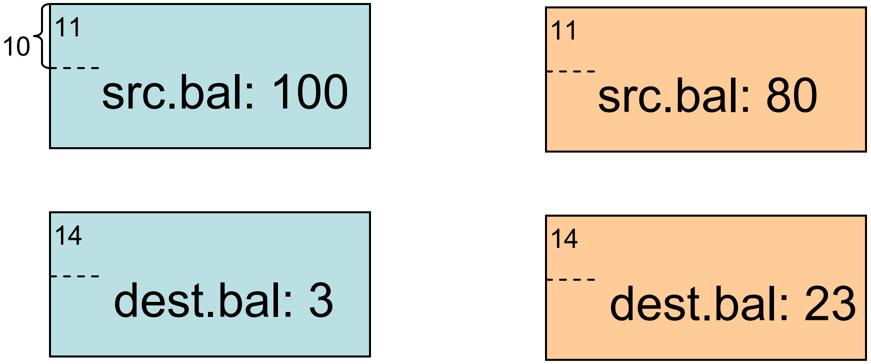
Dirty page table:

11: firstLSN = 860, lastLSN = 860

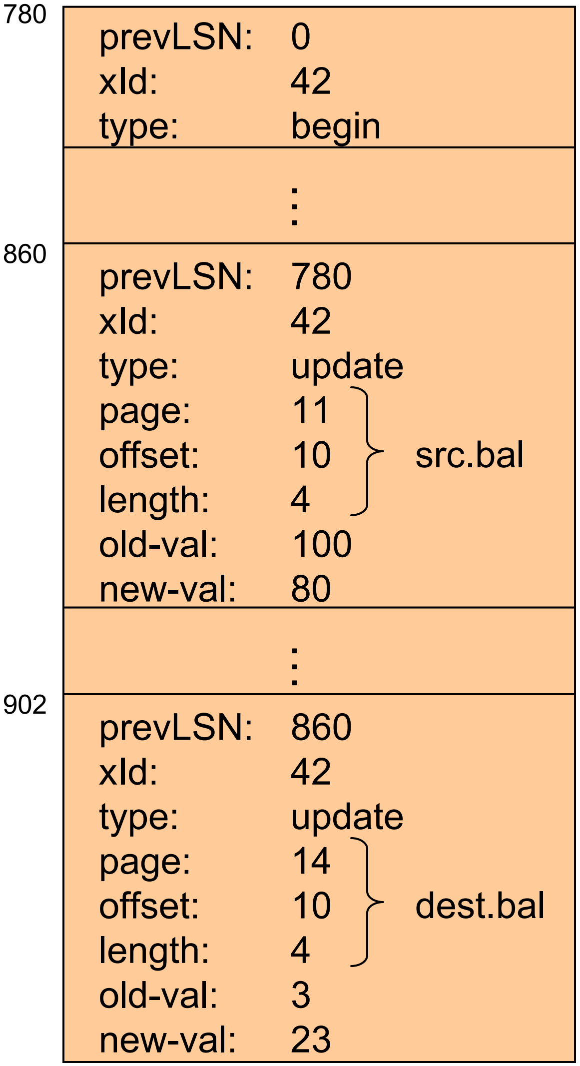
```
   xId = begin();   // suppose xId <- 42
   src.bal -= 20;
→  dest.bal += 20;
   commit(xId);
```

Disk:

Page cache:



Log:



780
prevLSN: 0
xId: 42
type: begin

⋮

860
prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
src.bal
old-val: 100
new-val: 80

⋮

902
prevLSN: 860
xId: 42
type: update
page: 14
offset: 10
length: 4
dest.bal
old-val: 3
new-val: 23

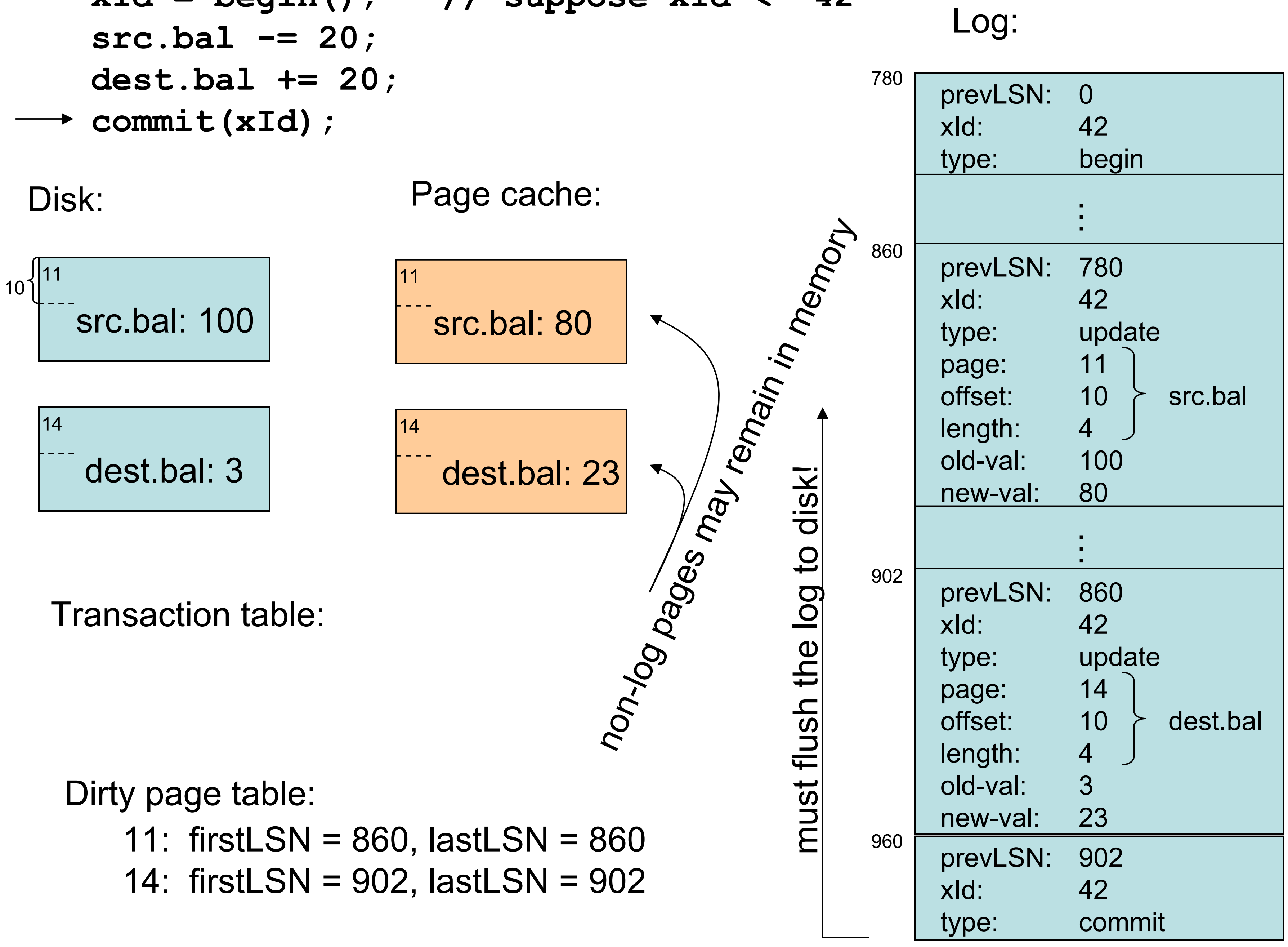
Transaction table:

42: prevLSN = 902

Dirty page table:

11: firstLSN = 860, lastLSN = 860
14: firstLSN = 902, lastLSN = 902

```
   xId = begin();   // suppose xId <- 42
   src.bal -= 20;
   dest.bal += 20;
→  commit(xId);
```



Disk:

Page cache:

non-log pages may remain in memory

must flush the log to disk!

Log:

780
prevLSN: 0
xId: 42
type: begin

⋮

860
prevLSN: 780
xId: 42
type: update
page: 11
offset: 10
length: 4
old-val: 100
new-val: 80
} src.bal

⋮

902
prevLSN: 860
xId: 42
type: update
page: 14
offset: 10
length: 4
old-val: 3
new-val: 23
} dest.bal

960
prevLSN: 902
xId: 42
type: commit

Transaction table:

Dirty page table:

11: firstLSN = 860, lastLSN = 860
14: firstLSN = 902, lastLSN = 902

The tail of the log

- The tail of the log can be kept in memory until a transaction commits
  - …or a buffer page is flushed to disk

Recovering from simple failures

- e.g., system crash
  - – For now, assume we can read the log
- “Analyze” the log
- Redo all (usually) transactions (forward)
  - – Repeating history!
  - – Use new-value in byte-level update records
- Undo uncommitted transactions (backward)
  - – Use old-value in byte-level update records

Why redo all operations?

- (Even the loser transactions)
- Interaction with concurrency control
  - Bring system back to a former state
- Generalizes to logical operations
  - Any operation with undo and redo operations
  - Can be much faster than byte-level logging

The performance of WAL

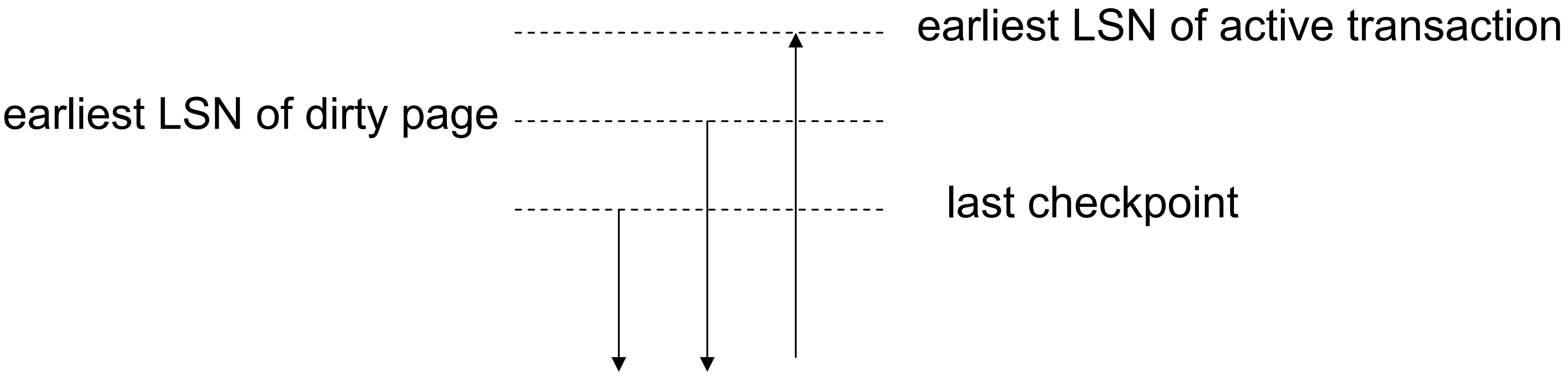
- Problems:
  - Must write disk twice?
    - Not always
  - For byte-level update logging, must know old value for the update record
- Writing the log is sequential
  - Might actually improve performance
    - Can acknowledge a write/commit as soon as the log is written

Improvements to this WAL

- Store LSN of last write on each data page
  - Can avoid unnecessary redoes
- Log checkpoint records
  - Flush buffer cache? Record which pages are in memory?
- Log recovery actions (CLR)
  - Speeds up recovery from repeated failures
- Ordered / metadata-only logging
  - Avoids needing to save old-value of files

Checkpoint records

- Can start analysis with last checkpoint
- Records:
  - Table of active transactions
  - Table of dirty pages in memory
    - And the earliest LSN that might have affected them



earliest LSN of active transaction

earliest LSN of dirty page

last checkpoint

Distributed Recovery:
Recovery in Two-Phase Commit

Recovery in Two-Phase Commit

- Easy: just log the state-changes
  - Participants: prepared, uncertain, committed/aborted
  - Coordinator: prepared, committed/aborted, done
  - The messages are idempotent!
    - In recovery, resend whatever message was next
    - If coordinator and uncommitted: abort

- Two cases:
  - Recovery after crashes and reboots
  - Recovery after timeouts

Handling Crash and Reboot

- Nodes cannot back out if commit is decided
- TC crashes just after deciding “commit”
  - Cannot forget about its decision after reboot
- A/B crashes after sending “yes”
  - Cannot forget about their response after reboot

Handling Crash and Reboot

- All nodes must log protocol progress
- What and when does TC log to disk?
- What and when does A/B log to disk?

Recovery Upon Reboot

- If TC finds no “commit” on disk, abort
- If TC finds “commit”, commit
- If A/B finds no “yes” on disk, abort
- If A/B finds “yes”, run termination protocol to decide

Handling Timeouts

- Examples:
  - TC times out waiting for A’s response
  - A times out waiting for TC’s outcome message
- Btw, timeouts aren’t necessarily due to network problems
  - They could due to slow, overloaded hosts

Handling Timeouts on A/B

- TC times out waiting for A (or B)’s “yes/no” response
- Can TC unilaterally decide to commit?
- Can TC unilaterally decide to abort?

Handling timeout on TC

- If B responded with “no” …
  - Can it unilaterally abort?
- If B responded with “yes” …
  - Can it unilaterally abort?
  - Can it unilaterally commit?

Possible termination protocol

- Execute termination protocol if B times out on TC and has voted “yes”
- B sends “status” message to A
  - If A has received “commit”/”abort” from TC …
  - If A has not responded to TC, …
  - If A has responded with “no”, …
  - If A has responded with “yes”, …

Resolves most failure cases except sometimes when TC fails

What about other failures?

- What if the log fails?
- What if the machine room is flooded?

- Solution: replication of the log or the data
- But handling replication with strong semantic is tough

- Next time: replicated state machines, consensus, and Paxos