Mutual Exclusion Algorithms

- Non-token based:
  - A site/process can enter a critical section when an assertion (condition) becomes true.
  - Algorithm should ensure that the assertion will be true in only one site/process.
- Token based:
  - A unique token (a known, unique message) is shared among cooperating sites/processes.
  - Possessor of the token has access to critical section.
  - Need to take care of conditions such as loss of token, crash of token holder, possibility of multiple tokens, etc.
General System Model

- At any instant, a site may have several requests for critical section (CS), queued up, and serviced one at a time.
- Site States: Requesting CS, executing CS, idle (neither requesting nor executing CS).
- Requesting CS: blocked until granted access, cannot make additional requests for CS.
- Executing CS: using the CS.
- Idle: action is outside the site. In token-based approaches, *idle* site can have the token.
Mutual Exclusion: Requirements

- Freedom from deadlocks: two or more sites should not endlessly wait on conditions/messages that never become true/arrive.
- Freedom from starvation: No indefinite waiting.
- Fairness: Order of execution of CS follows the order of the requests for CS. (equal priority).
- Fault tolerance: recognize “faults”, reorganize, continue. (e.g., loss of token).
Performance

- Number of messages per CS invocation: should be minimized.
- Synchronization delay, i.e., time between the leaving of CS by a site and the entry of CS by the next one: should be minimized.
- Response time: time interval between request messages transmissions and exit of CS.
- System throughput, i.e., rate at which system executes requests for CS: should be maximized.
- If $sd$ is synchronization delay, $E$ the average CS execution time: system throughput = $1 / (sd + E)$. 
Performance metrics

Last site exits CS

Next site enters CS

Synchronization delay

CS Request arrives

Messages sent

Enter CS

Exit CS

Response Time

Time
Performance ...

- **Low and High Load:**
  - Low load: No more than one request at a given point in time.
  - High load: Always a pending mutual exclusion request at a site.

- **Best and Worst Case:**
  - Best Case (low loads): Round-trip message delay + Execution time. \(2T + E\).
  - Worst case (high loads).

- **Message traffic:** low at low loads, high at high loads.

- **Average performance:** when load conditions fluctuate widely.
Simple Solution

- Control site: grants permission for CS execution.
- A site sends REQUEST message to control site.
- Controller grants access one by one.
- Synchronization delay: 2T -> A site releases CS by sending message to controller and controller sends permission to another site.
- System throughput: 1/(2T + E). If synchronization delay is reduced to T, throughput doubles.
- Controller becomes a bottleneck, congestion can occur.
Non-token Based Algorithms

- **Notations:**
  - $S_i$: Site $i$
  - $R_i$: Request set, containing the ids of all $S_i$s from which permission must be received before accessing CS.
  - Non-token based approaches use time stamps to order requests for CS.
  - Smaller time stamps get priority over larger ones.

- **Lamport’s Algorithm**
  - $R_i = \{S_1, S_2, \ldots, S_n\}$, i.e., all sites.
  - Request queue: maintained at each $S_i$. Ordered by time stamps.
  - Assumption: message delivered in FIFO.
Lamport’s Algorithm

- **Requesting CS:**
  - Send REQUEST\((tsi, i)\). \((tsi,i): \) Request time stamp. Place REQUEST in \(request\_queue\_i\).
  - On receiving the message; \(sj\) sends time-stamped REPLY message to \(si\). \(si\)’s request placed in \(request\_queue\_j\).

- **Executing CS:**
  - \(si\) has received a message with time stamp larger than \((tsi,i)\) from all other sites.
  - \(si\)’s request is the top most one in \(request\_queue\_i\).

- **Releasing CS:**
  - Exiting CS: send a time stamped RELEASE message to all sites in its request set.
  - Receiving RELEASE message: \(sj\) removes \(si\)’s request from its queue.
Lamport’s Algorithm…

- **Performance.**
  - 3(N-1) messages per CS invocation. (N - 1) REQUEST, (N - 1) REPLY, (N - 1) RELEASE messages.
  - Synchronization delay: T

- **Optimization**
  - Suppress reply messages. (e.g.,) Sj receives a REQUEST message from Si after sending its own REQUEST message with time stamp higher than that of Si’s. Do NOT send REPLY message.
  - Messages reduced to between 2(N-1) and 3(N-1).
Lamport’s Algorithm: Example

**Step 1:**
- **S1**: (2,1)
- **S2**: (1,2)
- **S3**: (1,2) (2,1)

**Step 2:**
- **S1**: (1,2) (2,1)
- **S2**: S2 enters CS
- **S3**: (1,2) (2,1)
Lamport’s: Example…

**Step 3:**

- S1
- S2 leaves CS
- S3

**Step 4:**

- S1 enters CS
- S2
- S3
Ricart-Agrawala Algorithm

- Requesting critical section
  - Si sends time stamped REQUEST message
  - Sj sends REPLY to Si, if
    - Sj is not requesting nor executing CS
    - If Sj is requesting CS and Si’s time stamp is smaller than its own request.
    - Request is deferred otherwise.
- Executing CS: after it has received REPLY from all sites in its request set.
- Releasing CS: Send REPLY to all deferred requests. i.e., a site’s REPLY messages are blocked only by sites with smaller time stamps
Ricart-Agrawala: Performance

- **Performance:**
  - 2(N-1) messages per CS execution. (N-1) REQUEST + (N-1) REPLY.
  - Synchronization delay: T.

- **Optimization:**
  - When Si receives REPLY message from Sj -> authorization to access CS till
    - Sj sends a REQUEST message and Si sends a REPLY message.
    - Access CS repeatedly till then.
  - A site requests permission from dynamically varying set of sites: 0 to 2(N-1) messages.
Ricart-Agrawala: Example

**Step 1:**
- S1
- S2
- S3

**Step 2:**
- S1
- S2
- S3

S2 enters CS
Step 3:

S1 enters CS

S1

S2

(2,1)

S3

S2 leaves CS

S1 enters CS
Maekawa’s Algorithm

- A site requests permission only from a subset of sites.
- Request set of sites $si \& sj$: $R_i, R_j$ such that $R_i$ and $R_j$ will have at least one common site ($S_k$). $S_k$ mediates conflicts between $R_i$ and $R_j$.
- A site can send only one REPLY message at a time, i.e., a site can send a REPLY message only after receiving a RELEASE message for the previous REPLY message.
- Request Sets Rules:
  - Sets $R_i$ and $R_j$ have at least one common site.
  - $Si$ is always in $R_i$.
  - Cardinality of $R_i$, i.e., the number of sites in $R_i$ is $K$.
  - Any site $Si$ is in $K$ number of $R_i$s. $N = K(K - 1) + 1 \rightarrow K = \text{square root of } N$. 
Maekawa’s Algorithm ...

- **Requesting CS**
  - Si sends REQUEST(i) to sites in Ri.
  - Sj sends REPLY to Si if
    - Sj has NOT sent a REPLY message to any site after it received the last RELEASE message.
    - Otherwise, queue up Si’s request.

- **Executing CS**: after getting REPLY from all sites in Ri.

- **Releasing CS**
  - send RELEASE(i) to all sites in Ri
  - Any Sj after receiving RELEASE message, send REPLY message to the next request in queue.
  - If queue empty, update status indicating receipt of RELEASE.
Maekawa’s Algorithm ...

- **Performance**
  - Synchronization delay: 2T
  - Messages: 3 times square root of N (one each for REQUEST, REPLY, RELEASE messages)

- **Deadlocks**
  - Message deliveries are not ordered.
  - Assume Si, Sj, Sk concurrently request CS
  - $R_i \cap R_j = \{S_{ij}\}$, $R_j \cap R_k = \{S_{jk}\}$, $R_k \cap R_i = \{S_{ki}\}$
  - Possible that:
    - $S_{ij}$ is locked by $S_i$ (forcing $S_j$ to wait at $S_{ij}$)
    - $S_{jk}$ by $S_j$ (forcing $S_k$ to wait at $S_{jk}$)
    - $S_{ki}$ by $S_k$ (forcing $S_i$ to wait at $S_{ki}$)
  - $\Rightarrow$ deadlocks among $S_i$, $S_j$, and $S_k$. 
Handling Deadlocks

- Si *yields* to a request if that has a smaller time stamp.
- A site suspects a deadlock when it is locked by a request with a higher time stamp (lower priority).
- Deadlock handling messages:
  - FAILED: from Si to Sj -> Si has granted permission to higher priority request.
  - INQUIRE: from Si to Sj -> Si would like to know Sj has succeeded in locking all sites in Sj’s request set.
  - YIELD: from Si to Sj -> Si is returning permission to Sj so that Sj can yield to a higher priority request.
Handling Deadlocks

- **REQUEST**(tsi,i) to Sj:
  - Sj is locked by Sk -> Sj sends FAILED to Si, if Si’s request has higher time stamp.
  - Otherwise, Sj sends INQUIRE(j) to Sk.

- **INQUIRE**(j) to Sk:
  - Sk sends a YIELD (k) to Sj, if Sk has received a FAILED message from a site in Sk’s set. (or) if Sk sent a YIELD and has not received a new REPLY.

- **YIELD**(k) to Sj:
  - Sj assumes it has been released by Sk, places Sk’s request in its queue appropriately, sends a REPLY(j) to the top request in its queue.

- Sites may exchange these messages even if there is no real deadlock. Maximum number of messages per CS request: 5 times square root of N.
Token-based Algorithms

- Unique token circulates among the participating sites.
- A site can enter CS if it has the token.
- Token-based approaches use sequence numbers instead of time stamps.
  - Request for a token contains a sequence number.
  - Sequence number of sites advance independently.
- Correctness issue is trivial since only one token is present -> only one site can enter CS.
- Deadlock and starvation issues to be addressed.
Suzuki-Kasami Algorithm

- If a site without a token needs to enter a CS, broadcast a REQUEST for token message to all other sites.
- Token: (a) Queue of request sites (b) Array LN[1..N], the sequence number of the most recent execution by a site j.
- Token holder sends token to requestor, if it is not inside CS. Otherwise, sends after exiting CS.
- Token holder can make multiple CS accesses.
- Design issues:
  - Distinguishing outdated REQUEST messages.
    - Format: REQUEST(j,n) -> jth site making nth request.
    - Each site has RNi[1..N] -> RNi[j] is the largest sequence number of request from j.
  - Determining which site has an outstanding token request.
    - If LN[j] = RNi[j] + 1, then Sj has an outstanding request.
Suzuki-Kasami Algorithm ...

- **Passing the token**
  - After finishing CS
  - (assuming Si has token), LN[i] := RNi[i]
  - Token consists of Q and LN. Q is a queue of requesting sites.
  - Token holder checks if RNi[j] = LN[j] + 1. If so, place j in Q.
  - Send token to the site at head of Q.

- **Performance**
  - 0 to N messages per CS invocation.
  - Synchronization delay is 0 (if the token holder repeats CS) or T.
## Suzuki-Kasami: Example

**Step 1:** S1 has token, S3 is in queue

<table>
<thead>
<tr>
<th>Site</th>
<th>Seq. Vector RN</th>
<th>Token Vect. LN</th>
<th>Token Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10, 15, 9</td>
<td>10, 15, 8</td>
<td>3</td>
</tr>
<tr>
<td>S2</td>
<td>10, 16, 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>10, 15, 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Step 2:** S3 gets token, S2 in queue

<table>
<thead>
<tr>
<th>Site</th>
<th>Seq. Vector RN</th>
<th>Token Vect. LN</th>
<th>Token Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10, 16, 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S2</td>
<td>10, 16, 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>10, 16, 9</td>
<td>10, 15, 9</td>
<td>2</td>
</tr>
</tbody>
</table>

**Step 3:** S2 gets token, queue empty

<table>
<thead>
<tr>
<th>Site</th>
<th>Seq. Vector RN</th>
<th>Token Vect. LN</th>
<th>Token Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>10, 16, 9</td>
<td>10, 16, 9</td>
<td>&lt;empty&gt;</td>
</tr>
<tr>
<td>S2</td>
<td>10, 16, 9</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S3</td>
<td>10, 16, 9</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Singhal’s Heuristic Algorithm

Instead of broadcast: each site maintains information on other sites, guess the sites likely to have the token.

Data Structures:
- Si maintains SVi[1..N] and SNi[1..N] for storing information on other sites: state and highest sequence number.
- Token contains 2 arrays: TSV[1..N] and TSN[1..N].
- States of a site:
  - R : requesting CS
  - E : executing CS
  - H : Holding token, idle
  - N : None of the above
- Initialization:
  - SVi[j] := N, for j = N .. i; SVi[j] := R, for j = i-1 .. 1; SNi[j] := 0, j = 1..N. S1 (Site 1) is in state H.
  - Token: TSV[j] := N & TSN[j] := 0, j = 1 .. N.
Singhal’s Heuristic Algorithm …

- Requesting CS
  - If Si has no token and requests CS:
    - \(SVi[i] := R\). \(SNi[i] := SNi[i] + 1\).
    - Send \(\text{REQUEST}(i,sn)\) to sites \(Sj\) for which \(SVi[j] = R\). (\(sn\): sequence number, updated value of \(SNi[i]\)).
  - Receiving \(\text{REQUEST}(i,sn)\): if \(sn \leq SNj[i]\), ignore. Otherwise, update \(SNj[i]\) and do:
    - \(SVj[j] = N \rightarrow SVj[i] := R\).
    - \(SVj[j] = R \rightarrow \) If \(SVj[i] \neq R\), set it to \(R\) & send \(\text{REQUEST}(j,SNj[j])\) to \(Si\). Else do nothing.
    - \(SVj[j] = E \rightarrow SVj[i] := R\).
  - Executing CS: after getting token. Set \(SVi[i] := E\).
Singhal’s Heuristic Algorithm …

- **Releasing CS**
  - $SV_i[1] := N$, $TSV[i] := N$. Then, do:
    - For other $S_j$: if $(SN_i[j] > TSN[j])$, then \{ $TSV[j] := SV_i[j]$; $TSN[j] := SN_i[j]$\}
  - If $SV_i[j] = N$, for all $N$, then set $SV_i[i] := H$. Else send token to a site $S_j$ provided $SV_i[j] = R$.

- Fairness of algorithm will depend on choice of $S_i$, since no queue is maintained in token.
- Arbitration rules to ensure fairness used.

- **Performance**
  - Low to moderate loads: average of $N/2$ messages.
  - High loads: $N$ messages (all sites request CS).
  - Synchronization delay: $T$. 
Singhal: Example

- Assume there are 3 sites in the system. Initially:
  Token: TSVs are N. TSNs are 0.

- Assume site 2 is requesting token.
  S2 sends REQUEST(2,1) to S1 (since only S1 is set to R in SV[2])

- S1 receives the REQUEST. Accepts the REQUEST since SN1[2] is smaller than the message sequence number.
  Send token to S2

  Updates SN, SV, TSN, TSV. Since nobody is REQUESTing, SV2[2] = H.

- Assume S3 makes a REQUEST now. It will be sent to both S1 and S2. Only S2 responds since only SV2[2] is H (SV1[1] is N now).
Raymond’s Algorithm

- Sites are arranged in a logical directed tree. Root: token holder. Edges: directed towards root.
- Every site has a variable `holder` that points to an immediate neighbor node, on the directed path towards root. (Root’s holder point to itself).
- Requesting CS
  - If Si does not hold token and request CS, sends REQUEST *upwards* provided its `request_q` is empty. It then adds its request to `request_q`.
  - Non-empty `request_q` -> REQUEST message for top entry in q (if not done before).
  - Site on path to root receiving REQUEST -> propagate it up, if its `request_q` is empty. Add request to `request_q`.
  - Root on receiving REQUEST -> send token to the site that forwarded the message. Set `holder` to that forwarding site.
  - Any Si receiving token -> delete top entry from `request_q`, send token to that site, set `holder` to point to it. If `request_q` is non-empty now, send REQUEST message to the `holder` site.
Raymond’s Algorithm …

- Executing CS: getting token with the site at the top of request_q. Delete top of request_q, enter CS.
- Releasing CS
  - If request_q is non-empty, delete top entry from q, send token to that site, set holder to that site.
  - If request_q is non-empty now, send REQUEST message to the holder site.
- Performance
  - Average messages: $O(\log N)$ as average distance between 2 nodes in the tree is $O(\log N)$.
  - Synchronization delay: $(T \log N) / 2$, as average distance between 2 sites to successively execute CS is $(\log N) / 2$.
  - Greedy approach: Intermediate site getting the token may enter CS instead of forwarding it down. Affects fairness, may cause starvation.
Raymond's Algorithm: Example

Step 1:
- S1: Token holder
- S2: Token request
- S3
- S4
- S5
- S6
- S7

Step 2:
- S1
- S2
- S3
- S4
- S5
- S6
- S7

Token

Token request
Raymond’s Algm.: Example...

Step 3:

```
S1
  └── S2
      ├── S4
      └── S5

  ┌── S3
      ├── S6
      └── S7
```

Token holder
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Non-Token</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Resp. Time(ll)</strong></td>
<td>2T+E</td>
<td>2T+E</td>
</tr>
<tr>
<td><strong>Sync. Delay</strong></td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td><strong>Messages(ll)</strong></td>
<td>3(N-1)</td>
<td>N</td>
</tr>
<tr>
<td><strong>Messages(hl)</strong></td>
<td>3(N-1)</td>
<td>N</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamport</td>
<td>2T+E</td>
<td>2T+E</td>
</tr>
<tr>
<td>Ricart-Agrawala</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Maekawa</td>
<td>2T+E</td>
<td>2T</td>
</tr>
<tr>
<td>Suzuki-Kasami</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Singhal</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>Raymond</td>
<td>T(log N)+E</td>
<td>Tlog(N)/2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Messages(hl)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lamport</td>
<td>3(N-1)</td>
</tr>
<tr>
<td>Ricart-Agrawala</td>
<td>2(N-1)</td>
</tr>
<tr>
<td>Maekawa</td>
<td>5*sq.rt(N)</td>
</tr>
<tr>
<td>Suzuki-Kasami</td>
<td>N</td>
</tr>
<tr>
<td>Singhal</td>
<td>N</td>
</tr>
<tr>
<td>Raymond</td>
<td>4</td>
</tr>
</tbody>
</table>