Distributed Systems

Synchronization In Distributed Systems
Synchronization in Distributed Systems

• Previously we saw how processes in a distributed system communicate with each other
  – RPC’s & Group Communications

• Another important consideration in distributed systems is how processes cooperate and synchronize with each other

• In single CPU systems we use critical regions, mutual exclusion and other system synchronization problems are generally solved using methods such as semaphores and monitors

• In a distributed system techniques involving semaphores and monitors do not work, and other techniques are needed

• Even determining whether event A happened before or after event B requires careful thought
Synchronization In Distributed Systems

- Critical to understanding synchronization in distributed systems is time and how it can be measured.
- Ensuring a consistent view of time is critical to managing synchronization in distributed systems.
  - We will look at clock synchronization mechanisms to achieve this.
- Based on clock synchronization we will look at mutual exclusion and election algorithms.
- Once these concepts are developed we will study a high-level synchronization technique called atomic transactions.
- Finally, we will look at deadlock and deadlock prevention in distributed systems.
Clock Synchronization

• In a distributed system it is generally not desirable to collect information from distributed processes in one place and then make decisions
  – Desire to make decisions based on distributed information

• Distributed systems (and associated algorithms) have the following properties
  – The relevant information is scattered among multiple machines
  – Processes make decisions based only on local information
  – A single point of failure in a distributed system should be avoided
  – No common clock or other precise global time source exists
Clock Synchronization

• In a non-distributed system, time is unambiguous
  – When a process needs to know the time it makes a system call and the kernel returns the time maintained by the hardware

• When multiple CPUs are present in a distributed system we cannot count on all clocks running at exactly the same rate

• Even if all clocks are initialized to exactly the same time at exactly the same instance, over time the clocks will differ
  – Due to clock skew because not all timers are exact

• If it is not possible to synchronize all clocks in a distributed system to produce a single, unambiguous time standard, then can it be permissible to have distributed processes agree on a consistent view of logical time?
Clock Synchronization

• Lamport (1978) showed that clock synchronization need not be absolute
• If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable
• Lamport realized that it is not important that all processes agree on time, but rather, that they agree on the order in which events occur
• Need to determine if two events are causally related, or if two events are concurrent
• If two events are causally related then one event must happen before the other event
• If two events are concurrent then they can safely occur in any order
Clock Synchronization Research

• The following material is derived from the research paper provided on the web by Raynal and Singhal entitled *Logical Time: A Way to Capture Causality in Distributed Systems*

• Before we investigate this paper, some fundamental Computer Science background needs to be reviewed:
  – Functions
  – Relations (especially binary relations)
  – Partial Orders
  – Total Orders
  – Monotonicity
  – Isomorphism
Functions and Relations

• The foundation for functions and relations is based on set theory

• Consider two sets A and B, the Cartesian product of A and B denoted AxB is denoted as the set of pairs in which the first component is chosen from A and the second component is chosen from B

• Formally the Cartesian product AxB is defined as:
\[ AxB = \{(a,b) \mid a \in A \text{ and } b \in B\} \]

• Example of Cartesian product:
Let A = \{1,2,3\} and B = \{4,5\} then
AxB = \{(1,4), (1,5), (2,4), (2,5), (3,4), (3,5)\}

• If the cardinality of A has \(n\) elements (notation \(|A|=n\)), and the cardinality of B has \(m\) elements, then the cardinality of AxB, denoted \(|AxB|\) has \(nm\) elements
Binary Relations and Functions

• A binary relation, denoted aRb, is a subset of the Cartesian product AXB where \( a \in A \) and \( b \in B \)

• A function is a binary relation with a special property: for each element \( a \in A \) there is exactly one ordered pair in R with the first component in a

• Stated another way a function, denoted \( f:A \rightarrow B \), is an assignment to each element a in A of a single element in B
  – \( f(a) = b \)
Binary Relations and Functions Examples

- Let \( C \) be the set of cities in the USA
- Let \( S \) be the set of states in the USA
- Consider
  - \( R_1 = \{ (x, y) : x \in C, y \in S, x \text{ is a city in state } y \} \)
  - \( R_2 = \{ (x, y) : x \in S, y \in C, y \text{ is a city in state } x \} \)

- Given \( R_1 \) and \( R_2 \):
  - \( R_1 \) is a function because each city is in one and only one state
  - \( R_2 \) is not a function, but it is a binary relation, because the same city (name) can be in more then one state
  - Example: Cambridge, Massachusetts and Cambridge Maryland
Special Properties of Functions

- We say a function is onto if $f(A) = B$, that is every $b$ in $B$ is the image of some $a$ in $A$

- We say a function is one-to-one if $a \neq a'$ implies $f(a) \neq f(a')$, that is distinct points in $A$ have distinct images

- We say a function is a bijection if it is both one-to-one and onto
Isomorphism

• We say a function $f: A \rightarrow B$ is an isomorphism if it has an inverse

• A function $f: A \rightarrow B$ is an isomorphism if and only if it is a bijection

• This makes sense because a bijection clearly defines a mapping between the domain and range where each element in the domain is mapped to a unique element in the range

• Thus, each element in the range of the function is clearly associated with a unique element in the range of the function
  – This is the inverse
Properties of Relations

- Reflexivity
  - A relation is reflexive if aRa holds
  - Example ≤ is reflexive, if a ≤ a is always true if a = a which is allowable under the ≤ relation

- Symmetry and Antisymmetry
  - A relation is symmetric if it is its own inverse, thus aRb holds and bRa holds
  - Example = is symmetric, if a = b then clearly b = a
  - A relation if antisymmetric if it is not symmetric
  - Example < is antisymmetric because if a < b then clearly b < a can not hold

- Transitivity
  - A relation is transitive if aRb and bRc then aRc holds
  - Example < is symmetric, if a < b and b < c then a < c is clearly true
Partial Orders and Total Orders

- A partial order is a transitive and antisymmetric binary relationship

- A total order is a partial order (transitive and antisymmetric) where every pair of elements in the domain are comparable
  - If R is a total order and a and b are two elements in the domain, then either aRb or bRa is true
  - Every total order is reflexive because a and b in aRb or bRa must hold and it must also hold that a=b due to the definition of total order (every pair)
Partial and Total Order
Examples

- The comparison operator $\leq$ is a total order
  - $\leq$ is transitive (inspection)
  - $\leq$ is antisymmetric. Except for the case of where $a=b$, $aRb$ or $bRa$ holds but both do not
  - Every element in the domain of $\leq$ is comparable. Consider all integers then for any two integers $a,b$ it is true that either $a \leq b$ or $b \leq a$. It is also true that if $a=b$ that both $a \leq b$ or $b \leq a$ hold because $a \leq a$ and $b \leq b$ holds

- The comparison operator $<$ is a partial order
  - $<$ is not transitive (inspection)
  - $<$ is antisymmetric (inspection)
  - $<$ is not comparable for every element. Consider $a=b$ then neither $aRb$ or $bRa$ is true because $a< a$ is false and $b< b$ is false
Logical Time Research

• Review research paper
Mutual Exclusion in Distributed Systems

• Mutual Exclusion, a review
  – Systems that involve multiple process often utilize critical regions
  – When a process has to read or update certain shared structures it
    • Enters a critical section
    • Performs its operation
    • Leaves the critical section
  – We use special constructs to serialize access to critical sections (semaphores, monitors, …)

• Most of the techniques that we know do not support distributed systems because there is no single shared memory image

• Need new techniques to achieve mutual exclusion
  – Centralized versus distributed techniques
A Centralized Algorithm for Distributed Mutual Exclusion

- Simulate how mutual exclusion is performed in a shared memory system
- One process acts as a coordinator
- Coordinator maintains a queue of requests for access to a critical section
- All processes send a message to the coordinator specifying the critical region that it wants to enter
- The coordinator either grants permission to the requesting process or queues the request if another process is using the critical section
- When a process exits the critical section it sends a message to the coordinator
- The coordinator now grants permission to the next process in the request queue, if any exists
A Centralized Algorithm for Distributed Mutual Exclusion

Algorithm can be implemented using RPC’s

Request

OK

Coordinator

Request

No Reply

Empty Queue

Release

OK
A Centralized Algorithm for Distributed Mutual Exclusion

• Algorithm guarantees mutual exclusion
• Algorithm is fair (FIFO ordering)
• Algorithm is easy to implement
  – RPC’s
  – Message Passing
  – Need 3 messages: Request, Grant, Release

• The coordinator is a single point of failure
• If the coordinator crashes the entire system will go down
• Process can not tell the difference between a request denied (blocking) and a dead coordinator
• A single coordinator may become a performance bottleneck
A Distributed Mutual Exclusion Algorithm

- Having a single point of failure and a potential bottleneck is generally unacceptable

- Researchers have proposed several distributed mutual exclusion algorithms

- Rickard and Agrawala’s algorithm
  - Based on clock synchronization techniques, scalar and vector
  - Requires that there be a total ordering of all events in the system
  - For any pair of messages it must be unambiguous which event happened first
Rucart abd Agrawala’s Algorithm

• When a process wants to enter a critical region, it builds a message containing
  – The critical region that it wants to enter
  – Its process number
  – The current time (logical time)

• The message constructed is sent to all processes, possibly including itself

• The sending of messages is assumed to be reliable (every message is acknowledged)

• May use reliable group communications

• When a process receives a message the action that it takes depends on its state with respect to the critical region named in the message

• Three separate cases must be handled
Ricart and Agrawala’s Algorithm

• Based on the received message, one of three cases must be handled:
  – **CASE 1:** The receiver is not in the critical region and does not want to enter it. Thus it sends back an “OK” message to the sender.
  – **CASE 2:** The receiver is already in the critical region. It handles this case by not replying to the sender and queues the request for later use.
  – **CASE 3:** The receiver wants to enter the critical region but has not yet done so. The receiver compares the timestamp in the message to its own logical timestamp.
    • If the timestamp in the message is lower, the receiver sends back an “OK” message to the sender.
    • If the timestamp is higher, the receiver queues the incoming request and sends nothing.
Ricart and Agrawala’s Algorithm

- After sending out requests asking permission to enter the critical region the process sits and waits until all other processes have given permission.
- Recall that the sender’s original message will be queued by receiver processes if the request can not be immediately granted.
  - CASE 2 & 3 (previous slide)
- As soon as the sender receives permission from all of the process in the distributed system the sender enters the critical section.
- The sender must also notify all processes in its queue when it exits the critical section.
  - The sender also deletes the entries from the queue.
A Token Ring Distributed Mutual Exclusion Algorithm

• This algorithm is based on organizing processes into a logical ring
• Each process knows about its neighbor
• A token is circulated around the ring
• If a process does not want to enter a critical region it passes the token to its neighbor upon receipt
• If a process wants to enter a critical region it waits until it receives the token then it:
  – Holds the token
  – Enters its critical region
  – Performs its operations
  – Leaves its critical region
  – Passes the token to its neighbor
• A process is not allowed to enter another critical region, it must pass the token to its neighbor and wait for it to circulate around the ring
A Token Ring Distributed Mutual Exclusion Algorithm

Problems
1. Token gets lost, it must be regenerated
2. Process crashes, the ring is broken
Election Algorithms

• Many distributed algorithms require one process to act as a coordinator
• It generally does not matter which process takes on the special coordinator responsibility, but one has to do it
• The problem is how do we select a coordinator → hold an election
• If a coordinator goes away how do we elect a new coordinator
  – Process normally ends
  – Process crashes and abnormally ends
• Requirements (for an election)
  – Every process must have a unique number, for example its network address
  – Every process knows the process number of every other process
  – All process must agree on on the new coordinator when the election ends
The Bully Algorithm

• When a process notices that the coordinator is no longer responding to requests, it initiates an election

• Process $P$ holds an election as follows:
  – $P$ sends and $ELECTION$ message to all processes with higher numbers
  – If nobody responds, $P$ wins the election and becomes the new coordinator
  – If one of the higher numbered processes answers, it takes over and $P$’s job is done

• Each higher numbered process now holds its own election based on the above process

• Eventually a process will hold an election and no other process will respond

• This process becomes the new coordinator and announces this fact to all of the other processes in the system
The Ring Algorithm

- The ring algorithm is based on arranging the distributed processes in a ring
  - No token is used in this algorithm
- Each process knows who its successor is
- When a processes notices that the coordinator is not responding it builds an \textit{ELECTION} message
  - This message includes the process number of the process that initiated the \textit{ELECTION}
- The \textit{ELECTION} message is then sent to the processes successor
- Once received, the process attaches its process number to the \textit{ELECTION} message
The Ring Algorithm

- Eventually the message will go around the ring
- When the original initiator of the *ELECTION* message receives the message again (after traversing the ring) the message will contain the process ID’s of all of the operating processes in the distributed environment
- The original initiator of the *ELECTION* message then extracts the ID of the largest process and generates a *COORDINATOR* message containing the largest processes ID
- This message is then sent around the ring
- Each process in the ring then records who is the new coordinator
Applications for Distributed Synchronization

• Atomic Transactions
  – Transactions are atomic
  – Difficult to manage with distributed processes
  – Distributed synchronization techniques can be scaled up to enable distributed atomic transactions

• Distributed Deadlock Management
  – The deadlock approaches discussed in MCS720 can be scaled up to work in the distributed world through the use of distributed synchronization techniques
    • The ostrich algorithm
    • Deadlock detection
    • Deadlock Prevention
    • Deadlock Avoidance