Maekawa's Algorithm

- Permission obtained from only a subset of other processes, called the *Request Set* (or *Quorum*)

- Separate Request Set, R_i, for each process i

- Requirements:
  - for all i, j: $R_i \cap R_j \neq \Phi$
  - for all i: $i \in R_i$
  - for all i: $|R_i| = K$, for some K
  - any node i is contained in exactly *D* Request Sets, for some *D*

- $\mathbf{K} = \mathbf{D} = \sqrt{\mathbf{N}}$ for Maekawa's

A Simple Version

- To request critical section:
  - i sends REQUEST message to all process in R_i
- On receiving a REQUEST message:
  - Send a REPLY message if no REPLY message has been sent since the last RELEASE message is received.
  - Update status to indicate that a REPLY has been sent.
  - Otherwise, queue up the REQUEST
- To enter critical section:
  - i enters critical section after receiving REPLY from all nodes in R_i

A Simple Version contd..

- To release critical section:
  - Send RELEASE message to all nodes in R_i
  - On receiving a RELEASE message, send REPLY to next node in queue and delete the node from the queue.
  - If queue is empty, update status to indicate no REPLY message has been sent.

Features

- Message Complexity: $3 * \sqrt{\mathbf{N}}$
- Synchronization delay =
  - 2*(max message transmission time)
- Major problem: DEADLOCK possible
- Need three more types of messages (FAILED, INQUIRE, YIELD) to handle deadlock.
  - Message complexity can be 5*sqrt(N)
- Building the request sets?

Token based Algorithms

- Single token circulates, enter CS when token is present
- Mutual exclusion obvious
- Algorithms differ in how to find and get the token
- Uses sequence numbers rather than timestamps to differentiate between old and current requests

Suzuki Kasami Algorithm

- Broadcast a request for the token
- Process with the token sends it to the requestor if it does not need it
- Issues:
  - Current versus outdated requests
  - Determining sites with pending requests
  - Deciding which site to give the token to

Suzuki Kasami Algorithm

- The token:
  - Queue (FIFO) Q of requesting processes
  - LN[1..n] : sequence number of request that j executed most recently

- The request message:
  - REQUEST(i, k): request message from node i for its k^{th} critical section execution

- Other data structures
  - RN_i[1..n] for each node i, where $RN_i[j]$ is the largest sequence number received so far by i in a REQUEST message from j.

Suzuki Kasami Algorithm

- To request critical section:
  - If i does not have token, increment $RN_i[\ i\ ]$ and send REQUEST(i, $RN_i[\ i\ ]$) to all nodes
  - If i has token already, enter critical section if the token is idle (no pending requests), else follow rule to release critical section

- On receiving REQUEST(i, sn) at j:
  - Set $RN_j[\ i\ ] = \max(RN_j[\ i\ ], sn)$
  - If j has the token and the token is idle, then send it to i if $RN_j[\ i\ ] = LN[\ i\ ] + 1$. If token is not idle, follow rule to release critical section

Suzuki Kasami Algorithm

- To enter critical section:
  - Enter CS if token is present

- To release critical section:
  - Set LN[i] = RN_i[i]
  - For every node j which is not in Q (in token), add node j to Q if RN_i[j] = LN[j] + 1
  - If Q is non empty after the above, delete first node from Q and send the token to that node

Notable features

- No. of messages:
  - 0 if node holds the token already, n otherwise
- Synchronization delay:
  - 0 (node has the token) or max. message delay (token is elsewhere)
- No starvation

Raymond’s Algorithm

- Forms a directed tree (logical) with the token-holder as root
- Each node has variable “*Holder*” that points to its parent on the path to the root.
  - Root’s Holder variable points to itself
- Each node i has a FIFO request queue Q_i