Advanced Operating Systems
Lecture notes

Dr. Dongho Kim
Dr. Tatyana Ryutov
University of Southern California
Information Sciences Institute

CSci555: Advanced Operating Systems
Lecture 3 – Distributed Concurrency, Transactions, Deadlock
9 September 2005

Dr. Tatyana Ryutov
University of Southern California
Information Sciences Institute
(lecture slides written by Dr. Katia Obraczka)

Administration

- Use web discussion board
- Email with questions to csci555@usc.edu
  - CSci555 in Subject of email
  - Try web board for questions too
- Assignment 1 is due Thursday

Today

- Concurrency and Synchronization [Hauser et al.]
- Transactions [Spector et al.]
- Distributed Deadlocks [Chandy et al.]
- Replication [Birman and Gifford]
- Time in Distributed Systems [Lamport and Jefferson]

Concurrency Control and Synchronization

- How to control and synchronize possibly conflicting operations on shared data by concurrent processes?
- First, some terminology.
  - Processes.
  - Light-weight processes.
  - Threads.
  - Tasks.

Processes

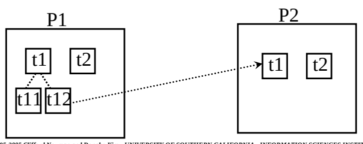
- Text book:
  - Processing activity associated with an execution environment, ie, address space and resources (such as communication and synchronization resources).

Threads

- OS abstraction of an activity/task.
  - Execution environment expensive to create and manage.
  - Multiple threads share single execution environment.
  - Single process may spawn multiple threads.
  - Maximize degree of concurrency among related activities.
  - Example: multi-threaded servers allow concurrent processing of client requests.

Other Terminology

- Process versus task/thread.
  - Process: heavy-weight unit of execution.
  - Task/thread: light-weight unit of execution.



Threads Case Study 1

- Hauser et al.
- Examine use of user-level threads in 2 OS's:
  - Xerox Parc's Cedar (research).
  - GVX (commercial version of Cedar).
- Study dynamic thread behavior.
  - Classes of threads (eternal, worker, transient)
  - Number of threads.
  - Thread lifetime.

Thread Paradigms

- Different categories of usage:
  - Defer work: thread does work not vital to the main activity.
    - Examples: printing a document, sending mail.
  - Pumps: used in pipelining; use output of a thread as input and produce output to be consumed by another task.
  - Sleepers: tasks that repeatedly wait for an event to execute; e.g., check for network connectivity every x seconds.

Synchronization

- So far, how one defines/activates concurrent activities.
- But how to control access to shared data and still get work done?
- Synchronization via:
  - Shared data [DSM model].
  - Communication [MP model].

Synchronization by Shared Data

- Primitives
  - Semaphores.
  - Conditional critical regions.
  - Monitors.

structure ↓ flexibility ↑

Synchronization by MP

- Explicit communication.
- Primitives send and receive
  - *Blocking send, blocking receive*: sender and receiver are blocked until message is delivered (redezvous)
  - *Nonblocking send, blocking receive*: sender continues processing receiver is blocked until the requested message arrives
  - *Nonblocking send, nonblocking receive*: messages are sent to a shared data structure consisting of queues (mailboxes)

  Deadlocks ?

- Mailboxes one process sends a message to the mailbox and the other process picks up the message from the mailbox

  Example:
  Send (mailbox, msg)
  Receive (mailbox, msg)

Transactions

- Database term.
  - Execution of program that accesses a database.
- In distributed systems,
  - Concurrency control in the client/server model.
  - From client's point of view, sequence of operations executed by server in servicing client's request in a single step.

Transaction Properties

- ACID:

  Atomicity: a transaction is an atomic unit of processing and it is either performed entirely or not at all

  Consistency: a transaction's correct execution must take the database from one correct state to another

  Isolation: the updates of a transaction must not be made visible to other transactions until it is committed

  Durability: if transaction commits, the results must never be lost because of subsequent failure

Transaction Atomicity

- "All or nothing".
- Sequence of operations to service client's request are performed in one step, ie, either all of them are executed or none are.
- Start of a transaction is a continuation point to which it can roll back.
- Issues:
  - Multiple concurrent clients: "isolation".
    1. Each transaction accesses resources as if there were no other concurrent transactions.
    2. Modifications of the transaction are not visible to other resources before it finishes.
    3. Modifications of other transactions are not visible during the transaction at all.
  - Server failures: "failure atomicity".

Transaction Features

- Recoverability: server should be able to "roll back" to state before transaction execution.
- Serializability: transactions executing concurrently must be interleaved in such a way that the resulting state is equal to some serial execution of the transactions
- Durability: effects of transactions are permanent.
  - A completed transaction is always persistent (though values may be changed by later transactions).
  - Modified resources must be held on persistent storage before transaction can complete. May not just be disk but can include battery-backed RAM.

Concurrency Control

Maintain transaction serializability:

- establish order of concurrent transaction execution
- Interleave execution of operations to ensure serializability

- Basic Server operations: read or write.
- 3 mechanisms:
  - Locks.
  - Optimistic concurrency control.
  - Timestamp ordering.

Locks

- Lock granularity: affects level of concurrency.
  - 1 lock per shared data item.
  - Shared Read
    - Exists when concurrent transactions granted READ access
    - Issued when transaction wants to read and exclusive lock not held on item
  - Exclusive Write
    - Exists when access reserved for locking transaction
    - Used when potential for conflict exists
    - Issued when transaction wants to update unlocked data
- Many Read locks simultaneously possible for a given item, but only one Write lock
- Transaction that requests a lock that cannot be granted must wait

Lock Implementation

- Server lock manager
  - Maintains table of locks for server data items.
  - *Lock* and *unlock* operations.
  - Clients *wait* on a lock for given data until data is released; then client is signalled.
  - Each client's request runs as separate server thread.

Deadlock

- Use of locks can lead to deadlock.
- Deadlock: each transaction waits for another transaction to release a lock forming a wait cycle.
- Deadlock condition: cycle in the wait-for graph.
- Deadlock prevention and detection.
  - require all locks to be acquired at once
    Problems?
  - Ordering of data items: once a transaction locks an item, it cannot lock anything occurring earlier in the ordering
- Deadlock resolution: lock timeout.

$$\begin{tikzcd}
\boxed{T1} \arrow[rr] & & \boxed{T2} \arrow[dl] \\
& \boxed{T3} \arrow[ul] &
\end{tikzcd}$$

Optimistic Concurrency Control 1

- Assume that most of the time, probability of conflict is low.
- Transactions allowed to proceed in parallel until *close transaction* request from client.
- Upon c*lose transaction*, checks for conflict; if so, some transactions aborted.

Optimistic Concurrency 2

- Read phase
  - Transactions have *tentative* version of data items it accesses.
    - Transaction reads data and stores in local variables
    - Any writes are made to local variables without updating the actual data
  - *Tentative* versions allow transactions to abort without making their effect permanent.
- Validation phase
  - Executed upon *close transaction*.
  - Checks serially equivalence.
  - If validation fails, conflict resolution decides which transaction(s) to abort.

Optimistic Concurrency 3

- Write phase
  - If transaction is validated, all of its tentative versions are made permanent.
  - Read-only transactions commit immediately.
  - Write transactions commit only after their tentative versions are recorded in permanent storage.

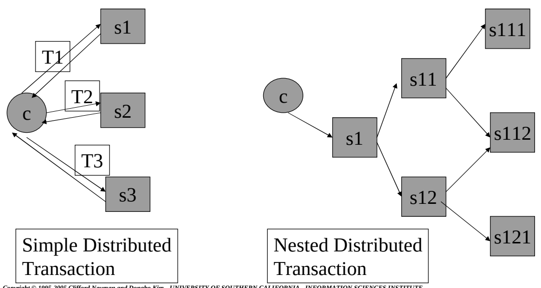
Timestamp Ordering

- Uses timestamps to order transactions accessing same data items according to their starting times.
- Assigning timestamps:
  - Clock based: assign global unique time stamp to each transaction
  - Monotonically increasing counter.
- Some time stamping necessary to avoid "livelock": where a transaction cannot acquire any locks because of unfair waiting algorithm

Local versus Distributed Transactions

- Local transactions:
  - All transaction operations executed by single server.
- Distributed transactions:
  - Involve multiple servers.
- Both local and distributed transactions can be simple or nested.
  - Nesting: increase level of concurrency.

Distributed Transactions 1


s1
T1
T2
c
s2
T3
s3
Simple Distributed Transaction
c
s1
s11
s12
s111
s112
s121
Nested Distributed Transaction

Distributed Transactions 2

- Transaction coordinator
  - First server contacted by client.
  - Responsible for aborting/committing.
  - Adding *workers*.
- Workers
  - Other servers involved report their results to the coordinator and follow its decisions.

Atomicity in Distributed Transactions

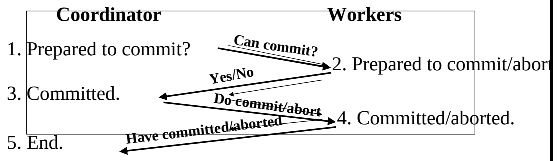
- Harder: several servers involved.
- Atomic commit protocols
  - 1-phase commit
    - Example: coordinator sends "commit" or "abort" to workers; keeps re-broadcasting until it gets ACK from all of them that request was performed.
    - Inefficient.
    - How to ensure that all of the servers vote + that they all reach the same decision. It is simple if no errors occur, but the protocol must work correctly even when server fails, messages are lost, etc.

2-Phase Commit 1

- First phase: voting
  - Each server votes to commit or abort transaction.
- Second phase: carrying out joint decision.
  - If any server votes to abort, joint decision is to abort.

2-Phase Commit 2

- **Phase I:**
  - **Each participant votes for the transaction to be committed or aborted.**
  - **Participants must ensure to carry out its part of commit protocol. (prepared state).**
  - **Each participant saves in permanent storage all of the objects that it has altered in transaction to be in 'prepared state'.**
- **Phase II:**
  - **Every participant in the transaction carries out the joint decision.**
  - **If any one participant votes to abort, then the decision is to abort.**
  - **If all participants vote to commit, then the decision is to commit.**



Concurrency Control in Distributed Transactions 1

- **Locks**
  - **Each server manages locks for its own data.**
  - **Locks cannot be released until transaction committed or aborted on all servers involved.**
  - **Lock managers in different servers set their locks independently, there are chances of different transaction orderings.**
  - **The different ordering lead to cyclic dependencies between transactions and a distributed deadlock situation.**
  - **When a deadlock is detected, a transaction is aborted to resolve the deadlock**

Concurrency Control in Distributed Transactions 2

- **Timestamp Ordering**
  - **Globally unique timestamps.**
    - **Coordinator issues globally unique TS and passes it around.**
    - **TS: <server id, local TS>**
  - **Servers are jointly responsible for ensuring that they performed in a serially equivalent manner.**
  - **Clock synchronization issues**

Concurrency Control in Distributed Transactions 3

- **Optimistic concurrency control**
  - **Each transaction should be validated before it is allowed to commit.**
  - **The validation at all servers takes place during the first phase of the 2-Phase Commit Protocol.**

Camelot [Spector et al.]

- **Supports execution of distributed transactions.**
- **Specialized functions:**
  - **Disk management**
    - **Allocation of large contiguous chunks.**
  - **Recovery management**
    - **Transaction abort and failure recovery.**
  - **Transaction management**
    - **Abort, commit, and nest transactions.**

Distributed Deadlock 1

- **When locks are used, deadlock can occur.**
- **Circular wait in wait-for graph means deadlock.**
- **Centralized deadlock detection, prevention, and resolutions schemes.**
  - **Examples:**
    - **Detection of cycle in wait-for graph.**
    - **Lock timeouts: hard to set TO value, aborting unnecessarily.**

Distributed Deadlock 2

- Much harder to detect, prevent, and resolve. Why?
  - No global view.
  - No central agent.
  - Communication-related problems
    - Unreliability.
    - Delay.
    - Cost.

Distributed Deadlock Detection

- Cycle in the global wait-for graph.
- Global graph can be constructed from local graphs: hard!
  - Servers need to communicate to find cycles.
- Example from book (page 533).

Distributed Deadlock Detection Algorithms 1

- [Chandy et al.]
- Message sequencing is preserved.
- Resource versus communication models.
  - Resource model
    - Processes, resources, and controllers.
    - Process requests resource from controller.
  - Communication model
    - Processes communicate directly via messages (request, grant, etc) requesting resources.

Resource versus Communication Models

- In resource model, controllers are deadlock detection agents; in communication model, processes.
- In resource model, process cannot continue until all requested resources granted; in communication model, process cannot proceed until it can communicate with at least one process it's waiting for.
- Different models, different detection alg's.

Distributed Deadlock Detection Schemes

- Graph-theory based.
- Resource model: deadlock when cycle among dependent processes.
- Communication model: deadlock when knot (all vertices that can be reached from *i* can also reach *i*) of waiting processes.

Deadlock Detection in Resource Model

- Use probe messages to follow edges of wait-for graph (aka edge chasing).
- Probe carries transaction wait-for relations representing path in global wait-for graph.

Deadlock Detection Example

1. **Server 1 detects transaction T is waiting for U, which is waiting for data from server 2.**
2. **Server 1 sends probe T->U to server 2.**
3. **Server 2 gets probe and checks if U is also waiting; if so (say for V), it adds V to probe T->U->V. If V is waiting for data from server 3, server 2 forwards probe.**
4. **Paths are built one edge at a time.**

Before forwarding probe, server checks for cycle (e.g., T->U->V->T).

5. **If cycle detected, a transaction is aborted.**

Replication 1

- **Keep more than one copy of data item.**
- **Technique for improving performance in distributed systems.**
- **In the context of concurrent access to data, replicate data for increase availability.**
  - **Improved response time.**
  - **Improved availability.**
  - **Improved fault tolerance.**

Replication 2

- **But nothing comes for free.**
- **What's the tradeoff?**
  - **Consistency maintenance.**
- **Consistency maintenance approaches:**
  - **Lazy consistency (gossip approach).**
    - An operation call is executed at just one replica; updating of other replicas happens by lazy exchange of "gossip" messages.
  - **Quorum consensus is based on voting techniques.**
  - **Process group.**

$\downarrow$ Stronger consistency

Quorum Consensus

- **Goal: prevent partitions from from producing inconsistent results.**
- **Quorum: subgroup of replicas whose size gives it the right to carry out operations.**
- **Quorum consensus replication:**
  - **Update will propagate successfully to a subgroup of replicas.**
  - **Other replicas will have outdated copies but will be updated off-line.**

Weighted Voting [Gifford] 1

- **Every copy assigned a number of votes (weight assigned to a particular replica).**
- **Read: Must obtain *R* votes to read from any up-to-date copy.**
- **Write: Must obtain write quorum of *W* before performing update.**

Weighted Voting 2

- **W > 1/2 total votes, R+W > total votes.**
- **Ensures non-null intersection between every read quorum and write quorum.**
- **Read quorum guaranteed to have current copy.**
- **Freshness is determined by version numbers.**

Weighted Voting 3

- On read:
  - Try to find enough copies, ie, total votes no less than R. Not all copies need to be current.
  - Since it overlaps with write quorum, at least one copy is current.
- On write:
  - Try to find set of up-to-date replicas whose votes no less than W.
  - If no sufficient quorum, current copies replace old ones, then update.

ISIS 1

- Goal: provide programming environment for development of distributed systems.
- Assumptions:
  - DS as a set of processes with disjoint address spaces, communicating over LAN via MP.
  - Processes and nodes can crash.
  - Partitions may occur.

ISIS 2

- Distinguishing feature: group communication mechanisms
  - Process group: processes cooperating in implementing task.
  - Process can belong to multiple groups.
  - Dynamic group membership.

Virtual Synchrony

- Real synchronous systems
  - Events (e.g., message delivery) occur in the same order everywhere.
  - Expensive and not very efficient.
- Virtual synchronous systems
  - Illusion of synchrony.
  - Weaker ordering guarantees when applications allow it.

Atomic Multicast 1

- All destinations receive a message or none.
- Primitives:
  - ABCAST: delivers messages atomically and in the same order everywhere.
  - CBCAST: causally ordered multicast.
    - "Happened before" order.
    - Messages from given process in order.
  - GBCAST
    - used by system to manage group addressing.

Other Features

- Process groups
  - Group membership management.
- Broadcast and group RPC
  - RPC-like interface to CBCAST, ABCAST, and GBCAST protocols.
  - Delivery guarantees
    - Caller indicates how many responses required.
      - No responses: asynchronous.
      - 1 or more: synchronous.

Implementation

- Set of library calls on top of UNIX.
- Commercially available.
- In the paper, example of distributed DB implementation using ISIS.
- HORUS: extension to WANs.

Time in Distributed Systems

- Notion of time is critical.
- "Happened before" notion.
  - Example: concurrency control using TSs.
  - "Happened before" notion is not straightforward in distributed systems.
    - No guarantees of synchronized clocks.
    - Communication latency.

Event Ordering

- Lamport defines partial ordering (→):
  1. If X and Y events occurred in the same process, and X comes before Y, then X→Y.
  2. Whenever X sends a message to Y, then X→Y.
  3. If X→Y and Y→Z, then X→Z.
  4. X and Y are concurrent if X↛Y and Y↛Z

Causal Ordering

- "Happened before" also called causal ordering.
- In summary, possible to draw happened-before relationship between events if they happen in same process or there's chain of messages between them.

Logical Clocks

- Monotonically increasing counter.
- No relation with real clock.
- Each process keeps its own logical clock Cp used to timestamp events.

Causal Ordering and Logical Clocks

1. *Cp* incremented before each event.
   Cp=Cp+1.
2. When *p* sends message *m*, it piggybacks *t=Cp*.
3. When q receives *(m, t),* it computes:
   Cq=max(Cq, t) before timestamping message receipt event.

Example: text book page 398.

Total Ordering

- Extending partial to total order.
- Global timestamps:
  - (T_a, p_a), where T_a is local TS and p_a is the process id.
  - (T_a, p_a) < (T_b, p_b) iff $T_a < T_b$ or $T_a=T_b$ and $p_a<p_b$
  - Total order consistent with partial order.

Virtual Time [Jefferson]

- Time warp mechanism.
- May or may not have connection with real time.
- Uses optimistic approach, i.e., events and messages are processed in the order received: "look-ahead".

Local Virtual Clock

- Process virtual clock set to TS of next message in input queue.
- If next message's TS is in the past, rollback!
  - Can happen due to different computation rates, communication latency, and unsynchronized clocks.

Rolling Back

- Process goes back to TS(last message).
- Cancels all intermediate effects of events whose TS > TS(last message).
- Then, executes forward.
- Rolling back is expensive!
  - Messages may have been sent to other processes causing them to send messages, etc.

Anti-Messages 1

- For every message, there is an anti-message with same content but different *sign*.
- When sending message, message goes to receiver input queue and a copy with "-" sign is enqueued in the sender's output queue.
- Message is retained for use in case of roll back.

Anti-Message 2

- Message + its anti-message = 0 when in the same queue.
- Processes must keep log to "undo" operations.

Implementation

- Local control.
- Global control
  - How to make sure system as a whole progresses.
  - “Committing” errors and I/O.
  - Avoid running out of memory.

Global Virtual Clock

- Snapshot of system at given real time.
- Minimum of all local virtual times.
- Lower bound on how far processes rollback.
- Purge state before GVT.
- GVT computed concurrently with rest of time warp mechanism.
  - Tradeoff?