ISSN: 2278 – 1323
International Journal of Advanced Research in Computer Engineering & Technology
Volume 1, Issue 3, May2012

Performance characterization of AODV protocol in MANET

Ruchi Rani
M.Tech Scholar
Department of Computer Science
Al-Falah School of engineering and Technology

Manisha Dawra
Lecturer
Department of Computer Science
Al-Falah School of engineering and Technology

***Abstract*— A mobile ad-hoc network (MANET) is a collection of mobile nodes which communicate over radio. These networks have an important advantage; they do not require any existing infrastructure or central administration. Therefore, mobile ad-hoc networks are suitable for temporary communication links. This flexibility, however, comes at a price: communication is difficult to organize due to frequent topology changes. In this paper we present a new on-demand routing algorithm for mobile, multi-hop ad-hoc networks. The algorithm is based on ant algorithms which are a class of swarm intelligence. Ant algorithms try to map the solution capability of ant colonies to mathematical and engineering problems. The main goal in the design of the algorithm was to reduce the overhead for routing. Furthermore, we compare the performance of AODV with varying the different parameters through simulation results in ns2 [1].**

***Index Terms*—AODV, MANET, ns2.**

I. INTRODUCTION

Mobile ad hoc networks (MANETs) is a self configuring network of mobile routers (and associated hosts) connected by wireless links. Communication must be set up and maintained on the fly over mostly by wireless links. Each node of a network can both route and forward data [2]. The exploding demand for computing and communication on the move has led to reliance for ad hoc networks. Although substantial attempts have been made on research towards design and development of ad hoc network parameters, there is relatively little understanding of their behavior.

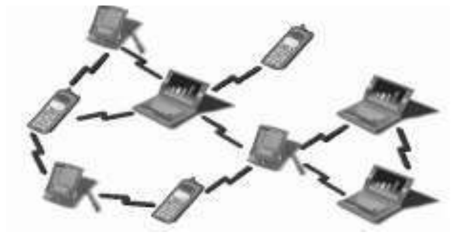


Fig1. Nodes of MANETS

So, in this paper on demand adhoc routing algorithm is used for the analysis of AODV protocol using different parameters in the environment of ns2.

II. Ad Hoc On-Demand Distance-Vector Protocol (AODV)

The Ad Hoc On-Demand Distance-Vector Protocol (AODV)[3] is a distance vector routing for mobile ad-hoc networks. AODV is an on-demand routing approach, i.e. there are no periodical exchanges of routing information.

A. AODV Route Discovery

When a node needs to determine a route to a destination node, it floods the network with a *Route Request (RREQ) message*. The originating node broadcasts a RREQ message to its neighboring nodes, which broadcast the message to their neighbors, and so on. To prevent cycles, each node remembers recently forwarded route requests in a route request buffer (see next section). As these requests spread through the network, intermediate nodes store reverse routes back to the originating node. Since an intermediate node could have many reverse routes, it always picks the route with the smallest hop count. When a node receiving the request either knows of a “fresh enough” route to the destination (see section on sequence numbers), or is itself the destination, the node generates a *Route Reply (RREP) message*, and sends this message along the reverse path back towards the originating node. As the RREP message passes through intermediate nodes, these nodes update their routing tables, so that in the future, messages can be routed though these nodes to the destination. Notice that it is possible for the RREQ originator to receive a RREP message from more than one node. In this case, the RREQ originator will update its routing table with the most “recent” routing information; that is, it uses the route with the greatest destination sequence number.

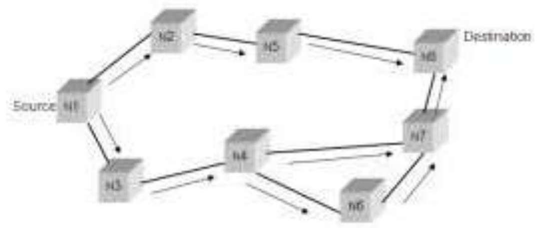


Fig2. . Route discovery of AODV

B. The Route Request Buffer

In the flooding protocol described above, when a node originates or forwards a route request message to its neighbors, the node will likely receive the same route request message back from its neighbors. To prevent nodes from resending the same RREQs (causing infinite cycles), each node maintains a *route request buffer*, which contains a list of recently broadcasted route requests. Before forwarding a RREQ message, a node always checks the buffer to make sure it has not already forwarded the request. RREQ messages are also stored in the buffer by a node that originates a RREP message. The purpose for this is so a node does not send multiple RREPs for duplicate RREQs that may have arrived from different paths. The exception is if the node receives a RREQ with a better route (i.e. smaller hop count), in which case a new RREP will be sent. Each entry in the route request buffer consists of a pair of values: the address of the node that originated the request, and a route request identification number (RREQ id). The RREQ id uniquely identifies a request originated by a given node. Therefore, the pair uniquely identifies a request across all nodes in the network. To prevent the route request buffers from growing indefinitely, each entry expires after a certain period of time, and then is removed. Furthermore, each node's buffer has a maximum size. If nodes are to be added beyond this maximum, then the oldest entries will be removed to make room.

C. Expanding Ring Search

The flooding protocol described above has a scalability problem, because whenever a node requests a route, it sends a message that passes through potentially every node in the network. When the network is small, this is not a major concern. However, when the network is large, this can be extremely wasteful, especially if the destination node is relatively close to the RREQ originator. Preferably, we would like to set the TTL value on the RREQ message to be just large enough so that the message reaches the destination, but no larger. However, it is difficult for a node to determine this optimal TTL without prior global knowledge of the network. To solve this problem, I have implemented an expanding ring search algorithm [4], which works as follows. When a node initiates a route request, it first broadcasts the RREQ message with a small TTL value (say, 1). If the originating node does not receive a RREP message within a certain period of time, it rebroadcasts the RREQ message with a larger TTL value (and also a new RREQ identifier to distinguish the new request from the old ones). The node continues to broadcast messages with increasing TTL and RREQ ID values until it receives a route reply. If the TTL values in the route request have reached a certain threshold, and still no RREP messages have been received, then the destination is assumed to be unreachable, and the messages queued for this destination are thrown out.

D. Sequence Numbers

Each destination (node) maintains a monotonically increasing sequence number, which serves as a logical time at that node. Also, every route entry includes a destination sequence number, which indicates the "time" at the destination node when the route was created. The protocol uses sequence numbers to ensure that nodes only update routes with "newer" ones. Doing so, we also ensure loop-freedom for all routes to a destination. All RREQ messages include the originator's sequence number, and its (latest known) destination sequence number. Nodes receiving the RREQ add/update routes to the originator with the originator sequence number, assuming this new number is greater than that of any existing entry. If the node receives an identical RREQ message via another path, the originator sequence numbers would be the same, so in this case, the node would pick the route with the smaller hop count. If a node receiving the RREQ message has a route to the desired destination, then we use sequence numbers to determine whether this route is "fresh enough" to use as a reply to the route request. To do this, we check if this node's destination sequence number is at least as great as the maximum destination sequence number of all nodes through which the RREQ message has passed. If this is the case, then we can roughly guess that this route is not terribly out-of-date, and we send a RREP back to the originator. As with RREQ messages, RREP messages also include destination sequence numbers. This is so nodes along the route path can update their routing table entries with the latest destination sequence number.

E. Link Monitoring & Route Maintenance

Each node keeps track of a *precursor list*, and an *outgoing list*. A precursor list is a set of nodes that route through the given node. The outgoing list is the set of next-hops that this node routes through. In networks where all routes are bi-directional, these lists are essentially the same. Each node periodically sends HELLO messages to its precursors. A node decides to send a HELLO message to a given precursor only if no message has been sent to that precursor recently. Correspondingly, each node expects to periodically receive messages (not limited to HELLO messages) from each of its outgoing nodes. If a node has received no messages from some outgoing node for an extended period of time, then that node is presumed to be no longer reachable. Whenever a node determines one of its next- hops to be unreachable, it removes all affected route entries, and generates a Route Error (RERR) message. This RERR message contains a list of all destinations that have become unreachable as a result of the broken link. The node sends the RERR to each of its precursors. These precursors update their routing tables, and in turn forward the RERR to their precursors, and so on. To prevent RERR message loops, a node only forwards a RERR message if at least one route has been removed.

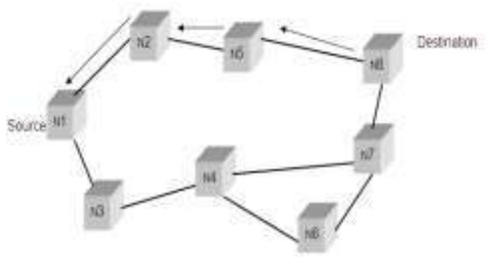


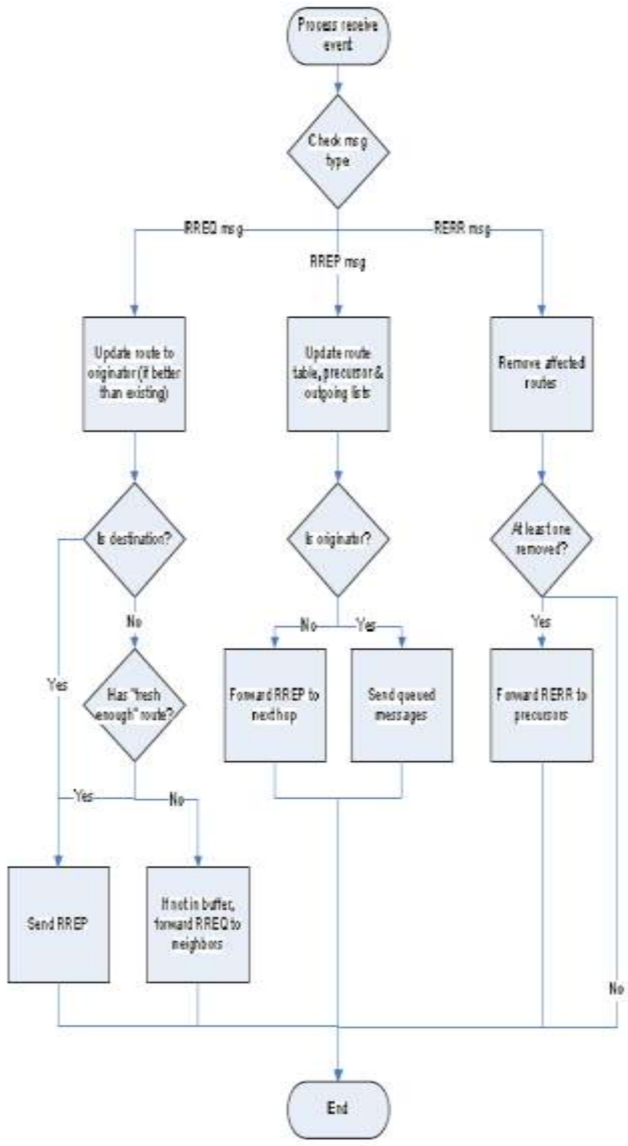
Fig3. Route maintenance of AODV

The following flow chart summarizes the action of an AODV node when processing an incoming message. HELLO messages are excluded from the diagram for brevity:

F. Code Explanation

State Variables and Data Structures:

- seqNum (int) – The node’s sequence number. This value is initialized to SEQUENCE_NUMBER_START and is incremented just before broadcasting a RREQ message.
- routeTable (RouteTable) – The routing table object. This structure stores route information in a HashMap, mapping NetAddress objects to RouteTableEntry objects. It contains methods for route addition/lookup/removal. It also contains methods for removing all routes though a given next hop, and for removing a list of route entries.



- RouteTableEntry – This class represents the route information for some destination. It includes: a next hop address (MacAddress), a destination sequence number, and a hop count.
- messageQueue (MessageQueue) – This message queue stores messages that are

waiting for routes. The messages are stored in a LinkedList object. The object has methods for sending queued messages, and removing messages (in case no route could be found).

- rreqList (LinkedList) – This structure contains a list of pending route requests (of type RouteRequest) originated by the node. Routes requests (represented as RouteRequest objects) are added to this list when the node initially requests a route. Requests are removed either when a RREP message is received, or when the RREQ with the maximum allowable TTL (TTL_THRESHOLD) times out.
- rreqBuffer (RreqBuffer) – The route request buffer object. This structure has a LinkedList of RreqBufferEntry objects, which keep track of recently sent RREQ messages so they do not get resent. It also contains methods for adding entries, and clearing expired entries. Entries expire after RREQ_BUFFER_EXPIRE_TIME. The clearExpireEntries() method gets called in the periodic timeout() event. The buffer has a maximum size of MAX_RREQ_BUFFER_SIZE.
- RreqBufferEntry – This class contains the RREQ ID and address of the node that originated the RREQ. It also contains the time (simulation time) that the message was sent.
- precursorSet (PrecursorSet) – This structure stores a list of the node's
  precursors, along with information for each precursor. This is stored as a HashMap, mapping the precursor's MacAddress to a PrecursorInfo object. The PrecursorInfo object contains the time that the message was last sent to the precursor. PrecursorSet includes a method for sending RERR messages to all precursors.

- outgoingSet (OutgoingSet) – This structure stores a list of outgoing nodes, along with a helloWaitCount for each outgoing node. helloWaitCount keeps track of the number of HELLO_INTERVALs that have passed since the last message was received from the outgoing node. If helloWaitCount exceeds a certain threshold specified by HELLO_ALLOWED_LOSS, then the outgoing node is considered unreachable.
- rreqIdSeqNum (int) – The sequence number for RREQ ID's. When sending a RREQ message, it assigns rreqIdSeqNum to the message's rreqId field, and then increments rreqIdSeqNum.

G. Core Methods

send (NetMessage) – This method, called by the network entity, attempts to send a message over the network. If routing information is available, it simply forwards the message to the appropriate next hop. Otherwise, the message is saved in the messageQueue and a route request is originated.

receive (…) – This method, called by the network entity, processes incoming

AODV messages. It checks the type of the message object and passes the message to the appropriate method:

receiveRouteRequestMessage() – Processes an incoming RREQ message. Updates routing tables, and then either sends a RREP message (by calling generateRouteReplyMessage(), or forwards the RREQ (by calling forwardRouteRequestMessage()).

receiveRouteReplyMessage() – Processes an incoming RREP message. Updates routing tables and precursor and outgoing lists. Then, if the node is the RREQ originator, it removes the pending route request, and sends the queued messages along the new route. If the node is not the RREQ originator, it forwards the RREP to the next hop.

receiveRouteErrorMessage() – Processes an incoming RERR message. Removes all affected routes. If at least one route removed, it calls precursorSet.sendRERR() to forward the RERR to all precursors.

receiveHelloMessage() – Processes an incoming HELLO message. This does nothing. (The peek() method takes care of the processing of HELLO messages).

peek () – This method is called by the network entity for every incoming packet (including non-AODV messages). If the last-hop of the incoming packet is in the outgoing set, the helloWaitCount for that outgoing node is reset (indicating that the node is still reachable).

timeout() – This method is an event that gets called every AODV_TIMEOUT for the duration of a simulation. It clears expired entries in the rreqBuffer and sends any HELLO messages that need to be sent. Then it updates the helloWaitCount counters for each outgoing node. If any of these helloWaitCount's have surpassed the HELLO_ALLOWED_LOSS, then routes are removed, and route error messages are sent.

RREQtimeout() – This timeout event gets scheduled for a future time whenever the node originates a RREQ message. When the timeout for a given route request occurs, if still no reply has been received (routeFound flag is false), then it sends another RREQ message with an increased TTL, and schedules another RREQtimeout(). This process continues until the routeFound flag has been set to true, or the TTL cannot be further increased (it is already at TTL_THRESHOLD).

sendIpMsg() – This method is used whenever a message needs to be sent over the network. This method sends the message using netEntity.send() after a brief, random delay. Additionally, if the next-hop node is a precursor, it renews the corresponding precursor entry with the current simulation time.

H. AODV Message Classes

There following four classes represent the different AODV messages. Each implements the jist.swans.misc.Message interface.

- RouteRequestMessage
- RouteReplyMessage
- RouteErrorMessage
- HelloMessage

Statistics

stats (AodvStats) – The stats object maintains global statistical information for a simulation. This object should be instantiated once by the simulation driver program, and each AODV node should contain a reference to this object. The reference can be set using the setStats() method.

Constants

The following constants can be set within the AODV code. Some of these can be used to tune AODV performance for different networks. All time durations are in simulation time.

DEBUG_MODE (Boolean) – If *true*, debugging statements are printed. Default is *false*.

HELLO_MESSAGES_ON (Boolean) – Activate/deactivate HELLO messages. Should always be *true*, except possibly for debugging. Default is *true*.

SEQUENCE_NUMBER_START (int) – Starting sequence number at each node. Default is *0*.

RREQ_ID_SEQUENCE_NUMBER (int) – Starting RREQ ID sequence number. Default is *0*.

RREQ_BUFFER_EXPIRE_TIME (long) – Maximum duration an entry may reside in the RREQ buffer before it may be removed. Default is *5 seconds*.

MAX_BUFFER_SIZE (int) – Strict maximum size of node's RREQ buffer. Default is *10*.

AODV_TIMEOUT (int) – Period of time between calls to timeout() event. Default is *30 seconds*.

HELLO_INTERVAL (long) – Duration of inactivity after which a HELLO message should be sent to precursor. Default is *30 seconds*.

HELLO_ALLOWED_LOSS (int) – Number of timeouts that must occur before determining an outgoing link unreachable. Default is *2*.

RREQ_TIMEOUT_BASE (long) – Constant term for RREQ timeout duration. Default is *1 second*.

RREQ_TIME_PER_TTL (long) – Variable term for RREQ timeout duration, which depends on the TTL value of the RREQ message. Defaut is *500 milliseconds (per TTL).*

III. RESULT

The important performance metrics which were evaluated are -:

Packet delivery ratio:---- The ratio of data packets deliver to the destination to those generated by cbr sources.

Normalized Routing Load:---- The number of routing packets transmitted per data packet delivered at the destination. Each hop wise transmission of a routing packet is counted as one transmission.

Bandwidth Utilization: ---- It is desirable that a routing protocol keeps this rate at a high level since efficient bandwidth utilization is important in wireless network where the available bandwidth is a limiting factor. This is an important metric because it reveals the loss rate seen by the transport protocol and also characterizes the completeness and correctness of routing protocol.

Node Misbehavior: ---- The percentage of packet lost in an adhoc network is called as node misbehavior.

We have evaluated the above parameters on the basis of varying mobility and varying node density.

Effect of Varying Node Density: ---- In our simulation, we have varied the numbers of nodes from 10 to 50 and evaluated the results by comparing those with the standard result for that variation. In case of packet delivery ratio as the number of nodes increases the packet delivery ratio increases. The presence of only 10 nodes present in the taken.

Simulation area is not sufficient to provide enough connectivity. This reflects in terms of poor packet delivery ratio with both protocol variants. But, as the number of nodes

Increased to 20 and above, the performance of AODV slightly improves packet delivery ratio.

As the number of nodes varies bandwidth utilization will increase. when the no. of nodes is below 20 then bandwidth utilization is less as compared to bandwidth utilization with 50 nodes.

As no. of nodes increases, the no. of misbehaving nodes are decreases and also normalize routing load will also decreases.

Effect of Varying Mobility: In our simulation, we have varied the speed of nodes from 0 to 20(m/sec) with keeping no. of nodes constant and evaluated the results by comparing those with the standard result for that variation.

In the presence of high mobility, link failure can happen very frequently. Link failures trigger new route discoveries in AODV since it has almost one route per destination in its routing table. Thus the frequently occurrences of route discoveries in AODV is directly proportional to the no. of route breaks.

So on varying the speed of nodes increases the packet delivery ratio will decreases because on increasing the speed the link between source and destination will break frequently and the no. of misbehaving nodes will increase because of link failure .we will now present the graphs generated in our simulation environment. These graphs were prepared by first simulating the MANET's in ns and nsnsm.

A. Graphs:

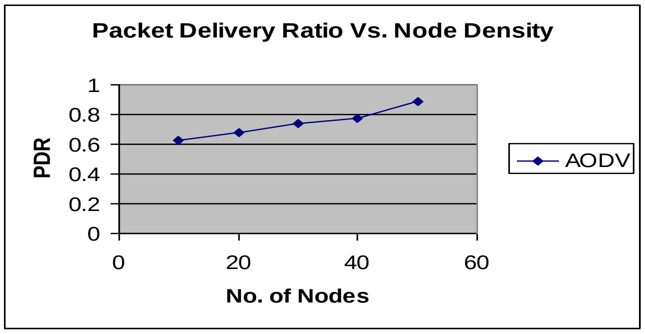


Fig 4 Packet Delivery Ratio Vs Node Density

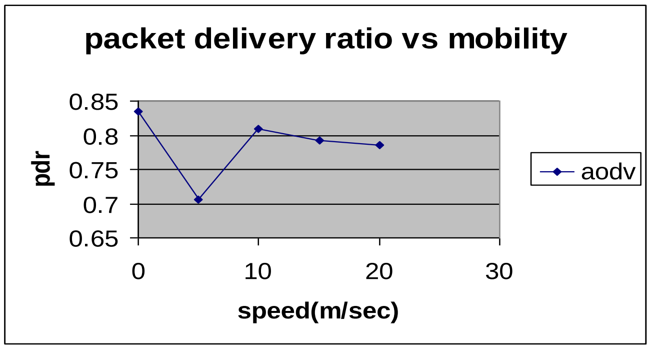


Fig 5 Packet Delivery ratio vs. Mobility

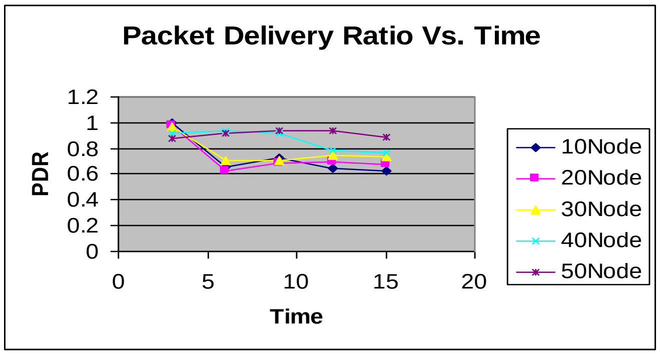


Fig 6 Packet Delivery Ratio vs. Time

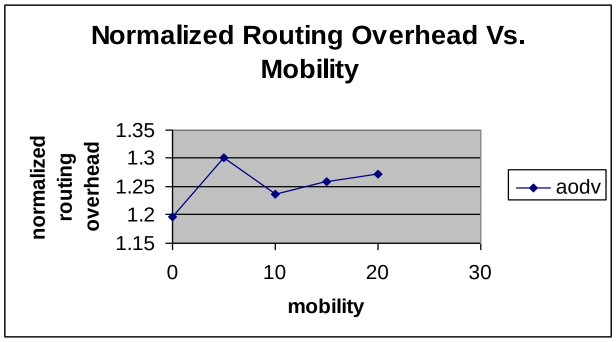


Fig 7 Normalized Routing overhead vs. Mobility

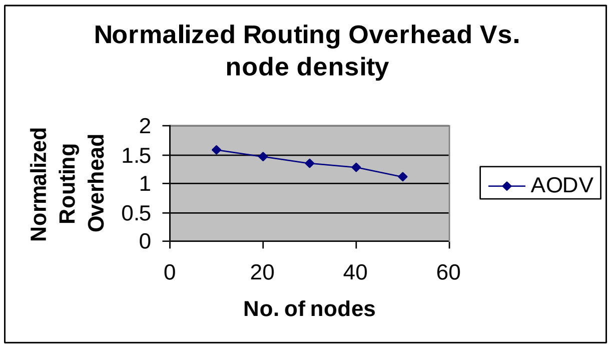


Fig 8 Normalized routing overhead vs. node density

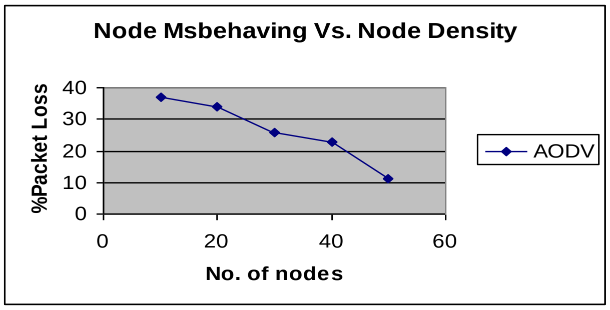


Fig 9 Node Misbehaving vs. Node density

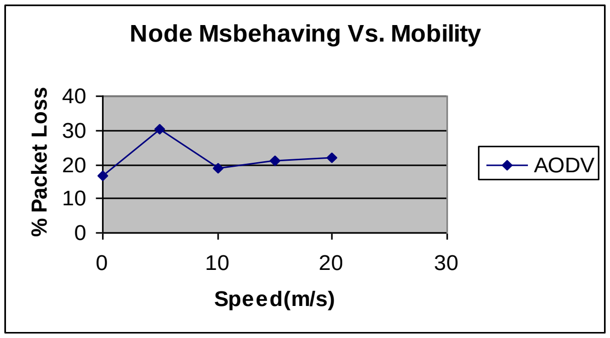


Fig 10 Node misbehaving vs. Mobility

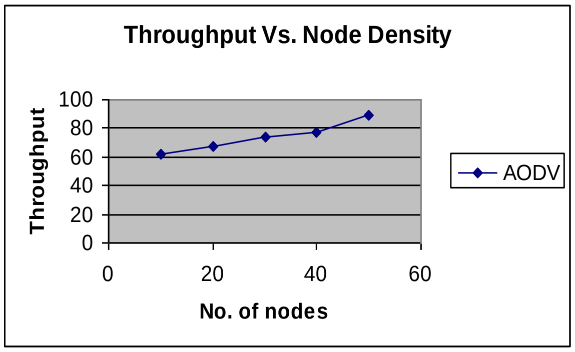


Fig 11 Throughput vs. node density

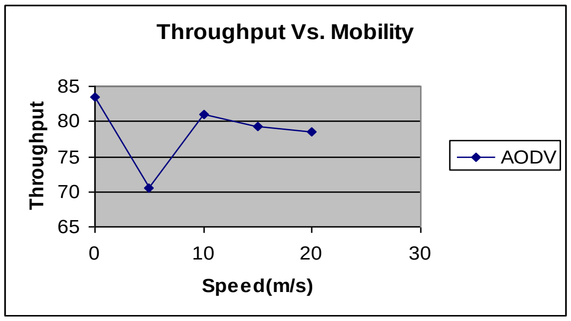


Fig 12 Throughput vs. Mobility

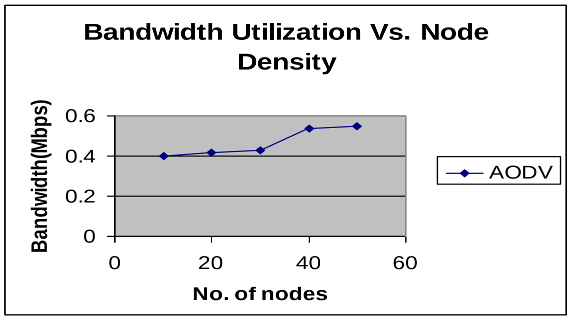


Fig 13 Bandwidth utilization vs. node density

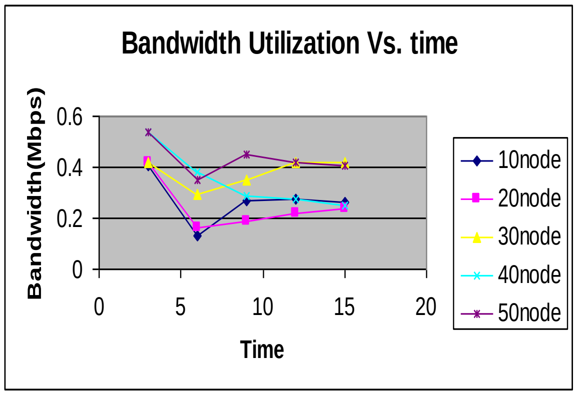


Fig 14 Bandwidth utilization vs. time

IV. CONCLUSION

AODV perform better results under low mobility and high node density. As we have seen in the graphs, the effects of different parameters with the variation of node density and mobility. Variation of node density: As the no. of nodes increases the no. of nodes behaving as intermediate nodes and the neighbor discovering time minimizes. This results in quicker path finding. So we obtain the better packet delivery ratio from source to destination. Simultaneously it will help in better bandwidth utilization, lesser node misbehaving, and lower normalize routing load. Variation of node mobility: As the speed of nodes increases, the link failure between the sources to destination occurs frequently. This will result in low packet delivery ratio, high normalized routing load, and high node misbehaving.

REFERENCES

[1]. ns-2 http://www.isi.edu/nsnam/ns.
[2]. Tope. M.A, McEachen, J.C, and Kinney. A.C, "Ad-hoc network routing using co-operative diversity", Advanced Information Networking and Applications, IEEE Conference, 18-20 April 2006.
[3]. Charles Perkins, Elizabeth Royer, and Samir Das. "Ad hoc on demand distance vector (AODV) routing". IETF RFC No. 3561, July 2003.
[4] Incheon Park, Jinguk Kim, Ida Pu." Blocking Expanding Ring Search Algorithm for Efficient Energy Consumption in Mobile Ad Hoc Networks".