Lecture 14: Coordination

Distributed Systems
Behzad Bordbar
School of Computer Science, University of Birmingham, UK

Overview

Co-ordination in distributed systems
- why needed, sources of problems

Mutual exclusion
- ring-based
- multicast & logical clocks [Ricart and Agrawala]

Leader election
- ring-based [Chang&Roberts]

Co-ordination algorithms

are fundamental in distributed systems:
- for resource sharing: concurrent updates of
  - records in a database (record locking)
  - files (file locks in stateless file servers)
  - a shared bulletin board
- to agree on actions: whether to
  - commit/abort database transaction
  - agree on a readings from a group of sensors
- to dynamically re-assign the role of master
  - choose primary time server after crash
  - choose co-ordinator after network reconfiguration

Why difficult?

Centralised solutions not appropriate
- communications bottleneck

Fixed master-slave arrangements not appropriate
- process crashes

Varying network topologies
- ring, tree, arbitrary, connectivity problems

Failures must be tolerated if possible
- link failures
- process crashes

Impossibility results
- in presence of failures, esp asynchronous model

Co-ordination problems

Mutual exclusion
- distributed form of critical section problems
- must use message passing

Leader elections
- after crash failure has occurred
- after network reconfiguration

Consensus (also called Agreement): next lecture
- similar to coordinated attack
- some based on multicast communication
- variants depending on type of failure, network, etc

Failure assumptions

Assume reliable links, but possible process crashes.

Failure detection service
- provides query if a process has failed
- how?
  - processes send ‘P is here’ messages every T secs
  - failure detector records replies
- unreliable, especially in asynchronous systems

Observations of failures:
- Suspected: no recent communication, but could be slow
- Unsuspected: no guarantee it has not failed since
- Failed: crash has been determined
Distributed mutual exclusion

The problem:
- N asynchronous processes, for simplicity no failures
- guaranteed message delivery (reliable links)
- to execute critical section (CS), each process calls:
  - enter()
  - resourceAccess()
  - exit()

Requirements
- (MC1) At most one process is in CS at the same time.
- (MC2) Requests to enter and exit are eventually granted.
- (MC3 - Optional, stronger) Requests to enter granted according to causality order.

Centralised mutual exclusion

Centralised service

Single server implements imaginary token:
- only process holding the token can be in CS
- server receives request for token
- replies granting access if CS free, otherwise, request queued
- when a process releases token, oldest request from queue granted

It works though...
- it does not respect causality order of requests (MC3) - why?
- but
- server is performance bottleneck!
- what if server crashes?

Ring-based algorithm

No server bottleneck, no master

Processes:
- continually pass token around the ring, in one direction
- if do not require access to CS, pass on to neighbour
- otherwise, wait for token and retain it while in CS
- to exit, pass to neighbour

How it works
- continuous use of network bandwidth
- delay to enter depends on the size of ring
- causality order of requests not respected (MC3) - why?

Ricart-Agrawala algorithm

Based on multicast communication
- N inter-connected asynchronous processes, each with
  - unique id
  - Lamport’s logical clock
- processes multicast request to enter critical section:
  - timestamped with Lamport’s clock and process id
- entry granted
  - when all other processes replied
  - simultaneous requests resolved with the timestamp

How it works
- satisfies the stronger property (MC3)
- if hardware support for multicast, only one message to enter
**Ricart-Agrawala algorithm**

**On initialization**
- \( \text{state} = \text{RELEASED} \)
- To enter the critical section:
  - \( \text{state} = \text{WANTED} \)
  - Multicast request to all processes; request processing deferred here
  - \( T \) = request’s timestamp;
  - Wait until number of replies received = \( N - 1 \);
  - \( \text{state} = \text{HELD} \)

**On receipt of a request \( \langle T_i, p_i \rangle \) at \( p_j \) (\( i \neq j \))**
- If \( \text{state} = \text{HELD} \) or (\( \text{state} = \text{WANTED} \) and \( (T_i, p_i) < (T_j, p_j) \))
  - queue request from \( p_i \) without replying;
  - else
    - reply immediately to \( p_i \);
  - end if

**To exit the critical section**
- \( \text{state} = \text{RELEASED} \)
  - reply to any pending requests;

---

**Multicast mutual exclusion**

1. **p_1**, **p_2** request access simultaneously, **p_3** does not want access to CS.
2. **p_2** does not reply to **p_1** since it has lower timestamp.

---

**Mutual exclusion summary**

**Performance**
- One request-reply enough to enter
- Relatively high usage of network bandwidth
- Client delay depends on the frequency of access and size of network

**Fault tolerance**
- Usually assume reliable links
- Some can be adapted to deal with crashes

**Other solutions**
- Sufficient to obtain agreement from certain overlapping subsets of processes voting set (Maekawa algorithm)

---

**Leader election algorithms**

**The problem**
- \( N \) processes
- For simplicity assume no crashes
- Must choose unique master co-ordinator amongst processes
- Election called after failure has occurred

**Requirements**
- **(LE1)** Every process knows \( P \), identity of leader, where \( P \) is unique process id (usually maximum) or is yet undefined.
- **(LE2)** All processes participate and eventually discover the identity of the leader (cannot be undefined).

---

**Chang&Roberts algorithm**

**Assumptions**
- Unidirectional ring, asynchronous, each process has UID

**Election**
- Initially each process non-participant
- Determine leader (election message):
  - initiate becomes participant and passes own UID on to neighbour
  - when non-participant receives election message, forwards maximum of own and the received UID and becomes participant
  - participant does not forward the election message
- Announce winner (elected message):
  - when participant receives election message with own UID, becomes leader and non-participant, and forwards UID in elected message
  - otherwise, records the leader’s UID, becomes non-participant and forwards it
**Chang&Roberts algorithm**

How it works
- if UIDs, then identity of leader unique
- two exchanges around the ring: election, announce winner
- if one process starts election
  - in worst case 3 round-trips needed – explain?

but
- does not tolerate failures (need reliable failure detector)
  - see bully algorithm (synchronous model)
  - works if more than one process simultaneously start election
- what if no UIDs?
  - nodes on a re-configurable network, ‘hot-pluggable’

---

**Summary**

Coordination algorithms
- Challenges
- deadlock
- fairness
- safety

Distributed mutual exclusion
- Central server algorithms
- Ring-based algorithms
- Multicast based algorithms
- Leader election algorithms

Further reading:
- pages 467-482

---

**Exercises:**

12.1 Is it possible to implement either a reliable or an unreliable (process) failure detector using an unreliable communication channel?

12.2 If all client processes are single-threaded, is mutual exclusion condition ME3, which specifies entry in happened-before order, relevant?

12.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of the synchronization delay.

12.4 In the central server algorithm for mutual exclusion, describe a situation in which two requests are not processed in happened-before order.