CS514: Intermediate Course in Computer Systems

Lecture 37: April 21, 2003

Transactions…

Transactions

CS514

- The most important reliability technology for client-server systems
- Today
  - How transactional systems really work
  - Implementation considerations
  - Limitations and performance challenges
  - WS_TRANSACTION
- Overall, the topic will span two lectures

Transactions

CS514

- There are several perspectives on how to achieve reliability
  - One approach focuses on reliability of communication channels and leaves application-oriented issues to the client or server – “stateless”
  - Major alternative is to focus on the *data* managed by a system. Stateful version yields transactional system
  - A third option exploits non-transactional replication. We’ve already seen this

Transactions in the real world

CS514

- In cs514 lectures, transactions are treated at the same level as other techniques
- But in the real world, transactions represent 90% of the existing market for distributed systems!!
  - The web is gradually starting to shift the balance (not by reducing the size of the transaction market but by growing so fast that it is catching up)
  - But even on the web, we use transactions when we buy products

Recall transactional model

CS514

- Applications are coded in a stylized way:
  - *begin* transaction
  - Perform a series of *read*, *update* operations
  - Terminate by *commit* or *abort*.
- Terminology
  - The application is the transaction manager
  - The transaction is the sequence of operations issued by the transaction manager while it runs
  - The data manager is presented with operations from concurrently active transactions
  - It schedules them in an interleaved but serializable order

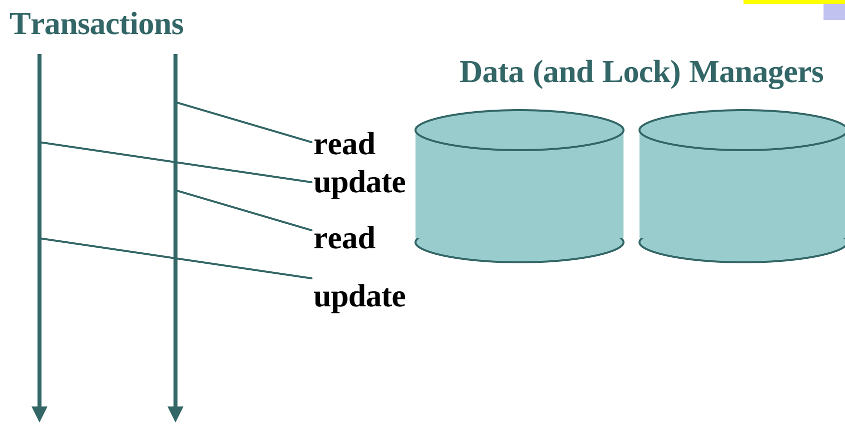
A side remark

CS514

- Each transaction is built up incrementally
  - Application runs
  - And as it runs, it issues operations
  - The data manager sees them one by one
- But often we talk as if we knew the whole thing at one time
  - We're careful to do this in ways that make sense
  - In any case, we usually don't need to say anything until a "commit" is issued

Transaction and Data Managers

CS514



transactions are stateful: transaction "knows" about database contents and updates

Typical transactional program

CS514

```
begin transaction;
    x = read("x-values", ....);
    y = read("y-values", ....);
    z = x+y;
    write("z-values", z, ....);
commit transaction;
```

What about the locks?

CS514

- Unlike other kinds of distributed systems, transactional systems *lock* the data they access
- They obtain these locks as they run:
  - Before accessing "x" get a lock on "x"
  - Usually we assume that the application knows enough to get the right kind of lock. It is not good to get a read lock if you'll later need to update the object
- In clever applications, one lock will often cover many objects

Locking rule

CS514

- Suppose that transaction *T* will access object *x.*
  - We need to know that first, *T* gets a lock that "covers" *x*
- What does coverage entail?
  - We need to know that if any other transaction *T'* tries to access *x* it will attempt to get the *same lock*

Examples of lock coverage

CS514

- We could have one lock per object
- … or one lock for the whole database
- … or one lock for a category of objects
  - In a tree, we could have one lock for the whole tree associated with the root
  - In a table we could have one lock for row, or one for each column, or one for the whole table
- All transactions must use the same rules!
- And if you will update the object, the lock must be a "write" lock, not a "read" lock

Transactional Execution Log

CS514

- As the transaction runs, it creates a history of its actions. Suppose we were to write down the sequence of operations it performs.
- Data manager does this, one by one
- This yields a "schedule"
  - Operations and order they executed
  - Can infer order in which transactions ran
- Scheduling is called "concurrency control"

Observations

CS514

- Program runs "by itself", doesn't talk to others
- All the work is done in one program, in straight-line fashion. If an application requires running several programs, like a C compilation, it would run as several separate transactions!
- The persistent data is maintained in files or database relations external to the application

Serializability

CS514

- Means that effect of the interleaved execution is indistinguishable from some possible serial execution of the committed transactions
- For example: *T1 and T2 are interleaved but it "looks like" T2 ran before T1*
- Idea is that transactions can be coded to be correct if run in isolation, and yet will run correctly when executed concurrently (and hence gain a speedup)

Need for serializable execution

CS514

T_1:	begin	read(x)	read(y)	write(x)	commit
T_2:	begin	read(x)	write(x)	write(y)	commit

DB: read(x) read(x) write(x) write(y) read(y) write(x)

Data manager interleaves operations to improve concurrency

Non serializable execution

CS514

T_1:	begin	read(x)	read(y)	write(x)	commit
T_2:	begin	read(x)	write(x)	write(y)	commit

DB: read(x) read(x) ~~write(x)~~ write(y) read(y) write(x)

Unsafe! Not serializable

Problem: transactions may "interfere". Here, T_2 changes x, hence T_1 should have either run first (read <u>and</u> write) or after (reading the changed value).

Serializable execution

CS514

T_1: begin read(x) read(y) write(x) commit

T_2: begin read(x) write(x) write(y) commit

↓

DB: read(x) write(x) read(x) write(y) read(y) write(x)

Data manager interleaves operations to improve concurrency but schedules them so that it looks as if one transaction ran at a time. This schedule "looks" like T_2 ran first.

Atomicity considerations

CS514

- If application ("transaction manager") crashes, treat as an abort
- If data manager crashes, abort any non-committed transactions, but committed state is persistent
  - Aborted transactions leave no effect, either in database itself or in terms of indirect side-effects
  - Only need to consider committed operations in determining serializability

How can data manager sort out the operations?

CS514

- We need a way to distinguish different transactions
  - In example, T_1 and T_2
- Solve this by requiring an agreed upon RPC argument list (“interface”)
  - Each operation is an RPC from the transaction mgr to the data mgr
  - Arguments include the transaction “id”
- WS_TRANSACTION standardizes this for web services. More later in this lecture.

Components of transactional system

CS514

- Runtime environment: responsible for assigning transaction id’s and labeling each operation with the correct id.
- Concurrency control subsystem: responsible for scheduling operations so that outcome will be serializable
- Data manager: responsible for implementing the database storage and retrieval functions

Transactions at a “single” database

CS514

- Normally use 2-phase locking or timestamps for concurrency control
- Intentions list tracks “intended updates” for each active transaction
- Write-ahead log used to ensure all-or-nothing aspect of commit operations
- Can achieve thousands of transactions per second

Two-phase locking: how it works

CS514

- Transaction must have a lock on each data item it will access.
  - Gets a “write lock” if it will (ever) update the item
  - Use “read lock” if it will (only) read the item. Can’t change its mind!
- Obtains all the locks it needs while it runs and hold onto them even if no longer needed
- Releases locks only after making commit/abort decision and only after updates are persistent

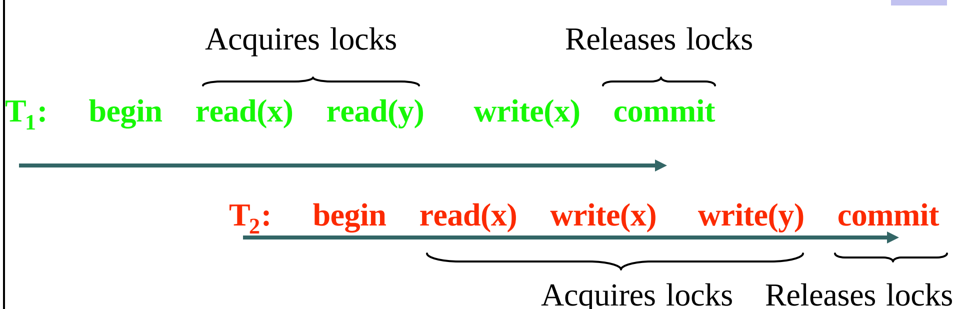
Why do we call it "two phase"?

CS514

- Locks are acquired during the pre-commit phase
- Locks are only released after the commit decision
  - Read locks don't conflict with each other (hence T' can read x even if T holds a read lock on x)
  - Update locks conflict with everything (are "exclusive")

Two-phase Locking

CS514



Notes

CS514

- Notice that locks must be kept even if the same objects won't be revisited
  - This can be a problem in long-running applications!
  - Also becomes an issue in systems that crash and then recover
    - Often, they "forget" locks when this happens
    - Called "broken locks". We say that a crash may "break" current locks…

Why does 2PL imply serializability?

CS514

- Suppose that T' will perform an operation that conflicts with an operation that T has done:
  - T' will update some data item that T read or updated
  - T updated some item and T' will read or update it
- T must have had a lock on that item that conflicts with the lock that T' wants
- T won't release it until it commits or aborts
- ... hence T' will wait until T commits or aborts

Acyclic conflict graph implies serializability

CS514

- Can represent conflicts between operations and between locks by a graph (e.g. first T1 reads x and then T2 writes x)
- If this graph is acyclic, can easily show that transactions are serializable
- Two-phase locking produces acyclic conflict graphs

Two-phase locking is "pessimistic"

CS514

- Acts to prevent non-serializable schedules from arising: pessimistically assumes conflicts are fairly likely
- Can deadlock, e.g. T1 reads x then writes y; T2 reads y then writes x. This doesn't always deadlock but it is capable of deadlocking
  - Overcome by aborting if we wait for too long,
  - Or by designing transactions to obtain locks in a known and agreed upon ordering

Contrast: Timestamped approach

CS514

- Using a fine-grained clock, assign a “time” to each transaction, uniquely. E.g. T1 is at time 1, T2 is at time 2
- Now data manager tracks temporal history of each data item, responds to requests as if they had occured at time given by timestamp
- At commit stage, make sure that commit is consistent with serializability and, if not, abort

Example of when we abort

CS514

- T1 runs, updates x, setting to 3
- T2 runs concurrently but has a larger timestamp. It reads x=3
- T1 eventually aborts
- ... T2 must abort too, since it read a value of x that is no longer a committed value
  - Called a cascaded abort since abort of T_1 triggers abort of T_2

Pros and cons of approaches

CS514

- Locking scheme works best when conflicts between transactions are common and transactions are short-running
- Timestamped scheme works best when conflicts are rare and transactions are long-running
- Weihl has suggested hybrid approaches but these are not common in real systems

Intentions list concept

CS514

- Idea is to separate *persistent state* of database from the *updates that have been done but have yet to commit*
- Intensions list may simply be the in-memory cached database state
- Say that transactions intends to commit these updates, if indeed it commits

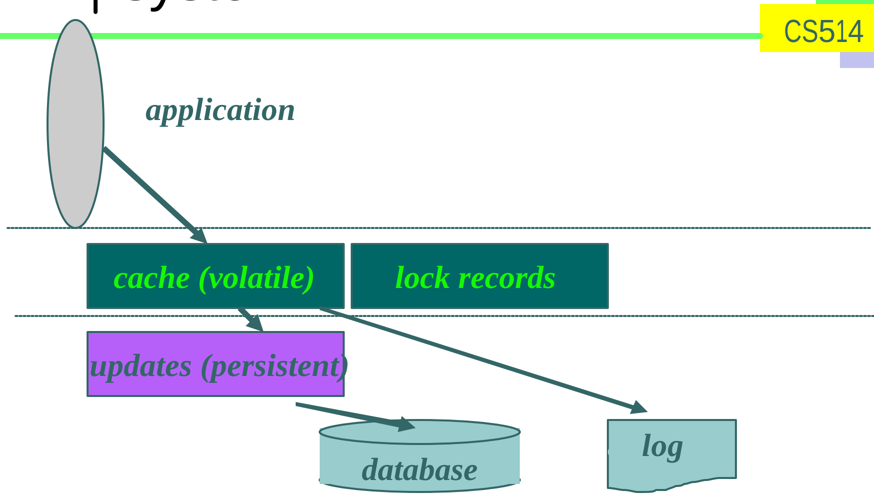
Role of write-ahead log

CS514

- Used to save either old or new state of database to either permit abort by rollback (need old state) or to ensure that commit is all-or-nothing (by being able to repeat updates until all are completed)
- Rule is that log must be written before database is modified
- After commit record is persistently stored and all updates are done, can erase log contents

Structure of a transactional system

CS514



Recovery?

CS514

- Transactional data manager reboots
- It rescans the log
  - Ignores non-committed transactions
  - Reapplies any updates
  - These must be "idempotent"
    - Can be done many times with same effect as a single time
    - E.g. x := 3, but not x := x.prev+1
- Then clears log records
- (In normal use, log records are deleted once transaction commits)

Transactions in distributed systems

CS514

- Notice that client and data manager might not run on same computer
  - Both may not fail at same time
  - Also, either could timeout waiting for the other in normal situations
- When this happens, we normally abort the transaction
  - Exception is a timeout that occurs while commit is being processed
  - If server fails, one effect of crash is to break locks *even for read-only access*

Transactions in distributed systems

CS514

- What if data is on multiple servers?
  - In a non-distributed system, transactions run against a single database system
    - Indeed, many systems structured to use just a single operation – a “one shot” transaction!
  - In distributed systems may want one application to talk to multiple databases

Transactions in distributed systems

CS514

- Main issue that arises is that now we can have multiple database servers that are touched by one transaction
- Reasons?
  - Data spread around: each owns subset
  - Could have replicated some data object on multiple servers, e.g. to load-balance read access for large client set
  - Might do this for high availability
- Solve using 2-phase commit protocol!

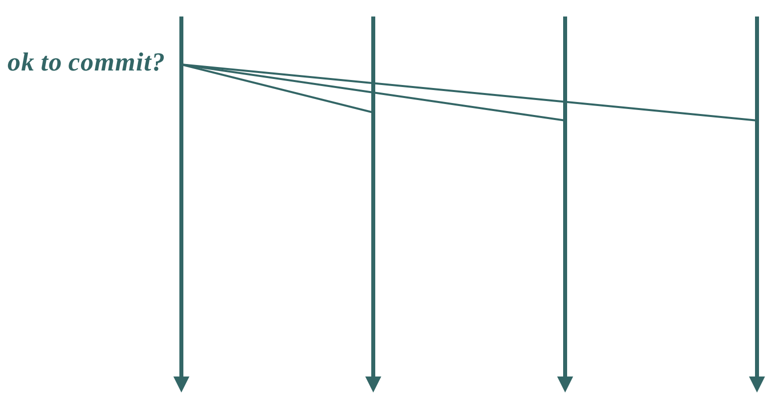
Two-phase commit in transactions

CS514

- Phase 1: transaction wishes to commit. Data managers force updates and lock records to the disk (e.g. to the log) and then say prepared to commit
- Transaction manager makes sure all are prepared, then says commit (or abort, if some are not)
- Data managers then make updates permanent or rollback to old values, and release locks

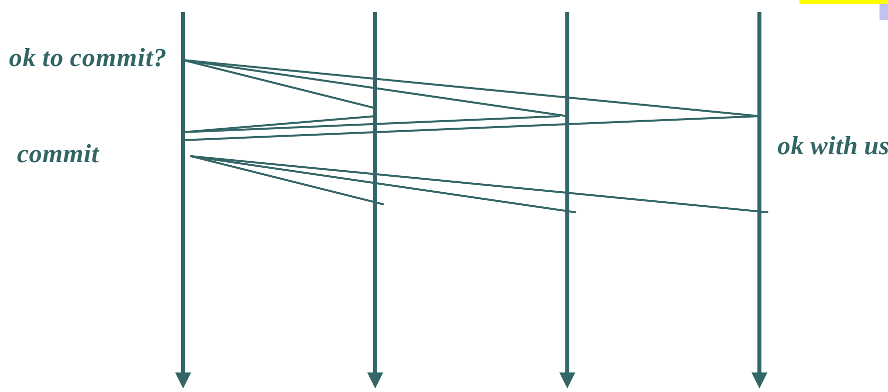
Commit protocol illustrated

CS514



Commit protocol illustrated

CS514



Note: garbage collection protocol not shown here

Unilateral abort

CS514

- Any data manager can unilaterally abort a transaction until it has said “prepared”
- Useful if transaction manager seems to have failed
- Also arises if data manager crashes and restarts (hence will have lost any non-persistent intended updates and locks)
- Implication: even a data manager where only reads were done must participate in 2PC protocol!

Notes on 2PC

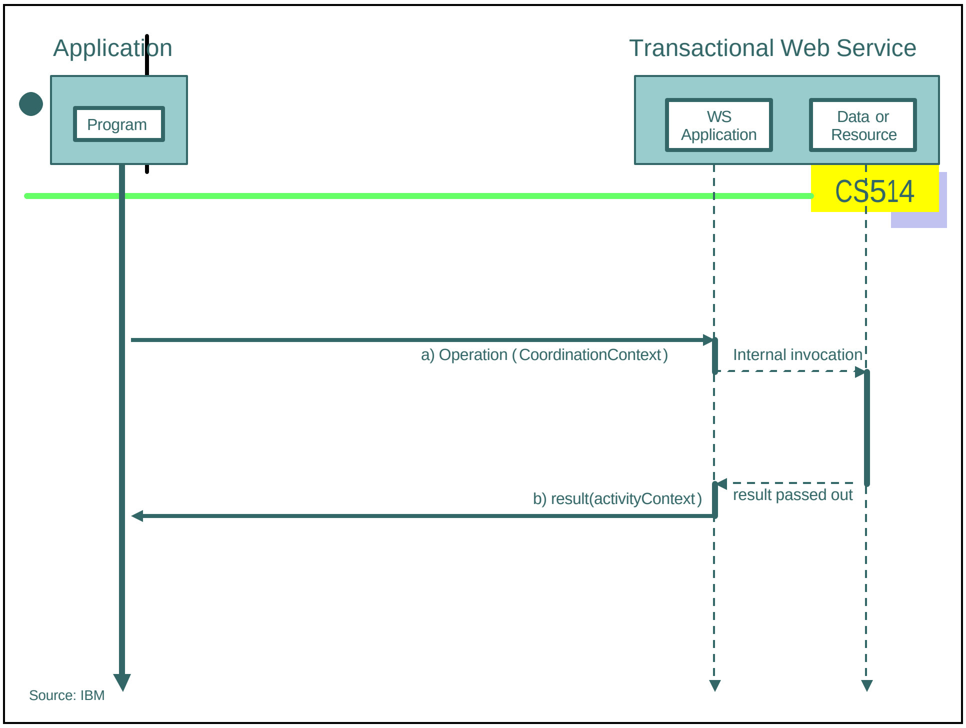
CS514

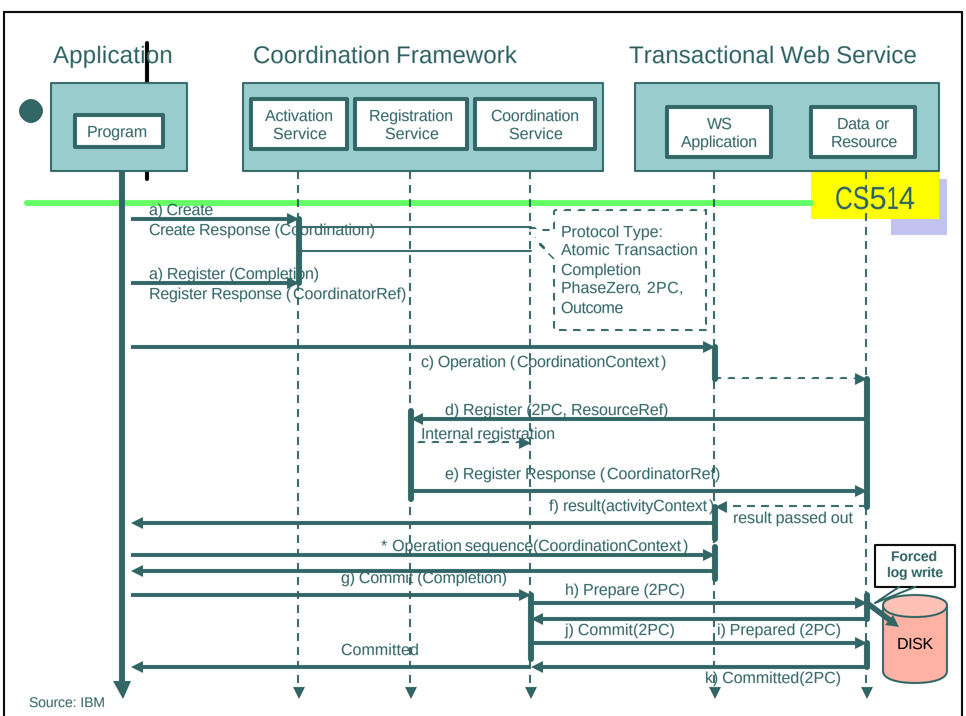
- Although protocol looks trivial, there are subtle issues! For example, who do you ask to find out how it terminated (after you crash and then reboot)
- Also, not a cheap protocol
  - Considered costly because of latency: few systems can pay this price
  - Hence most "real" systems run transactions only against a single server
  - For replication, they use log-based schemes

WS_Transaction

CS514

- Web Services optionally offer a transactional interface to users
  - So a client can talk to a database or a similar subsystem through a Web Service interface
  - Web Service and database must support the transactional mechanisms.
- We won't discuss "business transactions"


Application
Transactional Web Service
Program
WS Application
Data or Resource
CS514
a) Operation (CoordinationContext)
Internal invocation
b) result(activityContext)
result passed out
Source: IBM


Application
Coordination Framework
Transactional Web Service
Program
Activation Service
Registration Service
Coordination Service
WS Application
Data or Resource
CS514
a) Create
Create Response (Coordination)
Protocol Type: Atomic Transaction Completion PhaseZero, 2PC, Outcome
a) Register (Completion)
Register Response (CoordinatorRef)
c) Operation (CoordinationContext)
d) Register (2PC, ResourceRef)
Internal registration
e) Register Response (CoordinatorRef)
f) result(activityContext)
result passed out
* Operation sequence(CoordinationContext)
Forced log write
g) Commit (Completion)
h) Prepare (2PC)
j) Commit(2PC)
i) Prepared (2PC)
Committed
DISK
k) Committed(2PC)
Source: IBM

Impact on the server?

CS514

- A server must
  - Track the “identification” of requests
  - Maintain a way to back out of uncommitted requests
- At commit
  - Force uncommitted data to disk when a “prepare” request is first seen
  - Check status of pending requests after a reboot to see how things worked out

Impact on the client?

CS514

- Mostly a performance impact
- With a single server
  - Some analysts anticipate a 5x slowdown.
  - Perhaps this is pessimistic…
- With multiple servers
  - Overhead can be extremely high
  - *Multi-second* delays likely to be common
  - Stems from the “2 phase commit” delay