The total number of phases is at most

$$1 + \lceil \log N \rceil$$

the total number of messages is at most $8N(1 + \lceil \log N \rceil) \sim \mathcal{O}(N \log N)$ with constant factor of approx 8
Time Complexity is at most 3N if N is power of 2, otherwise 5N.

11 Mutual Exclusion (week 9)

In a monlithic system a critical section can be protected through mutual exclusion which is achieved by means of semaphores, monitors, or similar.

In a distributed system a critical section can be data that is distributed across various nodes. On several nodes we cannot lock with semaphores. New concepts are needed.

Mutual exclusion serves

- access of shared resource(eg. data)
- required atomic operations
- assumes no link or mode(process) failures

We look at three types of algorithms:

– time-stamp based

– token-based

– quorum-based

Since in a distributed system we cannot talk about events happening at the same time, we formalize mutual exclusion as requirement that no two processes access a shared resource in concurrent states.

Problem: Let a system consist of a fixed number of nodes and a critical section. A mutual exclusion algorithm must satisfy:

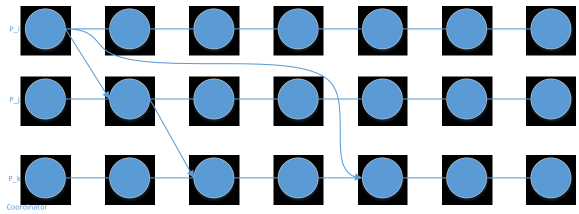
Safety: Two processes must not have permission to access the critical section in concurrent states (nothing bad will happen)

Liveness: Every request is eventually granted (something good will happen)

Fairness: Request must be granted in the order they are issued.

Remark :
The best and least expensive algorithm is centralized.

- safety and liveness are satisfied by a simple queue-based algorithm. One process is coordinator.



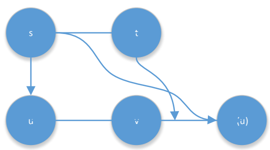
Why it this algorithm not fair?

Necessary assumptions:

– channels are reliable, no loss, no malicious insertions.

– channels are FIFO(first in, first out)

$$s \prec t \wedge s \rightsquigarrow u \wedge t \rightsquigarrow v \Rightarrow \underset{\text{FIFO}}{\neg(v \prec u)}$$



– define req(s) $\triangleq$ true, iff process $P_{s.p}$ has requested and not yet released critical section

– define cs(s) $\triangleq$ true, iff process $P_{s.p}$ has permission to enter critical section in state s

Formalize the properties:
suppose $t \prec u \prec v$ (request for cs in t access is granted in u released in v)
A process that is granted access eventually releases it:

$$cs(s) \Rightarrow (\exists t : s \prec t : \neg(t))\text{cooperation}$$

Further:

$$(s||t) \wedge (s \neq t) \Rightarrow \neg(cs(s) \wedge cs(t))\text{safety}$$

$$reg(s) \Rightarrow (\exists t : s \not\succeq t \wedge cs(t))\text{liveness}$$

Let next_cs(s) $\stackrel{def}{=} min\{t|s \not\succeq t \wedge cs(t)\}$
Let req_start(s) $\stackrel{def}{=} req(s)\neg req(prev(s))$
(req_start is true only if $P_{s.p}$ first made a request for that cs in s)

$$(req.start(s) \wedge req.start(t) \wedge s \rightarrow t) \Rightarrow nex\_cs(s) \rightarrow next\_cs(t) \text{ fairness}$$

Remark:

$$next\_cs(s) \rightarrow next\_cs(t) \Leftrightarrow \neg(next\_cs(t)) \rightarrow next\_cs(s))$$

11.1 Lamport's Algorithm

informal:

– each process has a logical clock and a queue:

Rules:

- to request a critical section (cs) a process sends a time-stamped message to all other processes and adds a time-stamped request to its queue.
- on receiving a reuquest message, the request and its time-stamp are stored in the queue and an acknowledgement is returned.
- to release, a process sends a release message to all other processes
- on receiving a release message, the corresponding request is deleted from the queue.
- a process determines that it can access the critical section iff
  1. it has a reuqest in the queue with time-stamp t
  2. t is smaller than all other requests in the queue
  3. it has received a message from all other processes with time-stamp $> t$ (ack).

formal algorithm

P_i:

```
var v: vector clock
    q: array[1..N] of int initially [∞..∞]
request
    q[i]:= V[i];
    send(q[i]) to all processes
release:
    q[i]:= ∞;
    send(q[i]) to all processes
receive(n)
    q[n.p]:= u.q[u.p];
    if event(u) = request then
        send ack to u.p
    end
```

Every process has two vectors to represent the queues:

s.q[j] timestamp of request by process P_j

s.v[j] timestamp of the last message from P_j
if $j \neq i, s.v[i]$ is logical clock in state s.

Complexity: $3(N - I)$messages nof N processes
Ricart & Agrawala's algorithm needs only $2(N - I)$ messages

Ricart and Agrawala's algorithm needs only $2(N - 1)$ messages. It combines ack and release. Channels need not be FIFO, ack may be sent later.

Rules:

Lamport

```
Pi:
var v: vector clock
  q: array[1..N] of int initially [∞..∞]
request
     q[i]:= V[i];
     send(q[i]) to all processes
release:
     q[i]:= ∞;
     send(q[i]) to all processes
receive(n)
     q[n.p]:= u.q[u.p];
     if event(u) = request then
          send ack to u.p
     end
```

Ricest and Agrawala

```
Pi:
var pendingQ: list of pro ids initially NaN
     myts: int initially ∞
     numOK: int initially 0
request
    myts:= logical clock
    send request with my myts to all other pro
    num OK := 0:
receive(u, request)
    if(u.myts < myts) then
       send OK to u.p
    else
       append( pendingQ, u.p);
receive(u, OK)
     numOK++
     if(numOK = N−1) then
          enterCS;
release:
     myts = ∞
     for j ∈ pendingQ do
          send OK to j
     pendingQ:= NULL;
```

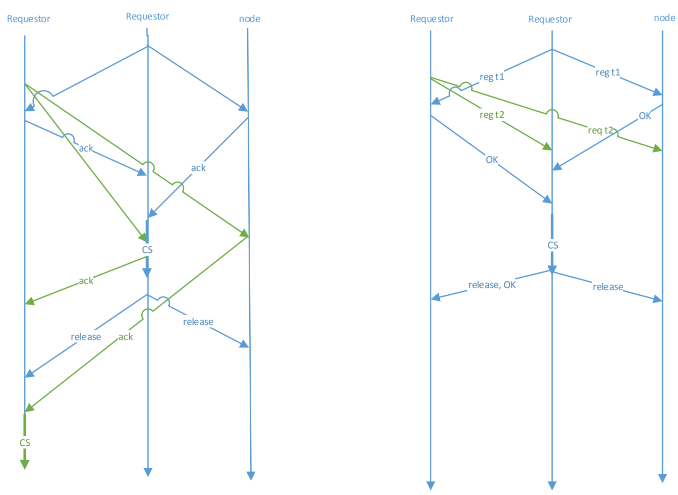


Figure 18: Example timeline for the two algorithms

- to request, a process sends a time-stamped message to all processes
- on receibing a request from another process the process sends ok message if either the process is not interested in the critical section or its own request has a higher time-stamp value. Otherwise, that process is kept in a pending queue.
- to release a critical section, process P_i sends an ok message to all processes in the pending queue
- process P_i is granted the critical section when it has requested the critical section and it has received an ok message from all other processes in response to the request message.