Distributed Systems

7. Coordination and Agreement

Werner Nutt

Co-ordination Algorithms

are fundamental in distributed systems:

- to dynamically re-assign the role of master
  - choose primary server after crash
  - co-ordinate resource access
- for resource sharing: concurrent updates of
  - entries in a database (data locking)
  - files
  - a shared bulletin board
- to agree on actions: whether to
  - commit/abort database transaction
  - agree on a readings from a group of sensors

Why is it Difficult?

- Centralised solutions not appropriate
  – communications bottleneck, single point of failure
- 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
  – impossibility of "coordinated attack"

Synchronous vs. Asynchronous Interaction

- Synchronous distributed system
  – Time to execute a step has lower and upper bounds
  – Each message is received within a given time
  – Each process has a local clock with a bounded drift

➔ *failure detection by timeout*

- Asynchronous distributed system
  – No bounds on process execution time
  – No bounds on message reception time
  – Arbitrary clock drifts

the common case

Co-ordination Problems

- Leader election
  - after crash failure has occurred
  - after network reconfiguration
- Mutual exclusion
  - distributed form of synchronized access problem
  - must use message passing
- Consensus (also called Agreement)
  - 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 answer if a process has failed
  - how?
    - processes send 'I am 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: but no guarantee it has not failed since
  - Failed: crash has been determined

Analysing (Distributed) Algorithms

- Qualitative properties
  - Safety: if there is an outcome,
    then it satisfies the specification of the algorithm
  - Liveness: there is an outcome
- Quantitative properties
  - Bandwidth: total number of messages sent around
  - Turnaround: number of steps needed to come to a result

Coordination and Agreement

7.1 Leader Election

1. **Leader Election**
2. Mutual Exclusion
3. Agreement

Leader Election

- The problem:
  – N processes, may or may not have unique IDs (UIDs)
  – must choose unique master co-ordinator amongst themselves
  – one or more processes can call election simultaneously
  – sometimes, election is called after failure has occurred
- Safety:
  – Every process has a variable *elected*, which contains the UID of the leader or is yet undefined
- Liveness (and safety):
  – All processes participate and eventually discover the identity of the leader (*elected* cannot be undefined).

Election on a Ring (Chang/Roberts 1979)

- Assumptions:
  – each process has a UID, UIDs are linearly ordered
  – processes form a unidirectional logical ring, i.e.,
    - each process has channels to two other processes
    - from one it receives messages, to the other it sends messages
- Goal:
  – process with highest UID becomes leader
- Note:
  – UIDs can be created dynamically, e.g., process i has the pair < $1/load_i$, pid_i >

Election on a Ring (cntd)

Processes

- send two kinds of messages: *elect(UID)*, *elected(UID)*
- can be in two states: *non-participant*, *participant*

Two phases

- Determine leader
- Announce winner

Initially, each process is *non-participant*

Algorithm: Determine Leader

- Some process with UID *id0* initiates the election by
  - becoming *participant*
  - sending the message *elect(id0)* to its neighbour
- When a *non-participant* receives a message *elect(id)*
  - it forwards *elect(idmax)*, where *idmax* is the maximum of its own and the received UID
  - becomes *participant*
- When a *participant* receives a message *elect(id)*
  - it forwards the message if *id* is greater than its own UID
  - it ignores the message if *id* is less than its own UID

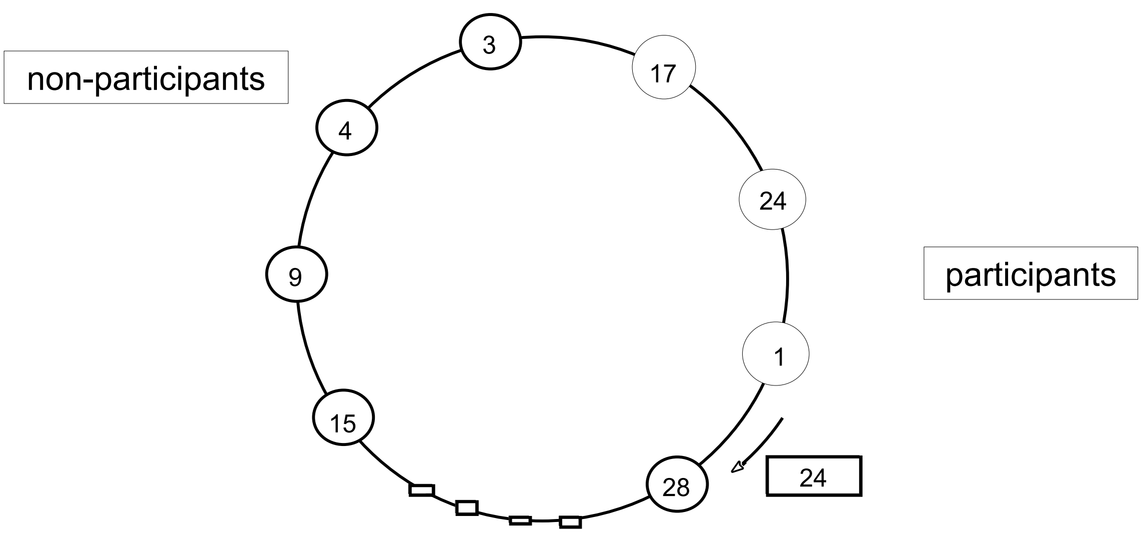
Algorithm: Announce Winner

- When a *participant* receives a message *elect(id)* where *id* is its own UID
  - it becomes the leader
  - it becomes *non-participant*
  - sends the message *elected(id)* to its neighbour

- When a *participant* receives a message *elected(id)*
  - it records *id* as the leader's UID
  - Becomes *non-participant*
  - forwards the message *elected(id)* to its neighbour

- When a *non-participant* receives a message *elected(id)*
  - …

Election on a Ring: Example



Properties

- Safety: ✓
- Liveness
  - clear, if only one election is running
  - what, if several elections are running at the same time?
    ➔participants do not forward smaller IDs
- Bandwidth: at most 3n – 1 (if a single process starts the election, what if several processes start an election?)
- Turnaround: at most 3n - 1

Under Which Conditions can it Work?

- What if the algorithm is run in an asynchronous system?
  - Synchronicity is not needed for the algorithm
    (but may be needed for detecting failure of the old leader)
- What if there is a failure (process or connection)?
  - the election gets stuck
  - ➔ assumption: no failures
    (in token rings, nodes are connected to the network by a connector, which may pass on tokens, even if the node has failed)
- When is this applicable?
  - token ring/token bus
  - when leader role is needed for a specific task
  - when IDs change, e.g., IDs linked to current load

Bully Algorithm (Garcia-Molina)

- Idea: Process with highest ID imposes itself as the leader
- Assumption:
  - each process has a unique ID
  - each process knows the IDs of the other processes
- When is it applicable?
  - IDs don't change
  - processes may fail
- Further assumption: synchronous system
  - to detect failure
  - to detect that there is no answer to a request

Bully Algorithm: Principles

- A process detects failure of the leader *How?*
- The process starts an election by notifying the potential candidates (i.e., processes with greater ID)
  - if no candidate replies (synchronicity!), the process declares itself the winner of the election
  - if there is a reply, the process stops its election initiative
- When a process receives a notification
  - it replies to the sender
  - and starts an election

Bully Algorithm: Messages

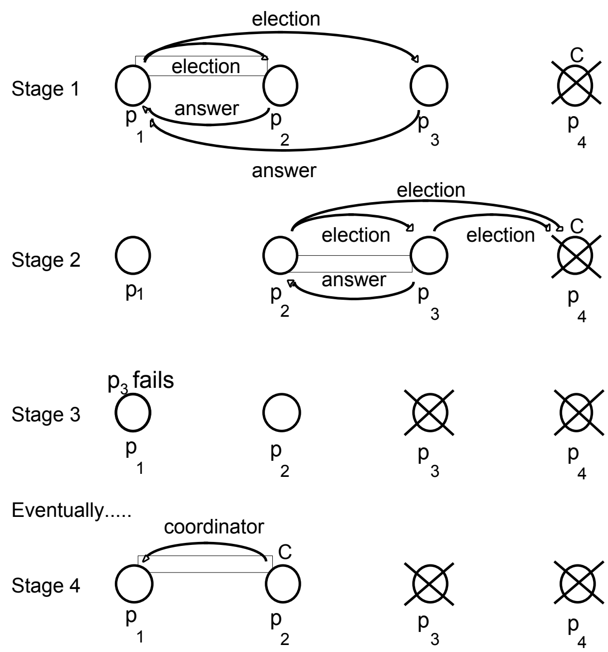
- Election message:
  - to "call elections" *(sent to nodes with higher UID)*
- Answer message:
  - to "vote" *(… against the caller, sent to nodes with lower UID)*
- Coordinator message:
  - to announce own acting as coordinator

Bully Algorithm: Actions

- The process with highest UID sends coordinator message
- A process starting an election sends an election message
  - if no answer within time $T = 2\ T_{transmission} + T_{process}$, then it sends a coordinator message
- If a process receives a coordinator message
  - it sets its **`coordinator`** variable
- If a process receives an election message
  - it answers and begins another election (if needed)
- If a new process starts to coordinate (highest UID),
  - it sends a coordinator message and "*bullies*" the current coordinator out

Bully Algorithm: Example

Processor p_2 is elected coordinator,
after the failure of p_4
and then p_3



Properties of the Bully Algorithm

- Liveness
  - guaranteed because of synchronicity assumption
- Safety
  - clear if group of processes is stable (no new processes)
  - not guaranteed if new process declares itself as the leader during election *(e.g., old leader is restarted)*
    - two processes may declare themselves as leaders at the same time
    - but no guarantee can be given on the order of delivery of those messages

Quantitative Properties

- Best case: process with 2nd highest ID detects failure
- Worst case: process with lowest ID detects failure

- Bandwidth:
  - N -1 messages in best case
  - $O(N^2)$ in worst case

- Turnaround:
  - 1 message in best case
  - 4 messages in worst case

Randomised Election (Itai/Rodeh)

- Assumptions
  - N processes, unidirectional ring, synchronous (?)
  - processes do not have UIDs
- Election
  - each process selects ID at random from set {1,…,K}
    - non-unique! but fast
  - processes pass all IDs around the ring
  - after one round, if there exists a unique ID then elect maximum unique ID
  - otherwise, repeat
- Question
  - how does the loop terminate?

Probabilistically!

Randomised Election (cntd)

- How do we know the algorithm terminates?
  - from probabilities: if we keep flipping a fair coin then after several *head*s you must get *tails*
- How many rounds does it take?
  - the larger the probability of a unique ID, the faster the algorithm
  - expected time: N=4, K=16, expected 1.01 rounds
- Why use randomisation?
  - symmetry breaker
  - no deterministic solution for the problem
- Only probabilistic guarantee of termination (with probability 1)

Coordination and Agreement

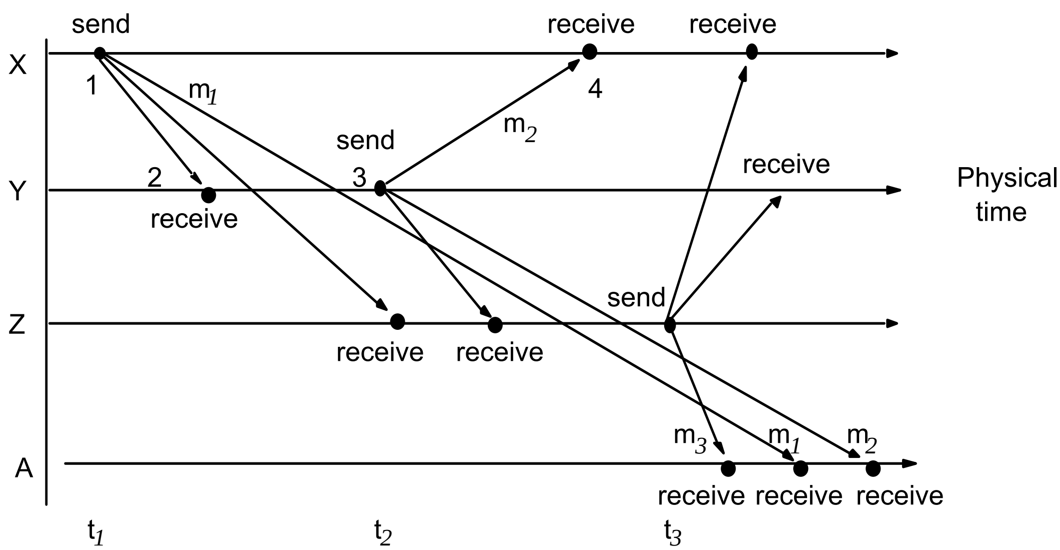
7.2 Mutual Exclusion

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
  - Safeness: At most one process is in CS at the same time
  - Liveness: Requests to enter and exit are eventually granted
  - Ordering: Requests to enter are served by a FIFO policy according to Lamport's causality order

Asynchronous Email



How can A know the order in which the messages were sent?

Time in Banking Scenario

A bank keeps replicas of bank accounts in Milan and Rome

- Event 1:
  Customer Rossi pays 100 € into his account of 1000 €
- Event 2:
  The bank adds 5% interest

Info is broadcast to Milan and Rome

➔ Make sure that replicas are updated in the same order!
➔ Give agreed upon time stamps to transactions!

Time Ordering of Events (Lamport)

Observation:
For some events E1, E2,
it is "obvious" that E1happened before E2
(written E1 → E2)

- If E1 happens before E2 in process P, then E1 → E2
- If E1 = send(M) and E2 = receive(M), then E1 → E2
  (M is a message)
- If E1 → E2 and E2 → E3 then E1 → E3

Logical Clocks

Goal: Assign “timestamps” ti to events Ei such that

$$E1 \rightarrow E2 \Rightarrow t1 < t2$$

not the converse!

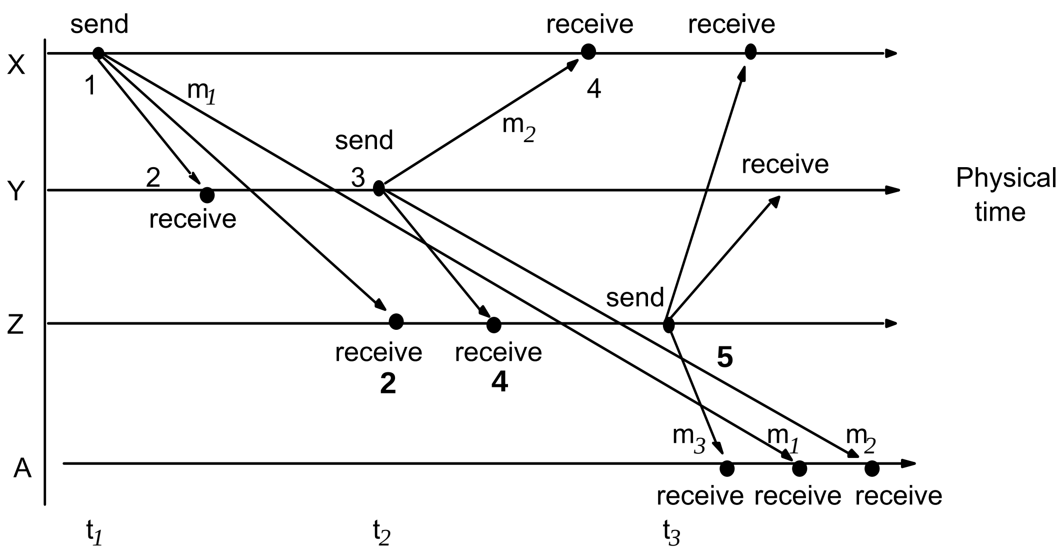
Approach: Processes

- incrementally number their events
- send numbers with messages
- update their “logical clock” to

  max(OwnTime, ReceivedTime) +1

  when they receive a message

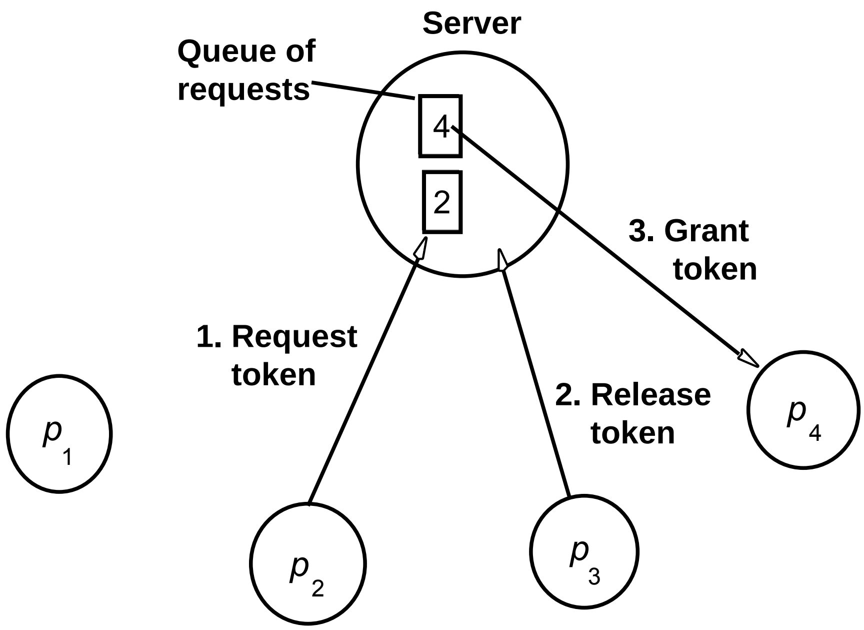
Logical Clocks in the Email Scenario



Messages carry numbers 5 1 3

For a tie break, use process numbers as second component!

Centralised Service



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 the token, oldest request from queue granted
- It works though...
  - does not respect causality order of requests – why?
- but
  - server is performance bottleneck!
  - what if server crashes?

Properties

- Safety: “No two processes are in the critical section at the same time” …
- Liveness: …
- Ordering: ???

- Bandwidth: 2 messages for request + 1 for release
- Client Delay: O(length of queue)
- Synchronisation Delay (= time between exit of current and enter of next process) : 2

Centralised Service: Discussion

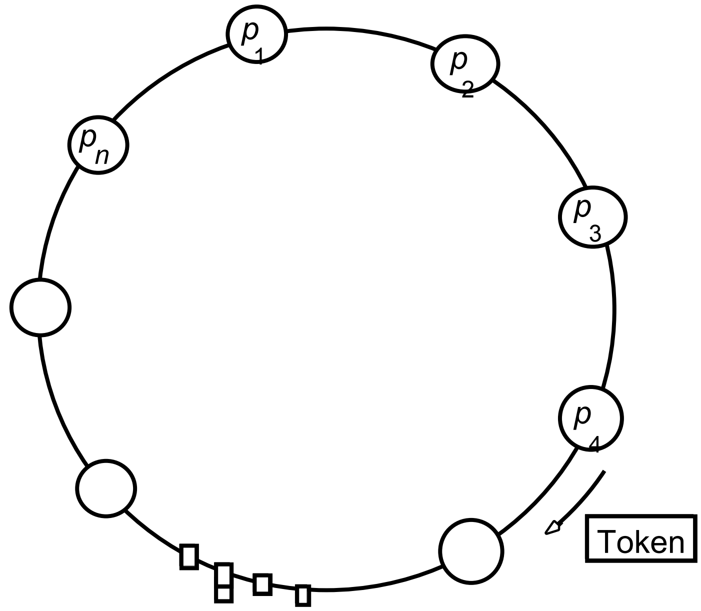
- Server is a single point of failure
  - How can one cope with server failure?
  - Which difficulties arise?

- How can a process distinguish between
  - “permission denied”
  - dead server?

- What about the following attempt to grant access with respect to Lamport‘s causality order:

  “Order requests in queue wrt Lamport timestamps”

Ring-based Algorithm



Arrange processes in a logical ring, let them pass token

Ring-based Algorithm

- No master, no server bottleneck
- 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

Why?

Properties

- Safety *("No two processes …")*: …
- Liveness: …
- Ordering: ???

- Bandwidth: continuous usage
- Client Delay: between 0 and N
- Synchronisation Delay: between 1 and N

Ring-based Algorithm: Discussion

- How many points of failure?

- Suppose the ring is a logical ring:
  How could one cope with failure of a node?

- How could one detect failure of a node?

Multicast Mutual Exclusion
(Ricart/Agrawala)

- Based on multicast communication
  - N inter-connected asynchronous processes, each with
    - unique id
    - Lamport's logical clock
  - processes multicast request to enter:
    - 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 ordering property
  - if support for multicast, only one message to enter

Multicast Mutual Exclusion

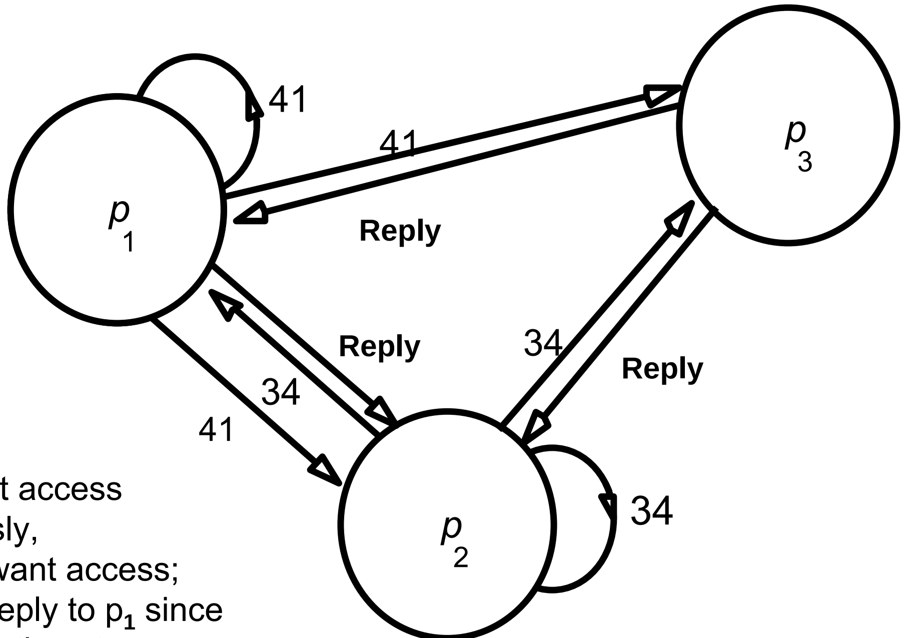
On initialization
    state := RELEASED;

To enter the 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;

Multicast Mutual Exclusion: Example



p_1, p_2 request access simultaneously,
p_3 does not want access;
p_2 does not reply to p_1 since it has a lower timestamp

Properties

- Safety: Indirect proof:

  p_i and p_j both in the critical section
  $\Rightarrow$ p_i and p_j replied to each other
  $\Rightarrow$ $(T_i, p_i) < (T_j, p_j)$ and $(T_j, p_j) < (T_i, p_i)$
- Liveness: …
- Ordering: if p_i makes its request "before" p_j,

  then $T_i,p_i < T_j,p_j$ …

- Bandwidth: 2 (N – 1) messages or 1 multicast + (N-1) replies
- Client Delay: 2
- Synchronisation Delay: 1

Multicast Mutual Exclusion: Discussion

- Is there a single point of failure?
- Comparison between multicast and centralised approach:
  - number of messages for granting a request
  - sensitivity to crashes
- Which approach would you expect to be used in practice?

Mutual Exclusion by Voting (Maekawa)

- Observation:
  - it is not necessary to have replies of all peers
  - one can use a voting mechanism
- Formalisation: With each process p_i, we associate a voting set $V_i \subseteq \{p_1, p_2, \ldots, p_N\}$ such that
  - $p_i \in V_i$
  - $V_i \cap V_j \neq \emptyset$ *there is at least one common member for any two voting sets*

Maekawa's Algorithm

On initialization

state := RELEASED;
voted := FALSE;

For p_i ***to enter the critical section***

Multicast *request* to all processes in V_i ;
Wait until (number of *repli*es received = $|V_i|$);
state := HELD;

On receipt *of a request from* p_i *at* p_j

if (*state* = HELD *or voted* = TRUE)
then
queue *request* from p_i without replying;
else
send *reply* to p_i;
voted := TRUE;
end if

Continues on next slide

Maekawa's Algorithm (cntd)

For p_i *to* ***exit the critical section***

state := RELEASED;
Multicast *release* to all processes in V_i;

On receipt of a *release from* p_i *at* p_j

if (queue of *request*s is non-empty)
then
remove head of queue – message from p_k, say;
send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

Qualitative Properties

- Safety: "No two processes are in the critical section at the same time"
  – Indirect proof:
    p_i and p_j both in the critical section $\Rightarrow V_i \cap V_j \neq \emptyset$

- Liveness: Deadlocks can occur, e.g., consider
  – $V_1 = \{p_1, p_2\}$, $V_2 = \{p_2, p_3\}$, $V_3 = \{p_3, p_1\}$
  – Suppose, p_1, p_2, p_3 concurrently send out requests
  – …

How could a deadlock be resolved?

And why is "$p_i \in V_i$" needed? Or is it?

Resolution of Deadlocks

- Process queues pending requests in "happened before" order

- Deadlock resolution:
  If node discovers that it has agreed to a "wrong" request (i.e., to a later request while an earlier request arrives only now),
  – it checks whether the requesting node is waiting or is in the critical section
  – revokes agreement to waiting nodes

- Why does it work?
  – Order is the same everywhere!

Quantitative Properties

Assume:

- all sets have the same size, say K
- there are M sets

- Bandwidth: 3K
- Client Delay: 2
- Synchronisation Delay: 2

- Questions:
  - How to choose K, i.e., the size of the voting sets?
  - Which value for M, i.e, how many different sets are best?
  - How can one choose the voting sets?

Maekawa's Algorithm: Optimised !?

On initialization
  state := RELEASED;
  voted := FALSE;

For p_i ***to enter the critical section***
  state := WANTED;
  Multicast *request* to all processes in $V_i - \{p_i\}$;
  Wait until (number of *repli*es received = $|V_i| - 1$);
  state := HELD;

On receipt *of a request from* p_i *at* p_j
  if (*state* = HELD *or voted* = TRUE)
  then
    queue *request* from p_i without replying;
  else
    send *reply* to p_i;
    voted := TRUE;
  end if

Continues on next slide

Maekawa's Algorithm: Optimised !?

For p_i to ***exit the critical section***
state := RELEASED;
Multicast *release* to all processes in $V_i - \{p_i\}$;

On receipt of a *release from p_i at p_j ($i \neq j$)*
if (queue of *request*s is non-empty)
then
remove head of queue – message from p_k, say;
send *reply* to p_k;
voted := TRUE;
else
voted := FALSE;
end if

Coordination and Agreement

7.3 Agreement

1. Leader Election
2. Mutual Exclusion
3. **Agreement**

Consensus Algorithms

- Used when it is necessary to agree on actions:
  - in transaction processing
    commit or abort a transaction?
  - mutual exclusion
    which process is allowed to access the resource?
  - in control systems
    proceed or abort based on sensor readings?
- The Consensus Problem is equivalent to other problems
  - e.g. reliable and totally ordered multicast

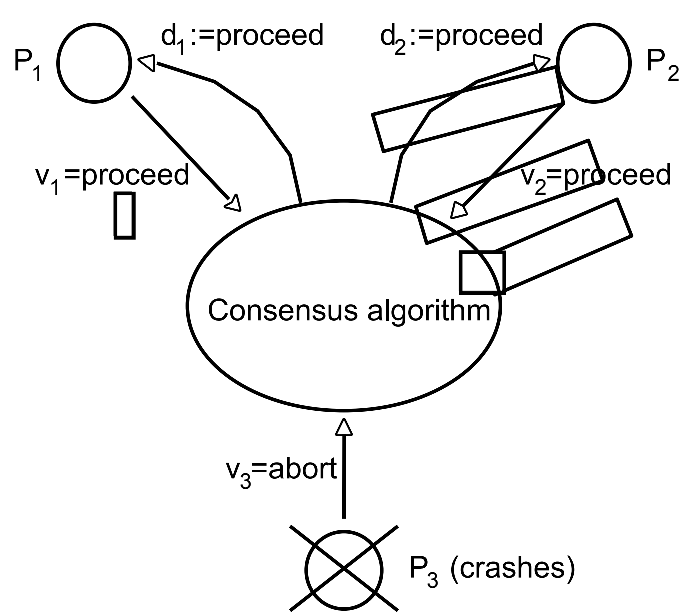
Model and Assumptions

- The model
  - N processes
  - communication by message passing
  - synchronous or asynchronous
  - communication reliable
- Failures!
  - Processes may crash
  - arbitrary (Byzantine) failures
    - processes can be treacherous and lie
- Algorithms
  - work in the presence of certain failures

Consensus: Main Idea

- Initially
  – processes begin in state “undecided”
  – 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!

Three Processes Reach a Consensus



Consensus: Requirements

- Termination
  - Eventually each correct process sets its decision variable.
- Agreement
  - Any two correct processes must have set their variable to the same decision value

    $\Rightarrow$ processes must have reached "decided" state
- Integrity
  - If all correct processes propose the same value, then any correct process that has decided must have chosen that value.

Ideas towards a Solution

- For simplicity, we assume no failures
  - processes *multicast* their proposed values to others
  - wait until they have collected all N values (including the own)
  - choose most frequent value among $v_1,\ldots,v_n$ (or special value $\perp$)
    - can also use minimum/maximum
- It works since ...
  - if multicast is reliable (Termination)
  - 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

Consensus in Synchronous Systems

- Uses basic multicast (= messages are sent individually)
  – guaranteed delivery by correct processes as long as the sender does not crash
- Admits process crash failures (but not byzantine failures)
  – assume up to f of the N processes may crash
- How it works ...
  – f +1 rounds
  – relies on synchronicity (timeout!)

Consensus in Synchronous Systems

- 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 the other ones (only values not sent before, that is, $V_{r-1} – V_{r-2}$)
  – receives multicast messages, records new values in V_r
- In round f+1
  – each process chooses $\min(V_{f+1})$ as decision value

The Algorithm (Dolev and Strong)

Algorithm for process p_i, proceeds in f+1 rounds

On initialization
$V_i^0 = \{v_i\}$; $V_i^{-1} = \{\}$

In round r ($1 \leq r \leq f+1$)
multicast($V_i^{r-1} - V_i^{r-2}$) // send only values that have not been sent
$V_i^r = V_i^{r-1}$
while (in round r)
{
if (p_j delivers V_j)
$V_i^r = V_i^r \cup V_j$
}

After f+1 rounds
$p_j = \min(V_i^{f+1})$

Consensus in Synchronous Systems

- Why does it work?
  - set timeout to maximum time for correct process to multicast message
  - one can conclude that process crashed if no reply
  - if process crashes, some value is not forwarded ...
- At round f+1
  - assume p_1 has a value v that p_2 does not have
  - then some p_3 managed to send v to p_1, but no more to p_2
    $\Rightarrow$ any process sending v in round f must have crashed (otherwise, both p_3 and p_2 would have received v)
  - in this way, in each round one process has crashed
  - there were f+1 rounds, but only f crashes could occur

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.

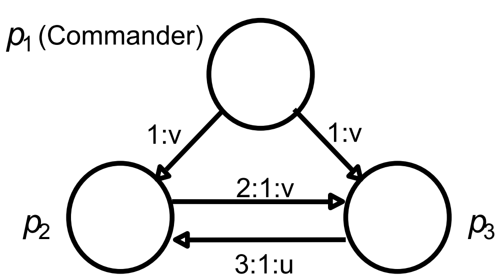
Byzantine Generals …

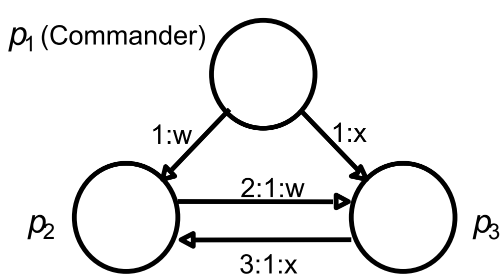
- Processes exhibit arbitrary failures
  - up to f of the N processes faulty
- In a synchronous system
  - can use timeout to detect absence of a message
  - cannot conclude process crashed if no reply
  - impossibility with $N \leq 3f$
- In an asynchronous system
  - cannot use timeout to reliably detect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

Impossibility with 3 Generals

- Assume synchronous system
  - 3 processes, one faulty
  - if no message received, assume ⊥
  - proceed in rounds
  - messages ‘3:1:u’ meaning ‘3 says 1 says u’
- Problem! ‘1 says v’ and ‘3 says 1 says u’
  - cannot tell which process is telling the truth!
  - goes away if digital signatures used...
- Show
  - no solution to agreement for N=3 and f=1
- Can generalise to impossibility for N ≤ 3f

Impossibility with 3 Generals





Faulty processes are shown in red

p3 sends illegal value p2
p2 cannot tell which value sent by commander

Commander faulty
p2 cannot tell which value sent by commander

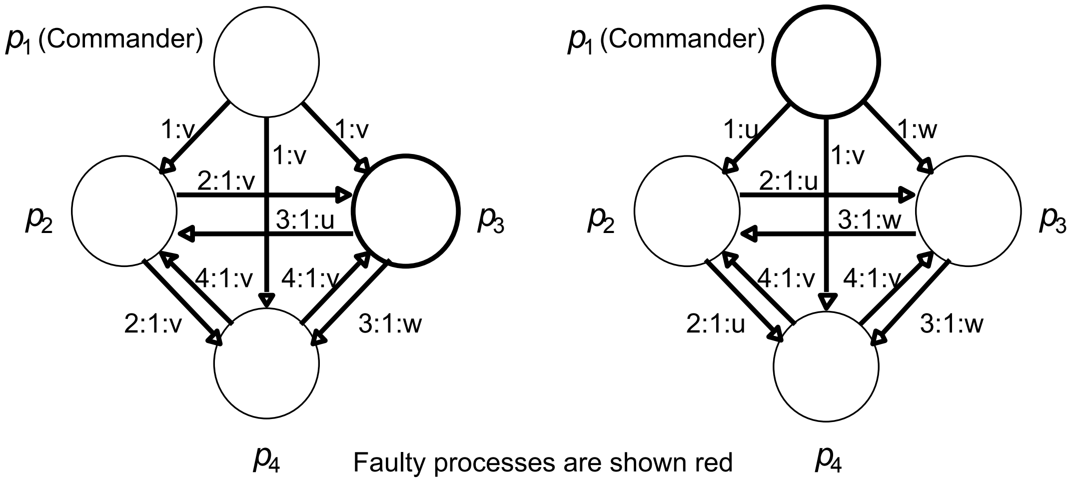
Impossibility with 3 Generals

- So, if the solution exists
  - p_2 decides on value sent by commander (v) when the commander is correct
  - and also when commander faulty (w), since it cannot distinguish between the two scenarios
- Apply the same reasoning to p_3
  - conclude p_3 must decide on x when commander faulty
- Thus
  - contradiction to Agreement!
    since p_2 decides on w, p_3 on x if commander faulty
  - no solution exists

But …

- A 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, then 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 $N \geq 3f + 1$

Four Byzantine Generals



Faulty processes are shown red

p2 decides majority(v,u,v) = v
p4 decides majority(v,v,w) = v

p2, p3 and p4 decide ⊥
(no majority exists)

In Asynchronous Systems …

- No guaranteed solution exists even for one failure!!!
  [Fisher, Lynch, Paterson ‘85]
  – does not exclude the possibility of consensus in the presence of failures
  – consensus can be reached with positive probability
- How can this be true?
  – The Internet is asynchronous, exhibits arbitrary failures and uses consensus?
- Practical solutions exist using
  – failure masking (processes restart after crash)
  – treatment of slow processes as “dead”
  (partially synchronous systems)
  – randomisation

Summary

- Consensus algorithms
  - are fundamental to achieve co-ordination
  - deal with crash or arbitrary (=Byzantine) failures
  - are subject t several impossibility results
- Solutions exist for synchronous systems
  - if at most f crash failures, in f+1 rounds
  - if no more than f processes of N are faulty, $N \geq 3f + 1$
- Solutions for asynchronous systems
  - no guaranteed solution even for one failure!
  - practical solutions exist

References

In preparing the lectures I have used several sources.
The main ones are the following:

Books:

- Coulouris, Dollimore, Kindberg. Distributed Systems – Concepts and Design (CDK)

Slides:

- Marco Aiello, course on Distributed Systems at the Free University of Bozen-Bolzano
- CDK Website
- Marta Kwiatkowska, U Birmingham, slides of course on DS