- 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.

11.2 Token-Based Algorithms (week 10)

11.2.1 A Centralized Algorithm

- coordinator P_0 for administration of token
- process holding token can access critical section
- safety and liveness guaranteed by coordinator
- fairness? $s \prec t$ then s served before t, even if request arrives first

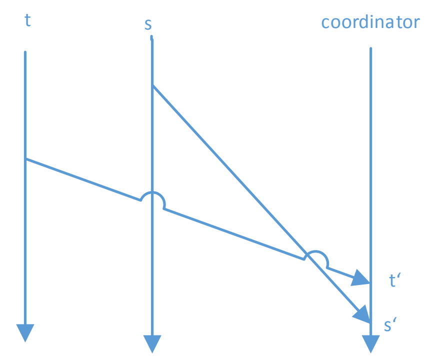


Figure 19: Coordinator must grant token in the order requests are generated

- request_list = requests that were not full filled
- request_done[i] # of requests by i that have been saved
- solution using logical clocks where s.v[j] $\triangleq$no. of requests made by process P_j
- on receiving request P_0 appends it to request_list
- if it has token, it checks for eligible requests and sends token to one them
- A request ω is eligible if ω.v is at most request_done, i.e. there is no request that happened before ω and has not yet been fulfilled.

algorithm(client)

```
P_i (client)
var v : array[1..,N] of int initially ∀ j v[j] = 0; (clocks, no. of requests)
    inCS: boolean initially false;
request
   v[i]++;
   send(request,v) to P_0

receive(token) from P_0
   in CS = true;

release
   send token to P_0
   incs = false

upon receive(n)(other message)
   c:= max(v,u.v);
```

Needs 3 messages per invocation of critical section.
Fails, if coordinator fails!(also fails when a token message is lost)

algorithm(coordinator)

```
P_0 (coordinator)
var reg\_done : array[1,..,N] of int initially 0
    req\_list : list of (pid, requestor) initially NULL;
    have\_token: boolean initially true;

receive(u, request)
    append(reg\_list , u);
    if     have\_token then checkreq();

receive(u, token)
        have\_token = true;
        checkreq();

checkreq()
```

eligible := $\{\omega \in \overset{\text{not fulfilled requests}}{request} \mid \forall j : j \neq \omega.p : \omega.v[j] < \underbrace{\underset{\text{fulfilled requests}}{request}}_{\text{there are no requests that happend before w and have not yet been fulfilled}}$

```
    if eligible ≠ \{\} then
       ω := first(eligible)
       delete(reqlist , ω);
       req\_done[ω.p]++; sendtokento P_{\omega .p};
       have\_token= false;
    end if
```

safety guranteed as a process s can only enter CS when s.has_token = true
i.e. $(s \neq t) \wedge (s \| t) \Rightarrow \neg(s.inCs \wedge t.inCS)$
liveness $r.req \Rightarrow \exists t : s \preccurlyeq t \wedge t.inCS$
Fairness $req\_start(s) \wedge req\_start(t) \wedge s \rightarrow t) \Rightarrow nextcs(s) \rightarrow next.cs(t)$

11.2.2 Decentralize the token algorithm

- replace the coordinator by token
- computation of coordinator is done by process holding the token
- the token carries variables:
  reqdone: array[1,..,N] of int initially 0
  reqlist: list of (pid, requestor) initially empty;
- request for token must go to all processes since it is not known who holds the token (could use use token ring)
- all nodes hold variable myRegList: list of (pid,reqlist) initially empty, that is used if token was in transit as the request was broadcast.

Decentralized algorithm(client)

```
Pi (client)
var v : array[1,..,N] of int initially ∀ j v[j] = 0; (clocks, no of requests)
    inCS: boolean initially false;
    havetoken : boolean initially false except for P0

request
   v[i]++;
   send(request,v) to all processes(including myself)

receive(request, u)
   v:=max(v,u.v)
   if havetoken then
      append(token.reqlist, u);
      if not inCS then checkreq();
      else append(myreqlist, u);

receive(u, token);
   inCS := true;
   havetoken := true;
   append(token.reqlist,{u|u ∈ myreqlist) ∧ (u > token.regdone)}
   myreqlist := NULL;

receive(u) // other message
v:= max(v,u.v);

release
   inCS := false;
   checkreg();
```

node holding token is single point of failure.

- N+2 messages for one access to CS
- $\mathcal{O}(N)$ messages

11.3 Quorum-based algorithms (week 11)

- no single point of failure
- permission from subset of processes(request set)
  Assume p = 0.1, i.e. if each node fails with Prob. 10%
  assume only votes from half of the nodes are needed to take decision, then half of the nodes may fail until the system is no longer able to come to a decision. This means, the system can tolerate $\frac{N}{2}$ failures, for N = 100 implies that $p^{50} = 0.1^{50} = (10^{-1})^{50} = 10^{-50}$ failure probability of the mutual exclusion algorithm is very low.
- permission from subset of processes (request set)
- if 2 request sets have non-empty intersection, we are guaranteed that at most one process can access cs
  (e.g. majority needs any subset with at least $[\frac{(N+1)}{2}]$ processes
- vote assignment is a set of group, s.t. each group has majority of votes

Define: Coterie: Let $U = \{u_1, u_2, ..., u_n\}$ be a set and C a non-empty family of subsets, called quorums of U.

Definition Coterie: $C = \{Q_1, Q_2, ..., Q_m\}$ where $\forall 1 \leq i \leq m : Q_i \subseteq U$ is a coterie under U if:

- No quorum is a subset of any other quorum

$$\forall i, j \; : \; i \neq j \neg (Q_i \subseteq Q_j) \text{ (Minimality)}$$

- Every two corums intersect

$$\forall i, j \; : \; Q_i \cap Q_j \neq \emptyset$$

Let C and D be coteries, C dominates D if C $\neq$ D and $\forall Q \in D : (\exists Q' \in C : Q' \subseteq Q)$
A coterie C is non dominated, if no coterie dominates C. Intuitively non-dominant coteries are in some sense optimal, because they can not be further reduced. A smaller coterie is better for reasons of algorithmic complexity.

Metrics for quorums:

1. Quorum size, smaller quorum needs less messages
2. Availability, probability p that there is at least one live quorum in the coterie
3. Load on the busiest node

These metrics are in conflict. There exists no quorum system that is optimal in all metrics.

Voting Systems

- Each process is assigned a number of votes (e.g. 1)
- Let the total number be V
  a quorum is a subset of processes with at least $\frac{V}{2}$ votes.
- Example: read(R) and write(W) quorums, such that

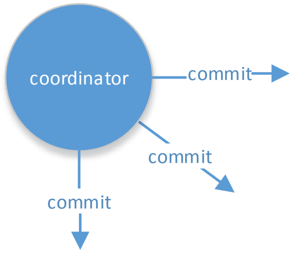


Figure 20:

1. R + W > V and
2. W > $\frac{V}{2}$

For consistency when there are V replicas of the data across the nodes.

1. prevents read-write conflicts
2. prevent w-w conflicts

To Read correct value R replicas must be read to write W replicas must be written.

12 Consistency and Consensus

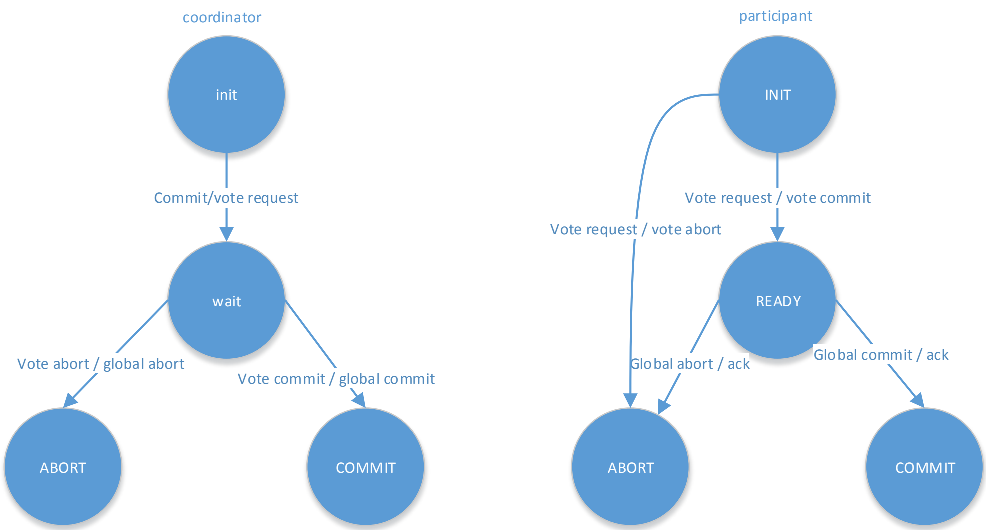
- distributed systems use replication to improve performance and/or reliability
- replication for scalability, since it allows to read (and write) in parallel
  How to keep replicas consistent?

12.1 distributed commit

Requirements and properties

- an operation is performed by a group or none of the group.
- reliable multicast: operations = delivery of a message
- distributed transactions: operations = executions of a transaction
- uses often coordinator
- one-phase commit:
- two-phase commit (2 PC)   Jim Gray(1978)
- distributed transaction involves several processors each on a different machine
  2 phases with each 2 steps:

1. coordinator $\xrightarrow{\text{vote request}}$ all participants.
2. participant $\underset{\text{vote abort}}{\xrightarrow{\text{vote commit}}}$ coordinator
3. if all commit, coordinator $\xrightarrow{\text{global commit}}$ all participants
   else coordinator $\xrightarrow{\text{global abort}}$ all participants.
4. if commit
   participant locally commits
   else
   participant locally aborts.



Problems if failures occur:

– coordinator blocks in state wait

– participant blocks in state ready

→ blocking commit protocol

– use "timeout" to unblock

– in state "ready" participant P can contact Q

  – if Q in in commit, coordinator died after sending to Q before sending to P ⇒ P can commit
  – if Q is in ABORT ⇒ ABORT
  – if Q is in INIT ⇒ ABORT
  – if Q ready ⇒ no decision, contact other participant

Blocking is resolved in 3 phase commit.