Appendix A

Helary et al's Algorithm

In this algorithm the processors, the request messages and the token messages are abstracted as three different type of objects. Each of them is associated with a set of information structures to implement the algorithm. Each processor $P_i$ ($i = 1..N$) maintains the following data structures to record necessary information, here $N$ is the total number of processors.

1. logic clock: long count_i ; /* This variable maintains the logical clock of the processor */

2. Token_here: boolean; /* This variable is true iff $P_i$ has the token */

3. Neigh[1..N]: It is a set of process ids, initialised with the name of $P_i$'s neighbours.

4. Req_array : This structure maintains the list of pending requests. The structure is as follows :

   ```
   Req_array {
   int neigh; /* neighbour id which sent the request */
   int s; /* state of the site as known by $P_i$ */
   long reqtime; /* time when request was created */
   }
   ```

   reqs[j].s = R implies $P_i$ has a pending request of site $j$ which was sent at reqs[j].reqtime and $P_i$ received it from its neighbour which is stored in reqs[j].neigh. If reqs[j].s = NR (i.e. non requesting), it implies that $P_i$ does not have a pending request of $P_j$.

Two kinds of messages are sent between processes in the algorithm:

1. Request message : These messages are created and sent by processors which do not own the token and want to access the CS. They are further propagated by the other
processors. A request message carries the following information:

\[
\text{ReqMsg} = \{
\begin{align*}
\text{int reqorigin; /* requestor's id */} \\
\text{long reqtime; /* logical clock value of requestor at the request creation time */} \\
\text{int sender; /* sender of this message */} \\
\text{int alreadyseen}[1..N]; /* set of process ids which have received or are about to receive the request */
\end{align*}
\}
\]

2. **Token**: There exists at most one such message in the system. A process can access the CS, only if it has the token. The token message carries the following information:

\[
\text{TokenMsg} = \{
\begin{align*}
\text{int elec; /* It is the process id of the token's final addressee */} \\
\text{long lud}[1..N]; /* The ith subcomponent of this array stores the value that P_i's logical clock had when P_j gave the token to another process */
\end{align*}
\}
\]

**Initialisation**: Initially the system selects one of the processors \(P_h\) as the token owner and initialises the information structures as follows:

a) **Token initialisation**:

\[
\begin{align*}
\text{TokenMsg.elec} &= 0; \\
\text{for (i}=0, i<N, i++) &
\text{TokenMsg.lud}[i] = -1;
\end{align*}
\]

b) **Processor initialisation**:

\[
\begin{align*}
\text{count} &= 0; \\
\text{Token here} &= \text{false}; \\
\text{for (i}=0, i<N, i++) &
\text{reqs[i].neigh} = -1; \\
\text{reqs[i].s} &= \text{NR}; \\
\text{reqs[i].reqtime} &= -1;
\end{align*}
\]

if processor = \(P_h\) then Token here = true;

/* **Procedure make_request** */

/* Processor i (P_i) produces a request for accessing the CS */
reqs[i].s = R;

if(token_here) {
    enter_cs();
    reqs[i].s = NR;
    if(any pending request) {
        TokenMsg.lud[i] = count_i;
        TokenMsg.elec = Processor id of highest priority request, 
        next_destn = TokenMsg.elec;
        count_i ++;
        token_here = false;
        send token to reqs[next_destn].neigh;
        reqs[next_destn].s = NR;
    }
} else {
    send ReqMsg(i, count_i, i, {neighbours of i} \ U \{i\}) to \{neighbours of i\};
}

/* Processor P_i receives a ReqMsg( req_origin, reqtime, sender, already_seen) */

if(reqtime > reqs[req_origin].reqtime ) {
    reqs[req_origin].s = R;
    reqs[req_origin].neigh = sender;
    reqs[req_origin].reqtime = reqtime;
    count_i = max(reqtime, count_i) + 1;
    send ReqMsg(req_origin, reqtime, i, \{already_seen\} \U \{neighbours of i\}); 
    to \{neighbours of i\} - \{already_seen\};
    if(token_here) {
        token_here = false;
        TokenMsg.elec = req_origin;
        TokenMsg.lud[i] = count_i;
        count_i ++;
        send token to reqs[req_origin].neigh;
        reqs[req_origin].s = NR;
    }
}

/* Processor P_i receives a TokenMsg(lud, elec) */

token_here = true;
if(TokenMsg.elec == i) {
    enter_CS;
    reqs[i].s = NR;
    if(any pending request) {


TokenMsg.lud[i] = count_i;
TokenMsg.elec = Processor id of highest priority request;
next_destn = TokenMsg.elec;
count_i++;
token_here = false;
send token to reqs[next_destn].neigh;
reqs[next_destn].s = NR;
}
else {
    token_here = false;
    send token to reqs[TokenMsg.elec].neigh;
}
Appendix B

Yan et al's Algorithm

In this algorithm, the processors, request messages and token messages are abstracted as different types of objects. Each of them is associated with a set of information structures to assist the token chasing process in the system. Each processor maintains the following information:

1. Token information: $R_{token\_i}$ maintains the latest known location of the token.

   $R_{token\_i}$
   
   ```
   int path[N], /* path to the latest known token location */
   long age, /* The latest known token age */
   ```

2. Processor's state information: The following structure maintains the latest known state and priorities of all the processors in the system.

   ```
   Structure Requests {
       State s,
       long pri,
   } Reqs\_i[N];
   ```

   ```
   enum State {N,R,H,E}
   ```

3. Logic clock: $Count\_i$;
   It is incremented by 1 when processor $P_i$ sends out a request, it may be set to a larger value by the received messages.

Token Message: The token message carries the following information:

   ```
   TokenMsg {
       long age, /* current age of the token */
       long max; /* the known largest logical clock value among processors */
       Requests Treqs[N]; /* The latest known states and priorities of all the processors */
       int path[N], /* path to the new token owner */
   }
   ```

Request Message: The request message carries the following information:
ReqMsg {
    int src; /* requester's id */
    long pri; /* priority of this request */
    long age; /* the age of the token it is chasing */
    long max; /* the largest logical clock value currently known in the system */
    int path[N]; /* chasing path to the token owner currently known */
    int hist_path[N]; /* intermediate processors for passing this message */
}

Initialisation : The information structure of the token message is initialised as follows:

age=0, max=0, path = null;
for(i=1; i<=N; i++) {
    Treqs[i].s = N, Treqs[i].pri = 0;
}  
Treqs[r].s = H, /* P_r is selected to be the initial token owner */

The information structure of each processor is initialised as follows:

Rtoken_i.path = one of the optimal paths from P_i to P_r.
Rtoken_i.age = 0, Count_i = 0;
for(j=1; j<=N; j++) {
    Reqs_i[j].pri = 0;
    if(j == r) Reqs_i[j].s = H;
    else Reqs_i[j].s = N;
}

/* Processor P_i produces a request for accessing the CS */
case (Reqs_i[i].s) {
    H : /* token is here */
        enter_cs();
        if(no processor of R state in Reqs_i)
            Reqs_i[i].s = H;
        else {
            reqs_i[i].s = N;
            Select the requester P_k with the smallest priority as the new token owner;
            Rtoken_i.path = an optimal path to P_k;
            Rtoken_i.age++;
            Send out TokenMsg(Rtoken_i.age, Count_i, Reqs_i, Rtoken_i.path);
        }
        break;
}
Req_i[i].s = R;
Count_i++;
Req_i[i].pri = Count_i;
src = i, pri = Req_i[i].pri;
age = Rtoken_i.age;
max = Count_i;
path = Rtoken_i.path;
hist_path = i;
Send out ReqMsg(src, pri, age, path, hist_path, max);
break;

/* P_i receives a request message ReqMsg(src, pri, age, path, hist_path, max) */

Count_i = max(Count_i, ReqMsg.max);
ReqMsg.max = max(Count_i, ReqMsg.max);
if(Req_i[src].pri >= ReqMsg.pri)
  Discard this request and return;
Req_i[src].s = R, Req_i[src].pri = ReqMsg.pri;
ReqMsg.hist_path = ReqMsg.hist_path U {i};
ReqMsg.path -= {i};
if(Rtoken_i.age > ReqMsg.age) {
  ReqMsg.path = Rtoken_i.path;
  ReqMsg.age = Rtoken_i.age;
}
if(Rtoken_i.age < ReqMsg.age) {
  Rtoken_i.path = ReqMsg.path;
  Rtoken_i.age = ReqMsg.age;
}
case (Req_i[i].s) {
  H: 
    Rtoken_i.age++;
    Rtoken_i.path = an optimal path from P_i to src;
    Req_i[I].s = N;
    age = Rtoken_i.age; Treqs = Req_i;
    path = Rtoken_i.path;
    max = Count_i;
    send out the token to the requester;
    break;
  N: 
    if(ReqMsg.hist_path & ReqMsg.path have any site in common)
      break;
    else
      Send the received ReqMsg to the next processor along
      ReqMsg.path;
      break;
R:
if(Reqs_i[I].pri < ReqMsg.pri)
break;
else {
    if(ReqMsg.hist_path & ReqMsg.path have any site in common)
        break;
    else
        Send ReqMsg to the next processor along ReqMsg.path;
}

/* P, receives the token message TokenMsg(age, max, Treq, path) */

TokenMsg.path = {I};
Count_I = TokenMsg.max = max(Count_I, Token_Msg.max);
Rtoken_i.path = TokenMsg.path;
Rtoken_i.age = TokenMsg.age;
for(j=1; j<=N; j++) {
    if(TokenMsg.Treqs[j].pri < Reqs_I[j].pri)
        TokenMsg.Treqs[j] = Reqs_I[j];
    else if( TokenMsg.Treqs[j].pri > Reqs_I[j].pri)
        Reqs_I[j] = TokenMsg.Treqs[j];
}
if(TokenMsg.path = = Null) {
    enter_cs();
    if(no processor of R state in Reqs_i)
        Reqs_i[i].s = H,
    else {
        Reqs_i[j].s = N;
        Select the requester P_k with the smallest priority as the new token
        owner;
        Rtoken_i.path = an optimal path to P_k;
        Rtoken_i.age++;
        Send out TokenMsg(Rtoken_i.age, Count_i, Reqs_i,
        Rtoken_i.path);
    }
}
else
    send TokenMsg to the next processor along the path;