



Distributed Systems

- What is a distributed system?
 - need a “network”
 - wire or wireless, Ethernet, high-speed interconnect, Myrnet, Infiniband, ...
 - CPU, disk (boot, data), memory, services, applications all accessed “remotely”, not “locally”
- Why use distributed systems?
 - share resources efficiently
 - parallel processing, manual or automated
 - reliability
 - communication over distance
 - Metcalfe's Law
- Examples
 - Internet, clusters, grids, P2P, Kazaa, Napster, [SETI@home](#)
- Types and topologies
 - LAN, WAN, Star, bus, tree, ...





Distributed System Problems

- same as single-system / single-OS...
 - process scheduling, memory/resource management, synchronization, caching, efficiency, administration/management
- ...but some are harder to solve
 - component failure
 - time synchronization
 - what TIME is it?
 - control coordination
 - who's in charge?
 - validating actions





Deutsch's 8 Fallacies of Distributed Computing

- the network is reliable
- latency is zero
- bandwidth is infinite
- the network is secure
- topology doesn't change
- there is ONE network administrator
- transport cost is zero
- the network is homogeneous





Distributed Systems Papers and Projects

- must relate to **core OS** and **core DS** concepts
- required:
 - 2 short papers, AND...
 - ...ONE of the following
 - long research paper
 - project
 - final exam
- select topic from course website, or propose & get approval for DS topic of your choice
- submission requirements
 - PDF preferred for report
 - send to hfoxwell@cs.gmu.edu
 - include appropriate citations & bibliography
 - use formatted tables or fixed-width font for columnar data





Chapter 17 Distributed Coordination

- Event Ordering
- Mutual Exclusion
- Atomicity
- Concurrency Control
- Deadlock Handling
- Election Algorithms
- Reaching Agreement





Event Ordering

- Happened-before relation (denoted by $<$).
 - If A and B are events in the same process, and A was executed before B, then $A < B$.
 - 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 $A < B$.
 - If $A < B$ and $B < C$ then $A < C$.
 - **simple, right?**





Implementation of $<$

- 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.
- Within each process P_i a logical clock, LC_i is associated. The logical clock can be implemented as a simple counter that is incremented between any two successive events executed within a process.
- A process advances its logical clock when it receives a message whose timestamp is greater than the current value of its logical clock.
 - **why?**
- If the timestamps of two events A and B are the same, then the events are concurrent. We may use the process identity numbers to break ties and to create a total ordering.





Distributed Mutual Exclusion (DME)

- Assumptions
 - 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.
- We present two algorithms to ensure the mutual exclusion execution of processes in their critical sections.





DME: Centralized Approach

- One of the processes in the system is chosen to coordinate the entry to the critical section.
- A process that wants to enter its critical section sends a request message to the coordinator.
- The coordinator decides which process can enter the critical section next, and it sends that process a reply message.
- When the process receives a reply message from the coordinator, it enters its critical section.
- After exiting its critical section, the process sends a release message to the coordinator and proceeds with its execution.
- This scheme requires three messages per critical-section entry:
 - request
 - reply
 - release
- what makes this difficult?





DME: Fully Distributed Approach

- When process P_i wants to enter its critical section, it generates a new timestamp, TS , and sends the message request (P_i, TS) to all other processes in the system.
- When process P_j receives a request message, it may reply immediately or it may defer sending a reply back.
- When process P_i receives a reply message from all other processes in the system, it can enter its critical section.
- After exiting its critical section, the process sends reply messages to all its deferred requests.
- **what makes this difficult?**





DME: Fully Distributed Approach (Cont.)

- The decision whether process P_j replies immediately to a 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.





Desirable Behavior of Fully Distributed Approach

- Freedom from Deadlock is ensured.
- Freedom from starvation is ensured, since entry to the critical section is scheduled according to the timestamp ordering. The timestamp ordering ensures that processes are served in a first-come, first served order.
- 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.

- this is a lot of overhead





Three Undesirable Consequences

- The 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 of the processes fails, then the entire scheme collapses. This can be dealt with 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. **This protocol is therefore suited for small, stable sets of cooperating processes.**





Atomicity

- Either all the operations associated with a program unit are executed to completion, or none are performed.
- Ensuring atomicity in a distributed system requires a transaction coordinator, which is responsible for the following:
 - Starting the execution of the transaction.
 - Breaking the transaction into a number of subtransactions, and distribution 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 (2PC)

- 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 $\langle \text{prepare } T \rangle$ record to the log.
- C_i sends $\langle \text{prepare } T \rangle$ message to all sites.
- When a site receives a $\langle \text{prepare } T \rangle$ message, the transaction manager determines if it can commit the transaction.
 - If no: add $\langle \text{no } T \rangle$ record to the log and respond to C_i with $\langle \text{abort } T \rangle$.
 - If yes:
 - add $\langle \text{ready } T \rangle$ record to the log.
 - force all log records for T onto stable storage.
 - transaction manager sends $\langle \text{ready } T \rangle$ 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
 - $\langle \text{abort } T \rangle$ or $\langle \text{commit } T \rangle$ 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.





Failure Handling in 2PC – Site Failure

- The log contains a `<commit T>` record. In this case, the site executes `redo(T)`.
- The log contains an `<abort T>` record. In this case, the site executes `undo(T)`.
- The log contains a `<ready T>` record; consult C_i . If C_i is down, site sends query-status T message to the other sites.
- The log contains no control records concerning T. In this case, the site executes `undo(T)`.



Failure Handling in 2PC – Coordinator C_i Failure



- If an active site contains a $\langle \text{commit } T \rangle$ record in its log, the T must be committed.
- If an active site contains an $\langle \text{abort } T \rangle$ record in its log, then T must be aborted.
- If some active site does not contain the record $\langle \text{ready } T \rangle$ 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 $\langle \text{ready } T \rangle$ 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 .

