J Inf Process Syst, Vol.10, No.1, pp.36~54, March 2014
http://dx.doi.org/10.3745/JIPS.2014.10.1.036

pISSN 1976-913X
eISSN 2092-805X

A Token Based Protocol for Mutual Exclusion in Mobile Ad Hoc Networks

Bharti Sharma*, Ravinder Singh Bhatia**, and Awadhesh Kumar Singh**

Abstract—Resource sharing is a major advantage of distributed computing. However, a distributed computing system may have some physical or virtual resource that may be accessible by a single process at a time. The mutual exclusion issue is to ensure that no more than one process at a time is allowed to access some shared resource. The article proposes a token-based mutual exclusion algorithm for the clustered mobile ad hoc networks (MANETs). The mechanism that is adapted to handle token passing at the inter-cluster level is different from that at the intra-cluster level. It makes our algorithm message efficient and thus suitable for MANETs. In the interest of efficiency, we implemented a centralized token passing scheme at the intra-cluster level. The centralized schemes are inherently failure prone. Thus, we have presented an intra-cluster token passing scheme that is able to tolerate a failure. In order to enhance reliability, we applied a distributed token circulation scheme at the inter-cluster level. More importantly, the message complexity of the proposed algorithm is independent of *N*, which is the total number of nodes in the system. Also, under a heavy load, it turns out to be inversely proportional to *n*, which is the (average) number of nodes per each cluster. We substantiated our claim with the correctness proof, complexity analysis, and simulation results. In the end, we present a simple approach to make our protocol fault tolerant.

Keywords—MANET, Inter-Cluster, Intra-Cluster, Mutual Exclusion, Token Ring

1. INTRODUCTION

Mutual exclusion is a classic coordination problem in distributed computing systems. The purpose of mutual exclusion protocols is to guarantee exclusive access to the critical resource. The code segment that is executed to access the shared resource is called the critical section (CS). The processes that compete for the shared resource cycle through the entry, critical section, exit, and remainder states. Fundamentally, designing the protocol for mutual exclusion is to design entry and exit protocols. It is one of the most highly researched problems in the computing community. Thus, a number of protocols that apply various approaches have been proposed in the literature on distributed computing. A survey of mutual exclusion protocols for static distributed systems is given in [1, 2]. However, the last two decades have witnessed tremendous development in the communication technology that resulted in the emergence of various types of networks (e.g., static distributed networks, cellular mobile networks, mobile ad hoc networks, and sensor networks). The change in networking technology has greatly altered the means of carrying out distributed computing. Unfortunately, the protocols developed for one

Manuscript received February 3, 2013; accepted January 28, 2014.
Corresponding Author: Awadhesh Kumar Singh (aksinreck@rediffmail.com)
* DIMT Kurukshetra, Haryana, INDIA (bharti_kanhiya@yahoo.co.in)
** NIT Kurukshetra, Haryana, INDIA (rsibhatia@yahoo.co.in, aksinreck@rediffmail.com)

Copyright ⓒ 2014 KIPS

type of network either fail completely or fail to work with matching performance in other types of networks. Thus, every change in the networking technology virtually triggers computer scientists to develop new protocols for the new environment. A two-tier principle for adapting protocols for mutual exclusion in cellular mobile networks is given in [3] and a method to restructure distributed algorithms, for the mobile computing environment, is presented in [4]. The use of logical structures is another popular approach to design message efficient distributed algorithms for dynamic networks. Recently, many mutual exclusion algorithms have been proposed that use logical structures, like the grid structure in [5], the two-dimensional array structure in [6, 7], and the hypercube structure in [8].

The ad hoc networks are installed for special purposes on a temporary basis. However, the constituting component nodes have various limitations (e.g., constrained battery power, a small amount of storage space, and limited computing capability) and thus, are susceptible to failures. Nevertheless, the ad hoc networks are easy to set up and can operate without any pre-existing infrastructure. Unlike cellular networks, they do not possess a base station and each host acts as a router, too. They try to provide connectivity beyond the range of fixed and cellular infrastructures. The mobile ad hoc networks (MANETs) use a wireless communication link between the nodes and have a three-dimensional movement of network nodes. Hence, we usually do not make assumptions about the patterns of movement in MANETs. There are many well-understood MANET algorithms. An overview of mutual exclusion protocols for MANETs is given in [9, 10]. The vehicular ad hoc network (VANET) [11] is a variant of MANET, where each vehicle is a wireless network enabled node. Although, to handle mutual exclusion, the VANET may use the existing MANET algorithms, we can enhance their performance by creating logical structures, (e.g., clusters) if we keep the restricted traffic movement pattern in mind when creating such structures. Hence, VANETs may use a cluster based MANET algorithm for various applications. For instance, it can be used in a scenario like war or rescue, where a group of military vehicles or people follows a synchronous movement (i.e., all nodes move in the same direction, move together to a new strategic area, or move along a highway). Thus, the relative position of nodes does not change despite a change in their absolute physical location. Hence, it is possible to predict that the nodes that are likely to remain together. Moreover, the life of communication links is also predictable by using the information about the position, speed, and direction of the movement of the nodes. Therefore, it is possible to create clustering algorithms based on traffic movement components, where the clustering structure would remain unperturbed for sufficiently longer durations. The clustering algorithms group the nodes into clusters to reduce communication overhead. One node from each cluster acts as the cluster head. All inter-cluster communication relays through the cluster head. The following Fig. 1 represents the schematic view of an assumed VANET environment. The dotted lines represent the wireless links that connect the cluster heads that form a virtual dynamic ring, which is computed on the fly.

In the literature on distributed computing, many algorithms exist for clustering and head selection. The network infrastructure is assumed to be fairly stable for a long time period. However, the movement of an individual cluster is unpredictable to a large extent because the clusters move autonomously in order to execute their specific assigned task. However, each cluster is assigned an individual task that is decided jointly by all the cluster heads, as dictated by the situation. The clusters share the critical resources that may be needed from time to time. Some of them are to be accessed in a mutually exclusive way.

Fig. 1. The Schematic View of VANET

The paper presents a token-based mutual exclusion algorithm that has been designed by keeping the controlled node movement, which is an inherent requirement of the ad hoc network, in mind. The token circulation schemes have a long latency and the token asking schemes need a coordinator. We intended to design a protocol that retains the advantages of both worlds without suffering too much from the above limitations. Hence, we used different schemes at the inter-cluster level and at the intra-cluster level. Consequently, our protocol is asymmetric. Nevertheless, it is message efficient and thus suitable for MANETs. The message complexity is independent of N, which is the total number of nodes in the system, and under a heavy load, it is inversely proportional to n, which is the (average) number of nodes per each cluster. Although the performance has been evaluated for fault-free setting, the method to provide fault tolerance is explained in detail, in Section 8.

2. THE SYSTEM MODEL

The system is constituted of mobile nodes that are arranged in clusters. Each node is assigned a unique id. Each cluster has a cluster head. All cluster heads know each other. Each node is assumed to remain in the communication range of its current cluster head, though it may move to some other cluster using a handoff procedure. A cluster head is not always in the communication range of all the remaining cluster heads. Thus, the neighbor set of a cluster head is dynamic. However, it is not empty. The cluster heads pass messages in the ring that is dynamic and computed on the fly. The algorithm uses a special message, which is called a "token," to support mutual exclusion. The token holder cluster has the privilege to use the critical resource. Once the token holder cluster is served, its cluster head passes the token to a neighboring cluster that is yet not served in the current round. The token receiving cluster head is called the successor of the token releasing cluster head. The successor does the identical action as its predecessor. In this way, a dynamic ring of cluster heads is formed. Any cluster member node may crash and the token may be lost. However, the crashing of the cluster head would require "reclustering," which is a problem beyond the scope of our paper. Due to node

mobility and node failure, the composition of clusters may undergo significant changes over time and it may also trigger the need for reclustering to occur. For instance, in military applications, a strategic unit (e.g., company, battalion, or brigade) may be constituted of various types of vehicles that can be modeled as mobile nodes. These strategic units can be modeled as clusters and their respective commanders as cluster heads. Although the vehicles of a particular strategic unit may not always (or, may not always need to) be in the communication range of each other, they always remain connected with their commander, in order to ensure strategic control and coordination. Similarly, a commander may not remain permanently connected with its other counterparts due to its autonomous and strategic movement.

3. THE ALGORITHM CONCEPT

The protocol supports mutual exclusion in clustered MANETs. It is token based and fault tolerant. Thus, it ensures exclusive access to shared resource even in the event of node and token failure. It uses the following two types of tokens: a unique global token, which circulates among the cluster heads, and a number of local tokens. The number of surviving local tokens is equal to the number of clusters in the system. Hence, each cluster has one local token, which is private to that cluster. The proposed algorithm uses both token circulation and token asking schemes.

The ad hoc networks execute the applications that have heavy contention for critical resources. Thus, each cluster always has some outstanding request. Hence, in general, the cluster heads remain hungry for a global token. It is well established that the token ring is very efficient under heavy contention. However, it is inefficient under light contention. Hence, we used the token circulation scheme among the cluster heads that are arranged in a dynamic ring, which is computed on the fly. Moreover, the token asking schemes incur a delay when used in some types of logical structures.

However, it is unlikely that all of the nodes of a cluster have an outstanding request. Therefore, we use the token asking scheme within the nodes of a cluster. As a result, our protocol has the following two components: the inter-cluster component and the intra-cluster component. The former is executed at the cluster heads and the latter is executed at the nodes of each cluster. In the existing literature on distributed computing, reference [12, 13] used clustering in their token based mutual exclusion algorithm for static distributed systems. Baldoni *et al.* [14], proposed a token-based algorithm for mutual exclusion in MANETs. The algorithm passes a token over the logical ring, which is dynamically created. For each round of token circulation there is a coordinator. The initial token holder node becomes the coordinator for the first round. The token circulation is initiated when a node becomes hungry for the token and sends a token request to the coordinator. The token is circulated in the entire ring, even if no other node is hungry and thus, the scheme incurs a large overhead [15]. The role of the coordinator is transferred to next node in the dynamic ring at the beginning of each round. The ID of the next coordinator is sent with the token in each round. Hence, the hungry nodes can pass their token requests directly to the coordinator. When a node is served, it passes the token to the next nearest node that has not been visited by the token. In this way, the algorithm ensures starvation freedom.

Unlike [12, 13, 14], our protocol does not have any ring coordinator. Despite having some advantages, the existence of a coordinator injects a centralized character into the protocol, which

results in reduced reliability. Although, as mentioned in the above three papers, the protocols rotate the coordinator and each change in coordinator results in the flow of a large number of messages to inform all the nodes about the new coordinator. Moreover, in MANETs, the communication links are volatile. Thus, the centralized coordination can't work effectively in the assumed environment. Hence, no cluster head has been elected or nominated as the ring coordinator. We have used a fully distributed approach to pass the global token among cluster heads. However, our clusters are assumed more reliable with robust cluster heads and, thus, in order to have better message efficiency, we have applied a centralized protocol to pass the local token within the nodes of a cluster. Nevertheless, in order to compensate for the loss of reliability, which is due to use of the centralized scheme, this scheme has been made fault tolerant. Next, we will discuss how both of the components work.

3.1 The Inter-Cluster Component

Each cluster head is aware of all of the other cluster heads. The cluster heads are assumed to be in a dynamic ring that is computed on the fly. In order to circulate the global token, the cluster heads that are not visited by it, are assumed to be hungry for token. The global token holder cluster head, after satisfying the pending token requests in its own cluster, forwards the global token to a token hungry neighbor (a.k.a. successor) cluster head. However, if there is no hungry neighbor cluster head, then it forwards the global token to a non-hungry neighbor cluster head that does not use the global token, and instead, it tries to find a hungry successor cluster head. In this way, all clusters are eventually served. When the global token holder observes that each cluster head has received the global token once in the current round, the current global token holder starts a new round of global token circulation. For this purpose, it resets (similar to initialization) the information contained in the global token, as if no cluster head has received the global token yet.

3.2 The Intra-Cluster Component

The cluster head of each cluster acts as a coordinator for the cluster. Each hungry node of a cluster sends a request for the local token to its cluster head, in order to enter the CS. The cluster head collects the requests sent by the hungry nodes and serves them with the help of the local token, as and when it receives the global token. When the requests, which are received before the reception of the global token, get served, the global token is forwarded to the successor cluster head in the dynamic ring. This scheme is used to avoid the starvation in token hungry clusters. In case, prior to receiving the global token, a cluster head could not receive any token request from its cluster members, it forwards the global token to the successor cluster head. However, after the reception of a global token at the cluster head, the token requests received at the cluster head are kept as pending until the global token is received by the cluster head in the next round.

4. Data Structures, Types of Messages, and Process Behavior

The protocol uses the data structures and types of messages as listed below.

4.1 At the Intra-Cluster Component

4.1.1 The Data Structures that are maintained at each Cluster Node

1. *try*: A Boolean variable, which is set to true, when the node wants to enter the CS
2. *In*: A Boolean variable, which is set to true, when node is in the CS
3. *co*: Identifier of a cluster head
4. *seqno*: This is the round number of a local token that is known to the cluster node
5. *type*: This is a variable that may assume any of the following values depending upon the type of message that is to be sent by cluster node to cluster head:
   (a) *req*: Request made to the cluster head when the node is hungry for the token
   (b) *over*: To inform the cluster head that the node has executed the CS.
   (c) *over&out*: To return the local token back to cluster head when there is no pending request entry in the local token.
6. *tokenholder*: This is the Boolean variable, which becomes true, after receiving a local token
7. *tokenvalue*: This is an integer, which is initially 0, that stores the number of local tokens at cluster node *i*
8. $neighbor_i$: This is the set of nodes that is currently in the communication range of cluster node *i*

4.1.2 Messages

A node communicates with other node within a cluster via the types of messages listed below.

1. *localtokenmsg<localtoken, roundno>*: This has two fields. The first is a queue of all of the requesting nodes of the cluster, and the second is the round number of the *localtokenmsg* that indicates how many rounds it has completed in the cluster. It is initialized to zero. The *localtokenmsg* is used to grant the privilege to execute the CS. Initially, the cluster head sends *localtokenmsg* to the requesting node that is entered at index zero in *localtoken*. The node receiving *localtokenmsg*, after executing the CS, deletes its entry from the *localtoken* and forwards *localtokenmsg* to the next hungry node by looking the entry in *localtoken*. When there is no entry left unserved in the *localtoken*, the site, which last executed the CS, then returns it to the cluster head.
2. *msg_to_ch*: *<Id, type, co>* The nodes of the cluster use this to send various types of messages to their cluster heads. It has three variables. The first field is the identifier of the sender node, the second one is a type of message, and third one has the ID of the cluster head.

4.1.3 The Behavior of a Process at Cluster Node

Each hungry node sends a token request to its cluster head. After receiving the *localtokenmsg* from its cluster head, the hungry node compares its local *seqno* with the global *roundno* that is available in the received *localtokenmsg*. If the local *seqno* is less than the global *roundno*, the node enters in the CS. After executing the CS, the node sets its local *seqno* to be equal to the global *roundno* that is available in the *localtokenmsg*. Furthermore, the node sends the *over* message to its cluster head and forwards the *localtokenmsg* to the next hungry node that has an outstanding request in the *localtokenmsg*. However, if the next hungry node is not in its communication range, the node returns the *localtokenmsg* back to its cluster head, which

forwards the *localtokenmsg* to the next hungry node that has an outstanding request in the *localtokenmsg*. Nevertheless, if the *localtokenmsg* has no unserved request, the node sends an *over&out* message to its cluster head and returns the *localtokenmsg* back to its cluster head. Listed below are the events and the actions taken on the occurrence of those events.

A. Cluster node *i* is hungry for token:

1. *try* ← True;
2. Send *msg_to_ch*<*i*, *req*, *co*> to the cluster head
3. Wait for the *localtokenmsg*;

B. On receiving *localtokenmsg* by cluster node *i*:

4. increment *tokenvalue; tokenholder* ← True;

Case 1: Single *localtokenmsg* is at a cluster node *i*

5. **IF** *tokenvalue* == 1 **THEN**
6. **IF** *seqno* < *localtokenmsg.roundno* **THEN**
7. *In* ← True; /* in critical section */
8. Execute **STEPS** 15-24;
9. **ELSE** execute **STEPS** 17-24;

Case 2: Two or more *localtokenmsg* are at a cluster node *i*

10. **IF** (*tokenvalue* ≥ 2) **THEN**
11. Keep the *localtokenmsg* with maximal *localtokenmsg.roundno*; consume all other *localtokenmsg;*
12. *tokenvalue* ← 1;
13. **IF** *seqno* ≥ *localtokenmsg.roundno* **THEN** execute **STEPS** 17-24;
14. **ELSE** execute **STEPS** 7-8;

C. On Exit CS by cluster node *i*:

15. *In* ← false;
16. *seqno* ← *localtokenmsg.roundno;*
17. dequeue node *i* from *localtoken*[];
18. **IF** *localtoken*[] == ∅ **THEN**
19. Send the *over&out* message to the cluster head;
20. Send the *localtokenmsg* to the cluster head; decrement *tokenvalue; token holder* ← False;

ELSE

21. Send an *over* message to the cluster head;
22. **IF** *localtoken*[head] ∉ $neighbor_i$ **THEN**
23. execute **STEP** 20;

24. **ELSE** send *localtokenmsg* to the *localtoken*[head]; decrement *tokenvalue*; *tokenholder* ← False;

4.2 At Inter-Cluster Component

4.2.1 The Data Structures that are maintained at each Cluster Head

1. *co*: This is the ID of the cluster head.
2. *globalno:* This is the round number of global token that is known to the cluster head.
3. *LQ*: This is queue of the token requests that are received at cluster head before the cluster head receives the global token. *LQ* is used to generate the local token. After receiving the global token, *LQ* is copied into the *localtokenmsg* by the cluster head before forwarding the *localtokenmsg* to the requesting node, which is entered at index zero.
4. $N\text{-}LIST_i$: This is the list of cluster heads that are the current neighbors of cluster head *i*. Initially, it is empty.
5. $hold_i$: This is a Boolean variable whose value is initially false, and then set to true when cluster head *i* receives the global token.
6. *hungry*: This is a Boolean variable that is set to true when some node within the cluster is hungry.
7. *globaltoken* < *globalvalue*, *color*[0…*N*-1]>: This contains the following two fields: an integer *globalvalue*, which is initialized to 0 and incremented in the beginning of each new circulation of a global token; and an array *color*[] that stores the status of all cluster heads. In the *globaltoken.color*[0…*N*-1] the entries are made as listed below.
   a. W: Signifies that the corresponding cluster head is yet to receive a *globaltoken*.
   b. R: Signifies that the corresponding cluster head has received the *globaltoken* and that it has used it to serve the token requests from its cluster.
8. *NLQ*: This is the queue of the token requests, which were received at the cluster head after the cluster head had received the *globaltoken*.

4.2.2 The Behavior of a Process at Cluster Head

Initially, one cluster head will arbitrarily become the holder of the *globaltoken*. If it has some outstanding token request(s) from its cluster, then it will use the *globaltoken* to generate a *localtokenmsg* in order to serve such requests. After serving its cluster, the cluster head makes its entry to a *globaltoken* as having "visited" and forwards the *globaltoken* to next reachable (a.k.a. successor) cluster head, whose entry in the *globaltoken* is labeled as "not visited." Here, the cluster head ID is used to do a tie break in case multiple cluster heads are reachable. The new recipient of the *globaltoken* also uses it in the identical manner. However, if all of the entries in the *globaltoken* have been "visited," the current *globaltoken* holder cluster head resets them to being "not visited" and starts a new round of token circulation. The token requests sent by the hungry sites are stored at the corresponding cluster head site by entering the requester ID in the local request queue. In a particular round, a cluster head only serves those token requests that were received at the cluster head before the cluster head could receive the *globaltoken* in that round. The remaining token requests, which were received at the cluster head after the cluster head received the *globaltoken* in that round, are kept as pending for the next round. This

mechanism ensures that the other clusters do not starve for the token. On the occurrence of various events, the cluster head takes the actions listed below.

A. On receiving *msg_to_ch* from cluster node *j* to cluster head *i*:

```
msg_to_ch.type == req
        IF globaltoken is not yet received THEN
           enter j in LQ;
        ELSE enter j in NLQ;
```

B. On receiving *globaltoken* at cluster head *i*:

```
IF globaltoken.color[i] == R THEN forward globaltoken to successor clus
    ter head like Case 3 below;
ELSE
hold ← True;
globaltoken.color[i] ← R;
IF hungry == False THEN
      forward the globaltoken to a successor cluster head j such that
      j≠i, color[j]=w, j∈N-LIST_i;
ELSE
      globalno ← globaltoken.globalvalue;
      generate localtokenmsg; increment localtokenmsg.roundno;
      send localtokenmsg to localtoken[head]; wait for message;

Case 1: IF the message received from the cluster node is over THEN
              Update LQ;
        ELSE wait for a timeout to receive the over or over&out message;

Case 2: IF timeout == true for the over or over&out message THEN
        generate a new localtokenmsg with the same roundno as it was in the
        lost localtokenmsg;
        send the new localtokenmsg to localtoken[head];

Case 3: IF localtoken[] == ∅ OR received over&out THEN hold ← False;
         //forward globaltoken to successor cluster head//
        begin
         IF j≠i ∧ globaltoken.color[j] == W THEN
              begin
              IF j ∈ N-LIST_i THEN send the globaltoken to j
              ELSE send the globaltoken to some cluster head k such that
              globaltoken. color[k] == R ∧ k ∈ N-LIST_i
              end
         ELSE //start a new circulation round of globaltoken//
         increment globaltoken.globalvalue; ∀j globaltoken.color[j] ← W;
```

forward the *globaltoken* to a successor cluster head *j* such that
$j \neq i, j \in N\text{-}LIST_i$;
end
IF *NLQ* is empty **THEN** *hungry* ← False **ELSE** *hungry* ← True;

5. The Correctness Proof

In this section, we show that the algorithm achieves mutual exclusion and it is free from deadlock and starvation.

5.1 Mutual Exclusion

The proof of mutual exclusion is trivial in token-based protocols. However, in our protocol, there is the provision of more than one type of token, namely the global token and local token. Hence, in order to show that it ensures mutual exclusion, we need to prove that one site, at most, holds the privilege to execute the CS. In our protocol, only the local token message is used to grant privilege to execute the CS. Moreover, a cluster head generates the local token message only after receiving the global token and the local token remains in circulation within its cluster until the cluster head holds the global token. In addition, the global token is the only token that is used to pass privilege to a cluster head to generate the local token message. Initially, only one cluster head holds the global token, say site S. There are three situations where site S sends the global token to another cluster head: (1) all of the hungry sites, within the cluster of site S that have requested for token before the receipt of global token, have finished the execution of the CS and the site that executed the CS last, has returned the local token back to site S; (2) no site, within the cluster of site S, was hungry before the receipt of a global token; or (3) site S has already received the global token once in the current round and therefore, it is prohibited to generate the local token message again in the current round. In either of these cases, site S sends the global token to only one of the reachable cluster heads that is yet to receive the global token in the current round. The new holder of the global token also uses it in the manner explained above, before sending it out. Therefore, one site at most holds the privilege to execute the CS.

5.2 Starvation Freedom

The starvation occurs when few sites repeatedly execute the CS, while other sites wait indefinitely for their turn to do so. In our algorithm, there is single global token that passes privilege among the cluster heads to ensure inter-cluster mutual exclusion. The only cluster head that is currently holding the global token can use its local token to serve the hungry nodes of its cluster. At each cluster head, the local token queue (LTQ) is a FCFS (First-Come-First-Serve) queue, which is used to contain the requests that have been received at the cluster head, before the cluster head itself receives the global token. The local token is passed around according to the order of requests in the LTQ. If site Y, whose request is in front of site X's request in the LTQ, has finished its execution of the CS, site Y's subsequent request will never be added to the current LTQ. Thus, no site will be executing CS repeatedly in the current LTQ round. Any site Y, whose request is in front of site X's request in the LTQ, will be able to make a subsequent token request and, therefore, will only get the local token again in the next LTQ round.

Moreover, the number of hungry nodes within the cluster is finite and so the length of the LTQ is also finite. Hence, site X will eventually get the local token. Once the current LTQ is served, the cluster head is bound to release the global token. Therefore, no cluster head will be holding the global token indefinitely. The system model assumes that the neighbor set of a cluster head is never empty. Consequently, when a cluster head releases the global token, an "unvisited" cluster head will eventually receive it. Since, the number of cluster heads is finite; every cluster head will eventually receive the privilege to use the critical resource. Hence, no cluster will be starving for the CS. Therefore, the algorithm is free from starvation.

6. THE COMPLEXITY ANALYSIS

6.1 Message Complexity

Assume there are n nodes in each cluster and that there are m cluster heads. However, each cluster may not have the same number of nodes. In that case, n is the average number of nodes per each cluster. The cluster heads has been identified as m_0, m_1, .. , m_{m-1}. The global token is assumed to traverse clockwise in the dynamic ring, which is computed on the fly. We will now analyze the performance of the protocol under both heavy and light load conditions.

(i) Heavily Loaded System

Under the heavy load condition, every node in all of the clusters is assumed to be hungry. Therefore, the total number of requesting nodes will be $m \times n$, which will generate $m \times n$ request messages. In order to serve these requests, the total number of global token messages generated will be m and the total number of local token messages generated will be $m \times n$. Therefore, the total number of all types of messages will amount to $(m \times n) + m + (m \times n)$. Hence, the number of messages required to fulfill one request will be equal to $\{2(m \times n)+m\}/(m \times n)$ that is $(2+1/n)$. Consequently, the message complexity under heavy load conditions would be $O(1/n)$. It is obvious that the message complexity is independent of m, which is the number of cluster heads. Here, it is worth noting that n is the (average) number of nodes per each cluster and not the total number of nodes in the system.

(ii) Lightly Loaded System

Under the light load condition, it is assumed that no request is pending. For a newly generated request, the worst-case scenario occurs when the request is from cluster head m_1 and the global token is at cluster head m_2. In order to fulfill this request, the algorithm would generate one local token request message, one local token message, and $(m-1)$ global token messages. Hence, in order to fulfill the request, the total number of required messages amounts to $(1+m)$. Thus, the message complexity under light load conditions would be $O(m)$. It is worth mentioning that in strategic networks, the number of cluster heads is $m << N$, where N is the total number of nodes in the system. Usually, in clustered VANETs, $m \leq \sqrt{N}$.

6.2 Synchronization Delay

It is the time beyond a site leaves the CS and ahead of the next site enters the CS. It is the

performance metric that has significance under high load condition. It is obvious from the message complexity analysis that there is a delay of only one message from a CS exit to the next CS entry. Therefore, the synchronization delay would be T, where T is the propagation time of a single message.

6.3 Response Time

The response time is important only for lightly loaded systems. It is the time interval a request waits for its CS execution to be over after its request messages have been sent out. The message complexity analysis show that the response time would be $(m + 1)T$, where m is the number of clusters and T is the propagation time of a single message. Although, the value of m is constant, the response time depends on the current position of the global token, because we have used token forwarding, rather than token asking, among the cluster heads. Thus, it is noteworthy that the response time would amount to $(m + 1)T$ only in the worst case scenario. In the best case scenario, when the global token is received at the requesting node's cluster head immediately after receiving the CS request, the response time would boil down to $2T$ (i.e., 1 request message + 1 local token message). Therefore, if we represent the response time as R, the following relation holds as: $2T \leq R \leq (m + 1)T$.

7. The Simulation Analysis

We simulated our protocol using NS-2 network simulator and considered a 500×500 grid. We assumed that a suitable clustering algorithm was in place. The total number of nodes was varied from 20 to 500. The mobility rate was considered from 8 m/sec to 40 m/sec with a pause time as 150 sec. The simulation time was given as 1,500 seconds. Initially, all clusters were assumed to have the same number of nodes. We experimented with our protocol for the following variations of network size: (i) by increasing the number of nodes per cluster and (ii) by increasing the number of clusters when the number of node increases significantly. The results were comparable. We also considered the random mobility model and we compared our protocol with

Fig. 2. The Request Load vs. the Number of Hops per CS Access

the Baldoni *et al.* [14] protocol. Since, the Baldoni *et al.* [14] protocol is non-fault tolerant, our performance comparison only applies to fault-free cases. Nevertheless, the fault tolerance is discussed in Section 8. Each point in the graphs shown below, were plotted after taking the average of the values obtained from ten different runs. It is clear from Fig. 2 that upon decreasing the request load, the number of hops per CS access monotonically decreased. However, the trend was the opposite in the case of the Baldoni *et al.* [14] protocol.

Similarly, upon decreasing the request load, the number of messages decreased significantly in our protocol. However, the decrease is not so significant in the case of the Baldoni *et al.* [14] protocol (ref. Fig. 3).

Fig. 3. The Request Load vs. the Number of Messages

It is more interesting to note from Figs. 4, 5, and 6 that the number of messages shows a downtrend with an increase in the number of requests under all three light, medium, and heavy load conditions, respectively.

Fig. 4. The Effect of the Light Request Load

Fig. 5. The Effect of Medium Request Load

Fig. 6. The Effect of the Heavy Request Load

In order to evaluate the effect of the varying mobility rate, we considered 5 – 25 m/s, 25 – 100 m/s, and 100 – 200 m/s as the low, medium, and high mobility rates, respectively. We concluded, as shown in Fig. 7, that the change in mobility rate only had a marginal effect on the number of messages for varying request loads. Similarly, as shown in Fig. 8, we observed that there is not significant change in the slope of the plots on low, medium, and high mobility rates. Therefore, we can conclude that the impact of mobility is very low on the request processing time as well. Moreover, the increase in request load does not trigger further increase in the request processing time beyond a certain point. Therefore, our protocol delivers consistent performance under heavy contention and high mobility.

Fig. 7. The Effect of the Varying Mobility Rate on the Message Count

Fig. 8. The Effect of the Varying Mobility Rate on the Processing Time

8. Fault Tolerance

We have proposed a dual global token ("primary" for mutual exclusion and "secondary" for fault tolerance) mechanism to combat the loss of global tokens. We assumed that both of the global tokens would not crash at the same time. We introduced a "secondary" global token that has the three attributes of a source address, destination address, and a round number, and included these attributes in the "primary" global token as well. For the sake of brevity, we will henceforth call the "primary" global token *primary* and the "secondary" global token *secondary*.

8.1 Handling the Loss of a Global Token

A site can use the *primary* if the token contains the same site ID in its destination address entry. Otherwise, the site forwards the *primary* towards the destination. Assume, that site i forwards *primary* to site j and subsequently forwards the *secondary* to site j. Now, as a channel in non-FIFO, there are two possibilities, as listed below.

(i) If the *secondary* reaches site j first, then site j holds the *secondary* and waits for the *primary* until timer T expires and sends a "generate a new *primary*" message to site i. Now, there could be two possible cases:

Case 1: The old *primary* was lost: The new *primary* will reach site j and site j forwards (after using it, in case site j is hungry) the new *primary* to a successor, say site g. Subsequently, site j forwards the *secondary* to site g.

Case 2: The old *primary* was blocked enroute: Now, the old (new) *primary* will reach site j. Site j forwards (after using it, in case site j is hungry) the old (new) *primary* to a successor, say site g. Subsequently, site j forwards the *secondary* to site g. The new (old) *primary*, which is received later, will be consumed by site j, on the basis of having a round number that is same as the old (new) *primary*.

(ii) If the *primary* reaches site j first, then site j uses it, in case site j is hungry, and then site j forwards the *primary* to a successor, say site g. In the mean time, if site j receives the *secondary* then site j forwards the *secondary* also, to site g, soon after sending the *primary*. However, if site j is not hungry, it forwards the *primary* to a successor, say site g and waits for the *secondary* until timer T expires. If site j receives the *secondary* then it forwards the *secondary* also, to site g. Otherwise, site j generates a new *secondary* and forwards it to site g. The old *secondary*, which is received later, will be consumed by site j on the basis of having the round number same as in the new *secondary*. However, if the old *secondary* could not find a direct route to site j, then it may reach some site, say $h \neq j$, which will pass the *secondary* to site i, which is the sender of the *secondary*. It will then generate the new *primary* and will send the new *primary* followed by the *secondary* to the successor cluster head.

8.2 Handling the Site Crash

If the crashed site was the holder of the *secondary*, then the system will not come across any new difficulty because the case is similar to the *secondary* that was lost/delayed, as mentioned above. However, if site j crashed immediately after receiving the *primary* then the *secondary* will not find a route to node j as it is unreachable, due to having crashed. Now, the *secondary* will be routed to some site, say $g \neq j$, which will check for site j in its 1-hop neighborhood. Now, there are two possibilities:

Case 1: If site j is a 1-hop neighbor of site g, then it is easy for site g to forward the *secondary* to site j.

Case 2: If site j is not a 1-hop neighbor of site g, then site g will pass the *secondary* to site i, which is the sender of the *secondary*. Now, site i will generate the new *primary* and will forward the new *primary* followed by *secondary,* to the successor cluster head.

8.3 Handling the Loss of a Local Token

The protocol handles the loss of a local token within the cluster by using the *over* and

over&out messages as follows. After executing the CS, each node sends an *over* message to the cluster head and forwards the local token, LTQ, to the next hungry node. When the cluster head receives an *over* message from a node it makes an entry in the copy of the LTQ that remains with the cluster head, corresponding to that node as its token request has been "served." If a local token is detected as lost in transit, the cluster head can regenerate the local token using the copy of the LTQ that is available at the cluster head. As the *over* message is the record of "served" nodes, the newly generated local token would be made available only to nodes that are yet to be "served" in the current round. However, this mechanism may lead to simultaneous availability of more than one local token within a cluster; thus, multiple nodes may enter CS, which is the violation of safety property (mutual exclusion). Nevertheless, our protocol can combat this situation using the following mechanism. Assume an arbitrary cluster C that has cluster head CH. Now, say node N1, after executing CS, sends an *over* message to node CH and passes the LTQ to node N2. Afterwards, say that the LTQ message suffered an excessive delay and, thus, CH was timed out while waiting for the next *over* message. Hence, CH will suspect the loss of LTQ and will generate a new local token, say LTQ'. However, it might so happen that LTQ is not lost and only it got delayed excessively. Now, there are three possible cases:

(i) LTQ' reached node N2 first: node N2 will enter the CS and, after executing the CS node N2 will discard the LTQ, which is received subsequently in case the LTQ contains the same round number as the LTQ'.

(ii) LTQ and LTQ' both reached node N2 at the same time: node N2 discards the LTQ in case LTQ contains the same round number as the LTQ' and subsequently node N2 enters the CS.

(iii) LTQ reached node N2 first: node N2 will enter the CS and, after executing CS node N2 discards the LTQ' in case LTQ' contains the same round number as the LTQ.

9. CONCLUSION

This paper proposes a protocol to handle the problem of mutual exclusion in ad hoc networks. In order to have the advantage of both worlds, the protocol uses both centralized and distributed schemes at different levels. The scheme is able to deal with faults. It is robust and suitable for long running applications. The message complexity of the protocol is remarkable under heavy load conditions, as well as under light load conditions. The token asking schemes suffer poor synchronization delay when they are used in the logical structures. Nevertheless, this loss has been largely compensated by our dual token-based approach. In the dual token approach, which is used by Wu-Cao-Raynal [16], if the primary token is lost in some round, say r, then the loss of a primary token is not detected unless the secondary token completes two rounds, particularly, rth and r+1th. Consequently, the latency involved in the new primary token generation is high. Thus, the critical resource remains unutilized for a longer duration. However, in our scheme, the token loss detection is timer based and the latency involved in the new primary token generation is equal to the time it takes to propagate a single message. Therefore, our dual token based fault tolerance scheme has less latency, as compared to the dual token approach used by Wu-Cao-Raynal [16].

REFERENCE

[1] A. Kshemkalyani and M. Singhal, Distributed Computing: Principles, Algorithms, and Systems, Cambridge University Press, NY, 2008, pp. 327-336.

[2] P. Saxena and J. Rai, "A Survey of Permission-based Mutual Exclusion Algorithms," Journal of Computer Standards and Interface, vol. 25, no. 2, 2003, pp. 159-181.

[3] B. Badrinath, A. Acharya and T. Imielinski, "Designing Distributed Algorithms for Mobile Computing Networks," Computer Communications, 19, 1996, pp. 309-320.

[4] R. Ghosh and H. Mohanty, "On Restructuring Distributed Algorithms for Mobile Computing," IWDC 2002, LNCS 2571, 2002, pp. 224-233.

[5] M. Bertier, L. Arantes and P. Sens, "Distributed Mutual Exclusion Algorithms for Grid Applications: A Hierarchical Approach," Journal of Parallel and Distributed Computing, vol. 66, no. 1, January 2006, pp. 128–144.

[6] H. Taheri, P. Neamatollahi and M. Naghibzadeh, "A Hybrid Token-based Distributed Mutual Exclusion Algorithm using Wraparound Two-Dimensional Array Logical Topology," Information Processing Letters, vol. 111, no. 17, September 2011, pp. 841–847.

[7] P. Neamatollahi, H. Taheri and M. Naghibzadeh, "Info-based Approach in Distributed Mutual Exclusion Algorithms," Journal of Parallel and Distributed Computing, vol. 72, no. 5, May 2012, pp. 650–665.

[8] L. Rodrigues, J. Cohen, L. Arantes and E. Duarte, "A Robust Permission-based Hierarchical Distributed k-Mutual Exclusion Algorithm," IEEE 12th International Symposium on Parallel and Distributed Computing (ISPDC), 2013, pp. 151–158.

[9] M. Benchaiba, A. Bouabdallah, N. Badache and M. Ahmed-Nacer, "Distributed Mutual Exclusion Algorithms in Mobile Ad Hoc Networks: An Overview," ACM SIGOPS Operating Systems Review, vol. 38, no. 1, 2004, pp. 74-89.

[10] B. Sharma, R. Bhatia and A. Singh, "DMX in MANETs: Major Research Trends Since 2004," Int. Conf. on Advances in Computing and Artificial Intelligence, ACAI'11, 2011, pp. 50-55.

[11] H. Hartenstein and K. Laberteaux, "A Tutorial Survey on Vehicular Ad Hoc Networks," IEEE Communications Magazine, June 2008, pp. 164-171.

[12] V. Kumar, J. Place and G. -C. Yang, "An Efficient Algorithm for Mutual Exclusion using Queue Migration in Computer Networks," IEEE Trans. Knowledge and Data Engineering, vol. 3, no. 3, 1991, pp. 380-384.

[13] P. Chaudhury and T. Edward, "An O(√n) Distributed Mutual Exclusion Algorithm using Queue Migration," Journal of Universal Computer Science, vol. 12, no. 2, 2006, pp. 142-159.

[14] R. Baldoni, A. Virgillito and R. Petrassi, "A Distributed Mutual Exclusion Algorithm for Mobile Ad Hoc Networks," 7th IEEE Symposium on Computer and communications (ISCC'02), July 2002, pp. 539–545.

[15] S. Tamhane and M. Kumar, "A Token Based Distributed Algorithm for Supporting Mutual Exclusion in Opportunistic Networks," Pervasive and Mobile Computing, vol. 8, no. 5, October 2012, pp. 795–809.

[16] W. Wu, J. Cao and M. Raynal, "A Dual-token-based Fault Tolerant Mutual Exclusion Algorithm for MANETs," LNCS 4864, 2007, pp. 572–583.

Bharti Sharma

She received the MSc degree in Information Technology from MD Univ. Rohtak, HR, India in 2003 and the MCA degree from the same Univ. in 2004. She is faculty in the Department of Computer Application at DIMT Kurukshetra, HR, India. Presently, she is working towards her PhD degree in the Department of Computer Application at National Institute of Technology (NIT) Kurukshetra, HR, India. Her field of research is Mobile Computing.

Ravinder Singh Bhatia

He received the BTech degree in Electrical Engineering from GND Univ. Amritsar, PB, India in 1987; MTech and PhD degrees in the same area from NIT Kurukshetra in 1993 and 2008, respectively. Presently, he is Professor in the Department of Electrical Engineering at NIT Kurukshetra. His research interests include Distributed Systems, Wireless Sensor Networks, and Power Quality.

Awadhesh Kumar Singh

He received the BTech degree in Computer Science from Gorakhpur Univ. Gorakhpur, UP, India in 1988; MTech and PhD degrees in the same area from Jadavpur Univ. Kolkata, WB, India in 1998 and 2004, respectively. Since August 1991, he is faculty in the Department of Computer Engineering at NIT Kurukshetra, where he is Professor, at present. His research interests include Distributed Algorithms, Mobile Computing, and Fault Tolerance.