Distributed Systems (5DV147)
Mutual Exclusion and Elections
Fall 2016

Computer Clock synchronization (previous lecture)

• Algorithms
  - Cristian’s algorithm
  - Berkeley algorithm
    - Master/slave relationship
    - Master polls slaves
    - Get current time from each slave
    - Send the offset from own time to each slave
  - Network Time Protocol
    - Unlike the others, designed for WAN rather than LAN use
    - Time servers close to the time source are more trusted
    - Redundant paths → survives disconnects

Motivation

- Is needed to coordinate access to a shared resource
  ➢ Concurrent access to a shared resource is serialized
  ... but the solution need to be based on message passing
- Three basic approaches
  ➢ Token-based
  ➢ Permission-based (Timestamp-based)
  ➢ Quorum-based

Distributed mutual exclusion

OS Refresher- semaphores

- Two types
  - Binary = 0 or 1
  - Counting 0 to n
- When wait() is called by a thread:
  - If semaphore is open, thread continues
  - If semaphore is closed, thread blocks on queue
- Then signal() opens the semaphore:
  - If a thread is waiting on the queue, the thread is unblocked
  - If no threads are waiting on the queue, the signal is remembered for the next thread

OS Refresher- mutexes

- Owned by a thread
  - Must be unlocked by the locker
  - May be locked/unlocked in separate functions
- Attempts to get a locked mutex force caller to sleep
Assumptions

- The system is asynchronous, process do not fail, and message delivery is reliable
- N processes \( p_i \) \( (i=1, 2, ..., N) \) that do not share variables
  - \( p_i \) access shared resources in a critical section
  - \( p_i \)'s are well behaved, finite time on the critical section

```
enter()
resourceAccesses()
exit()
```

Application level protocol for executing a critical section

Fairness

- Absence of starvation
- Maintain the order in which requests are made
  - No global clocks
  - Happened-before ordering:
    - it is not possible for a process to enter the critical section more than once while another waits to enter

Essential requirements

Safety: At most 1 process may enter the critical section at a time

Liveness: requests to enter and exit the critical section eventually succeed
  - Freedom of deadlock and starvation
  - Ordering: if a request to enter the critical section happened-before another, then access is granted according to that order

Criteria for evaluating algorithms

- Bandwidth consumed
  - Number of messages for entry and exit operations
- Client delay
  - Depends how many processes want access and how (typically) long are those accesses
    - Short and rare, dominant factor is the algorithm
    - Long and frequent, dominant factor is waiting for everyone to take a turn
- Throughput of the system
  - Synchronization delay, one process exiting and another one entering the critical section

MUTEX Algorithms

**Central Server**

- Send request to server, oldest process in queue gets access (a token), return token when done
- No process has token \( \Rightarrow \) reply (enter) immediately
- Otherwise \( \Rightarrow \) queue request
- Oldest process in the queue gets token after released
Properties
- Safety? Yes! (at most one can enter)
- Liveness? Yes (assuming no crashes)
- Ordering? No! Why not?
- Performance
  - Entering: 2 messages (request + grant)
  - Exiting: 1 message
  - Client delay: 2 messages (request + grant)
  - Synchronization delay: between 1 – N messages

Ring-based
- Token is passed around a ring of processes
- Safety?
- Liveness? Yes (assuming no crashes)
- Ordering? No!
- Performance
  - Single point of failure
  - Single point of failure

Ricart and Agrawala
- Distributed algorithm, no central coordinator
- Use Lamport's timestamps to order requests
- Multicast a request message
  - Enter critical section only when all other processes have given permission
  - Processes work cooperatively to provide access in a fair order
- Use multicast primitive or each process needs a group membership list

Details
- Each process
  - Has unique process ID
  - Has communication channels to the other processes
  - Maintains a logical (Lamport) clock
  - Is in a state ∈ {wanted, held, released}
- Requests are multicasted to group
- (process ID and clock value) (id, value)
- Lowest clock value gets access first
- Equal values? Check process ID!
Maekawa’s voting

Optimization: need only to ask a subset of the processes
- Key is how to build the subset
- At least one common member in any two voting sets
- Every voting set is the same size
- Works as long as subset overlaps
- Process can vote in one election at a time

Properties
- Safety? Yes
- Liveness? No, deadlocks can happen! \( V_1 = \{p_1, p_2\}, V_2 = \{p_2, p_3\}, V_3 = \{p_3, p_1\} \)
- \( \rightarrow \) ordering? No, but can
  - Add Lamport’s clocks
  - Retrieve votes if an earlier request arrives to a processor that has already voted
- Performance
  - Bandwidth: \( 3 \sqrt{N} \), \( 2 \sqrt{N} \) messages for entering and \( \sqrt{N} \) messages to exit
  - Client delay: 1 round-trip
  - Synchronization delay: 1 round-trip

Comparison of algorithms
- Central server:
  - Simple and error-prone!
  - \( \rightarrow \) but otherwise good performance!!!
- Ring-based algorithm:
  - Also simple, but not single point of failure
  - Not fair at all!
- Ricart and Agrawala:
  - Completely distributed and decentralised
  - Slower, more expensive, and less robust
  - \( \rightarrow \) but fair!
- Maekawa’s voting algorithm:
  - Only a subset of processes grant access; works if subsets are overlapping
... more comparison

- Message loss?
  - None of the algorithms handle this
- Crashing processes?
  - Ring? No! Others? depends
    - Central – not server nor process holding or having requested token
    - Ricart & Agrawala – no
    - Maekawa’s – only if crashed process is not in voting set.

Summary

- Control access to shared resources
- Algorithms
  - Central server
  - Ring-based
  - Ricart and Agrawala
  - Maekawa’s voting algorithm

Motivation

- How to choose a process to play a particular role in the system
- Start with all processes in same state
  - One process will reach state leader
  - Other processes will reach state lost
- Each process requires a unique identifier (totally ordered)
- Every process knows the id(s) of other (all) processes

Essential requirements

Safety:
A participant has elected, \( e \text{ or elected, } = P \),
where \( P \) is chosen as the non-crashed process

Liveness:
All processes participate and eventually set elected, to not \( e \text{ or} false \text{ or crash} \)
Election Algorithms

Ring-based algorithm

- Goal is to elect a single process – the coordinator
  - process with the largest identifier
- During election, pass $\max(\text{own ID}, \text{incoming ID})$ to next process
  - If a process receives own ID, it must have been highest and may send that it has been elected

Bully algorithm

- Requires:
  - Synchronous system
  - All processes know of each other (which ones have higher ids)
  - Reliable failure detectors
  - Reliable message delivery
- Allows
  - Crashing processes

Details

- Safety? Liveness? Yes!
- Tolerates no failures (limited use)

Worst case, $N$ messages until reaching peer with largest identifier

$N$ messages to complete another circuit

$N$ messages advertising the election

$3N-1$ messages

Example

- Safety? No
  - if process IDs can be reused!
- Liveness? Yes
  - if message delivery is reliable

Example

- The election of coordinator $p_2$ after the failure of $p_4$ and then $p_3$
- Eventually, $p_2$ becomes coordinator
Summary

• Election algorithms
  – Seems like a simple problem, but non-trivial solutions are... non-trivial
  – Ring and Bully algorithms
• Want to read more about non-trivial election algorithms?
  – http://www.sics.se/~ali/teaching/dalg/l06.ppt

Next Lecture

Group Communication

Reading

Chapter 15 “Coordination and Agreement”, Distributed systems, 5th ed. By Coulouris, Dollimore, Kindberg and Blair