Co-ordination algorithm

- 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

资源共享

- to agree on actions: whether to
  - commit/abort database transaction
  - agree on a readings from a group of sensors

行为共识、

- to dynamically re-assign the role of master
  - choose primary time server after crash
  - choose co-ordinator after network reconfiguration

master动态确定

Difficulty

- Centralised solutions not appropriate
  - communications bottleneck

中心化方案不适合

- Fixed master-slave arrangements not appropriate
  - process crashes

固定的master-slave不适合

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

网络拓扑变化多端

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

容错

- Impossibility results
  - in presence of failures, esp. asynchronous model

不可能的结果

Co-ordination problems

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

互斥

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

领导人选举

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

共识、

Failure Assumption

- Assume reliable links, but possible process crashes.

假设连接可靠,但process会crash.

- Failure detection service
  - processes 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

失效检测服务:
processes每隔T秒发送信息,当crash时detector会检测这条信息.
在异步系统中不可靠

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

Distributed mutual exclusion 分布式互斥

- 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

  (**ME1**) At most one process is in CS at the same time.

  (**ME2**) Requests to **enter** and **exit** are eventually granted.

  (**ME3** - Optional, stronger) Requests to **enter** granted according to causality order.

同时最多只有一个进程在critical section

enter和exit请求最终都会接受

enter的请求顺序按照因果顺序

Centralised service

- Single server implements imaginary token:
  - only process holding the token can be in CS
  - server receives **request** for token
  - replies **grant**ing access if CS free; otherwise, request queued
  - when a process **release**s token, oldest request from queue granted
- It works though...
  - does not respect causality order of requests (MC3) - why?
- but
  - server is performance bottleneck!
  - what if server crashes?

有一个共享的token(互斥锁、条件变量)

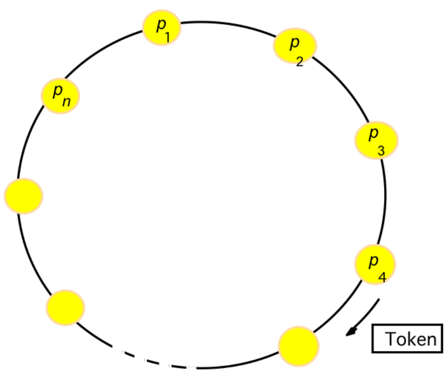
只有拥有token的才能进入CS

不遵循MC3,时间序列上的.

瓶颈性能

Server可能会crash

Ring-based algorithm



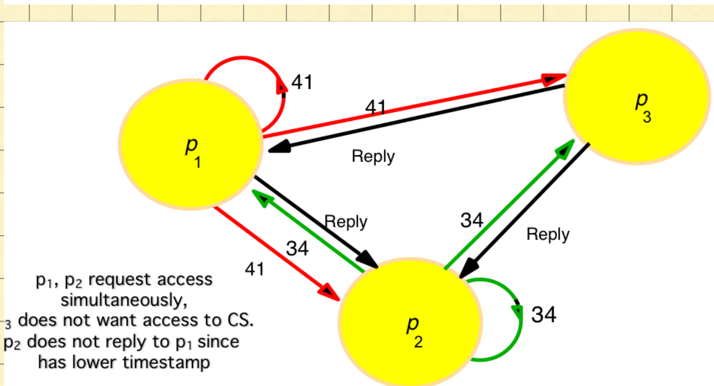
- Arrange processes in a logical ring, let them pass token.
- No server bottleneck, no master
- Processes:
  - continually pass token around the ring, in one direction
  - if do not require access to CS, pass on to neighbour
  - otherwise, wait for token and retain it while in CS
  - to exit, pass to neighbour
- How it works
  - continuous use of network bandwidth
  - delay to enter depends on the size of ring
  - causality order of requests not respected (ME3) - 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 (ME3)
  - if hardware support for multicast, only one message to enter

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

On receipt of a request $<T_i, p_i>$ *at* p_j $(i \neq j)$
    if (*state* = HELD) or ((*state* = WANTED) *and* $((T, p_j) < (T_i, p_i))$)
    then
        queue *request* from p_i without replying;
    else
        reply immediately to p_i;
    end if
*To **exit the critical section***
    state := RELEASED;
    reply to any queued requests;



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)

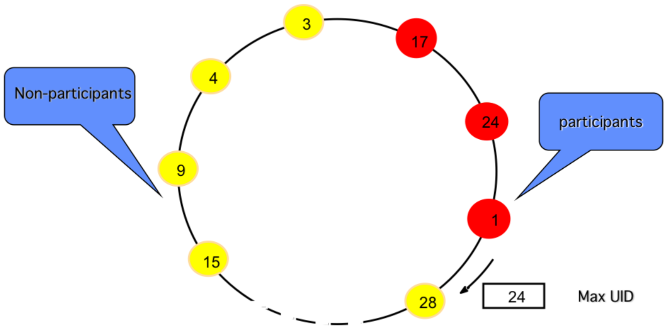
Leader election algorithms

- The problem
  - N processes
  - for simplicity assume no crashes
  - must choose unique master co-ordinator amongst processes
  - election called after failure has occurred
  - one or more processes can call election simultaneously
- 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 & Robert algorithm



- Leader election in a ring: asynchronous model, UIDs known.

- Assumptions
  - unidirectional ring, asynchronous, each process has UID
- Election
  - initially each process non-participant
  - determine leader (*election* message):
    - initiator 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

- 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'

Agreement

- Agreement (= Consensus) problems
  - why & where needed
  - definition
- Byzantine generals
  - in synchronous systems
  - in asynchronous systems
- And finally
  - impossibility results!
  - practical implications

Consensus algorithms

- used when it is necessary to agree on actions:
  - in transaction processing
    - commit or abort transaction?
  - mutual exclusion
    - which process should enter the critical section?
  - in control systems
    - proceed or abort based on sensor readings?

The model & assumptions

- The model
  - N processes
  - message passing
  - synchronous or asynchronous
  - communication reliable
- Failures!
  - process crashes
  - arbitrary (Byzantine) failures
    - processes can be treacherous and lie
- The algorithm
  - works in presence of certain failures

Consensus : main idea

- Initially
  - processes begin in **undecided** state
  - propose an initial value from a set D
- Then
  - processes communicate, exchanging values
  - attempt to decide
  - cannot change the decision value in **decided** state
- The difficulty
  - must reach decision even if crash has occurred
  - or arbitrary failure!

Consensus : requirements

- Termination
  - Eventually each correct process sets its decision value.
- Agreement
  - Any two correct processes must have decided on the same decision value.
- Integrity
  - If all correct processes propose the same value, then any correct process that has decided must have chosen that value.

Towards a solution

- For simplicity, assume no failures
  - processes multicast its proposed value to others
  - wait until all N values collected (including own)
  - decide through majority vote (⊥ special value if none)
    - can also use minimum/maximum
- It works since...
  - all processes end up with the same set of values
  - majority vote ensures Agreement and Integrity
- But what about failures?
  - process crash - stops sending values after a while
  - arbitrary failure - different values to different processes

Byzantine generals

- The problem [Lamport 1982]
  - three or more generals are to agree to **attack** or **retreat**
  - one (commander) issues the order
  - the others (lieutenants) decide
  - one or more generals are treacherous (= faulty!)
    - propose attacking to one general, and retreating to another
    - either commander or lieutenants can be treacherous!
- Requirements
  - Termination, Agreement as before.
  - Integrity: If the commander is correct then all correct processes decide on the value proposed by commander.

Consensus in synchronous system

- Uses basic multicast
  - guaranteed delivery by correct processes assuming the sender does not crash
- Admits process crash failures
  - assume up to f of the N processes may crash
- How it works...
  - f +1 rounds
  - relies on synchrony (timeout!)

- Initially
  - each process proposes a value from a set D
- Each process
  - maintains the set of values V_r known to it at round r
- In each round r, where $1 \leq r \leq f+1$, each process
  - multicasts the values to each other (only values not sent before, $V_r - V_{r-1}$)
  - receives multicast messages, recording any new value in V_r
- In round f+1
  - each process chooses minimum V_{f+1} as decision value

- Why it works?
  - sets timeout to maximum time for correct process to multicast message
  - can conclude process crashed if no reply
  - if process crashes, some value not forwarded...
- At round f+1
  - all correct process arrive at the same set of values
  - hence reach the same decision value (minimum)
  - at least f+1 rounds needed to tolerate f crash failures
- What about arbitrary failures?

Byzantine generals

- Processes exhibit arbitrary failures
  - up to f of the N processes faulty
- In a synchronous system
    - can use timeout to detect absence of a message
  - cannot conclude process crashed if no reply
  - impossibility with $N \le 3f$
- In asynchronous system
  - cannot use timeout to reliably detect absence of a message
  - impossibility with even one failure!!

- Solution exists for 4 processes with one faulty
  - commander sends value to each of the lieutenants
  - each lieutenant sends value it received to its peers
  - if commander faulty, then correct lieutenants have gathered all values sent by the commander
  - if one lieutenant faulty, the each correct lieutenant receives 2 copies of the value from the commander
- Thus
  - correct lieutenants can decide on majority of the values received
- Can generalise to $N \ge 3f + 1$

In asynchronous systems

- No guaranteed solution exists even for one failure!!! [Fisher, Lynch, Paterson '85]
  - does not mean never reach consensus in presence of failures
  - but that can reach it with positive probability
- But…
  - Internet asynchronous, exhibits arbitrary failures and uses consensus?
- Solutions exist using
  - partially synchronous systems
  - randomisation [Aspnes&Herlihy, Lynch, etc]