AODV Routing Implementation for Scalable Wireless Ad-Hoc Network Simulation (SWANS)

Clifton Lin
cal36@cornell.edu

Introduction

The Ad-Hoc On-demand Distance Vector (AODV) routing protocol [2] is one of several published routing protocols for mobile ad-hoc networking. Wireless ad-hoc routing protocols such as AODV are currently an area of much research among the networking community. Thus, tools for simulating these protocols are very important.

For my project, I have implemented the AODV protocol as part of a scalable wireless ad hoc network simulation (SWANS). SWANS is built upon a novel Java-based simulation framework called JiST [1].

Since one of the goals is scalability, I have strived to make the code as efficient as possible. For example, I implemented an expanding ring search algorithm to limit the flood of RREQ messages. I also attempted to keep memory utilization low by doing things like capping buffer sizes, removing expired entries, and by reducing the number of events. Finally, I used efficient data structures, such as hash tables, to improve performance.

The code is correct, to the best of my knowledge. Throughout the development process, I have used simulation programs which I have written to test and harden the code.

In the remainder of this report, I describe my protocol design, discuss performance results, and give an overview of the code for future developers.

Protocol Design

My implementation of AODV is based on a recent draft of the AODV specification [2]. I have implemented all the essential functionality of AODV, including:

- RREQ and RREP messages (for route discovery)
- RERR messages, HELLO messages, and precursor lists (for route maintenance)
- Sequence numbers
- Hop counts
- Expanding ring search

Some functionality described in the specification has been omitted, such as Gratuitous RREP messages, RREP acknowledgements, and multicast support, because they are either not essential to the algorithm, or inapplicable given our network model.

The Basic Protocol

Each AODV router is essentially a state machine that processes incoming requests from the SWANS network entity. When the network entity needs to send a message to another node, it calls upon AODV to determine the next-hop.

Whenever an AODV router receives a request to send a message, it checks its *routing table* to see if a route exists. Each routing table entry consists of the following fields:

- Destination address
- Next hop address
- Destination sequence number
- Hop count

If a route exists, the router simply forwards the message to the next hop. Otherwise, it saves the message in a *message queue*, and then it initiates a route request to determine a route. The following flow chart illustrates this process:

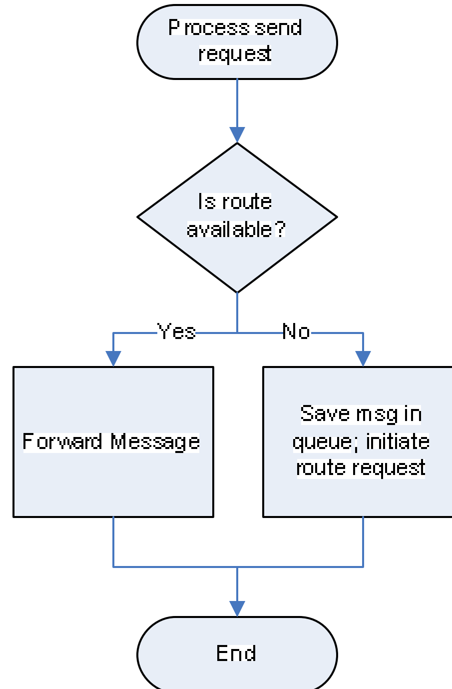


Figure 1

Upon receipt of the routing information, it updates its routing table and sends the queued message(s).

AODV nodes use four types of messages to communicate among each other. *Route Request (RREQ)* and *Route Reply (RREP)* messages are used for route discovery. *Route Error (RERR)* messages and *HELLO* messages are used for route maintenance. The following sections describe route determination and route maintenance in greater detail.

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. (See section on sequence numbers).

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.

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, 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.

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.

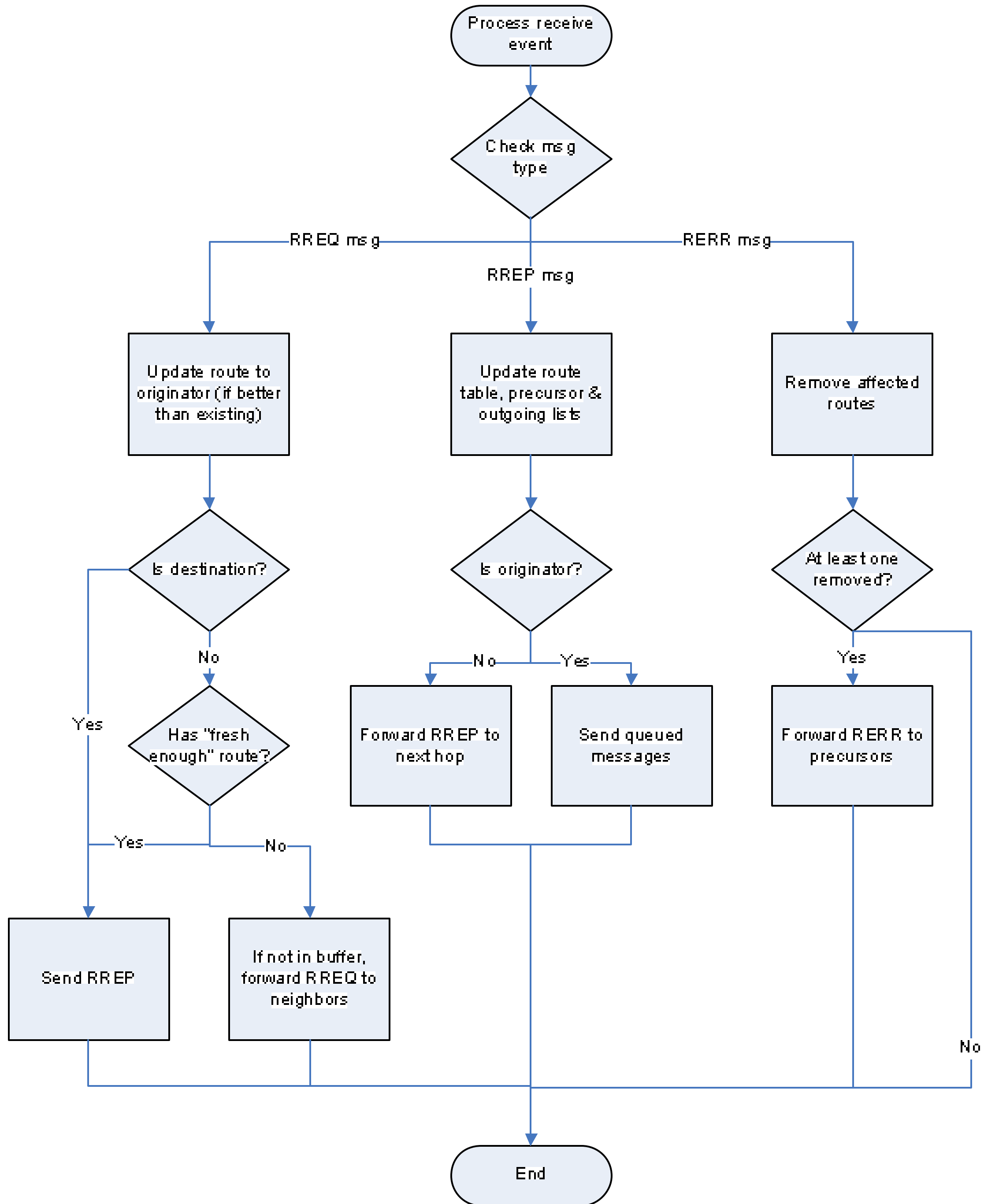
Link Monitoring & Route Maintenance

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.

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:



Performance Results - Scalability

One of the goals in simulating AODV is to determine how well it scales. How does the protocol performance vary with respect to the number of nodes in the network?

Attempting to answer this question, I conducted experiments measuring message activity, varying the number of nodes. I compute total message activity as the total number of AODV messages sent and received at each node. It is important to count both sent and received messages, as they will generally differ, for not all sent messages are received, while some messages are received many times (broadcasts). Additionally, I measured memory usage and elapsed time.

Varying the number of nodes can be accomplished in two basic ways. One is by varying field size, keeping node density constant. Another is by keeping the field size constant and increasing the density. I performed experiments using both these approaches.

In all my simulated experiments, each node sent messages to random destinations at an average rate of one message per minute. The nodes sent messages for ten minutes, and then statistics were recorded one minute afterwards.

Increasing Density in a Fixed Field

In this first experiment, I attempted to determine the effects of increasing the density of mobile nodes within a fixed area. I varied the number of nodes, from 4 to 1024 nodes, within a fixed field (3000x3000 meters). All message sending rates and durations were held constant. Also, to minimize randomness, the nodes were arranged in a grid, with equal spacing between nodes.

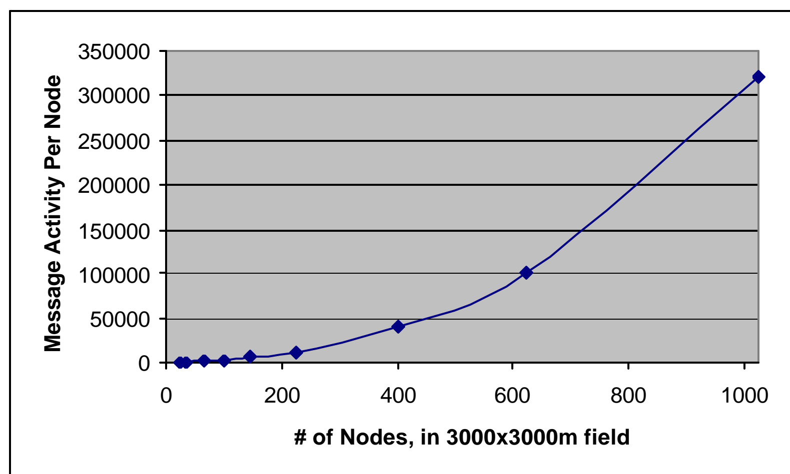


Figure 3

From Figure 3, we see that as the node density increases, the number of messages sent and received per node appears to increase quadratically. This can be explained by the

observation that when nodes broadcast RREQ messages, those messages are received by more nodes. As more nodes come close together, each node receives a greater number of RREQ messages, thus performing an increased amount of work.

Memory and Time Usage

Memory and time are critical resources that can limit scalability. In these experiments, I measure memory usage and elapsed time with respect to the number of nodes. For consistency, the simulations I ran for these experiments were identical to the ones I ran for the above experiment. Memory usage information was obtained from the Java Virtual Machine at the end of each simulation. The experiments were performed on a mosix-enabled cluster machine (Dell 1550) with dual 1.2 Ghz processors, 1 GB memory, using Java 2 v.1.4.2 on a RedHat 9 Linux kernel.

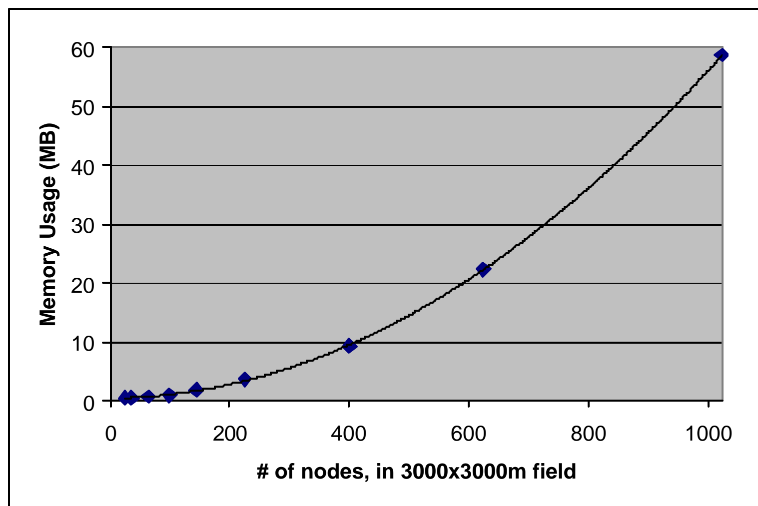


Figure 4. Memory usage grows quadratically. Best-fit curve: $m=.0561n^2+.593n+543.6$.

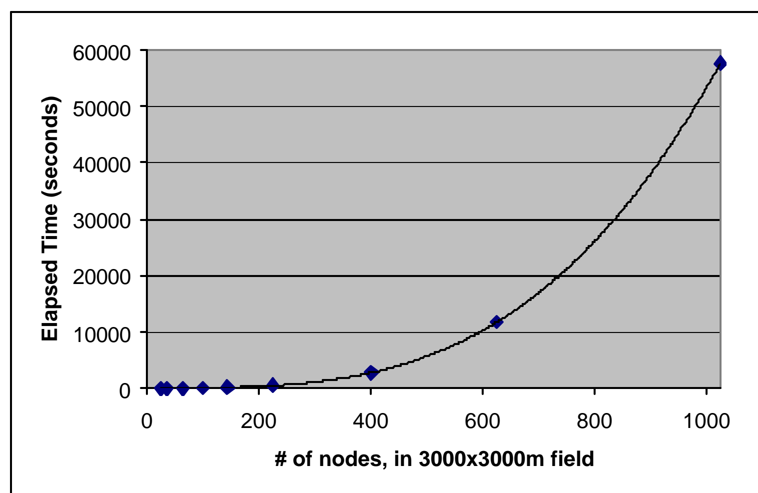


Figure 5. Elapsed time exhibits cubic growth. Best-fit curve: $t=6.6\times 10^{-5}n^3-.0154n^2+2.84n-99.2$.

In Figure 4, the best-fit curve was computed using quadratic regression. The equation of the curve was $m = .0561n^2 + .593n + 543.6$, where m represents memory usage in megabytes, and n represents the number of nodes. This regression curve gives us a model with which we can predict memory usage for any number of nodes. The model suggests a $O(n^2)$ relationship between memory usage and nodes. By this model, with one gigabyte of memory, we would expect to be able to run this simulation with about 4200 nodes.

While the *Memory Usage* plot in Figure 4 could be fit nicely with a quadratic regression curve, the *Elapsed Time* plot in Figure 5 was best-fit using cubic regression. This model suggests an $O(n^3)$ relationship between running time and number of nodes. Figure 5 shows that running this particular simulation with 1000 nodes takes roughly 58,000 seconds, or 16 hours. By this model, running the same simulation with twice as many nodes would require over 131 hours, or 5½ days!

Mobility effect in constant-density field

In this experiment, I compared message activity per node with and without mobility, in a constant-density field. Nodes were initially arranged in a grid format with each node separated by 625 meters, which is also the maximum range of the node radios. Nodes moved according to the *random walk* mobility model. Every minute, each node moved a random distance of up to 200 meters in a random direction.

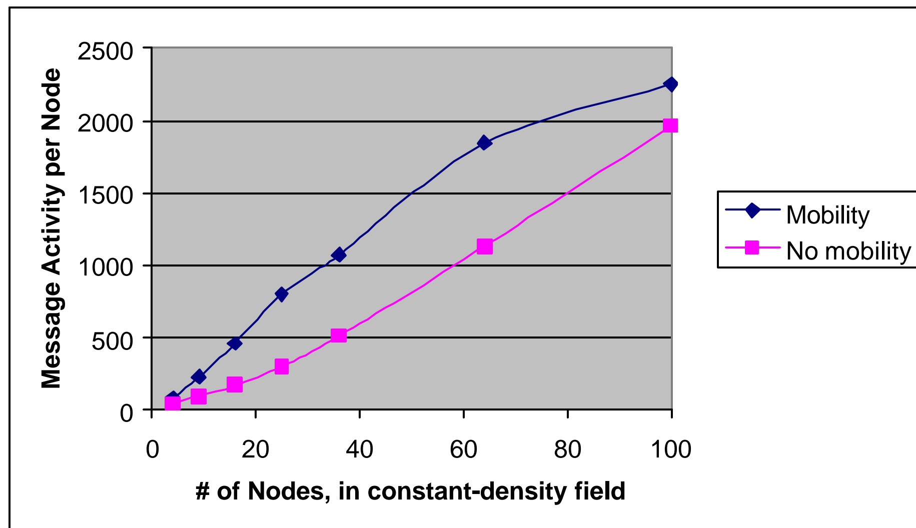


Figure 6

From Figure 6, we see that mobility causes an increase in message activity. With mobility, destinations can become unreachable, causing route error messages to be sent and routes to be removed. To create those routes again, new route requests need to be originated, resulting in the increased message activity.

Figure 7 shows a breakdown of the different AODV messages in the above experiment, with mobility.

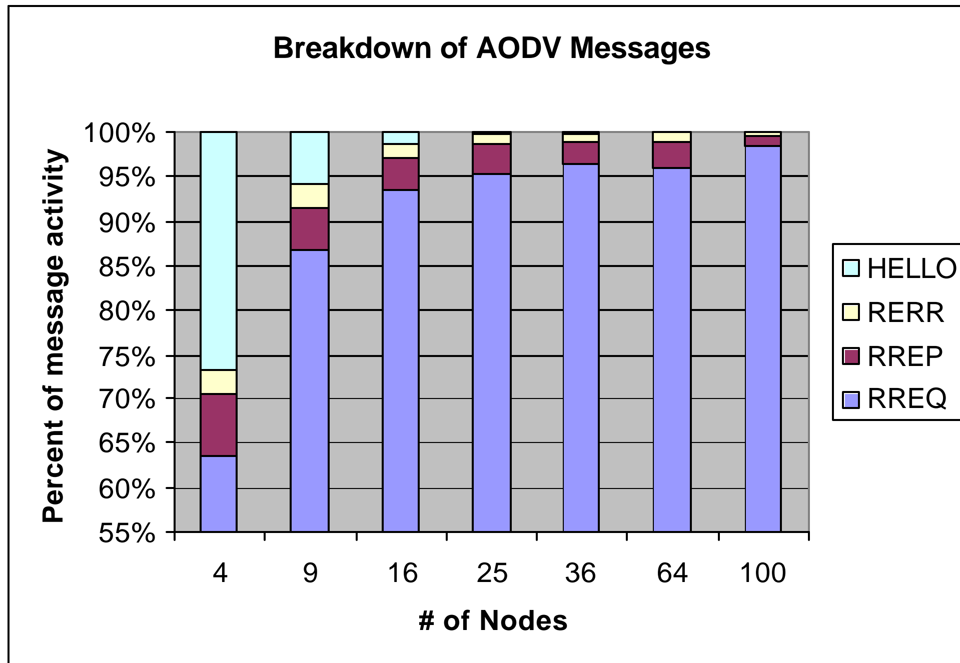


Figure 7

Figure 7 illustrates that RREQ messages make up the majority of the messages passed throughout the AODV network. Furthermore, the proportion of RREQ messages increases as the number of nodes increases. The reason there are proportionally many more RREQ messages is because they are flooded through the network upon each route request. In contrast, RREP, RERR, and HELLO messages are always sent to specific neighbors.

In some sense, RREQ messages are the "bottleneck" of the protocol. Reducing the number of RREQ messages would significantly improve the overall performance. The expanding ring search mechanism does help, on average, by limiting the spread of RREQ broadcasts; yet, RREQ messages are still the main limiting factor of performance.

Code Explanation

Files

The AODV code is part of the SWANS code base. The following is a list of files within SWANS that relate to AODV:

```
src/jist/swans/route/
      RouteAodv.java        - the bulk of the AODV code
      RouteInterface.java - contains the AODV routing interface
src/driver/
      aodvsim.java     - a simulation driver for running large-scale simulations
      aodvtest.java    - a simple simulation with just a few nodes
```

`RouteAodv.java` contains the bulk of the AODV code. All of the code explanations in the following sections refer to this file.

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.
    - o `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.

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.

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).*
- `TRANSMISSION_JITTER` (long) – The maximum delay before sending any packet.

Running Simulations:

The *aodvsim* driver program can be used to run large-scale AODV simulations. Via options, the user can specify input variables such as the number of nodes, field dimensions, node arrangement, mobility model, packet loss, send rate, and node activity timing. When the simulation is complete, the program outputs statistics of packet counts, memory usage, and elapsed time.

For usage help, type '`swans driver.aodvsim`' without any additional options. Figure 8 shows a sample execution.

```
% swans driver.aodvsim -n 25 -a grid:5x5 -f 3000x3000 -t 10,600,60 -s
1.0 -m static -l none
-------------
Packet stats:
-------------
Rreq packets sent = 1396
Rreq packets recv = 4554
Rrep packets sent = 318
Rrep packets recv = 317
Rerr packets sent = 0
Rerr packets recv = 0
Hello packets sent = 255
Hello packets recv = 254
Total aodv packets sent = 1969
Total aodv packets recv = 5125
Non-hello packets sent = 1714
Non-hello packets recv = 4871
--------------
Overall stats:
--------------
Messages to deliver = 250
Route requests      = 65
Route replies       = 86
Routes added        = 65

freemem:  1493248
maxmem:   839909376
totalmem: 2031616
used:     539112
start time  : Wed Apr 14 18:32:04 EDT 2004
end time    : Wed Apr 14 18:32:07 EDT 2004
elapsed time: 3239
```

Figure 8. Sample execution for simulation with 25 nodes arranged in a 5x5 grid format in a 3000x3000m field. After 10 seconds, nodes begin sending messages an average rate of 1.0 message per minute, for 600 seconds; the simulation ends one minute after nodes stop sending messages. There is no mobility or packet loss.

The driver program *aodvtest* is an example of a simple AODV simulation, with just three nodes and a single message. Throughout the development process, I used small simulations such as this to test correctness of the code. When running these small simulations, it is often useful to set the `RouteAodv.DEBUG_MODE` flag to `true` to see how AODV is operating internally.

References:

[1] R. Barr. JiST—Java in Simulation Time: User Guide and Tutorial. Sept. 2003.

[2] C. Perkins, E. Belding-Royer, S. Das. Ad hoc On-Demand Distance Vector (AODV) Routing. Feb. 2003. `http://www.ietf.org/internet-drafts/draft-ietf-manet-aodv-13.txt`.