CIS 415: Operating Systems
Distributed Coordination

Spring 2013
Prof. Kevin Butler
Event Ordering

• *Happened-before* relation (denoted by $\rightarrow$).

  ‣ If $A$ and $B$ are events in the same process, and $A$ was executed before $B$, then $A \rightarrow B$.

  ‣ If $A$ is the event of sending a message by one process and $B$ is the event of receiving that message by another process, then $A \rightarrow B$.

  ‣ If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$. 
Happened-Before

• Properties of the happened-before relation
  ‣ If A, B are events in the same process and A executes before B then $A \rightarrow B$
  ‣ If A is a send message event from a process and B is a receive message event from another process then $A \rightarrow B$
  ‣ If $A \rightarrow B$ and $B \rightarrow C$ then $A \rightarrow C$ (what property is this?)

• If events A and B are not related by $\rightarrow$ then they can execute concurrently (no effect of A on B)
Implementing

- Associate a timestamp with each system event
  - Require that for every pair of events A and B, if A → B, then the timestamp of A is less than the timestamp of B
- Associate logical (Lamport) clock $LC_i$ with process $P_i$
  - Implement as a simple counter incremented between any two successive events executed within a process.
- Process advances logical clock when receiving message with timestamp > current value of LC
- If $TS_A == TS_B$, events are concurrent (use process ID to break ties and create a total ordering)
Distributed Mutual Exclusion

• Assumptions for DME
  ‣ The system consists of $n$ processes; each process $P_i$ resides at a different processor.
  ‣ Each process has a critical section that requires mutual exclusion.

• Requirement
  ‣ If $P_i$ is executing in its critical section, then no other process $P_j$ is executing in its critical section.
DME: Centralized Approach

1. One of the processes in the system is chosen to coordinate the entry to the critical section.

2. A process that wants to enter its critical section sends a request message to the coordinator.

3. The coordinator decides which process can enter critical section next, sends that process a reply message.

4. When the process receives a reply message from the coordinator, it enters its critical section.

5. After exiting its critical section, process sends release message to coordinator, proceeds with its execution.
This scheme requires three messages per critical-section entry:

- request
- reply
- release
DME: Fully Distributed Approach

1. When process $P_i$ wants to enter its critical section, it generates a new timestamp, $TS$, and sends the message \textit{request} ($P_i, TS$) to all other processes in the system.

2. When process $P_j$ receives a \textit{request} message, it may reply immediately or it may defer sending a reply back.

3. When process $P_i$ receives a \textit{reply} message from all other processes in the system, it can enter its critical section.

4. After exiting its critical section, the process sends \textit{reply} messages to all its deferred requests.
Process P0

Enter CS

Request(P0, TS)

Process P1

Not in CS

Reply(P1)

Process P2

Not in CS

Request(P0, TS)

Reply(P2)

Process P3

In CS

Finish CS

Request(P0, TS)

Reply(P3)
The decision whether process $P_j$ replies immediately to a $\text{request}(P_i, TS)$ message or defers its reply is based on three factors:

- If $P_j$ is in its critical section, then it defers its reply to $P_i$.
- If $P_j$ does not want to enter its critical section, then it sends a reply immediately to $P_i$.
- If $P_j$ wants to enter its critical section but has not yet entered it, then it compares its own request timestamp with the timestamp $TS$.
  - If its own request timestamp is greater than $TS$, then it sends a reply immediately to $P_i$ ($P_i$ asked first).
  - Otherwise, the reply is deferred.
Fully Distributed: The Good

- Freedom from Deadlock
- Freedom from starvation
  - entry to CS is scheduled by TS ordering
  - ordering ensures that processes are served FCFS
- The number of messages per critical-section entry is $2 \times (n - 1)$.

This is the minimum number of required messages per critical-section entry when processes act independently and concurrently.
Fully Distributed: The Bad

• Processes need to know the identity of all other processes in the system, which makes the dynamic addition and removal of processes more complex.

• If one process fails, entire scheme collapses
  ‣ Mitigated by continuously monitoring the state of all the processes in the system

• Processes that have not entered their critical section must pause frequently to assure other processes that they intend to enter the critical section
  ‣ Protocol is best suited for small, stable sets of cooperating processes
Atomicity

- Either all the operations associated with a program unit are executed to completion, or none performed.
- Ensuring atomicity in a distributed system requires a transaction coordinator, which is responsible for:
  - Starting execution of the transaction.
  - Breaking transaction into subtransactions, and distributing these subtransactions to the appropriate sites for execution.
  - Coordinating the termination of the transaction, which may result in the transaction being committed at all sites or aborted at all sites.
Two-Phase Commit Protocol

• Assumes fail-stop model
• Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
• When the protocol is initiated, the transaction may still be executing at some of the local sites.
• The protocol involves all the local sites at which the transaction executed.
• Example: Let $T$ be a transaction initiated at site $S_i$ and let the transaction coordinator at $S_i$ be $C_i$. 
Phase 1: Obtaining a Decision

• $C_i$ adds $<\text{prepare } T>$ record to the log.

• $C_i$ sends $<\text{prepare } T>$ message to all sites.

• When a site receives a $<\text{prepare } T>$ message, the transaction manager determines if it can commit the transaction.
  
  ‣ If no: add $<\text{no } T>$ record to the log and respond to $C_i$ with $<\text{abort } T>$.

  ‣ If yes:
    • add $<\text{ready } T>$ record to the log.
    • force all log records for $T$ onto stable storage.
    • transaction manager sends $<\text{ready } T>$ message to $C_i$. 
Phase 1 (Cont.)

- Coordinator collects responses
  - All respond “ready”, decision is *commit*.
  - At least one response is “abort”, decision is *abort*.
  - At least one participant fails to respond within time out period, decision is *abort*.
Phase 2: Recording Decision in the Database

- Coordinator adds a decision record
  
  `<abort T>` or `<commit T>`

  to its log and forces record onto stable storage.

- Once that record reaches stable storage it is *irrevocable* (even if failures occur).

- Coordinator sends a message to each participant informing it of the decision (commit or abort).

- Participants take appropriate action locally.
2PC – Site Failure

• The log contains a \(<\text{commit } T>\) record. In this case, the site executes \text{redo}(T).

• The log contains an \(<\text{abort } T>\) record. In this case, the site executes \text{undo}(T).

• The contains a \(<\text{ready } T>\) record; consult \(C_j\). If \(C_j\) is down, site sends \text{query-status } T message to the other sites.

• The log contains no control records concerning \(T\). In this case, the site executes \text{undo}(T).
2PC – Coordinator Failure

• If an active site contains a <commit $T$> record in its log, then $T$ must be committed.

• If an active site contains an <abort $T$> record in its log, then $T$ must be aborted.

• If some active site does not contain the record <ready $T$> in its log then the failed coordinator $C_i$ cannot have decided to commit $T$. Rather than wait for $C_i$ to recover, it is preferable to abort $T$.

• All active sites have a <ready $T$> record in their logs, but no additional control records. In this case we must wait for the coordinator to recover.
  
  ▶ Blocking problem – $T$ is blocked pending the recovery of site $S_i$. 
Concurrent Control

- Modify the centralized concurrency schemes to accommodate the distribution of transactions.

- Transaction manager coordinates execution of transactions (or subtransactions) that access data at local sites.

- Local transaction only executes at that site.

- Global transaction executes at several sites.
Locking Protocols

• Can use the two-phase locking protocol in a distributed environment by changing how the lock manager is implemented.

• Nonreplicated scheme – each site maintains a local lock manager which administers lock and unlock requests for those data items that are stored in that site.
  ‣ Simple implementation involves two message transfers for handling lock requests, and one message transfer for handling unlock requests.
  ‣ Deadlock handling is more complex.
Single-Coordinator Approach

• A single lock manager resides in a single chosen site, all lock and unlock requests are made at that site.
  ✓ Simple implementation
  ✓ Simple deadlock handling
  ✗ Possibility of bottleneck
  ✗ Vulnerable to loss of concurrency controller if single site fails

• Multiple-coordinator approach distributes lock-manager function over several sites.
Majority Protocol

• Avoids drawbacks of central control by dealing with replicated data in a decentralized manner.

• More complicated to implement

• Deadlock-handling algorithms must be modified; possible for deadlock to occur in locking only one data item.
Biased Protocol

• Similar to majority protocol, but requests for shared locks prioritized over requests for exclusive locks.

• Less overhead on read operations than in majority protocol; but has additional overhead on writes.

• Like majority protocol, deadlock handling is complex.
• One of the sites at which a replica resides is designated as the primary site. Request to lock a data item is made at the primary site of that data item.

• Concurrency control for replicated data handled in a manner similar to that of unreplicated data.

• Simple implementation, but if primary site fails, the data item is unavailable, even though other sites may have a replica.
• Generate unique timestamps in distributed scheme:
  ‣ Each site generates a unique local timestamp.
  ‣ The global unique timestamp is obtained by concatenation of the unique local timestamp with the unique site identifier.
  ‣ Use a *logical clock* defined within each site to ensure the fair generation of timestamps.

• Timestamp-ordering scheme – combine the centralized concurrency control timestamp scheme with the 2PC protocol to obtain a protocol that ensures serializability with no cascading rollbacks.
Generation of Unique Timestamps

- Local unique timestamp
- Site identifier
- Global unique identifier
Deadlock Prevention

• Resource-ordering deadlock-prevention – define a global ordering among the system resources.
  ‣ Assign a unique number to all system resources.
  ‣ A process may request a resource with unique number $i$ only if it is not holding a resource with a unique number greater than $i$.
  ‣ Simple to implement; requires little overhead.

• Banker’s algorithm – designate one of the processes in the system process that maintains the information necessary to carry out Banker’s algorithm.
  ‣ Also implemented easily, but may require too much overhead.
Timestamped Deadlock-Prevention

• Each process $P_i$ is assigned a unique priority number.

• Priority numbers are used to decide whether a process $P_i$ should wait for a process $P_j$; otherwise, $P_i$ is rolled back.

• The scheme prevents deadlocks. For every edge $P_i \rightarrow P_j$ in the wait-for graph, $P_i$ has a higher priority than $P_j$. Thus a cycle cannot exist.
Wait-Die Scheme

• Based on a nonpreemptive technique.

• If Pi requests a resource currently held by Pj, Pi is allowed to wait only if it has a smaller timestamp than does Pj (Pi is older than Pj). Otherwise, Pi is rolled back (dies).

• Example: Suppose that processes P1, P2, and P3 have timestamps 5, 10, and 15 respectively.
  ‣ if P1 request a resource held by P2, then P1 will wait.
  ‣ If P3 requests a resource held by P2, then P3 will be rolled back.
Wound-Wait Scheme

• Based on a preemptive technique; counterpart to the wait-die system.

• If $P_i$ requests a resource currently held by $P_j$, $P_i$ is allowed to wait only if it has a larger timestamp than does $P_j$ ($P_i$ is younger than $P_j$). Otherwise $P_j$ is rolled back ($P_j$ is wounded by $P_i$).

• Example: Suppose that processes $P_1$, $P_2$, and $P_3$ have timestamps 5, 10, and 15 respectively.
  ‣ If $P_1$ requests a resource held by $P_2$, then the resource will be preempted from $P_2$ and $P_2$ will be rolled back.
  ‣ If $P_3$ requests a resource held by $P_2$, then $P_3$ will wait.
Two Local Wait-For Graphs

Site $S_1$

Site $S_2$
Global Wait-For Graph
Centralized Deadlock Detection

• Each site keeps a local wait-for graph. The nodes of the graph correspond to all the processes that are currently either holding or requesting any of the resources local to that site.

• A global wait-for graph is maintained in a single coordination process; this graph is the union of all local wait-for graphs.
Constructing Wait-for Graph

• There are three different options (points in time) when the wait-for graph may be constructed:
  ‣ 1. Whenever a new edge is inserted or removed in one of the local wait-for graphs.
  ‣ 2. Periodically, when a number of changes have occurred in a wait-for graph.
  ‣ 3. Whenever the coordinator needs to invoke the cycle-detection algorithm.

• Unnecessary rollbacks may occur as a result of false cycles.
Local and Global Wait-For Graphs

site $S_1$

site $S_2$

coordinator
Summary

• Distributed computers add a layer of complexity starting with how to order messages

• Solutions such as two-phase commit allow for distributed synchronization and atomicity

• Deadlock detection can be centralized, somewhat distributed, or fully distributed
CIS 415: Operating Systems
Cloud Computing & Virtualization

Spring 2013
Adam Bates
A View From Above

- Clouds provide *computing as a utility*.
- Three essential aspects of cloud computing.
  - **Scalability**: the illusion of infinite resources.
  - **Elasticity**: pay-as-you go, only for what you need.
  - **No money down**: elimination of up-front capital.
- Business Incentive: cloud providers *statistically multiplex* customers onto the same machine, ensuring high utilization and profitability.
Virtualization

• Clouds services sell different kinds of virtualized hardware and platforms.

• Infrastructure-as-a-Service Clouds rent virtual machines (VMs) by the hour.

• A virtual hypervisor is a lightweight OS that manages resource sharing between VMs.
Xen: CPU Sharing

- **Algorithm:** *Borrowed Virtual Time (BVT).*
- **Proportional Sharing:** Different VM’s can receive different priority weights (e.g. dom1 = 20%, dom2 = 80%).
- **Work Conserving:** If only one VM has work to do, it can take all of the CPU.
- **Low-Latency Support:** Quick wake-up that prioritizes VMs that have just received an event. Improves performance of I/O-intensive work and TCP congestion controls.
Xen: Networking

• All domU packets pass through dom0 as they enter/exit the machine.
• Packet ordering algorithm is Simple Round Robin.
• domU’s enqueue packets onto packet rings inside of dom0. Lots of overhead!
• Hypervisor enforces isolation, prevents guest VMs from mapping to one another’s memory frames.

• Guest VMs can’t access physical memory! Maintain *shadow tables* to keep a mapping between guest’s view of RAM and the physical RAM.
Of course, different hypervisors behave differently.
Questions?

amb@cs.uoregon.edu