Lamport's Algorithm

Lamport was the first to give a distributed mutual exclusion algorithm as an illustration of his clock synchronization scheme. Let $\mathbf{R_i}$ be the ***request set*** of site S_i, i.e. the set of sites from which S_i needs permission when it wants to enter CS. In Lamport's algorithm, $\forall i : 1 \leq i \leq N :: \mathbf{R_i} = \{S_1, S_2, ..., S_N\}$. Every site S_i keeps a queue, $request\_queue_i$, which contains mutual exclusion requests ordered by their timestamps. This algorithm requires messages to be delivered in the FIFO order between every pair of sites.

The Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends REQUEST(ts_i, i) message to all the sites in its request set $\mathbf{R_i}$ and places the request on $request\_queue_i$ (ts_i is the timestamp of the request).
2. When a site S_j receives the REQUEST(ts_i, i) message from site S_i, it returns a timestamped REPLY message to S_i and places site $S'_i s$ request on $request\_queue_j$.

Executing the critical section.

1. Site S_i enters the CS when the two following conditions hold**:**

   a) [L1:] S_i has received a message with timestamp larger than (ts_i, i) from all other sites.
   b) [L2:] $S'_i s$ request is at the top $request\_queue_i$.

Releasing the critical section.

1. Site S_i, upon exiting the CS, removes its request from the top of its request queue and sends a timestamped RELEASE message to all the sites in its request set.
2. When a site S_j receives a RELEASE message from site S_i, it removes S'_is request from its request queue.

When a site removes a request from its request queue, its own request may come at the top of the queue, enabling it to enter CS. The algorithm executes CS requests in the increasing order of timestamps.

The Ricart-Agrawala Algorithm

The Ricart-Agrawala algorithm is an optimization of Lamport's algorithm that dispenses with RELEASE messages by cleverly merging them with REPLY messages. In this algorithm also, $\forall i : 1 \leq i \leq N : \mathbf{R_i} = \{S_1, S_2, ..., S_N\}$.

The Algorithm

Requesting the critical section.

1. When a site S_i wants to enter the CS, it sends a timestamped REQUEST message to all the sites in its request set.
2. When site S_j receives a REQUEST message from site S_i, it sends a REPLY message to site S_i if site S_j is neither requesting nor executing the CS or if site S_j is requesting and S'_is request's timestamp is smaller than S_j's own request's timestamp. The request is deferred otherwise.

Executing the critical section

1. Site S_i enters the CS after it has received REPLY messages from all the sites in its request set.

Releasing the critical section

1. When site S_i exits the CS, it sends REPLY messages to all the deferred requests.

A site's REPLY messages are blocked only by sites that are requesting the CS with higher priority (i.e., a smaller timestamp). Thus, when a site sends out REPLY messages to all the deferred requests, the site with the next highest priority request receives the last needed REPLY message and enters the CS. The execution of CS requests in this algorithm is always in the order of their timestamps.

Maekawa's Algorithm

The Construction of request sets. The request sets for sites in Maekawa's algorithm are constructed to satisfy the following conditions:

M1: $(\forall i\ \forall j : i \neq j, 1 \leq i, j \leq N :: \mathbf{R_i} \cap \mathbf{R_j} \neq \Phi)$.

M2: $((\forall i : 1 \leq i \leq N :: S_i \in \mathbf{R_i})$

M3: $(((\forall i : 1 \leq i \leq N :: |\mathbf{R_i}| = K)$

M4: Any site S_j is contained in K number of $\mathbf{R_i}$s, $1 \leq i, j \leq N$.

The Algorithm

Requesting the critical section.

1. A site S_i requests access to the CS by sending REQUEST(*i)* messages to all the sites in its request set $\mathbf{R_i}$.
2. When a site S_j receives the REQUEST(*i*) message, it sends a REPLY(*j)* message to S_i provided it hasn't sent a REPLY message to a site from the time it received the last RELEASE message. Otherwise, it queues up the REQUEST for later consideration.

Executing the critical section.

1. Site S_i accesses the CS only after receiving REPLY messages from all the sites in $\mathbf{R_i}$.

Releasing the critical section.

1. After the execution of the CS is over, site S_i sends RELEASE(*i*) message to all the sites in $\mathbf{R_i}$.
2. When a site S_j receives a RELEASE(*i)* message from site S_i, it sends a REPLY message to the next site waiting in the queue and deletes that entry from the queue. If the queue is empty, then site updates its state to reflect that the site has not sent out any REPLY message.

Suzuki-Kasami's broadcast algorithm
The Algorithm

Requesting the critical section

1. If the requesting site S_i does not have the token, then it increments its sequence number, $RN_i[i]$, and sends a REQUEST(*i, sn*) message to all other sites (*sn* is the updated value of $RN_i[i]$).
2. When a site S_j receives this message, it sets $RN_j[i]$ to max($RN_j[i]$, *sn*). If S_j has the idle token, then it sends the token to S_i if $RN_j[i]=LN[i]+1$.

Executing the critical section.

1. Site S_i executes the CS when it has received the token.

Releasing the critical section. Having finished the execution of the CS, site S_i takes the following actions:

1. It sets $LN[i]$ element of the token array equal to $RN_i[i]$.
2. For every site S_j whose ID is not in the token queue, it appends its ID to the token queue if $RN_i[j]=LN[j]+1$.
3. If token queue is nonempty after the above update, then it deletes the top site ID from the queue and sends the token to the site indicated by the ID.

Raymond's Tree-Based Algorithm

The Algorithm

Requesting the critical section.

1. When a site wants to enter the CS, it sends a REQUEST message to the node along the directed path to the root, provided it does not hold the token and its *request_q* is empty. It then adds its request to its *request_q*. (Note that a nonempty *request_q* at a site indicates that the site has sent a REQUEST message to the root node for the top entry in its *request_q*).
2. When a site recives a REQUEST message, it places the REQUEST in its *request_q* and sends a REQUEST message along the directed path to the root provided it has not sent out a REQUEST message on its outgoing edge (for a previously received REQUEST on its *request_q*).
3. When the root site receives a REQUEST message, it sends the token to the site from which it received the REQUEST message and sets its *holder* variable to point at that site.
4. When a site receives the token, it deletes the top entry from its *request_q,* sends the token to the site indicated in this entry, and sets its *holder* variable to point at that site. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by *holder* variable.

Executing the critical section.

1. A site enters the CS when it receives the token and its own entry is at the top of its *request_q*. In this case, the site deletes the top entry from its *request_q* and enters the CS.

Releasing the critical section. After a site has finished execution of the CS, it takes the following actions:

1. If its *request_q* is nonempty, then it deletes the top entry from its *request_q*, sends the token to that site, and sets its *holder* variable to point at that site.
2. If the *request_q* is nonempty at this point, then the site sends a REQUEST message to the site which is pointed at by the *holder* variable.