A note on token based control in rings and linear arrays*

P Thangavel# & R Rajendra Prasathδ

Department of Computer Science, University of Madras,
Chepauk, Chennai - 600 005, India

In this paper, we describe token based message routing strategy for the allocation of a single shared resource among *n* processing entities that are arranged in the variant of a ring network and a linear array. First we present an algorithm for single shared resource allocation in a bidirectional ring network in which token and requests can pass through in opposite directions. Then we propose a new algorithm for a linear array (a ring with a faulty node) in which requests are allowed to pass through the processors according to the direction of the token movement. These protocols are built on request based token control strategy in which token moves only after the initialization of a request. We ensure that the proposed algorithms prevent starvation of requests which are served within a maximum delay.

Keywords : Distributed computing, Shared resource allocation, Token passing, Bidirectional ring network.

1. Introduction

Consider the allocation problem of a single shared resource among a set of *n* processors that are arranged in the variants of a ring network. We provide the solution by means of a request based token passing strategy in which token is used to control the allocation of the shared resource. At first, the token is informed by the request sent by processors which are in need of the shared resource. Then the free token is converted into a busy token. After servicing the processors with a pending request, the busy token is destroyed and a new free token is again circulated in the network. This type of control mechanism is used in various communication problems such as election problem [9], mutual exclusion problem[10,12,13,17], etc. Data transmission [1,14,15] using ring networks and access to the transmission medium in bus networks [3,7] were solved by constructing ring networks.

These problems use the same procedure: an entity holding token will pass it along the ring as soon as it no longer needs it. In this case, the number of requests exchanged may be unbounded even for a finite number of requests. Feuerstein *et* al.[6] has overcome this problem by implementing request based token passing technique that allows the token to move only if it is informed of a processor asking the shared resource. They have described this strategy for an unidirectional fault free ring network in which token as well as requests are allowed to pass through in a single direction. Feuerstein *et* al.[6] have proposed two protocols Q and D using request message based token control strategy in which requests and token move in single direction only. Algorithm Q requires *n* messages per request and service traffic $3n^2/2$. Algorithm D requires $2n$ messages per request and service traffic $(3n - 3)$.

Although a vast amount of literature exists on the token based control strategy, but little is known about the use of circular token based control mechanism. In the context of distributed computing, the research has been mainly focused on the detection of token loss and on self-stabilizing aspects[2,4,9]. They have assumed that the request message carries routing information about the source-destination identities. Using these details, there are numerous message routing techniques that make use the shortest paths on which routing from source to destination node is made [5,11].

The analysis of the proposed algorithms uses two distinct measures as in Feuerstein *et* al.[6,16]. The first measure is the average number of messages per request necessary to satisfy a sequence of requests, that is, the

* *This work was partially supported by Council of Scientific and Industrial Research (CSIR), Government of India.*
# *Corresponding author. Email: thangavelp@yahoo.com*
δ *email: rrajprasath@yahoo.com*

worst case ratio between total number of (token and request) messages and the number of requests in the sequence. The second measure is the service traffic, defined as the worst case ratio between the time in which a processor sends a request and the time in which the token reaches the processor.

In this paper, we assume that requests and token can move in opposite directions. Such a ring is referred to a bidirectional ring. First we present an improved algorithm D′ of Feuerstein *et* al. algorithm D. In D′, the additional check tour is not required as in Feuerstein *et* al [6]. It requires $2(n - 1)$ messages per request and service traffic $(3n - 3)$. Here an additional service traffic of $(n - 1)$ is due to requests which may be initiated by the processors after it has been served by the token. Then we propose an algorithm L1 for a linear array with a single shared resource. It requires $2(n - 1)$ messages per request and service traffic is bounded by $(3n - 3)$.

Algorithm D′ allocates a single shared resource in a bi-directional ring network which is a stable network structure suitable for high speed LANs. The message routing is reliable because of bi-directional movements over the processors. The second algorithm L1 proposed for a linear array provides better performance and improved service reliability for the processors to access the single shared resource. Also this algorithm provides a better lower bound with bi-directional message movements.

In the next section, we present a controlling algorithm for a bidirectional ring network. In section 3, we describe an algorithm for a linear array. Finally, section 4 concludes the paper.

2. Allocation of a shared resource in rings

Let us consider n processors P_1, P_2,..., P_n that are arranged in a logically structured ring in which the communications can be in either direction, clockwise or anticlockwise direction. For $i = 1, 2, \ldots, n$ processor, P_i is connected to P_{i+1} and P_{i-1} and P_n is connected to P_1. The requests are not related with the information about the processors that generate them. Without loss of generality, we can assume that a processor can ask the resource again only after the previous request has been satisfied. The processors have neither a shared memory nor a common clock and speeds are not related.

First we present the modified algorithm D′ of Feuerstein et al. algorithm D. Here we assume that token is allowed to pass through only in anticlockwise direction and requests are allowed to pass through clockwise direction. Also assume that each processor has a local variable M_p such that $M_p = 1$, if either a request has passed through the processor P_i or P_i has a pending request of its own and $M_p = 0$, otherwise. Processor P_i, willing the token, sends a request that passes through atmost $(n - 1)$ processors to each the token. Token may also be moving atmost $(n - 1)$ processors to serve a request. The behaviour of a processor P_i is formally described as follows:

Algorithm - D′

/* This algorithm describes the single shared resource allocation in a bi-directional ring network. In this network, token moves in anticlockwise direction and requests move in clockwise direction */

int i // $1 \le i \le n$; the index of processor P_i

boolean M_p; // takes value 0 or 1

(1) When P_i needs resource then

Begin

If P_i has token then it enters critical section;

Otherwise

if $M_p = 0$ then it sets $M_p = 1$ and sends the request in clockwise direction to the next processor

else ($M_p = 1$) no message is sent

End

(2) When P_i receives a request then

Begin

If P_i has token then it sets $M_p = 0$ and sends token to the next processor in anticlockwise direction

If P_i does not have token then

if $M_p = 0$ then it sets $M_p = 1$ and sends the request in clockwise direction
else ($M_p = 1$) no message is sent

End

(3) When P_i receives token then

Begin

If P_i has a pending request then it enters critical section in which it sets $M_p = 0$.

At the end of critical section, pass token to the next processor in anticlockwise direction

If P_i has no pending request then

if $M_p = 1$ then it sets $M_p = 0$ and sends token to the next processor

else ($M_p = 0$) token stops at P_i.

End

This algorithm D′ does not require an additional check tour as in algorithm D of Feuerstein *et* al.[6].

Theorem : 2.1

Algorithm D′ satisfies all requirements of processors. It needs atmost $2(n - 1)$ messages per request and its service traffic is $(3n - 3)$.

Proof:

In this algorithm D′, requests are allowed to travel in clockwise direction and token T is allowed to move in anticlockwise direction. Assume that initially T is at P_i. Then requests coming from P_2 to P_n will be received by token. Each processor receives a request will check its M_p value, if it is zero then that request will be forwarded; otherwise, it will be stopped, as this processor would have sent or forwarded a request signal. Movement of the token is controlled by M_p value. The requests are passed as long as no request has been forwarded from that processor by setting M_p=1. This will be carried out until the request reaches the processor which has the token. The token is moving along anticlockwise direction. Along the way it serves all pending requests and stops at the processor at which M_p=0. Thus the algorithm D′ serves all the requests.

Now to prove the first measure, we see that the processor which has token T need not send any request. So when token receives request from remaining $(n - 1)$ processors, the token may have to move all processors upto its previous processor to satisfy its requests. Thus $(n - 1)$ processors are eligible to send requests at every movement. But a processor can ask the resource only after that the previous request has been satisfied. Thus atmost $(n - 1)$ messages are needed to inform the token. In addition, token may be moved over $(n - 1)$ processors to satisfy the pending requests in the worst case. Thus the number of message exchanges per request is atmost $2(n - 1)$.

The service traffic scales the total number of message exchanges per request between the time in which a processor P_i sends a request and the time in which it gets served. Also we assume that all other requests are generated only after P_i's request. Here two different types of messages may be sent between the processor P_i and the time at which it receives the token. The first type is the request from P_i, which requires atmost $(n - 1)$ movements to reach the token in the clockwise direction. In order to serve the request of P_i, token may have to pass atmost $(n - 1)$ processors. While passing each processor, it serves all pending (or waiting) requests. Further it is possible that the processors along the path may generate new requests after the token has passed that processor. In this process, additional $(n - 1)$ messages may be generated in the worst case before token reaches the requested processor. Thus the service traffic in the worst case amounts to $(3n - 3)$. ■

3. Token-based Control in Linear Arrays

Consider n processing elements arranged in the form of a linear array, which might result in from a ring with single faulty node. Here also we use similar settings as above, except that tokens move along one direction serving all pending requests until there is a pending request; otherwise, the token changes its direction if there is pending request in the other direction. Request messages are sent using M_p value as above, by keeping track of the movement of token using direction bit d. Token has two flag bits T_l and T_d, which will be modified inside the critical section to record pending requests from left or right respectively. The behaviour of a processor P_i that executes algorithm L1 as follows:

Algorithm : L1

/* This algorithm allocates the single shared resource in a linear array – a ring with a faulty node */

int i // $1 \le i \le n$; the index of processor P_i

boolean M_p, d, T_l,T_r; // each takes value 0 or 1

(1) When P_i needs resource then

Begin

If P_i has token then it enters critical section

If P_i has no token then

if $M_p = 0$ then it sends a request along left port

if $d = 0$; otherwise

through right port by setting $M_p = 1$

if $M_p = 1$ then no message is sent.

End

(2) When P_i receives request then

Begin

If P_i has token then

If $i = 1$ (or $i = n$) then set $d = 0$(1); $T_{l(r)} = 1$; $M_p = 0$ and pass token along right (left) port of P_1.

If $1 < i < n$ then

If request is from left port then set $T_l = 1$; otherwise set $T_r = 1$;

If $d = 0$ then

if $T_r = 1$ then set $d = 1$; send token along right port;

if $T_l = 0$ then set $M_p = 0$

else($T_r = 0$ and $T_l = 1$) set $M_p = 0$ and pass token along left port of P_i.

else ($d = 1$)

if $T_l = 1$ then set $d = 0$ (if $T_r = 0$); $M_p = 0$ and pass token along left port

else set $M_p = 0$ and pass token along right port of P_i.

else (if P_i has no token)

if $M_p = 0$ then

set $M_p = 1$ and pass request to next processor in the same direction otherwise nothing is done.

End

(3) When P_i receives token then
Begin
If P_i has a pending request then it enters critical section; at the end of critical section, pass token as follows:
if $i = 1$(or n) then set $T_{l(r)} = 1$; $M_p = 0$ and
if $T_{r(l)} = 1$ then set $d = 1$(0) and pass token along right(left) port otherwise token stays at $P_1(P_n)$.
else($1 < i < n$)
if $d = 0$ then set $d = 1$
if $T_l = 0$ then set $M_p = 0$ and pass token along right port
else set $d = 0$ if $T_r = 0$ then set $M_p = 0$ and pass token along left port.
else (if P_i has no pending request then)
if $M_p = 0$ then
if $d = 0$ then set $T_r = 0$
if $T_l = 1$ then pass token along left port
else token stays at P_i
else ($d = 1$) set $T_l = 0$
if $T_r = 1$ then pass token along right port;
else token stays at P_i
else ($M_p = 1$) if $d = 0$[or 1] then
if $T_{l[\text{or } r]} = 0$ then set $M_p = 0$; $d = 1$ and pass token along right [or left] port
End

Theorem : 3.1
Algorithm L1 serves all requests and requires 2 (n - 1) messages per request and service traffic is 3 (n - 1).

Proof :
Proof is similar to theorem 2.1. ■

Example:

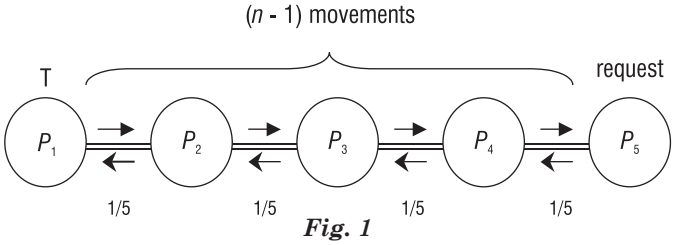

Fig. 1

To illustrate the performance of the proposed algorithm L1, consider a linear array of $n = 5$ processors as shown in fig. 1. We assume that token resides at P_1 and processor P_5 generates a request to access the single shared resource. The generated request advances over the processors only if M_P value of that processor is zero; otherwise the request would be stopped as already a request had been sent. So the generated request from processor P_5 requires (n - 1) = 4 movements to inform the token in the worst case. Similarly as token moves from processor P_1 to P_5, it serves all the pending requests in its way if any request is generated in the mean time. This type of movements require (n - 1) = 4 movements. Finally additional (n - 1) message movements are required as processors may generate request messages after the token had left that processor. Hence the algorithm serves the request of processor P_5 and requires 8 message exchanges. In this case, the service traffic amounts to 12 message exchanges.

4. Conclusion

In this paper, we have proposed two protocols in a bidirectional ring network and a linear array using token based message passing technique. The algorithm D′ for a bidirectional ring network does not require an additional check tour as we require in the algorithm D of Feuerstein *et* al.[6]. Also we have generalized an algorithm, namely L1 for a linear array, which efficiently describes bidirectional message as well as token passing strategy. A *skewed tree* [8], it may be skewed to left or right, is similar to a linear array structure and thus the algorithm L1 could be applicable to a binary tree skewed to either left or right regardless of its degree. Our future work includes this problem with various unexplained measures to have a deep insight, for example, fault-tolerance aspects like node faults or link faults during transmission in higher dimensional topologies like k-ary n-cubes, butterfly networks and permutation networks.

References

1] Andrews. D, Schulz. G, A token-ring Architecture for local area networks: An update, in : Proc.IEEE COMPCON, Fall 1982.
2] Ben-Dor. A, Haveli. S and Schuster. A, Potential function analysis of Greedy hot potato routing, Theory of Computing Systems, 31 (1998) 41-61.
3] Bertsekas. D, Gallager. R , Data Networks, 2nd Ed., Prentice Hall, 1992.
4] Burns. J.E, Pachl. J, Uniform self-stabilizing rings, ACM Trans. On Prog. Languages and Systems, 11(2) (1989) 330-334.
5] Chalamiah. N and Ramamurthy. B, Finding shortest paths in distributed loop networks, Inform. Process. Lett. 67 (1998) 157-161.
6] Feuerstein.E, Leonardi.S, Marchetti-Spaccamela.A and Santoro.N, Efficient Token Based Control in rings, Inform. Proc. Lett, 66 (1998) 175-180.
7] Hajek.B, Bounds on evacuation time on deflection routing, Distributed Computing, 5 (1991) 1-6.
8] Horowitz. E, Sahni. S and Rajasekaran. S, Computer Algorithms, Galgotia Publications, 1998.
9] Lann. G. Le, Distributed Systems – Towards a formal approach, in B.Gilchrist(ed.) Inform. Proc. Lett, 77 (1977) 155-160.
10] Lodya. S and Kshemkalyani. A, A fair distributed mutual exclusion algorithm, IEEE Trans on Parallel and Dist. Systems, Vol. 11, No. 6 (2000) pp. 537-549.
11] Mukhopadhyaya. K and Sinha. S.B, Fault-tolerant routing in distributed loop networks, IEEE Trans on Computers, 44(12) (1995) 1452-1456
12] Raynal. M, Distributed Algorithms and protocols, John Wiley and sons, 1988.
13] Reisig. W, Elements of Distributed Algorithms, Springer-Verlag, Berlin, Heidelberg, 1998.
14] Ross. F.E, FDDI-A tutorial, IEEE Communication Magazine, 24(1986) 10-17.
15] Stallings. W, Data and Computer Networks, 5th Ed., Prentice Hall, 1997.
16] Tarjan. R.E, Amortized Computational Complexity, SIAM J. Alg. Discrete Math.,6(2) (1985) 306-318.
17] Wu. J, Distributed System Design, CRC Press, 1999.