Chapter 3

Token based Delay Optimal Algorithms

3.1 Introduction

Existing Token based algorithms for arbitrary network topologies follow FCFS (First Come First Serve) serialization discipline to order the requests. Most of these algorithms assume the network to be fully connected. Therefore when they strictly follow the FCFS discipline, it poses no extra overheads on the algorithm. But consider a practical situation when the physical network is not fully connected. The token has to take an optimal path to reach a requesting site. On its path the token might pass over some sites which are in requesting state, but it would not service their request as it has to serialize access to CS in strictly first come first serve basis. Because of this reason the token might have to travel back and forth in the network in order to service the requests in strict FCFS fashion. As message propagation delay between two sites is much higher than the CS execution time, this back and forth movement of the token introduces considerable delay in servicing the requests. Based on this observation we propose a new technique, on-route servicing which uses a weak FCFS serialization discipline. The token services the requests of the sites that fall on the route of its destination. We have modified two existing token based algorithms [HPR88, YZY96] by inserting on-route servicing capability. Performance of the modified algorithms is evaluated and compared with the existing algorithms. The comparison reveals that this technique improves upon the average response time under
heavy load conditions to a considerable extent. The effect of varying the network connectivity is also studied on the performance of these algorithms.

We also present a new delay optimal token based algorithm using broadcast approach, for arbitrary topologies, this algorithm has two distinguished features:

- **On-route servicing**: The token serves the requests of the sites which fall on the route to its destination.

- **Token location propagators (tlp)**: The information of the token's current destination is disseminated in the system by special messages called token location propagators (tlp).

At very high loads, on-route servicing proves to be of great advantage as instead of moving from one site to other in order to serve the requests in strictly FCFS basis, the token services the requesting sites, if they fall on its path to its destination, this saves on message complexity as well as delay under heavy load conditions. Token location propagators are useful under light load situations. When a token leaves for a requesting site, information is spread in the network using tlp's so that when a site wants to access CS, it has the latest information about the token. Both these techniques together make the proposed algorithm, highly efficient.

We compare the performance of our algorithm with two existing algorithms, Helary et al [HPR88] and Yan et al [YZY96]. These are also token based and use broadcast approach to achieve mutual exclusion in arbitrary topologies.
3.2 System Preliminaries

The System Model, Performance Model and the Network Topologies used to evaluate the performance of the proposed algorithms is same as given in Chapter 2. The network size is taken to be of 16 nodes. A brief description of Helary et al’s algorithm and Yan et al’s algorithm is as follows. Section 3.4 gives the modified version of these algorithms and exhibits the performance gain achieved.

3.2.1 Helary et al’s Algorithm [HA]

In this algorithm, no assumption is made about the network topology, except connectivity. The only knowledge owned by a process is the name of its neighbours (i.e. sites connected to it by a direct link. The main principles of the algorithm are:

1. When a process wants to enter the CS it checks whether it owns the token or not. If it has the token it accesses the CS otherwise it sends a request message to its neighbours and waits for the token message. The request is propagated in the network with a flooding broadcast technique i.e. everytime a process receives a request from one of its neighbours it further propagates it to its other neighbours. The request message carries a control part with it which consists of a subset S of the processes, which have received the request message or are about to receive it.

2. On receiving the request message, a process (P) computes the subset T of its neighbours not belonging to S and sends the request message to them. The request is added to P’s set of known pending requests. If P owns the token, then it sends the token to requesting process via the neighbour from which it received the request.
3. Upon receiving the token message, process P keeps it if the token addressee is itself and accesses the CS otherwise P hands it over to N, the neighbour which sent him the request being serviced.

4. On exiting the CS, the process P extracts the oldest request R from its pending requests set. It then sends the token to the creator of R through N, the neighbour of P which sent him the request R.

The complete algorithm is given in Appendix A.

3.2.2 Yan et al’s Algorithm (YA)

This algorithm focuses on minimising the request delay by taking good use of dynamic state information and network topology information. It uses a unique token message to ensure the safety property and Lamport’s logical clock to provide fairness property. In this algorithm there are two active objects: request message and token message. Certain information structures are associated with these objects to speed up the dissemination of new state information and the latest location of the token. Each site only sends out one request message to chase the token. While chasing the token, a request message dynamically adjusts its chasing path based on the local information at intermediate nodes.

Each site can be in one of the following four states:

1. R: The requesting state for accessing the CS.
2. E: The executing state in the CS.
3. H: The holding state for the idle token when no requests have been sensed.
4. N: Non Requesting state, when the site is not holding the token and has no pending request for CS.
Van et al's algorithm consists of the following four phases:

1. Processor $P_i$ produces a request for accessing the CS: If $P_i$ owns the token, it immediately accesses the CS. If not, it sends a request message to the token owner (as per its local information).

2. Processor $P_i$ receives a request message: Processor $P_i$ compares the state information between the request message and the local information of $P_i$ and aligns them to the latest information. If $P_i$ is in state H or E, it implies that the token is at $P_i$, therefore the request ends its chasing process. If $P_i$ is in the state N or R, before passing the request to the next processor, $P_i$ checks whether the request will revisit any processor. If so, it means that the request has been or will be known by the token, hence request ends its chasing at $P_i$. Moreover when $P_i$ is in state R, it stops the chasing process of the received request if it has a lower priority value than $P_i$’s request message. Because by fairness, the token should be transferred to $P_i$ earlier than to the received request.

3. Processor $P_i$ receives the token message: $P_i$ compares the state information between the token message and the local information of $P_i$ and aligns them to the latest state information. If $P_i$ is the destination of the token, it accesses the CS, otherwise $P_i$ transfers the token message to the next processor as defined by the token’s path.

4. Processor $P_i$ exits from the CS: If $P_i$ has some pending requests, it sends the token to the processor whose request has the smallest priority value, Otherwise $P_i$ enters state H.

The complete algorithm is given in Appendix B.
3.3 A New Delay Optimal Algorithm

3.3.1 Basic Idea

The new mutual exclusion algorithm proposed here is a token based, and supports all the basic assumptions about token based mutual exclusion algorithms. There is only one token message in the system at any given moment of time. The token is passed from node to node, and only the token holder is permitted to enter its CS. Uniqueness of the token in the system guarantees mutual exclusive access to the CS. Each site maintains a copy of an adjacency matrix describing the network connections of the system and the routing is statically determined. When site \( i \) sends a message to site \( j \), the message is routed along a precalculated path. Lamport’s [Lam78] logical clock are used to provide the fairness property. Each site can be in one of the four states at a given point of time:

1. **R**: The site is requesting the CS.
2. **E**: The site is executing the CS.
3. **NR**: The site is not requesting the CS.
4. **H**: The site is holding the idle token.

Each site maintains the information of the current state of all the sites of the system and the information about the token location. This information is constantly updated by the messages exchanged between the sites. The following messages are exchanged between the sites:

1. **Request Message**: This message is generated when a site \( i \) wants to access the CS and it does not own the token. The request message is sent to the latest known token
location as per the information held by site i. The request message may be updated about the current location of the token by the intermediate sites through which it is routed and is directed towards the current location of the token.

2. **Token message**: This message gives a site a unique privilege to access the CS. To ensure faster information dissemination, the token message carries the information of the latest known states and priorities of all the sites and updates the nodes which fall on its path.

3. **Token location propagators (tlps)**: This message is used for faster dissemination of token's latest location. When the token leaves from a site i for its new destination j, these tlps are sent to inform other sites about the token's destination. Hence the information of token's latest location is propagated in the network in parallel with its transmission from site i to site j.

The main features of the algorithm are:

1. **On-route servicing**: Existing token based mutual exclusion algorithms for arbitrary network topologies follow FCFS serialization discipline to order the requests. It implies that the token on its way to the requesting site might pass over some sites which are in requesting state and it would update its information structure about the current state of these sites but it does not service them. The sites are served in strictly FCFS mode. This induces a larger delay in the system as the token has to travel back and forth in the network in order to service the requests in FCFS basis. We follow a weak FCFS discipline, in which the token's destination is chosen using FCFS discipline, but it also services the requests of the sites which fall on its route to the destination site. This
technique improves upon the average response time specially under heavy load conditions because at high loads token would find more requesting sites on its path.

2. **Information dissemination using tlp's**: In token based algorithms the focus is on minimizing the request delay i.e. faster a request is able to register itself with the token, the better is the algorithm. For this the requesting site should have the latest information of the location of the token. In our algorithm we use special messages called token location propagators (tlps) to disseminate the information about the latest location of the token in the network. Suppose site i sends the token to site j which is h hops away from it. At the same time site i sends tlp's with the information that the new token location is site j. These tlp's are sent to all sites which are (h-1) hops away from site i and whose optimal path to site i does not intersect with the path of the token. The site on-route the tlp's destination are also updated about the token's latest location. Tlp carries the information about the destination of token, age of the token, list of sites to which tlp has to be sent (tlp_set) and the maximum number of hops the tlp has to cover. When a site x receives a tlp, it updates its information about the token. hops is decremented. If hops becomes equal to zero (i.e. site x is the destination of the tlp) it does not further generate the tlp's. Otherwise (site x is the intermediate node) site x generates tlp messages and sends it to the sites specified by the tlp_set. To minimize the number of tlp messages, site x sends tlp's only to those sites of tlp_set which are hops distance away from x. Moreover only one tlp is sent through a particular link from site x i.e. if tlp is to be sent to two or more sites whose optimal paths to site x intersect with each other, then only one tlp is sent for all of them. Separate tlp's would be generated by the intersecting site and
sent further. This way the information about the token's destination is propagated in the network in parallel to the token's travel to the destination and hence requests would be able to register themselves faster with the token, as the token's latest information is spread over a large area of the network.

For example. In fig 3.1 site 1 sends the token to site 5 which is four hops away from it. Tips are sent to all sites which are 3 hops away i.e. site 9. Now sites 0,9,10 along with site 1,2,3,4 have the latest information about the token. Now if site 9 wants to access the CS, it will send the request directly to site 5 which is 4 hops away, instead of sending request to site 1 first and then getting the information that site 5 has the token. Similarly in fig 3.2 if site 1 sends the token to site 6, it follows the path [1,2,3,6] which has 3 hops. The tips are sent to all nodes, which are 2 hops away from site 1, i.e. 3,5,7,8,10. Optimal paths of 3&5 intersect with the token's path, therefore tips are only sent to 7,8, and 10. Tip reaches site 7 via site 4, site 8 via site 9, and site 10 via site 0. Therefore by the time token reaches site 6, sites 0,1,2,3,4,6,7,8,9,10 receive the information about the latest token location in parallel. Now if any of these sites want to send a request message, they will directly send it to site 6. This way the request would catch the token very fast and hence response time is reduced. At lighter loads, the information dissemination through request and token messages is very low, so the tips play an important role in propagating the latest token location.
When a site $i$ wants to access the CS, it checks if it has the token, if so it accesses the CS otherwise it sends request message to latest known location of the token as per its own knowledge and waits for the token. When site $j$ receives a request message it updates its state information about site $i$. The information about the current location of token is updated and the ongoing path of the request message is decided. If site $j$ is holding the token, it sends the token to site $i$ and in parallel also sends tips which propagate the information about latest token location in the network. If site $j$ is in requesting state when it receives the request message from site $i$, then the request message is forwarded further only if priority of requester has a lower value then site $j$, otherwise by fairness, the token should be transferred to site $j$ earlier then to the received request. When the token would come to site $j$ it would register site $i$’s request also. If site $j$ is in non-requesting state then the request message is forwarded further on its path to the latest known token location.
When site i receives the token, it updates its own state and that of the token by the newest state information. If site i is the destination of the token or if it is in requesting state, it accesses the CS, Otherwise the token message is transferred to the next processor in the path to the destination. When site i exits from the CS, It searches for a request with the smallest priority and sends the token to it. If no request is found then site i enters holding state (H). When a requesting site i is serviced on-route then after servicing the request, the token updates its state information and moves to the next site on path to the destination.

### 3.3.2 Data Structures

Each site i maintains the following information:

1. **Token Information**: At each site i the record of the latest known token location is kept.

   ```
   Struct token_info {
       int loc; /* latest known token location */
       int age; /* latest known token age */
   }
   ```

   token_info.age decides whether the recorded token location is latest or out of date.

2. **System state information**: Each site maintains the record of the states and priority values of all sites.

   ```
   Struct {
       int s; /* latest known state */
   }
   ```
int pri; /* latest known priority */

} site_info[N];

3. **Logical clock**: A logical clock is maintained by each site which is incremented by 1 when the site sends a request message for itself. It may be set to a larger clock value by the received messages. This clock value is used to achieve fairness in the system. When a request message is generated by site i, the priority of the request is this logical value at site i.

    Long l_clock;

4. **On-route service flag**: On exiting the CS, this variable tells the site whether it was serviced on-route or not. Further action is taken accordingly.

    Boolean on_route;

5. **Routing information**: At site i, link[j] gives the next site id which falls on path from site i to j.

    int link [N];

The following messages are used to exchange information among the sites in the network:

1. **Request Message**: The format of the request message is

    Reqst_msg(id,pri,tok_loc,tok_age,max_clock)

These are created and sent by any site which does not own the token and wants to access the CS. The parameters have the following meaning.
• \textit{id} : This gives the request creator site name.

• \textit{pri} : This is the logical clock value of the requestor at the request creation time. The requests are serviced by the token in the priority order defined by this value.

• \textit{tok\_loc} : This gives the latest known token location. This is the destination of the request message and may be updated by the intermediate nodes.

• \textit{tok\_age} : This gives the age of the token which this request message is searching. It may be updated by the intermediate nodes.

• \textit{max\_clock} : This is the largest logical clock value currently known in the system and is constantly updated at each intermediate node.

2. \textbf{Token message} : The format of the token message is

\[ \text{token(age, state\_info, destn, t\_clock)} \]

Only one such message exists in the system at a given point of time. The site which receives this message gets the privilege to access the CS. The parameters have the following meaning:

• \textit{age} : It gives the current age of the token and is used to distinguish old token information from new token information. It is incremented by one whenever token changes its ownership.

• \textit{state\_info} : This maintains the state information of all sites in the system and is used to disseminate this information along the token’s path.

• \textit{destn} : This is the token’s current destination. The token message on its path informs the sites about its current destination.
- $t_{clock}$: This is the known largest logical clock value among the sites and is constantly updated at each site.

3. **Token location propagators**: These messages are used to disseminate the information about the token's latest location. The format of token location propagator is:

\[ \text{tlp}(tok\_loc, tok\_age, hops) \]

The parameters have the following meaning:

- $tok\_loc$: This is the token's destination i.e. would be the location of the token.
- $tok\_age$: This is the current age of the token whose destination is $tok\_loc$.
- $hops$: This gives the maximum number of hops the tlp's have to go. $hops$ are decremented by one at each intermediate node and further propagation of tlp's is stopped if $hops$ becomes equal to zero.

### 3.3.3 Initialization

Initially, the system selects one of the sites, say site $r$ as the token owner and the information structure of all the sites is initialized as follows:

```c
l\_clock = 0;
for(i=0; i<N;i++) {
    site\_info[i].s = NR; /* all sites are non-requesting initially */
    site\_info[i].pri = 0;
    if(i==r) site\_info[i].s = H; /* site r holds the token */
}
token\_info.loc = r; /* the token is located at site i */
token\_info.age = 0; /* Initially the token age is set to 0 */
```
3.3.4 Description of the Algorithm

Each site $i$ [ $i = 1-N$ ] in the network is driven by the following events:

1. **Site i produces a request for accessing the CS**

   When a site $i$ produces a request to access the CS, it checks if it has the token, if so it accesses the CS otherwise sets its own state as requesting, increments the logical clock and sends the request message with priority value equal to its current logical clock to the current token owner as per its knowledge. The detailed description of the procedure follows:

   /* This procedure is executed by a site i when it wants to access the CS */

   **Make_reqst ( )**
   
   ```c
   if( site_info[i].s == H) {
       site_info[i].s = E; /* site i goes in executing state */
       enter_cs ( ); /* site i accesses the CS */
       on_route = false;
       exit_cs( on_route ); /* site i exits from CS */
   }
   else {
       site_info[i].s = R; /* site i is in requesting state */
       l_clock++; /* logical clock is incremented */
       site_info[i].pri = l_clock; /* pri of the request is the current logical clock value */
       send Reqst_msg(i,site_info[i].pri,token_info.loc,token_info.age,l_clock)
       to link[token_info.loc], /* sends the request to the next node on path to token location */
   }
   }

2. **Site i receives a request message Reqst_msg(i,pri,tok_loc,tok_age,max_clock)**
When site i receives a request message from site j, then the logical clock of site i and \textit{max\_clock} variable of the request message are updated to the maximum value of the two. The received request message is discarded if:

- The \textit{pri} of the request message is lower than the priority value of site j as recorded with site i. This implies that the received request is out of date.
- If the \textit{pri} of the request message is equal to the priority value of site j as recorded with site i and the state of site j at site i is non-requesting. This implies that site j’s request has already been serviced.

Site i updates its state information about site j. The information about the latest known location of token is updated both at site i and request message structure. Depending upon the site i’s current state, the following actions are taken:

1. Holding: If site i holds the token, it sends it to site j. After sending the token, site i generates the token location propagators and sends them in the network to disseminate the information about the token’s current destination.

2. Non-Requesting: If site i is in NR state then the request message is forwarded to the next site on the path to the latest known token location.

3. Requesting: If site i is in requesting state then the request message is forwarded further only if the priority value of site i’s request is higher than that of the received request. By fairness, if the priority of site i’s request has a lower value than the received request, the token should be transferred to site i earlier than to the received request.

/* This procedure is executed by site i when it receives a \texttt{Reqst\_msg(j,pri,tok\_loc,tok\_age,max\_clock)} */
recv_rqst(j, pri, tok_loc, tok_age, max_clock) {
    if (1_clock > max_clock) /* set the clock to the largest logical clock value */
        max_clock = 1_clock;
    else
        1_clock = max_clock;
    if ((site_info[j].pri < pri) || ((site_info[j].pri) == pri) && (site_info[j].s == R))) {
        /* Received request is a new request */
        site_info[j].s = R; /* update site i's state information about site j */
        site_info[j].pri = pri;
        if (token_info.age > tok_age) { /* updates the request message about the latest known token location */
            tok_loc = token_info.loc;
            tok_age = token_info.age;
        }
    } else if (token_info.age < tok_age) { /* update site i about the latest known token location */
        token_info.loc = tok_loc;
        token_info.age = tok_age;
    }
    switch(site_info[i].s) {
    case H: /* if site i holds the token, it sends it to site j */
        token_info.age++ /* increment the age of token */
        token_info.loc = j /* update site i about token's destination */
        site_info[i].s = NR;
        send token (token_info.age, site_info, token_info.loc, 1_clock) to link[j];
        send_tlp(); /* send the token location propagators */
        break;
    case NR:
        send Reqst_msg(j, pri, tok_loc, tok_age, max_clock) to link[tok_loc];
        break;
    case R:
        if (site_info[i].pri >= pri) /* send the request message further only if pri of site i's request is greater than site j's request */
            sendReqst_msg(j, pri, tok_loc, tok_age, max_clock) to link[tok_loc];
        break;
    }
}
4. Site i receives a token message token (age,state_info,destn,t_clock)

When site i receives the token message, the state information held by site i and the token message are updated to the latest known values. Note that when pri value of a request is same with site i and token, then site i's information is updated. This is because token may have serviced the request on-route, so token has the latest information. Site i accesses the CS if the destination of the token is site i or if site i is in requesting state, otherwise the token is forwarded to next site on the path to the destination. If the site i was not the destination of the token but was in requesting state, the on_route service flag is set. When site i exits from the CS, the action is taken depending upon the value of the on_route service flag. The detailed description of the protocol is as follows:

/* This procedure is executed by site i when it receives the token(age,state_info,destn,t_clock) */

Recv_token (age,state_info,destn,t_clock)
{
    if((l_clock > t_clock) /* the logical clock of site i is set to the largest known logical clock value */
        t_clock = l_clock;
    else
        l_clock = t_clock;
    token_info.loc = destn; /* site i is updated about the token's current destination */
    token_info.age = age;
    for(j=0; j<N; j++)
    
    if(state_info[j].pri < site_info[j].pri) {
        state_info[j].s = site_info[j].s;
        state_info[j].pri = site_info[j].pri;
    }
    else {
        site_info[j].s = state_info[j].s;
        site_info[j].pri = state_info[j].pri;
    }
}
if ( destn == i) { /* If the destination of the token is i, then site i accesses the CS */
    site_info[i].s = E;
    enter_cs();
    on_route = false;
    exit_cs(on_route);
}
else if (site_info[i].s == R) { /* site i is in requesting state */
    site_info[j].s = E;
    enter_cs();
    on_route = true;
    exit_cs(on_route);
}
else /* token is sent to the next site on the path to destination */
    send token (age,state_info,destn,t_clock) to link[destn];

5. Site i receives a token location propagator tlp(tok_loc,tok_age,tlp_set,hops)

When a site i receives a tlp, it compares the age of the token propagated by the tlp with its own information about the token. The tlp is discarded if the age of the token at site i has higher value than the age of the token propagated by tlp. Otherwise site i is updated about the token’s current location. The number of hops is decremented. If hops become equal to zero (i.e. site i is the destination of the tlp), site i does not send the tlps further. Otherwise (site i is the intermediate node) site i generates tlp messages and sends them to the sites as given in tlp_set. The detailed procedure is as follow:

/* This procedure is executed when site i receives a token location propagator
   tlp(tok_loc,tok_age,tlp_set,hops) */

Recv_tlp(tok_loc,tok_age,tlp_set,hops)
{
    if(token_info.age < tok_age) {
        token_info.age = tok_age;
        enter_cs();
        on_route = true;
        exit_cs(on_route);
    }
}

else /* token is sent to the next site on the path to destination */
    send token (age,state_info,destn,t_clock) to link[destn];

5. Site i receives a token location propagator tlp(tok_loc,tok_age,tlp_set,hops)

When a site i receives a tlp, it compares the age of the token propagated by the tlp with its own information about the token. The tlp is discarded if the age of the token at site i has higher value than the age of the token propagated by tlp. Otherwise site i is updated about the token’s current location. The number of hops is decremented. If hops become equal to zero (i.e. site i is the destination of the tlp), site i does not send the tlps further. Otherwise (site i is the intermediate node) site i generates tlp messages and sends them to the sites as given in tlp_set. The detailed procedure is as follow:

/* This procedure is executed when site i receives a token location propagator
   tlp(tok_loc,tok_age,tlp_set,hops) */

Recv_tlp(tok_loc,tok_age,tlp_set,hops)
{
    if(token_info.age < tok_age) {
        token_info.age = tok_age;
        enter_cs();
        on_route = true;
        exit_cs(on_route);
    }
}

else /* token is sent to the next site on the path to destination */
    send token (age,state_info,destn,t_clock) to link[destn];
token_info_loc = tok_loc;
hops--;
if( hops != 0) { /* site i is the intermediate node */
    for(j=0;j<N;j++) {
        if((tlp_set[j] == 1) && (dist(i,j,hops)) /* distance from i

        else
            tlp_set[j] = 0;
    }

    for(j=0;j<N;j++)
        link_invok[j]=0;

    for(j=0;j<N;j++)
        if((tlp_set[j] == 1) && (link_invok[link[j]] != 1)) {
            send tlp(tok_loc,tok_age,tlp_set,hops) to link[j],
            tlp_set[j] = 0;
            link_invok[link[j]] = 1;
        }
}
}

5. Site i exits from the CS

When a site exits the CS, it receives all the request messages that have arrived while it
was accessing the CS. The information structure of site i and the token are updated by
the information carried by the request messages. If site was serviced on-route by the
token, its state is set to NR both in the token and at the site itself. The token is sent to the
next site on the path to its destination. If site i was the destination of the token, then on
exiting the CS, site i searches for the smallest Pri value request in the site_info
information structure. If a pending request is found, the token is sent to it, otherwise the
site i enters the holding state. After sending the token to the smallest Pri value requester,
token location propagators are sent to inform the sites about the token's current destination. The detailed procedure is as follows:

/* This procedure is executed when a site i exits from the CS */

exit_cs (on_route) {
    /* receive the waiting request messages and update the information structure of the token and site i itself */

    while (!end) {
        receive a Reqst_msg (j,pri, tok_loc,tok_age,max_clock);
        if( l_clock < max_clock)
            l_clock = max_clock;
        if( site_info[j].pri < pri) {
            site_info[j].s = R;
            site_info[j].pri = pri;
            state_info[j].s = R;
            state_info[j].pri = pri;
        }
        if(l_clock > t_clock)
            t_clock = l_clock;
        if(on_route) {
            site_info[i].s = NR;
            state_info[i].s = NR;
            send token(age,state_info,destn,t_clock) to link[destn];
        } else {
            if(next_destn = high_pri(i) != -1) { /* high_pri(i) returns the highest priority pending request at site i */
                site_info[i].s = NR;
                token_info.loc = next_destn;
                token_info.age++;
                send token(token_info.age, site_info, token_info.loc,t_clock) to link[token_info.loc];
                send_tlp();
            } else
                site_info[i].s = H;
        }
    }
}
/* This function generates and sends the token location propagators */

send_tlp (loc, age)
{
    hops = No. of hops token will travel to reach its destination;
    hops--; /* send tlp's to sites which are (hops-1) distance away from site i */
    if(hops == 0)
        return;
    for(j=0; j<N; j++) {
        if (no. of hops from i to j is equal to hops and the path from i to j does not
            intersect with the path of the token from i to token's destination)
            tlp_set[j] = 1;
        else
            tlp_set[j] = 0;
    } 
    for(j=0; j<N; j++) {
        if((tlp_set[j] == 1) && (link_invok[link[j]] != 1)) {
            send tlp(loc, age, tlp_set, hops) to link[j];
            tlp_set[j] = 0;
            link_invok[link[j]] = 1;
        }
    }
}

3.3.5 Algorithm Correctness

1. Mutual exclusion

As there is a unique token in the system, and a site is allowed to access the CS if and only
if it holds the token, mutual exclusion is trivially guaranteed. At any point of time there
exists atmost one site which is in executing (E) or holding (H) state. By construction it
holds in the initial state. Later every sending of the token is possible only by a site i
which holds the token and sending of the token sets the state of site i as non-requesting
(NR). Only on receipt of token message, a site i can enter executing state or holding state.
Hence mutual exclusion is achieved.
2. Fairness

As the serialisation discipline is not strictly FCFS, it introduces an inherent unfairness in the system. The sites which have a higher degree of connectivity would fall in the path of token more often, and therefore the service rate of these requests would be higher. For example in the star topology of N+1 nodes, the central node’s request would be served at N times faster rate, because to service any request, the token will have to pass through the central node, and if it is in requesting state, service its request first and then move forward. We observe that sites with higher connectivity are higher responsibility sites, as their failure has a larger impact in the network, therefore if they get better access to CS, it becomes a fair scheme.

3. Liveness

Liveness means that any site requesting the CS, eventually gets the token and accesses the CS in finite time period. On-route servicing introduces a small delay in reaching the destination but this delay is finite as the token’s path is acyclic and finite. Once a request is registered with the token, it would be eventually serviced, this is ensured by the priority value attached with each request. Each site maintains a logical clock which is set to the largest known clock value by the incoming token and request messages. When a request is generated, by a site, the logical clock value is incremented by 1 and this value becomes the priority value of the request. Since the token services the requests in priority order and new requests in the system have higher priority as explained above, therefore once a request is registered with the token, it is eventually serviced. Now we prove that a request generated in the system is registered with the token. When a site i makes a
request to access the CS the following situations may occur:

1. If the site \( i \) is holding the token, then it immediately accesses the CS otherwise, it sends a request message to the token location known by site \( i \).

2. When site \( i \) receives a request message from site \( j \) then:
   - If site \( i \) owns the token, it sends it to site \( j \).
   - If site \( i \) is executing the CS, then after executing the CS, site \( i \) receives the request message and registers it with the token.
   - If site \( i \) is in R state and the request of site \( i \) has lower priority value than request from site \( j \), the request of site \( j \) is detained at site \( i \). When the token arrives at site \( i \), the request of site \( j \) gets registered with the token.
   - The request message moves further only if site \( i \) is in NR state or if in R state with higher priority value then site \( j \)'s request message.

As the request message moves from one site to another, it is updated about the latest known token location and finally it gets itself registered with the token. Moreover the request message on its path updates all sites about the requesting status of site \( j \). Therefore if the token reaches any of these sites, then it registers the request of site \( j \). Initially all sites know about the token location, so a request can directly register itself with the token and then whenever the token moves it updates the intermediate sites about its destination. Tips help to spread this information faster and wider in the network. Therefore as the network is a connected graph with no partitions, a request is always able to register itself with the token. Hence liveness is ensured.
3.3.6 Performance Analysis

The performance of the proposed algorithm is evaluated and compared with Yan et al algorithm and Helary et al algorithm. The performance evaluation has been done on 4 network topologies (Ring, Chordal Ring I, Chordal ring II and Barrel shifter) of increasing connectivity as discussed in chapter 2. The metrics that have been used to evaluate the performance are:

- **Time Delay**: It is the period of time between the instant a site invokes mutual exclusion and the instant when the site enters the CS. It is also referred to as the average response time per CS of the algorithm.

- **Message Traffic**: Average number of message hops incurred per CS execution.

3.3.6.1 Time delay

The average time delay per CS on varying the arrival rate of CS requests is shown in fig.3.3. All the three algorithms exhibit similar delay curve. The delay in granting the CS at higher loads is high. As the load is reduced, the delay in receiving the token also decreases and finally becomes stable. The delay is high at high loads, because at a given time token has a large number of pending requests. Therefore even if a request registers itself with the token, it is serviced only after the previous pending requests have been serviced. As the load decreases, pending requests also decreases and hence delay also decreases. Helary et al algorithm exhibits maximum delay at high loads because it does not make use of dynamic state information and network topology information. Yan et al makes use of the information about the network topology and token and request messages help in information dissemination in the network, therefore its delay performance is.
better than that of Helary et al's algorithm at high loads. Our algorithm exhibits the best performance at high loads. This is because at high loads more sites on the path of the token are in requesting state. As these requests are serviced by the token on its route to its destination, the delay in servicing these requests is very less. Hence average delay is reduced. As the load is decreased i.e. number of requests per unit time becomes less, the probability of finding request site on the token's path reduces and therefore at lower loads on-route requesting does not contribute much in reducing the delay. At light loads, the efficiency of the algorithm depends upon how quickly a request message finds the site holding the token. Helary et al's algorithm has the lowest delay at light loads. This is because request messages are sent to all neighbors and search for token in parallel. Our algorithm has lower delay as compared to Yan et al's algorithm. This is because in our algorithm, token location propagators spread the token location information wider and faster in the network. The average delay of all the three algorithms reduces as the connectivity increases. This is because as connectivity increases, network diameter decreases and hence request is able to reach token faster. Moreover when diameter is decreased, token also reaches the requesting site faster. As the connectivity is increased, the delay curve of all the three algorithms converges to same value.

### 3.3.6.2 Message traffic

The average number of message hops per CS on varying the arrival rate of CS requests is shown in fig. 3.4. We observe that Helary et al's algorithm has the maximum message traffic, whereas Yan et al's algorithm generates the least message traffic. Our algorithm has slightly higher message complexity than Yan et al's algorithm. At very high loads
the message complexity of our algorithm is similar to that of Yan et al's algorithm as more requests are served on-route and less tips are generated. As the load is decreased, number of messages per CS increase as a request has to search for the token at longer distances in Yan et al's algorithm. Tips contribute to the increased message traffic in our algorithm. Helary et al's algorithm has the highest message complexity because if a site i wants to access the CS, it sends request messages to all its neighbors, which further send these request messages to their neighbors. This way the entire network is flooded by the request messages, Helary et al's algorithm does not make use of the state information at intermediate sites. For example, in the ring network, a requesting site sends two requests to its adjacent sites, which search for the token along the ring simultaneously in clockwise and anti-clockwise direction. The two requests stops when they meet the token and then one of them sends the token to the requestor. So for the network of 16 sites, on an average 24 message hops are made for one CS entry. The experimental results in fig.3.4 confirms this. As the connectivity of the network increases, the message complexity of both, our algorithm and Yan et al's algorithm decreases. This is because as connectivity increases the diameter of the network decreases and the messages have to travel lesser number of hops. But the message complexity of Helary et al's algorithm increases with the increase in the connectivity of the network. This is because, as the connectivity of the network increases, the number of neighbors of each site increases and therefore more number of request messages are flooded in the network.
Fig. 3.3 (a) Avg. time delay on ring

Fig. 3.3 (b) Avg. time delay on chordal ring
Fig. 3.3 (c) Avg. time delay on chordal ring II

Fig. 3.3 (d) Avg. time delay on barrel shifter
Fig. 3.4 (a) Avg. message traffic on ring

Fig. 3.4 (b) Avg. message traffic on chordal ring 1
Fig. 3.4 (c) Avg. message traffic on chordal ring II

Fig. 3.4 (d) Avg. message traffic on barrel shifter
3.3.7 Conclusion

Therefore we conclude that at high loads our algorithm has the least delay, but a slightly higher message traffic than Yan et al's algorithm. At light loads Helary et al's algorithm has the least delay but the message traffic generated by Helary et al's algorithm is very high. The performance analysis proves that our algorithm achieves an optimal message complexity and the smallest delay. Moreover, the effect of varying the network connectivity is also studied and we find that the proposed algorithm has the best performance specially under heavy loads and on topologies with low connectivity.

3.4 Inserting On-route servicing in Token Based Algorithms

We propose a modification in the Helary et al's [HPR88] and Yan et al's [YZY96] token based algorithms by inserting on-route servicing capability in these algorithms. Further, we make a performance comparison of these algorithms with their modified versions on networks of varying connectivity.

3.4.1 Modified Helary et al's Algorithm (MHA)

In HA, each site follows strict FCFS discipline to service the requests. The process which receives a token message is defined as a token owner if the token is addressed to it otherwise it is a token handler. The token handler cannot use the token to enter the CS, it is involved only in transferring token to its final addressee. We propose that if the token finds a site in requesting state, it should first service its request and then move towards its destination. By doing so, the algorithm would save on the average response time to a
considerable extent as the message propagation delay is much higher than the CS execution time. Moreover in HA, even if a request message reaches a site which has the token, it is further broadcast which only adds to message complexity. In the modified algorithm we stop the further broadcast of the request message if it reaches a site which has the token. This site deletes the request message and sends the token to the requesting site.

The modification is made in the module where a processor receives a token message, and when it receives a request message:

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

```c

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 if (reqs[i].s == R) {
    enter_cs;
    reqs[i].s = NR;
    count_i ++;
    TokenMsg.lud[i] = count_i;
    token_here = false;
    send token to reqs[TokenMsg.elec].neigh;
}
else {
    token_here = false;
    send token to reqs[TokenMsg.elec].neigh;
}
```

```
/* Processor \( P_i \) receives a \( \text{ReqMsg}( \text{req\_origin}, \text{reqtime}, \text{sender}, \text{already\_seen}) \) */

\[ \text{if}(\text{reqtime} > \text{reqs}[\text{req\_origin}].\text{reqtime}) \{ \]
\[ \quad \text{reqs}[\text{req\_origin}].s = \text{R}; \]
\[ \quad \text{reqs}[\text{req\_origin}].\text{neigh} = \text{sender}; \]
\[ \quad \text{reqs}[\text{req\_origin}].\text{reqtime} = \text{reqtime}; \]
\[ \quad \text{count}_i = \max(\text{reqtime}, \text{count}_i) + 1; \]
\[ \quad \text{if}(\text{token\_here}) \{ \]
\[ \quad \quad \text{token\_here} = \text{false}; \]
\[ \quad \quad \text{TokenMsg.elec} = \text{req\_origin}; \]
\[ \quad \quad \text{TokenMsg.lud}[i] = \text{count}_i; \]
\[ \quad \quad \text{count}_i++; \]
\[ \quad \quad \text{send} \text{token} \text{to} \text{reqs}[\text{req\_origin}].\text{neigh}; \]
\[ \quad \quad \text{reqs}[\text{req\_origin}].s = \text{NR}; \]
\[ \quad \} \]
\[ \quad \text{else} \{ \]
\[ \quad \quad \text{send} \text{ReqMsg}(\text{req\_origin}, \text{reqtime}, i, \{\text{already\_seen}\} \cup \{\text{neighbours of } i \}) ; \]
\[ \quad \quad \text{to} \{\text{neighbours of } i \} - \{\text{already\_seen}\}; \]
\[ \} \]

### 3.4.2 Modified Yan et al's Algorithm (MYA)

In YA, when a process receives the token it compares the state information between the token and the local information of \( P_i \) and aligns them to the latest state information. If \( P_i \) is in R state, then this information is updated in the token's information structure but this request would be serviced depending upon its timestamp (i.e. priority) value. The modification we propose is that if the token finds a site in R state on its path to the final addressee, it services this request first and then moves towards its destination. In YA, a request message is discarded by a site \( P_i \) if it is of lower priority value then the local information held by \( P_i \) about the requesting site. In MYA algorithm, a process \( P_i \) discards
a received request message if:

1. It is out of date i.e. old request.
2. This request would be serviced by the token on its path to the new token owner.

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

In this module on receiving the request message P_i discards the request if the following condition is true:

\[ \text{if}((\text{Reqs}_i[\text{src}]\text{.pri} \geq \text{ReqMsg.pri}) \&\& ((\text{rtoken}_i\text{.path} \text{includes site } I) \&\& (\text{age} \leq \text{rtoken}_i\text{.age})) \]

Rest of the procedure is same as that of YAN's.

/* P_i 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[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);
    }
} else if(Reqs_I[I].s == R) {
    enter_cs();
}
Reqs_I[I].s = NR;
TokenMsg.Treqs[I].s = NR;
send TokenMsg to the next processor along the path;
}

else
    send TokenMsg to the next processor along the path;

3.4.3 Algorithm Correctness

1. **Mutual exclusion**: As there is unique token in the system and the algorithm allows a process to enter the CS iff it holds the token, therefore the mutual exclusion is trivially guaranteed in both the algorithms.

2. **Freedom from deadlock and starvation**: The algorithms are free from deadlock and starvation as proved in [HPR88, YZY96]. In MHA & MYA, the token moves towards its destination as defined in HA & YA algorithms. On its way it allows the requesting sites to access CS. This introduces a small delay in reaching the destination but this delay is finite as the token's path diameter of the network is also finite, therefore a finite delay might be introduced in reaching the destination. Hence the algorithm is starvation free. As the serialisation discipline is not strictly FCFS, it introduces an inherent unfairness in the system. The sites which have a higher degree of connectivity would fall in the path of token more often, and therefore the service rate of these requests would be higher. For example in the star topology of N+1 nodes, the central node's request would be served at N times faster rate, because to service any request, the token will have to pass through the central node, and if it is in
requesting state, service its request first and then move forward. We observe that sites with higher connectivity are higher responsibility sites, as their failure has a larger impact in the network, therefore if they get better access to CS, it becomes a fair scheme.

3.4.3 Performance Analysis

1. Time Delay: In fig. 3.5 we see that at higher loads both MHA and MYA show considerably low average response time than HA & YA respectively. The reason is that at very high loads, more sites on the path of the token would be in requesting state. As these requests would be serviced by the token on its route to the destination, the delay in servicing these requests would be very less and it would also save in token hops. As the load is decreased i.e. number of requests per unit time becomes less, the probability of finding requesting sites on the token's path would reduce and therefore the waiting time per invocation of the modified versions approaches that of the original algorithms as load decreases.

2. Message Traffic: Number of messages per invocation are also less in case of MHA as compared to HA as shown in fig 3.6, this is because in HA the request message is broadcast to all nodes, whereas in MHA, if the request reaches a site which holds the token, it sends the token to the requesting site and the request message is not further broadcast. This reduces the number of messages per invocation. Number of messages are more in MYA at high load as shown in fig 3.6. This is because at higher loads on route servicing of requests is more. In YA the
request stops chasing the token if it reaches a site which is in requesting state & has a lower priority than itself. But in MYA algorithm, the token would service all the requests that fall on its path. Therefore a request message may not find a site in requesting state and therefore it will have to chase the token for a longer period and hence more number of store & forwards of the request message would be there. As load decreases the number of messages per invocation in MYA algorithm becomes equal to the YA algorithm. At heavy load, in ring topology, the token continues to move in one direction servicing requests of all sites falling on its path. This is like the migrating token mutex algorithm. In this the token moves from one site to other in the ring, if the site has a pending request, service it and then move to the next site. For a ring topology this is considered to be the most efficient solution.
Fig. 3.5 (a) Avg. Time Delay on Ring

Fig. 3.5 (b) Avg. Time Delay on Chordal Ring I
Fig. 3.5 (c) Avg. Time Delay on Chordal Ring II

Fig. 3.5 (d) Avg. Time Delay on Barrel Shifter
Fig. 3.6 (a) Avg. Message Traffic on Ring

Fig. 3.6 (b) Avg. Message Traffic on Chordal Ring I
Fig. 3.6 (c) Avg. Message Traffic on Chordal Ring II

Fig. 3.6 (d) Avg. Message Traffic on Barrel Shifter