Synchronize Thy Actions for Coordination
Trying to have processes coordinate?
The key is getting them to wait
For it’s exclusive authorization
That predicates entry into the critical section
Either a token a process must posses
Or permissions it should collect for the access

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

☐ Why does Spark Streaming have the odd choice of 2 total copies, instead of the typical 3?
Topics covered in this lecture

- Distributed Coordination
- Distributed Mutual Exclusion

THE JOURNEY SO FAR
Networking
IP, TCP, UDP, Ethernet

Threads: Safety

Correctness

Throughput

Scaling

Response Times

Liveness / Deadlocks

Spark

Ease of Use

Cloud-scale systems
MapReduce

Fault Tolerance

Frameworks

HDFS

Hadoop

DISTRIBUTED COORDINATION
What we will cover

- Collection of algorithms whose goals vary, but share an aim that is fundamental in distributed systems
  - For a set of processes to:
    - Coordinate their actions
    - Agree on one or more values

Communication styles

- Asynchronous communications
  - No timing assumptions

- Synchronous communications have bounds on
  - Maximum message transmission delay
  - Time to execute each step of a process
  - Clock drift rates

Allows us to use timeouts to detect process crashes
 Coordination & Agreement

- A set of processes need to *coordinate* actions or *agree* on a set of values
- Must be able to do so even when *hierarchical* relationships do not exist
  - E.g.: Controller-Worker where a single point of failure exists

Example: Spaceship

- Multiple computers
- Computers that control spaceship must agree on several conditions
  - E.g., Status: Proceed or abort mission
- Coordinate access to shared resources
  - Sensors, actuators, etc.
DISTRIBUTED MUTUAL EXCLUSION

Distributed processes often need to coordinate their activities

- If a collection of processes share a set of resources *mutual exclusion* is needed to:
  - Prevent interference
  - Ensure consistency

- This is the critical section problem in OS
Distributed mutual exclusion

- Extension to distributed systems of the familiar problem of avoiding race conditions
  - In kernels and multi-threaded applications
- Shared variables or facilities provided by a local kernel cannot be used to solve this
- Solution must be based solely on message passing

Consider a set of $N$ processes $p_i \, i=1, 2, \ldots, N$
- These do not share variables
- Processes access common resources
  - They do so in a critical section
SUMMARY OF APPROACHES

April 11, 2019

Approaches to distributed mutual exclusion

- Token-based solutions
- Permission-based solutions
Token-based solutions

- Mutual exclusion is achieved by passing a special message (token) between the processes.
- There is only one token:
  - Whoever has that token is allowed to access shared resource.
- When finished, token is passed to another process.

Token-based solutions: Advantages

- Depending on how processes are organized, fairly easy to avoid starvation.
- Deadlocks can also be avoided.
Token-based solutions: Disadvantages

- When the token is lost – for e.g., process holding the token crashes, complex actions need to be taken
- After a failure, intricate distributed process needs to be initiated
  - Ensure that a new token is created
  - But above all, make sure that that is the only token

Permission-based solutions

- Process wanting to access resource first requests permission from other processes
- Many different ways to granting this permission
Structural considerations for the solution

- With a central server
- Without a central server
  - Peer processes must coordinate their accesses to shared resources
  - Occurs routinely on Ethernets and IEEE 802.11 wireless
    - Network interfaces cooperate as peers so that only one node transmits at a time on the shared medium
    - Ethernet: Method of operation “Carrier Sensing, Multiple Access with Collision Detection” or CSMA/CD
    - Wireless: “Carrier Sensing, Multiple Access with Collision Avoidance” CSMA/CA
Assumptions in our algorithms

- The system is asynchronous
- Processes do not fail
- Message delivery is reliable
  - Delivered eventually and exactly-once

Application level protocol for entering the critical section

- `enter()`
  - Block if necessary
- `resourceAccesses()`
  - Access shared resources in the critical section
- `exit()`
  - Allow other processes to enter
Requirements for distributed mutual exclusion

- **ME1**: At most one process may execute in the critical section at a time
  - Safety

- **ME2**: Requests to enter and exit the critical section eventually succeed
  - Liveness: Freedom from deadlocks and starvation

- **ME3**: If one request happened-before another, then entry to the CS is granted in that order
  - Fairness

Evaluation of the algorithms

- **Bandwidth consumed**: Proportional to number of messages sent in each entry and exit operation

- **Client delay** incurred by process for each entry or exit operation

- Effect on throughput of the system
  - Synchronization delay between one process exiting critical section and next process entering it
  - Throughput is greater when synchronization delay is shorter
The Central Server Algorithm

April 11, 2019

The central server algorithm

- Simplest way to achieve mutual exclusion
- Central server grants authorization to enter the critical section
- To enter a critical section, process sends request message to the server
  - Awaits reply from server
  - Reply constitutes token signifying authorization to enter critical section
Acquisition of token

- If no process holds the token?
  - Server replies immediately granting token

- If the token is held by another process?
  - Server does not reply, but queues the request
  - When that process exits the critical section, it sends a message giving server back the token
    - If the queue of waiting processes is non-empty, server chooses oldest entry in the queue and sends it the token

Server managing a mutual exclusion token

- a. Request Token
- b. Release Token
- c. Grant Token

Queue of Requests
Evaluating the central server algorithm [1/2]

- Entering critical section
  - Requires 2 messages: Request followed by grant
  - Delay at the requesting process?
    - Round trip delay
    - There is also the queuing delay for messages residing in the queue
- Exiting the critical section requires one release message
  - Assuming asynchronous communications means that this does not delay the exiting process

Evaluating the central server algorithm [2/2]

- Synchronization delay
  - Release message to server followed by grant to another process: Round trip time
- Server is a performance bottleneck for the system
  - Single point of failure as well
Ring-based algorithm

- Arrange mutual exclusion between \( N \) processes \textbf{without} requiring an additional process.
- Each process \( p_i \) has a communication channel to the next process in the ring, \( p_{(i+1) \mod N} \).
- Exclusion is conferred by obtaining a token that is \textit{passed from process to process} in a single direction around the ring.
  - E.g. clockwise.
Ring topology is unrelated to physical connections between underlying nodes

Acquisition of token

- When a process that does not need to enter critical section receives the token?
  - Immediately forwards token to its neighbor

- Process that requires token, **waits** until it receives it and then retains it

- To **exit** the critical section, process **sends token** to neighbor
Properties satisfied by the ring algorithm

- Satisfies ME1 and ME2
- Token is not necessarily acquired in a happened-before manner (ME3)

Performance analysis

- **Continuously** consumes network bandwidth (except when process is in critical section)
  - Processes send messages around ring even when no process requires critical section entry
- Delay experienced by process requesting entry to critical section?
  - 0: when it has just received the token
  - N messages when it has just passed on the token
Performance analysis

- Exit from critical section
  - Requires only 1 message

- Synchronization delay between one process' exit and another process' entry into critical section
  - Anywhere between 1 and N message transmissions

---

**MUTUAL EXCLUSION USING MULTICAST AND LOGICAL CLOCKS**

RICART & AGARWALA'S ALGORITHM

---
**LOGICAL CLOCKS**: If two processes do not interact with each other

- Their clocks need not be synchronized
- Lack of synchronization is not observable
  - Does not cause problems

---

**Lamport’s logical clocks**

- The **happens-before** relation

- $a$ and $b$ are events in the process; and $a$ occurs before $b$
  - Then $a \Rightarrow b$ is true

- $a$ is event of message sent by one process; $b$ is event of message being received in another process
  - Then $a \Rightarrow b$ is true
Some more things about the happens-before relation

- If \( a \xrightarrow{\text{b}} b \) and \( b \xrightarrow{\text{c}} c \), then \( a \xrightarrow{\text{c}} c \)
  - Transitive

- If events \( x \) and \( y \) occur in processes that do not exchange messages, then ...
  - \( x \xrightarrow{} y \) is not true
  - But, neither is \( y \xrightarrow{} x \)
  - These events are said to be concurrent

An example of Lamport’s algorithm:

Each clock runs at a constant (but different rate)
Implementing Lamport’s clocks

1. Before executing an event, $P_i$ executes
   $$C_i \leftarrow C_i + 1$$

2. When $P_i$ sends a message $m$ to $P_j$, it sets $m$’s timestamp $ts(m)$ to $C_i$ in previous step

3. Upon receipt of message $m$, $P_j$ adjusts its own local counter
   $$C_j \leftarrow \max\{C_i, ts(m)\}$$
   do step (1) and deliver message
An application of Lamport’s clock:
User has $1000 in bank account initially

Add $100 to account

San Francisco

Update with 1% interest

New York

REPLICATED DATABASE

Add $100 ... Total: $1100
Give 1% interest on total: $11
Balance: $1111

Give 1% interest ... Total: $1010
Add $100
Balance: $1110

There is a difference when the orders are reversed

- Our objective for now is consistency
- Both copies must be exactly the same

- Situations like this require totally-ordered multicast
  - All messages are delivered in the same order to each receiver
  - Lamport’s logical clocks allow us to accomplish this in a completely distributed fashion
Using Lamport’s clock to order messages

- Process puts received messages into local queue
  - Ordered according to the message’s timestamp

- Message can be delivered only if it is **acknowledged** by all the other processes

- If a message is at the head of the queue, and acknowledged by all processes
  - It is delivered and processed

Other types of logical clocks

- Vector clocks
- Matrix clocks
The contents of this slide set are based on the following references