Many challenges have been facing in Mobile Ad hoc networking due to frequent changes in the network topology and the lack of resources. Now a day’s a lots of research is going on to support QoS in the Internet and other networks, although they are not sufficient for mobile Ad hoc networks and still QoS support for such networks remains an open problem. In this paper, a new scheme has been proposed for achieving QoS in terms of packet delivery. The proposed method adopts the snapshot algorithm of distributed systems to store information and the same will be forwarded to destination using dynamic linking. The performance of the proposed method is assessed through its low processing overhead and loop freedom.

Keywords: Deadlock, Snapshot Algorithm, MANET, QoS.

A mobile ad hoc network is also called MANET. The main characteristic of MANET strictly depends upon both wireless link nature and node mobility.
features. Basically this includes dynamic topology, bandwidth, energy constraints, security limitations and lack of infrastructure [2]. MANET is viewed as suitable systems which can support some specific applications as virtual classrooms, military communications, emergency search and rescue operations, data acquisition in hostile environments, communications set up in Exhibitions, conferences and meetings, in battle field among soldiers to coordinate defence or attack, at airport terminals for workers to share files etc. Several routing protocols for ad hoc networks have been proposed as DSR and AODV. Major emphasis has been on shortest routes in all these protocols in response whenever break occurs.

In this paper a new technique is proposed to avoiding deadlock situation between nodes based on snapshot algorithm. Due to the frequent changes in network topology and the lack of the network resources both in the wireless medium and in the mobile nodes, mobile ad hoc networking becomes a challenging task [1]. Effect of this technique a MANET is free from deadlock situation.

2. CLASSIFICATION OF ROUTING PROTOCOLS

A routing protocol is needed to send packets from source node to destination node. A routing protocol has to find a route for packet delivery and make the packet delivered to the correct destination [3]. Routing Protocols have been a QOS based Routing for Ad Hoc Mobile networks. Routing Protocols in Ad Hoc Networks can be categorized into two types:

2.1 Proactive Protocols

In Proactive or Table Driven routing protocols each node maintains one or more tables containing routing information to every other node in the network. All nodes keep on updating these tables to maintain latest view of the network. Some of the famous table driven or proactive protocols are: DBF [4], GSR [5].

2.2 Reactive Protocols

In Reactive or On Demand routing protocols, routes are created as and when required. When a transmission occurs from source to destination, it invokes the route discovery procedure. The route remains valid till destination is achieved or until the route is no longer needed. Some famous on demand routing protocols are: DSR [6], AODV [8].

3. SNAPSHOT ALGORITHM

The proposed scheme takes care of on detecting deadlock situation between source node and destination node based on snapshot algorithm. The Snapshot algorithm helps to MANET to detect deadlock based on maintaining state information of presenting nodes in the conversation.

The goal of this algorithm is to record a set of processes of nodes and channels states for a set of processes p, i=1, 2,…, N such that, even though the combination of recorded processes may have occurred at the same time, the recorded global state is consistent.

The assumptions of algorithm:

- Neither process of nodes nor channels fail.
- All channels are uni directional.
- FIFO based services are provided channels.
The graph of channels and processes are strongly connected. Any processes may initiate a global snapshot at any time.

The algorithm is defined two rules, the marker sending rule and the marker receiving rule.

**Algorithm**

**Marker receiving rule for process** $P_i$  
On $P_i$’s receipt of a marker message over channel $C$:  
if($P_i$ has not yet recorded its state)  
it records its process state now;  
records the state of $C$ as the empty set;  
turns on recording of messages arriving over other incoming channels;  
else  
$P_i$ records the state of $C$ as the set of messages it has received over $C$ since it saved its state.  
endif

**Marker sending rule for process** $P_i$  
After $P_i$ has recorded its state, for each outgoing channel $C$; $P_i$ sends one marker message over $C$ (before it sends any other messages over $C$).

The marker sending rule obligates processes to send a marker after they have recorded their state. The marker receiving rule obligates a process that has not recorded its state to do so. In that case, this is the first marker that it has received. It notes which messages subsequently arrive on the other incoming channels. When a process that has already saved its state receives a marker (on another channel), it records the state of that channel as the set of messages it received on it’s since it saved its state.

Any process may begin the algorithm at any time. It acts as though it has received a marker and follows the marker receiving rule. Thus it records its state and begins to record messages arriving over all its incoming channels, several processes may initiate recording concurrently in this way.

Illustration of the algorithm is for a system for a system of two process, p1 and p2 connected by two unidirectional channels, c1 and c2.

![Figure 1: Two processes and their initial states](image)

The two processes trade in ‘widgets’. Process $p1$ sends orders for widgets over $c2$ to $p2$, enclosing payment at the rate of $10 per widget. Sometime later, process $p2$ sends widgets along channel $c1$ to $p1$. Process $p2$ has already received an order for five widgets, which it will shortly dispatch to $p1$.

![Figure 2: The execution of the processes](image)

The above diagram shows an execution of the system while the state is recorded. Process $p1$ records its state in the actual global state $S0$, when the state of $p1$ is <$1000, 0$>. Following the marker...
sending rule, process $p_1$ then emits a marker message over its outgoing channel $c_2$ before it sends the next application-level message: (Order 10, $100), over channel $c_2$. The system enters actual global state $S_1$.

Before $p_2$ receives the marker, it emits an application message (five widgets) over $c_1$ in response to $p_1$’s previous order, yielding a new actual global state $S_2$.

Now process $p_1$ receives $p_2$’s message (five widgets), and $p_2$ receives the marker. Following the marker receiving rule, $p_2$ records its state as <$50, 1995> and that of channel $c_2$ as the empty sequence. Following the marker sending rule, it sends a marker message over $c_1$.

When process $p_1$ receives $p_2$’s marker message, it records the state of channel $c_1$ as the single message (five widgets) that it received after it first recorded its state. The final actual global state is $S_3$.

The final recorded state is $p_1$: <$1000, 0>; $p_2$: <$50, 1995>; $c_1$: <(five widgets)>; $c_2$: <>. Note that this state differs from all the global states through which the system actually passed.

Characterizing the observed state • The snapshot algorithm selects a cut from the history of the execution. The cut, and therefore the state recorded by this algorithm, is inconsistent. To see this, let $e_i$ and $e_j$ be events occurring at $p_i$ and $p_j$, respectively, such that $e_i \rightarrow e_j$. We assert that if $e_j$ is in the cut, then $e_i$ is in the cut. That is, if $e_j$ occurred before $p_j$ recorded its state, then $e_i$ must have occurred before $p_i$ recorded its state. This is obvious if the two processes are the same, so we shall assume that $j \neq i$. Assume, for the moment, the opposite of what we wish to prove: that $p_i$ recorded its state before $e_i$ occurred. Consider the sequence of $H$ messages $m_1, m_2, ..., m_H (H \geq 1)$, giving rise to the relation $e_i \rightarrow e_j$. By FIFO ordering over the channels that these messages traverse, and by the marker sending and receiving rules, a marker message would have reached $p_j$ ahead of each of $m_1, m_2, ..., m_H$. By the marker receiving rule, $p_j$ would therefore have recorded its state before the event $e_j$. This contradicts our assumption that $e_j$ is in the cut, and we are done.

We may further establish a reachability relation between the observed global state and the initial and final global states when the algorithm runs. Let $Sys = e_0, e_1, ..., e_i$ be the linearization of the system as it executed (where two events occurred at exactly the same time, we order them according to process identifiers). Let $S_{init}$ be the global state immediately before the first process recorded its state; let $S_{final}$ be the global state when the snapshot algorithm terminates, immediately after the last state-recording action; and let $S_{snap}$ be the recorded global state.
We shall find a permutation of Sys, Sys’=e_0’,e_1’,e_2’,… such that all three states Sinit, Ssnap and Sfinal occur in Sys’. Ssnap is reachable from Sinit in Sys’, and Sfinal is reachable from Ssnap in Sys’.

Figure 3: Reachability between states in the snapshot algorithm

The above diagram shows this situation, in which the upper linearization is Sys and the lower linearization is Sys’.

We derive Sys’ from Sys by first categorizing all events in Sys as pre-snap events or post-snap events. A pre-snap event at process pi is one that occurred at pi before it recorded its state; all other events are post-snap events. It is important to understand that a post-snap event may occur before a pre-snap event in Sys, if the events occur at different processes. (Of course, no post-snap event may occur before a pre-snap event at the same process.)

We shall show how we may order all pre-snap events before post-snap events to obtain Sys’. Suppose that ej is a post-snap event at one process, and ej+1 is a pre-snap event at a different process. It cannot be that ej → ej+1 for then these two events would be the sending and receiving of a message, respectively. A marker message would have to have preceded the message, making the receipt of the message a post-snap event, but by assumption ej+1 is a pre-snap event. We may therefore swap the two events without violating the happened-before relation (that is, the resultant sequence of events remains a linearization). The swap does not introduce new process states, since we do not alter the order in which events occur at any individual process.

We continue swapping pairs of adjacent events in this way as necessary until we have ordered all pre-snap events e_0’,e_1’,e_2’,… e’_{r-1} prior to all post-snap events e’_R, e’_R+1,e’_R+2, … with Sys’ the resulting execution. For each process, the set of events in e_0’,e_1’e_2’,...,e’_R–1 that occurred at it is exactly the set of events that it experienced before it recorded its state. Therefore the state of each process at that point, and the state of the communication channels, is that of the global state Ssnap recorded by the algorithm. We have disturbed neither of the state’s Sinit or Sfinal with which the linearization begins and ends. So we have established the reachability relationship.

Stability and the reachability of the observed state: The reachability property of the snapshot algorithm is useful for detecting stable predicates. In general, any non-stable predicate we establish as being True in the state Ssnap may or may not have been True in the actual execution whose global state we recorded. However, if a stable predicate is True in the state Ssnap then we may conclude that the predicate is True in the state Sfinal, since by definition a stable predicate that is True of a state S is also True of any state reachable from S. Similarly, if the predicate evaluates to False for Ssnap, then it must also be False for Sinit.

Global states

The examples of distributed garbage collection, deadlock detection, termination detection and debugging:

Distributed garbage collection: An object is considered to be garbage if there are no longer any references to it anywhere in the distributed system. The memory taken
upby that object can be reclaimed once it is known to be garbage. To check that an object is garbage, we must verify that there are no references to it anywhere in the system.

Distributed deadlock detection: A distributed deadlock occurs when each of a collection of processes waits for another process to send it a message, and where there is a cycle in the graph of this ‘waits-for’ relationship.

Distributed termination detection: The problem here is how to detect that a distributed algorithm has terminated. Detecting termination is a problem that sounds deceptively easy to solve: it seems at first only necessary to test whether each process has halted. To see that this is not so, consider a distributed algorithm executed by two processes $p_1$ and $p_2$, each of which may request values from the other. Instantaneously, we may find that a process is either active or passive — a passive process is not engaged in any activity of its own but is prepared to respond with a value requested by the other. Suppose we discover that $p_1$ is passive and that $p_2$ is passive. To see that we may not conclude that the algorithm has terminated, consider the following scenario: when we tested $p_1$ for passivity, a message was on its way from $p_2$, which became passive immediately after sending it. On receipt of the message, $p_1$ became active again — after we had found it to be passive. The algorithm had not terminated.

Distributed debugging: Distributed systems are complex to debug, and care needs to be taken in establishing what occurred during the execution. For example, suppose Smith has written an application in which each process $p_i$ contains a variable $x_i (i = 1, 2, \ldots, N)$. The variables change as the program executes, but they are required always to be within a value $\delta$ of one another. Unfortunately, there is a bug in the program, and Smith suspects that under certain circumstances $|x_i - x_j| > \delta$ for some $i$ and $j$, breaking her consistency constraints. Her problem is that this relationship must be evaluated for values of the variables that occur at the same time.

Each of the problems above has specific solutions tailored to it; but they all illustrate the need to observe a global state, and so motivate a general approach.

It is possible in principle to observe the succession of states of an individual process, but the question of how to ascertain a global state of the system — the state of the collection of processes — is much harder to address. The essential problem is the absence of global time. If all processes had perfectly synchronized clocks, then we could agree on a time at which each process would record its state — the result would be an actual global state of the system. From the collection of process states we could tell, for example, whether the processes were deadlocked. But we cannot achieve perfect clock synchronization, so this method is not available to us. So we might ask whether we can assemble a meaningful global state from local states recorded at different real times.

The answer is a qualified ‘yes’, but in order to see this we must first introduce some definitions.

Let us return to our general system of $N$ processes $p_i (i = 1, 2, \ldots, N)$, whose execution we wish to study. We said above that a series of events occurs at each process, and that we may characterize the execution of each process by its history:

$$\text{history}(p_i) = h_i = e_{i0}, e_{i1}, e_{i2}, \ldots$$
Similarly, we may consider any finite prefix of the process’s history:

\[ H^i_0 = \langle e^0_i, e^1_i, \ldots, e^k_i \rangle \]

Each event either is an internal action of the process (for example, the updating of one of its variables), or is the sending or receipt of a message over the communication channels that connect the processes.

In principle, we can record what occurred in \( \varphi \)'s execution. Each process can record the events that take place there, and the succession of states it passes through. We denote by \( s_i^k \) the state of process \( p_i \) immediately before the \( k \)th event occurs, so that \( s_i^0 \) is the initial state of \( p_i \). We noted in the examples above that the state of the communication channels is sometimes relevant. Rather than introducing a new type of state, we make the processes record the sending or receipt of all messages as part of their state. If we find that process \( p_i \) has recorded that it sent a message \( m \) to process \( p_j (i \neq j) \), then by examining whether \( p_j \) has received that message we can infer whether or not \( m \) is part of the state of the channel between \( p_i \) and \( p_j \).

We can also form the global history of \( \varphi \) as the union of the individual process histories:

\[ H = h_0 \cup h_1 \cup \ldots \cup h_{N-1} \]

Mathematically, we can take any set of states of the individual processes to form a global state \( S = (s_1, s_2, \ldots, s_N) \). But which global states are meaningful – that is, which process states could have occurred at the same time? A global state corresponds to initial prefixes of the individual process histories. A cut of the system’s execution is a subset of its global history that is a union of prefixes of process histories:

\[ C = h_1^{c_1} \cup h_2^{c_2} \cup \ldots \cup h_N^{c_N} \]

The state \( s_i \) in the global state \( S \) corresponding to the cut \( C \) is that of \( p_i \) immediately after the last event processed by \( p_i \) in the cut – \( e^i_c (i = 1, 2, \ldots, N) \). The set of events \( \{ e^i_c; i = 1, 2, \ldots, N \} \) is called the frontier of the cut [7].

4. PROPOSED METHOD

The routing protocols provide a route but not detect a deadlock situation. The snapshot algorithm detects a deadlock situation between source and destination.

The channels in Ad-Hoc networks will be added dynamically. Even though, the processes identified by “snapshot” algorithm will be forwarded through, by linking the channels dynamically.

The snapshot algorithm detects the deadlock situation in between nodes and the information maintained in separate file similar to log file. By verifying this by file one can assess and avoids the sending information to other node in the channel.

5. CONCLUSION

A new scheme has been proposes to detect deadlock situation between source and destination. This can be incorporated effectively in MANETs to improve low processing overhead and loop freedom.
6. REFERENCES


