Dynamic Source Routing for SWANS

Ben Viglietta
bsv3@cornell.edu

Introduction

I have implemented the Dynamic Source Routing Protocol for Mobile Ad Hoc Networks (DSR) for the SWANS network simulator, built upon Rimon Barr's Java in Simulation Time (JiST) simulation platform. This paper describes the design of the routing protocol, the methods I used to test it, and current and possible future optimizations of the protocol. This implementation of DSR was based upon the DSR draft specification found at http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-09.txt, and it is conformant to that specification, except in some minor details that will be noted later.

Design

Unlike some other routing protocols, DSR adds its own header to all packets sent through the network; for data packets this header usually contains the intended source route, among other things. DSR can therefore be thought of simply as another level in the network protocol stack, existing between the network layer and the link layer. Each DSR header contains one or more DSR options indicating the purpose of the packet: route request, route error, acknowledgement, and so on. There are three main tasks that DSR has to perform: routing a data packet through the network; route discovery, in which DSR determines a new route from one node to another; and route maintenance, in which DSR recognizes and corrects broken routes.

Route Cache

The central data structure for each node using DSR is the route cache, which contains the node's knowledge of the current network topology. The DSR specification suggests two possible implementations for the route cache. The first is the "path cache," in which each node keeps a list of all paths that it knows of between any two nodes, and the second is the "link cache," in which each node maintains a graph representing the current state of the network.

The route cache I have implemented is a variation of the path cache. There is no reason in the current DSR implementation for a node to keep track of routes in which it is not one of the endpoints, so each node just keeps a list of all known routes from itself to each possible destination. The cache is stored as a hash table, mapping a destination to a list of known routes to that destination. The benefits of this implementation are that it is very fast to look up a route or determine that none exists, that it is a smaller data structure than the straightforward path cache, and that it is easy to ensure that all routes are loop-free. The main drawback is that it is probably a larger data structure than the link cache, especially if the network is very dense. Also, it is not as flexible as a link cache: if a link cache learned of a route from A to B and a route from B to C, it would be able to deduce a route from A to C, while a path-based cache cannot.

It is possible for the route cache to contain more than one route to a given destination. In that case, shorter routes are preferred over longer routes, with ties broken arbitrarily.

Route Discovery

When a node wants to send a packet to a destination for which there is no route in the route cache, it begins the process of route discovery. A DSR packet containing a route request option is broadcast to all nodes within range. Each node receiving the route request proceeds to re-broadcast it to all nodes within range. If there is a route from the originator of the route request to the destination, then the destination will eventually see the route request and send a reply to the originator.

With all the broadcasting that occurs during route discovery, you have to be very careful to avoid bogging down the network with floods of route requests. To this end, each node maintains a data structure called a route request table. The route request table keeps track of all route requests that a node has recently seen. If a node receives a route request that exists in the route request table, it will not propagate it. This ensures that each node broadcasts a given route request no more than once. The route request table also keeps track of how frequently a given node has originated route requests for each destination, thus ensuring that route requests are originated at a reasonable rate. Figure 1 displays the logic used by a node upon receiving a route request.

When route discovery is in progress, any packets intended for a destination for which the source does not yet know a route are kept in a queue called the send buffer. Every time the source node receives a DSR message with a route reply option, it adds the new route to the route cache and sends any messages waiting in the send buffer that can use the new route.

Route Maintenance

Every time a node sends a packet using a DSR source route it is required to verify the reachability of the next hop on the route. To do this, it can add an acknowledgement request option to the DSR header. The next-hop node, upon receiving the packet, will send back a DSR message containing an acknowledgement option. If the sender does not receive such an acknowledgement within a specified time period, it retransmits the packet along with another acknowledgement request.

The DSR specification mandates a data structure called a maintenance buffer that holds the packets that are currently awaiting acknowledgement. However, no such data structure exists explicitly in my code: this is one area where I found the JiST architecture to be very useful. Instead of maintaining an explicit queue, for each packet sent I simply

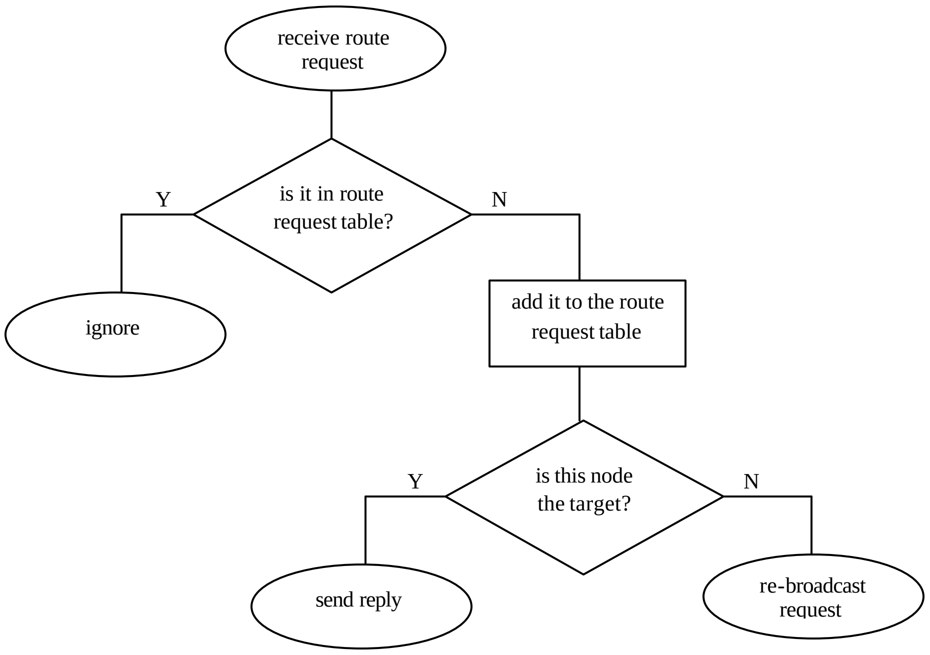


Figure 1 – Receipt of a route request

scheduled an event to occur after an appropriate timeout that would either retransmit or drop the packet, depending on whether an acknowledgement had been received at that time. In effect, the JiST event queue became my maintenance buffer, which I found simplified the route maintenance code significantly.

After a certain number of attempts to send the packet without any acknowledgement, the sender will consider the link broken. (The number of transmission attempts is currently three.) After discovering the broken link, the sender sends a DSR message to the originator of the message containing a route error option indicating the link in its route that was broken. The originator then removes from its route cache all paths that use the broken link. Unless the originator’s route cache contained multiple routes to some destinations, it will initiate a new route discovery the next time a packet must be sent to one of the destinations whose route relied on the broken link.

Sending a data packet

When a data packet is passed to DSR from the network layer, DSR strips off the IP header, encapsulates it in a DSR header with the appropriate source route taken from the route cache, and adds a new IP header. (If the route cache contains no route to the destination, the packet waits in the send buffer while route discovery takes place.) The packet is then sent along the given source route, with acknowledgements taking place at every hop along the way. Figure 2 displays the logic used by a node upon receiving a data packet.

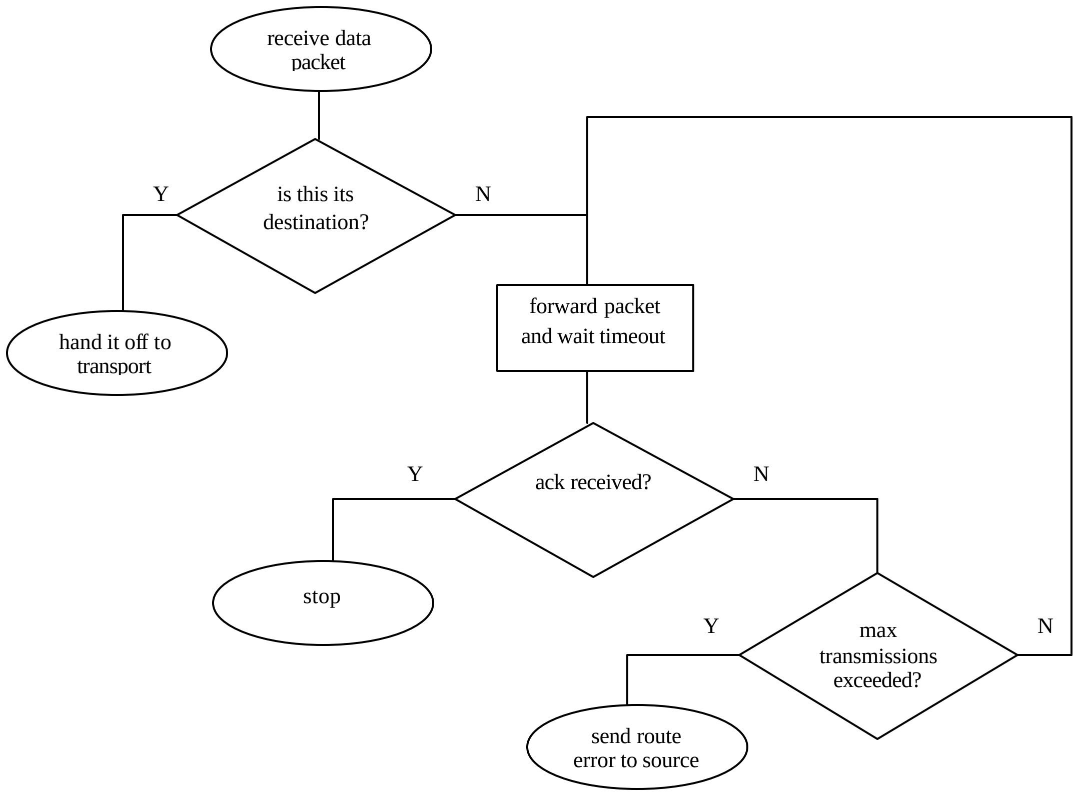


Figure 2 – Receipt of a data packet

Note that DSR does not provide any exactly-once semantics. If an acknowledgement is lost, it can cause a message to be retransmitted unnecessarily, and so the same message can be received more than once. Likewise, if a message is sent along a route that is broken, it will be dropped and therefore not received at all.

Testing

To test the performance of DSR, I built a simulation program called CBR (for Constant Bit Rate) that creates a two-dimensional field and places a number of nodes randomly inside it. Among the nodes are client-server pairs that transmit packets to each other once per second as the simulation progresses. The following parameters can be set:

- Routing protocol. Default is DSR.
- Dimensions of the field. Default is 1000x1000.
- Number of nodes. Default is 100.
- Number of client-server pairs. Default is 10.
- Number of transmissions. Default is 100.
- Packet loss probability. Default is 0.
- Node movement rate. Default is 0. Nodes move at discrete time intervals – the higher this value is, the more frequently they move. When the time comes for nodes to move, every node immediately moves to a random location in the field.

This is not a very realistic model of node movement, but it suffices to test DSR's route maintenance features.

This program was used to perform all the tests described in this section. (Note: All the tests described in this section were actually performed after applying the optimizations described in the next section.)

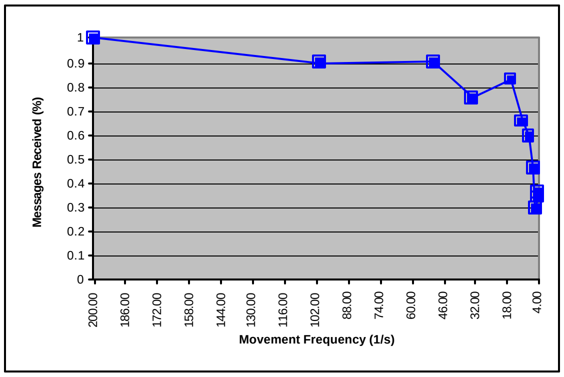
Testing correctness

I first tested DSR in a small field. Experimentation revealed that the range of a node in SWANS using default parameters for the field is around 700 meters. Therefore in a 300x300 field every node should be in range of every other node. Running CBR several times in such a field containing fifty nodes with five client-server pairs with 100 transmissions each and no packet loss or movement yielded an average total of about 515 out of 500 messages delivered. In theory, with no packet loss or node movement, exactly 500 out of 500 messages should be delivered; however, packets do get lost in SWANS even when you tell the network layer not to lose them. The culprit is the link layer, which drops packets if it is too busy to send them. DSR has this problem a lot because every packet is broadcast at the link layer. When packet loss is light, DSR actually ends up sending *more* than the appropriate number of packets, because lost acknowledgements cause packets to be retransmitted unnecessarily. Therefore 515 out of 500 messages delivered indicates that DSR is working correctly for small fields.

I then tried a larger field that would require longer source routes, still without adding any packet loss or node movement. In a 3000x3000 field with 200 nodes and five client-server pairs with 100 transmissions each, the average number of messages received was almost exactly 500. The average route length was about 2.5, so this indicates that DSR is able to find and use non-trivial routes as well.

If you go to more client-server pairs, the number of messages received tends to get worse as more messages get lost due to conflicts as the link layer. This is a regrettable side effect of the fact that DSR always broadcasts at the link layer, but there is not a lot that can be done in DSR to fix this problem, besides increasing the number of retransmissions.

Adding packet loss to the simulation doesn't really test anything new, since the simulations already have packet loss even without specifying it. Adding node movement, on the other hand, is a significant test, since DSR's ability to identify and correct broken routes is essential functionality. Figure 3 on the next page shows a graph of movement rate versus the percentage of messages successfully received for a simulation as before (3000x3000 field, 200 nodes, four client-server pairs, 100 transmissions each). Whenever nodes are moving we can expect some packets to be lost, since a packet sent using a broken source route will always be lost. Therefore the drop from 100% message receipt on the left side of the graph to 90% in the middle is to be expected. Somewhere around one movement every 15 seconds there is a precipitous drop in message receipt; presumably this is the point at which the nodes are moving too fast for DSR to catch up.



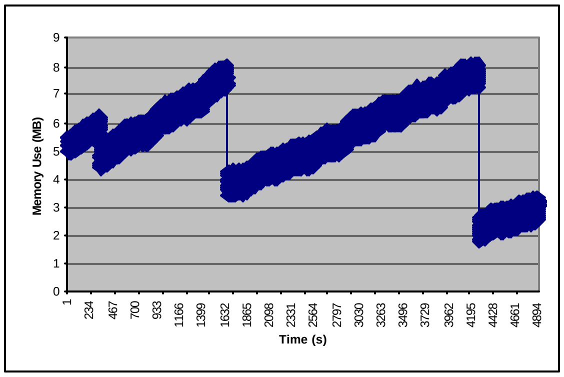


Figure 3 – Modeling node movement

Figure 4 – Measuring memory usage

(Recall that each time the nodes move, the entire field is shuffled randomly, which can be a very drastic change to the network topology. In real life node movement would be more subtle, and repairing broken links would undoubtedly be faster.) From this graph we can determine that DSR does in fact identify and correct broken routes, as long as the nodes are not moving too rapidly.

Testing memory consumption

The tests in the preceding section indicate that DSR works, even to a reasonable degree in the face of packet loss and node movement, but we must also consider resource costs. The first priority is to ensure that DSR does not have any memory leaks. To that end, I ran a simulation with several hundred nodes, a few of them transmitting continuously at a rate of once per second, for 500 seconds. The resulting memory use is plotted in Figure 4. The steep drops in memory use in that graph are no doubt points where Java's garbage collector kicked in. Taking the garbage collection into account, we can see that overall memory use is not increasing with time; if anything it appears to be decreasing. I'm at a loss to explain that – maybe it has to do with some part of JiST that I'm not familiar with. At any rate, it's not increasing, which is what I intended to show.

The next step is to determine the amount of memory actually used by each node. To calculate this, I ran simulations using various numbers of nodes and measured the average memory use over a lengthy period. I did this both for DSR and for a baseline routing protocol all of whose functions were no-ops. For the DSR tests there was a single node transmitting continuously and all other nodes were in range of it, and for the baseline tests of course nothing no traffic was occurring. The results are shown in Figure 5 on the next page.

Both graphs are ultimately linear, as expected. By calculating the slope of the line we can determine the amount of memory used by a node in each test: for the baseline, it is 0.95 KB, and for DSR it is 3.22 KB. Taking the difference indicates that DSR by itself is using about 2.27 KB for a single node with light traffic. That strikes me as a relatively large number, but given the number of different data structures DSR has to maintain,

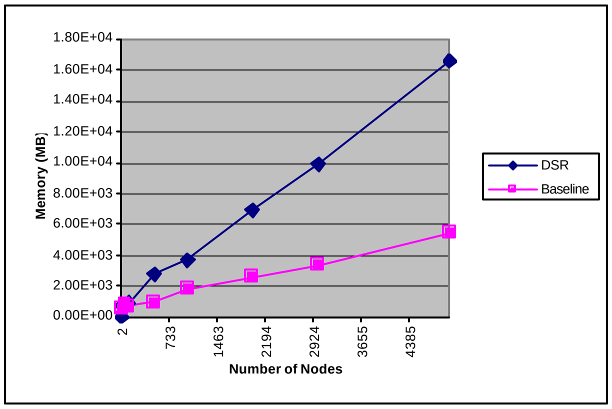


Figure 5 – Memory used per node

including route cache, send buffer, route request table, maintenance buffer, and gratuitous route reply table, it may not be. It's hard for me to evaluate this number by itself without any other implementations of routing algorithms to compare it to. When other routing algorithms become available, the memory usage of DSR can be re-evaluated to determine if it needs to be improved.

Optimizations

There are many potential optimizations that can be applied to DSR. The specification itself describes many features that can improve performance but are not strictly required. The testing as described above revealed several areas that seemed fruitful for optimization. The two main optimization features that I ended up implementing were automatic route shortening and passive acknowledgements.

Automatic route shortening

I found in my simulations that nodes were very frequently using routes that were longer than necessary. For example, in a simulation of 100 nodes all of which were in range of each other, the average route length among 1000 messages was 1.6, meaning that more than half the time nodes were using a route other than the direct hop from source to destination. I implemented automatic route shortening in an attempt to ameliorate this situation.

If a packet is being sent along a sub-optimal route, it may happen that a node on the route receives the packet before it is received by some of the nodes preceding it on the intended source route – this means that those nodes preceding it on the source route are superfluous and can safely be removed from the route. In this situation, the node receiving the packet sends a gratuitous route reply to the originator of the message informing the originator of the shorter available route.

It is important to limit the rate at which gratuitous route replies are sent to avoid flooding the originator with them. Therefore a data structure called a gratuitous route reply table is maintained in order to keep track of which nodes have recently been sent gratuitous route replies. A gratuitous route reply will be sent to the originator of a packet only if the originator does not exist in the gratuitous route reply table. The entries in the table are removed after a certain timeout, thus limiting the rate of replies to a given originator.

The automatic route shortening seems to be effective at reducing route length. After implementing automatic route shortening, the aforementioned test of 1000 messages sent among 100 nodes all within range of each other now had an average route length of 1.07. In a larger simulation with 2000 messages among 100 nodes not all in range of each other, and with the nodes moving at a rate of once every 20 seconds, the average route length dropped from 2.27 to 1.49, so the improvement seems to be appreciable.

Passive acknowledgements

Another issue I discovered while analyzing DSR's performance was that there was a very high ratio of administrative packets to data packets. This was due largely to the fact that every data packet sent required a corresponding acknowledgement to be returned, ensuring that the ratio was at least one to one. In fact, after factoring in route requests, route replies, and route errors, the ratio was often much higher than one to one: in experiments featuring substantial node movement or packet loss I observed ratios of up to eight or nine administrative packets for every one data packet.

Passive acknowledgements were an attempt to address this problem. Instead of requesting an explicit network-level acknowledgement, after forwarding a packet along its route, a node would wait to overhear the same packet being forwarded to its next hop. This would be taken as an indication that the packet had been received, and it would therefore not be retransmitted. Passive acknowledgements are completely free compared to network-level acknowledgements: if the passive acknowledgement is overheard, no extra messages need to be sent besides the data packet itself.

If no passive acknowledgement is overheard within a specified timeout, the message will be retransmitted, this time using the standard network-level acknowledgement request. If no acknowledgement is received after the normal number of retransmissions, then a route error is sent as usual to the originator of the message.

Obviously passive acknowledgements cannot be used on the last hop of the route, since the packet will not be retransmitted by the final destination node. Therefore they are of little benefit for short routes. However, for longer routes there is significant benefit: in a 500-node simulation transmitting a total of 3000 packets with an average route length around 5.5 and substantial node movement, the ratio of administrative packets to data packets dropped from 8.25:1 to 1.35:1. Since there is no cost to using passive acknowledgements, this is therefore a clear win.

Future improvements

There are many different ways that the current implementation of DSR could be improved. Some of them are:

- Currently when a packet is sent along a route that contains a broken link, that packet will be lost when it arrives at the broken link. It should be possible to salvage such a packet if the node just before the broken link has in its route cache an alternate route to the destination.
- Currently a route request can be answered only by the node being sought. However, it should be possible for nodes to respond to route requests using information from their route caches. If this were done, care would have to be taken to avoid flooding the originator of the request with replies, but the DSR specification lays out a possible method for handling this situation.
- The DSR specification describes an optional "flow state" extension to the protocol that allows data packets to be sent without explicit DSR headers. This can supposedly greatly reduce the overhead of DSR. This optimization would require a significant amount of work.

Specification conformance

As mentioned earlier, there are a couple of areas in which my implementation of DSR does not conform to the published specification. The first concerns link layer protocols that sometimes have unidirectional links. The DSR specification requires elaborate logic and data structures for handling the possibility of network links that only work in one direction. Since unidirectional links are not an immediately important feature, I have completely neglected this issue in implementing DSR.

The other area of nonconformance concerns networks that use DSR to route among some of the nodes and other routing protocols to route among other nodes – for example, a wireless network in which one of the nodes is also connected to the Internet. The logic for handling these kinds of cases is not very complicated, but this seemed like a relatively unimportant feature for simulation purposes, so I ignored it as well.

To the best of my knowledge, the DSR implementation conforms to the requirements of the specification in every other respect.

Code overview

The following section provides a more detailed description of the data structures and functions used to implement DSR. Nearly all of the DSR data structures and logic are contained in the file `jist/swans/route/RouteDsr.java`. The classes describing the layout of DSR headers and options are in `jist/swans/route/RouteDsrMsg.java`.

The code implementing the CBR test program can be found in `driver/CBR.java`. It is not described in any further depth here, since it is simple enough that the code should be self-explanatory.

Data Structures

The route cache has already been extensively discussed. It is a `Hashtable` mapping IP addresses of destinations to `LinkedLists` of routes. Each route is an array of IP addresses giving one possible path from this node to the destination. The `LinkedLists` are kept sorted in order of increasing route length. Figure 6 on the next page is a conceptual image of one entry in the route cache.

The send buffer is a `LinkedList` of `BufferedPackets`. Each entry contains a packet waiting to be sent and the time at which it was inserted into the buffer. Entries in the send buffer are evicted after a timeout of `SEND_BUFFER_TIMEOUT`.

The route request table serves two purposes: it limits the rate at which this node sends out route requests, and it limits the rate at which this node replies to route requests. It is also a `Hashtable` mapping destination IP addresses to table entries. Each table entry contains the following information used to limit the rate at which requests are made: the TTL of the last request sent to this destination, the time at which the last request was sent, the number of requests for this destination sent since we last received a reply from it, and the timeout before another route request can be made. Table entries also contain a `LinkedList` of integer values indicating the identification values of the most recently seen route requests coming from this destination; this is used to limit the rate of sending route replies to this destination.

There is, as discussed before, no explicitly maintained maintenance buffer for packets that use network-level acknowledgements. There is, however, a maintenance buffer for packets awaiting passive acknowledgements. It is a `Hashtable` whose keys are structures containing the following information: the source and destination addresses of the buffered packet, the network protocol number, and the IP identification and fragmentation offset fields. Each entry in the buffer hashes to a number which is the value of the Segments Left field in the DSR header at the time the packet was buffered. All of these data are necessary to determine if an overheard packet can be taken as a passive acknowledgement of a previously transmitted packet.

The gratuitous route reply table is a `HashSet` containing pairs of IP addresses: these are the originator and last-hop IP addresses of packets that have recently triggered gratuitous route replies. Entries in the gratuitous route reply are evicted after a timeout of `GRAT_REPLY_HOLDOFF`.

Also, a small `HashSet` is maintained that holds the values of the outstanding network-level acknowledgements.

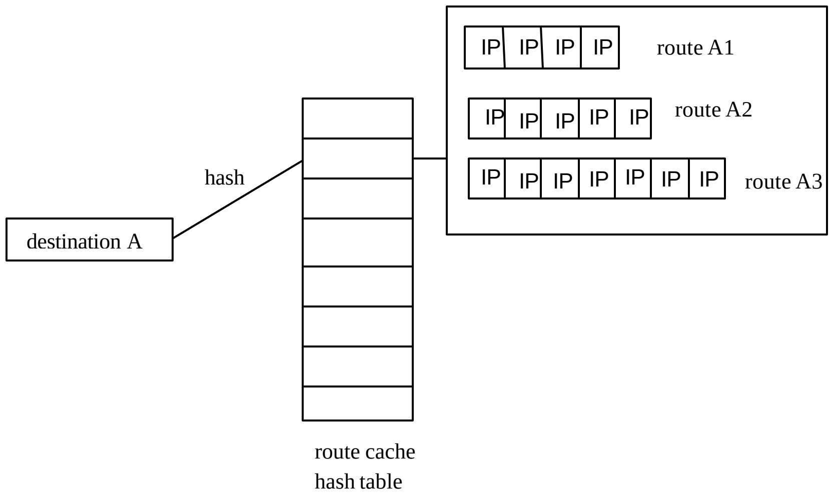


Figure 6 – Route cache

Control Flow

When a packet is overheard by DSR through the `peek` method, it is passed to a function called `ProcessOptions` that handles each of the options in the DSR header in turn. Every option is handled more or less independently of all the other options, and for each option `ProcessOptions` hands off control to one or two methods that handle specific option types. The overall control flow is summarized by the graph in Figure 7.

Route Request

Route requests are handed off to the `HandleRequest` method. `HandleRequest` first checks the route request table to see if this request has been seen recently. If it has, it is ignored; if it has not, its ID number is entered into the route request table.

Then `HandleRequest` examines the route request option to see if this is the node the request is seeking. If it is, the `SendRouteReply` method is called to send a route reply to the originator of the request. If it is not, the `ForwardRequest` method re-broadcasts the request to every node within range.

Route Reply

Route replies are handed off to the `HandleReply` method. This method simply adds the new route information into the route cache via the `InsertRouteCache` method. After updating the route cache, `InsertRouteCache` checks the send buffer to see if any waiting packets can be sent; if so, they are sent.

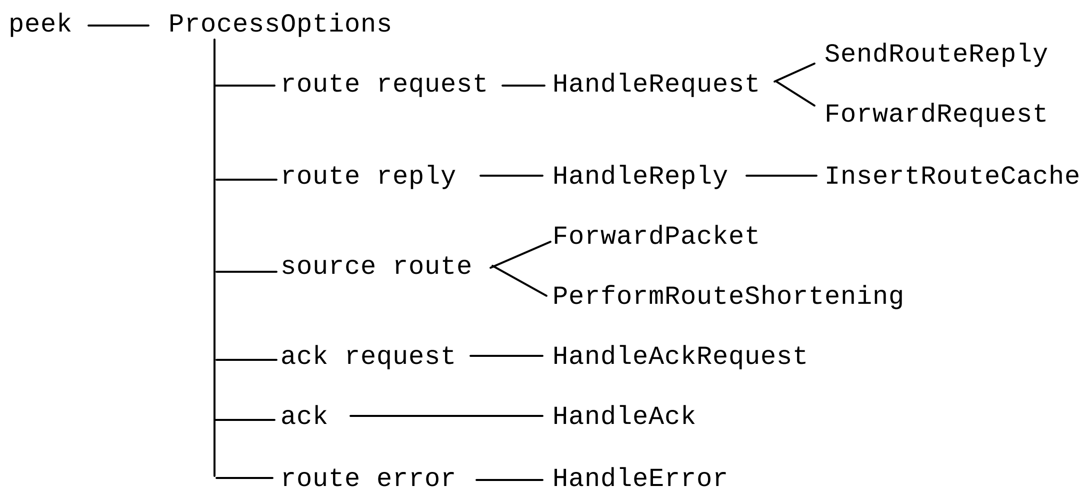


Figure 7 – Control Flow

Source Route

Given a source route option, `ProcessOptions` checks to see if this node is the intended next-hop destination of this packet. If it is, then the `ForwardPacket` method is called to forward the packet on to its next hop. Acknowledgements and retransmission work as described above.

If this node is not the next-hop destination of the packet, `PerformRouteShortening` is called to see if this node occurs later in the source route. If it does, `PerformRouteShortening` calls `SendGratuitousRouteReply` to send a gratuitous route reply, and the gratuitous route reply table is updated accordingly.

Acknowledgement Request

Acknowledgement requests are handed off to the `HandleAckRequest` method. This method checks the packet's source route to determine if this node is the intended next-hop destination of the packet. If it is, an acknowledgement is returned to the previous-hop node.

Acknowledgement

Acknowledgements are handed off to the `HandleAck` method. If this node is determined to be the destination of the acknowledgement, then the acknowledgement's ID value is removed from the table of outstanding acks.

Route Error

Route errors are handled by the `HandleError` method. If the error is of type `NODE_UNREACHABLE` (the only error that is currently supported), then the route cache is updated to remove the link that apparently no longer works.

The only significant bit of logic not described above is the `Transmit` method. This is the method that is called any time a packet has to be sent with acknowledgements and retransmission: data packets, route replies, route errors, etc. The `Transmit` method calls either `TransmitWithNetworkAck` or `TransmitWithPassiveAck`, as appropriate, and they function according to the acknowledgement rules discussed previously.

Constants

The class `RouteDsr` contains various constants that can be tweaked to modify DSR's behavior. They are as follows:

`BROADCAST_JITTER` – Small amounts of jitter are added to packet retransmission timeouts. This constant is the maximum jitter that can be added.
`SEND_BUFFER_TIMEOUT` – the maximum amount of time a packet will be in the send buffer before being dropped.
`REQUEST_PERIOD` – the initial timeout before retransmitting an unanswered route request.
`MAX_REQUEST_PERIOD` – the maximum timeout ever used for retransmitting an unanswered route request.
`MAX_MAINT_REXMT` – the maximum number of times a packet will be retransmitted using network-level acknowledgements.
`MAINT_PERIOD` – the timeout before retransmitting a packet using network-level acknowledgements.
`GRAT_REPLY_HOLDOFF` – the minimum time between sending gratuitous route replies to a given node.
`PASSIVE_ACK_TIMEOUT` – the timeout before retransmitting a packet using passive acknowledgements.
`TRY_PASSIVE_ACKS` – the maximum number of times to try using passive acknowledgements before switching to network-level acknowledgements.
`MAX_REQUEST_TABLE_IDS` – the maximum size of a single entry in the route request table.

Using the simulation platform

My impressions as a user of JiST and SWANS may be of interest to the people who created and maintain them, so I will say a few words about them.

JiST

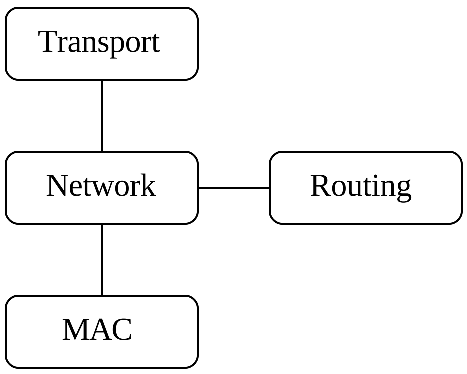
JiST is largely transparent as a simulation platform, which is to say that 95% of the code I wrote for JiST would have been exactly the same if I had been writing it for real. This makes it in general very easy to use. The only difficulty comes when trying to do things that are time-dependent, such as creating delays or timers. JiST's timing functions are, for me at least, a little counter-intuitive and awkward, but after some experience using them I became fairly comfortable using them.

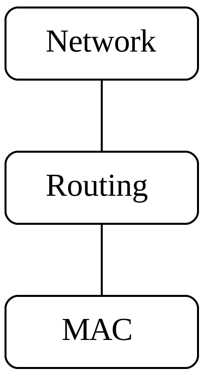
I will mention one enduring source of frustration for me, which is that using a debugger on JiST is made very difficult by the class rewriting that goes on. You cannot step into or out of an entity invocation as a normal function, nor can you obtain a meaningful stack trace within an entity method. For the same reason, any exceptions thrown within an entity method don't get propagated up to the function that called it. I suppose, though, that these effects are unavoidable given JiST's architecture.

SWANS

I only dealt with a small portion of the entire SWANS infrastructure, so I can only comment on the parts I used, namely the network, routing, and MAC layers. I never had any problem understanding or using any of the interfaces, barring a known bug I encountered in the 802.11 implementation, so overall SWANS performed admirably for me.

There is, I should mention, sort of a conceptual mismatch between how SWANS anticipates a routing protocol should work and how DSR actually works. SWANS seems to expect the routing layer to be sort of an adjunct to the network layer, as in the diagram on the left, whereas DSR is really a protocol layer unto itself, as on the right.





Furthermore, SWANS expects the routing protocol to use MAC addresses for its addressing scheme, whereas DSR uses IP addresses. Neither of these issues, though, is really critical, since I was still able to implement DSR without any serious problems.

Another issue with JiST/SWANS is the representation of network messages as Timeless objects. Messages are not really "timeless" in DSR, since they change at every hop: decrementing TTL, modifying source routes, etc. Because messages are Timeless objects they can't be modified, so this requires a lot of copying of message objects in order to make small changes to them. I haven't done any tests to measure what the actual costs of the copying are, so I can't say for sure if the use of Timeless objects here is a win or not.