[Faculty of Engineering, Computing and Mathematics](http://www.directory.uwa.edu.au/view?dn=ou%253DFaculty%2520of%2520Engineering%255C%252C%2520Computing%2520and%2520Mathematics%252C%2520ou%253DFaculties%252C%2520o%253DThe%2520University%2520of%2520Western%2520Australia)

School of Computer Science and Software Engineering

CITS4419: Mobile and Wireless Computing

DSR:

A Raspberry Pi Implementation

Author: Ash Tyndall Asra Alshabib

Bo Chuen Chung Dayang Abang Mordian

Hui Li Leow Max Ward

Raphael Byrne Timothy Raphael

Vincent Sun Zhiqiang (Cody) Qiu

Supervisor: Prof. Amitava Datta

Semester 2, 2013

Acknowledgement

We would like to acknowledge and express our deep gratitude to the following persons who made the completion of this project possible:

Our project supervisor, Prof. Amitava Datta, for his guidance, enthusiastic encouragement and valuable support.

The Faculty of Engineering, Computing and Mathematics for providing laboratory facilities.

Finally we would like to express our thanks and appreciation to our colleagues and industry persons for giving me such attention and time.

Table of Contents

1. The project 1

2. DSR Architecture 1

2.1. General description of DSR 1

2.2. Optimizations 2

3. Documentation 3

3.1. DSR Packet 3

3.1.1. Attributes 3

3.1.2. Packet representation 4

3.2. DSR Routing 4

3.2.1. Constants and Attributes 4

3.2.2. Packet construction 5

3.2.3. Network methods 5

3.2.4. Routing methods 6

3.2.5. Acknowledgement 7

3.2.6. Send Packet 7

3.2.7. Receive packet 8

3.2.8. DSR Updates 8

3.3. Route Cache 9

3.3.1. Constant and Attributes 9

3.3.2. Methods 10

4. Simulation 10

4.1. Network Simulation 10

4.2. Real-time simulation 10

5. Results 11

6. Discussions 12

Appendix: DSR Test Cases 13

# The project

The goal of the project is to implement a Dynamic Source Routing (DSR) protocol. The DSR implementation (described in Section0) is written using Python 3.3.2. The hardware and software used in this project is tabulated in the following table.

|  |  |
| --- | --- |
| **Hardware/Software** | **System Requirements** |
| Raspberry Pi | Linux platform: Raspbian, Debian  Memory: 512MB RAM CPU & GPU  Power: Rechargeable battery  LAN: Wi-Pi |
| Cnet simulator | Windows Vista, 7 and above  Linux and UNIX platforms  Macintosh OS-X 10.4 and above |

Table : Hardware and Software requirement

# DSR Architecture

This section will cover the important components of the implemented DSR protocol as well as the optimization algorithms used.

The assumptions used for the implementation are:

1. Mobile ad hoc networks.
2. Links between two nodes are bi-directional.
3. Message received by the nodes will always be complete.
4. All nodes fully participate in DSR and cooperate to forward packets.
5. Distance between two nodes is the number of hops.

## General description of DSR

In this project, we have implemented a DSR protocol that is capable of performing the tasks of a simple DSR protocol. DSR nodes can send or receive route requests, route reply and sending a packet. Since DSR operates on an on-demand basis, each node also has a route cache and mechanism for route maintenance. When a source node S wants to send message to the destination node D, S would perform route discovery to D by broadcasting route request to the neighbouring nodes. Once D received the request, it will send a route reply back to S based on the same path. When S received the reply, it will send the packet to D, and D will send an acknowledgement message to S upon receiving the packet.

The path used is stored in the route cache for S and D, as well as those nodes that listened to the packet (i.e. nodes that participated in transporting the packet to the destination). As each node in the ad hoc networks moves around frequently, there will naturally be variation in the network topology. If a node that is forwarding the packet discovers a broken link, it will attempt to fix the broken link, and the error will be piggybacked on top of the route request. Each node maintains its route cache by expiring route at certain period, and updates its cache whenever it is receive routing or message packets.

## Optimizations

There are a number of optimizations used in our implemented DSR protocol, briefly outlined as follows:

Shortest Path in *route cache*

There are many useful algorithms for finding the shortest path such as Breath-First Search (BFS), Dijkstra's Algorithm and Ant Colony Optimization. Since the weights between any two links in the network are assumed to be 1 (assumption #5), we have used BFS to find the shortest path in the route cache (refer to Section0 for more details). BFS is the best option, with time complexity of , where and denotes the number of vertices and edges respectively, and .

Automatic Route Shortening in *route send*

When a node N is forwarding a package from the source node S to destination D, it will check its route cache for an alternative path to the D. If N has a shorter path to D compared to the current path used, N will use that path so that the total number of hops from S to D is lessened.

Exponential Back-off

Exponential Back-off algorithm is used in processing the unproductive route requests sent by the node (refer to *check\_send\_buffer* method in 3.2.8 DSR Updates). Route request is only re-broadcasted by the node if the duration of it in the buffer exceeds a specified timeout. Every time the node re-initiates the route discovery, this timeout will double so that node will not re-initiate the route discovery too frequently as it is a costly process.

Partition and Route Request Storm

To prevent route request storm in DSR, each node keeps track of the route request it has already seen in a buffer. In our implementation, any route request that are already seen by the node or any path that is already in the path of the route request will be ignored to avoid route request storm or network partition.

# Documentation

This section will cover the documentation of the python implementation and the data structure of the DSR protocol for the project. The implementation are organised into three components: 1) Packet; 2) Routing Algorithm; and 3) Route Cache.

## DSR Packet

Messages sent to each DSR node are embedded in the form of packet. The types of messages are summarized in Table 2. This section will describe the packet class (*dsr\_packet.py*).

|  |  |  |
| --- | --- | --- |
| **Message Type** | **ID** | **Descriptions** |
| REQUEST | 1 | Route request to initial the routing |
| REPLY | 2 | Route reply for the route request received |
| ERROR\* | 3 | Route error |
| SEND | 4 | Send Message |
| ACK | 5 | Acknowledgement for the message received |
| ERREQ\* | 6 | Error route request to propagate a broken link |

Table : Message type (\* denotes message type not in used)

### Attributes

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| Type | Packet’s message type represented by its corresponding ID as shown in Table 2. |
| Path | The path of the packet from source node to destination node. |
| Contents | The contents of the packet. If message type is send and ack, the content will be the actual message; for other message types, the content will be toID. |
| ID | The identifier of the packet. By default, this is set to be -1. |
| FromID | The identifier of the DSR source node. By default, this is set to be -1. |
| OriginatorID | The ID of the original packet. By default, this is set to be -1. |
| OriginatorNodeID | The ID of the node that first create the original packet. By default, this is set to be -1. |
| toID | The identifier of the DSR destination node. By default, this is set to be -1. |
| brokenLink | A tuple (*A, B*) that represents the broken link between the path *A* and *B*. By default, this is set to be (-1, -1). |

Table : Data structure of a DSR Packet.

Note: *ID, OriginatorID* and will always be a positive integer.When a node broadcast a packet, it resets the packet’s *FromID* and *toID* to -1.

### Packet representation

The packet is translated as string, passed from a DSR node to the network layer, which is represented in the following format:

***Type***| ***Path*** | ***Content***| ***ID***| ***fromID***| ***originatorID***| ***toID***| ***brokenLink*** | ***originatorNodeID***

where each item in *Path* and *brokenLink* are separated by ‘>’.

When a DSR node received the packet string from the network, the string is parsed into the packet object using the method from\_str(packetStr).

## DSR Routing

This section will describe the methods for the DSR routing protocol in details.

### Constants and Attributes

|  |  |
| --- | --- |
| **Constants** | **Descriptions** |
| MAX\_transmissions | The maximum number of transmissions taken before sending route error. |
| MAX\_time\_between\_ack | The maximum time of waiting for an acknowledgement before retransmitting. |
| MAX\_time\_between\_request | The maximum time DSR node should wait for a route reply before broadcasting the route request again. |

|  |  |
| --- | --- |
| **Parameters** | **Descriptions** |
| Node\_addr | The address of the DSR node. |

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| ID | The identifier of this DSR node. This ID is set to be *node\_addr*. |
| Next\_packet\_id | The id of next packet, initialized to be 0. |
| Received\_queue | Keeps track of the packets that are ready to be received by this DSR node on a first in first out basis. |
| Send\_queue | Keeps track of packets that are ready to be sent by this DSR node on a first in first out basis. Each item in the *send\_queue* is represented in the form (*contents, toID*). |
| Send\_buffer | Buffer that keeps track of route request packet, waiting for the route reply. Each item in the buffer is represented in the form (*msg*, *broadcast\_msg*, *start*, *counter*).  Note:   * *msg* is the packet to send * *broadcast\_msg* is the broadacasted request for *msg* * *start* is the starting time for which the packet is sent * *counter* is the number of times the packet has been sent. |
| Done\_buffer | Buffer that keeps track of packets that are sent to this DSR node. |
| Outbox | A list of packets represented as string, which are ready to be sent on the network. Each item in the *outbox* is represented as tuple in the form (*str(pkt), toId*). |
| Awaiting\_acknowledgement\_buffer | Buffer that stores a list of sent packets that are waiting for acknowledgement. Each item in the buffer is represented as (*pkt*, *start*, *timetransmitted*),  Note:   * *start* is the starting time for which the packet *pkt* is being processed * *timetransmitted* is the number of times the packet has been transmitted. |
| Route\_cache | The route cache that store the path for sending message *(refer to Section* 0*. Route Cache).* |
| Seen\_errors | A set of tuples in the form of (*fromID*, *originatorID*), where the error of the originator packet sent by *fromID* has already been seen by this DSR node. |
| Seen\_route\_request | A set of route request which have been already been processed by this node. Each item is represented as a tuple *(fromID, originatorID)*. |
| Already\_received\_msgs | A set of messages sent which have already been processed by this node. These messages are intended for this node to receive. Each item is represented as a tuple *(origintatorID, originatorNodeID)*. |

Table : Data structure of DSR Node.

### Packet construction

Make\_packet(type, path, contents): The method construct a packet by taking in packet *type*, *path* and *contents* as parameters. A packet’s id and *originatorID* are the current value of *next\_packet\_id*. Every time the method is called, *next\_packet\_id* will increment by 1 and return the newly made packet.

Make\_packet\_o(type, path, contents, originator, originatorNode): This method is used to construct a new packet that acknowledge the originators. Similar to *make\_packet*, except that the packet’s *originatorID* and *originatorNodeID* are the respective parameters *originator* and *originatorNode*.

### Network methods

Network\_broadcast(pkt): This method broadcasts *pkt* to its neighbour nodes. *Pkt*.*fromID* and *pkt*.toId are reset to -1. This *str(pkt)* is appended to the *outbox* queue.

Network\_sendto(pkt, toID): This method is called when the DSR node is sending *pkt* to *toID.* Firstly, the *pkt.fromID* and *pkt.toID* are updated to *this*.*ID* and *toID* respectively. Then the *str(pkt)* is appended to the *outbox*.

### Routing methods

Route\_request(msg): This method is used to process the route request (*msg*). Firstly, any *msg.brokenLink* will be removed from the *route\_cache*. It will check two attributes of *msg* for route request: *ms*g.*contents* and *msg.path*. The *msg*.*content* is the addressof the recipient node. There are three cases in method:

1. If this is equal to the DSR node’s *ID*, this node will make a route reply packet *make\_packet\_o(REPLY, rev\_path, toID, msg.originatorID, msg.originatorNodeID)*, where *rev\_path* is the reversed *path* and *toID* is the first address in the *path*. This packet is sent to the next address on the *rev\_path*.
2. However, if condition 1 is not met, and if *this*.*ID* is one of the addresses in *msg*.*path*, or the *msg* is contain in *seen\_route\_requests*, then the DSR node will do nothing to avoid any cyclic requests or seen route requests.
3. If the first two conditions are not met, then the *msg* will be added to node’s *seen\_route\_request.* This node will then find the shortest path to the destination from its *route\_cache* if such path exist and have no broken links:
   1. If such path exists, then extend this path to *msg.path*. This node will then make a new reply packet using *make\_packet\_o(REPLY, rev\_path, source\_ID, msg.originatorID)*, where *rev\_path* is the reversed path of *msg.path* and source\_ID is the first item of *msg.path*. This *reply* packet is sent to the next address on the *rev\_path*.
   2. The DSR node will include itself in *msg.path* and *broadcast* the *request* using *make\_packet\_o(REQUEST, msg.path, msg.contents, msg.originatorID, msg.originatorNodeID)*, together with the *msg.brokenLink*.

Route\_reply(msg): This method is used to process the route reply (msg). There are two cases based on the *msg.contents*, which is used to identify the recipient’s node.

1. If msg.contents is this.ID, then the DSR node will retrieve the contents from the send\_buffer via remove\_from\_send\_buffer(msg.originatorID). If the contents is not empty, this node use the contents to make a packet using make\_packet(SEND, rev\_path, contents), where rev\_path is the reversed path of msg.path. This packet is sent to the next address in the rev\_path, and also added to acknowledgement buffer via *add\_to\_ack\_buffer* method.
2. If condition 1 is not met, then this node will make a reply packet using make\_packet\_o(REPLY, msg.path, msg.contents, msg.originatorID, msg.originatorNodeID) and forward it to the next address in the path after itself.

Route \_send(msg): This method is used to process the route send (*msg),* which denote the sent packet that contains the intended message for the destination node. Firstly, the DSR node will send an acknowledgement to the sender of *msg*. There are two cases here:

1. If the destination of *msg* is *this.ID*, then this node will add *msg* to the *already\_received\_msgs* and *done\_buffer* if *msg* is not in *already\_received\_msgs*.
2. If condition 1 is not met, this node will send *msg* to the next address in the path. This node will look for the shorter path to the destination in its *route\_cache*.
   1. If such *path* is found, and is shorter than the *msg.path* used, then this node make a new packet using *make\_packet\_o(SEND, path, msg.contents, msg.originatorID, msg.originatorNodeID)*. This packet is sent to the next address in the new *path* and also added to acknowledgement buffer via *add\_to\_ack\_buffer* method.
   2. If *path* not found or *path* is longer than *msg.path*, this node will make a new packet using *make\_packet(SEND, msg.path, msg.contents)*. This packet is sent to the address of the next node in the path, and also added to acknowledgement buffer via *add\_to\_ack\_buffer* method.

Route\_discover(data, toID): This method is used by the node to discover route path. Firstly, this node will look at its route cache for *path* to *toID* via *get\_shortest\_path(toID)* – (Refer to Section 0 Route Cache). There are two cases here:

1. If *path* to *toID* is in the route cache, then this node will make a send packet via *make\_packet(SEND, path, msg)*. This packet is sent to the next address in the *path* and also added to acknowledgement buffer via *add\_to\_ack\_buffer* method.
2. If no such *path* is found, this node will make a request packet *pkt* via *make\_packet(REQUEST, new\_path, toID)*, where *new\_path* is a list containing only *this.ID*. This node will broadcast the request packet and record the *time* of the broadcast, and append *(data, pkt, time, 1)* to *send\_buffer*.

Route\_discover\_with\_error(originalPkt, brokenLink): This method is used to do a route discovery whilst propagating error. Firstly, *brokenLink* will be removed from *route\_cache*. This node will then make a new route *request* using *make\_packet\_o(REQUEST, new\_path, destinationID, originalPkt.originatorID, originalPkt.originatorNodeID)*, where *new\_path* is a list that contains only *this.ID* and *destinationID* is the destination of the *originalPkt*. The *request* will piggyback the *brokenLink* which is then broadcasted on the network to notify the neighbouring nodes about the *brokenLink*. This node will record the *time* of the broadcast, and this information is appended to *send\_buffer* in the form *(originalPkt.contents, request, start, 1)*.

### Acknowledgement

Msg\_acknowledgement(msg): This method is used to process the acknowledgement (*msg*) in that this node is waiting for. In the acknowledgement *msg, msg.contents* contains *id* of the packet that is waiting to be acknowledged. Once the item in the *awaiting\_acknowledgement\_buffer* matches that *id*, it is removed from the buffer.

### Send Packet

Send\_message(contents, toID): This method is used to add the to-be-send packet to the *send\_queue*. (*contents, toID*) is appended to the *send\_queue*.

### Receive packet

Receive\_packet(pkt):This method is used to process the *pkt sent to this node* transported by the network. It will parse the string packet *pkt* which is then added to the *receive\_queue* if and only if this node is the recipient of *pkt*. Additionally, this node will look at *pkt.fromID* and *pkt.path* to make sure that *pkt* is not from itself and add the *pkt.path* into the *route\_cache* (via *offer\_route(pkt.path) –* See Section 0 Route Cache*)* if the length of *pkt.path* is greater than 1.

Pop\_inbox(): This method will pop and return all the received messages in the *done\_buffer* that are directed to the DSR node.

Pop \_outbox():This method will pop and return all the sent messages in the *outbox* that are sent from the DSR node.

Remove\_from\_send\_buffer(ID):This method remove the packet from *send\_buffer* based on the packet’s *ID*. It will loop through all the sent messages in the *send\_buffer* to find the message with *originatorID* matching the *ID*. Once this message is found, the method will remove and return the message from *send\_buffer.*

### DSR Updates

Check\_ack\_buffer(): This method will update all items in *awaiting\_acknowledgement\_buffer* for route maintenance. It will loop through the acknowledgements *ack* in the *awaiting\_acknowledgement\_buffer*. There are two cases here:

1. If the number of packet transmission to *next* node has exceeded the *MAX\_transmissions*, this DSR node will broadcast the error message to its neighbours about the *brokenLink*, where *brokenLink* is *(this.ID, next)*. This node will call *route\_discover\_with\_error(ack[pkt], brokenLink)* in an attempt to fix the routing, and remove the *ack* from *awaiting\_acknowledgement\_buffer*.
2. If the *ack* did not exceed the *MAX\_transmissions* duration, but the time taken to received the *ack* exceed the *MAX\_time\_between\_ack* interval, then this node will send the packet again.

Add\_to\_ack\_buffer(pkt): This method will add acknowledgement packet *pkt* to the waiting list.There are two cases for this method.

1. If *pkt* is already in the *awaiting\_acknowledgement\_buffer* in the form *(pkt, start, timetransmitted)*, it will be removed from the buffer. To retransmit the *pkt,* *(pkt, new\_start, timetransmitted)* is added to the buffer, where *new\_start* is the time of the retransmission.
2. If condition 1 is not met, then *(pkt, start, 1)* is added to *awaiting\_acknowledgement\_buffer*, where *start* is the time of the transmission.

Check\_send\_buffer(): This method will update all items in the *send\_buffer*. It will loop through the sent item (*msg*, *broadcast\_msg*, *start*, *counter*) in the *send\_buffer* with two cases:

1. If counter > MAX\_route\_discoveries, then the item is removed from *send\_buffer*.
2. Otherwise if the item exceeds the sent duration, . Then the item is removed and re-appended as *(msg, broadcast\_msg, new\_start, counter+1)* to the *send\_buffer*, where *new\_start* is the new time of packet retransmission. This *broadcast\_msg* is broadcasted in the network again.

Update(): This method will update the attributes of the DSR node. When the method is called, this node will perform three actions as follows:

1. Firstly, process items in *ack\_buffer* and *send\_buffer* via *check\_ack\_buffer* and *check\_send\_buffer* respectively.
2. Next, process and pop all items in *receive\_queue*. For each message *msg* in the *receive\_queue,* the DSR node will ignore any messages that are from itself. For each message, there are five possible scenarios for each message type:
   * REQUEST: run route\_request(msg)
   * REPLY: run route\_reply(msg)
   * SEND: run route\_send(msg)
   * ACK: run msg\_acknowledgement(msg)
3. Finally, process and pop the all items in *send\_queue*. For each *send* message [in the form of *(contents, toID)* in the *send\_queue*, the DSR node will call the *route\_discover(contents, toID)*.

## Route Cache

The data structure of route cache uses an adjacency matrix for representing a graph. The methods in the route cache will be discussed in this section.

### Constant and Attributes

|  |  |
| --- | --- |
| **Constant** | **Descriptions** |
| MAX\_DELTA | The maximum age of a link measured in milliseconds. |

|  |  |
| --- | --- |
| **Parameters** | **Descriptions** |
| **myID** | **The *ID* of the node that uses this route cache.** |

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| Edge\_list | A collection of the edge list for graph representation. Each item in *edge\_list* is represented in the form *{A: {B1, B2, …}}*, where the set *{B1, B2, …}* does not contain any duplicates, and node *A* is linked to *B1, B2*, and so on in the graph. |
| Edge\_age | A collection of the age for each the edges in the graph. Each item in *edge\_age* is represented in the form *{A: {B: n}}*, where the link from *A* to *B* has age *n*. |
| me | The identifier of the root node. This is set to be *myID*. |

Table : Data structure of a route cache.

### Methods

Offer\_route(route): The *route* from node *A* to *B* is represented as list in the form *[A, …, B]* with length *n*. This method will add the route information into the cache via call *add\_link(route[i], route[i+1])*, where *0 ≤ i < n*.

Add\_link(fromID, toID): This method adds a single link *(fromID, toID)* to the route cache by adding it into the *edge\_list* and also *edge\_age* with the current time measured in milliseconds.

Remove\_link(fromID, toID): This method remove a single link *(fromID, toID)* from the route cache by removing it from the *edge\_list* and *edge\_age*.

Get\_shortest\_path(toID): This method is used to find the shortest path to the destination node *toID*. It is used to expire any old links so that the route cache. This is done by removing the link that has an *age* greater than *MAX\_DELTA*. To make sure route cache is up-to-date, this method will expire old links via *expire\_link*(). Breath-first-search (BFS) is used to find the single source shortest path to *toID*.

# Simulation

The simulator is used to simulate the communication between the nodes in DSR. In this project, we have implemented a network simulator to run repeatable experiments for evaluating the performance of the implemented DSR. For real-time simulation, we have implemented the DSR in Raspberry Pi, a small mobile system.

## Network Simulation

The network simulator is used to simulate the communication between 5 mobile nodes. The communication matrix between nodes is initialized prior to the start of simulation. We currently initialized three communication matrixes in three distinct time stamp to represent which node can communicate to each other at different time stamp. There are 50 simulation steps take place in one simulation cycle. When the simulation starts, the first step is to randomly pick a source and destination nodes (the nodes are represented by integer 0-4), and, generate a random message in the form of 6 ASCII characters. Then, the network simulator passes these values to DSR to process the communication between the nodes. If the current iteration step matches the next network change, the network simulator will switch to the next communication matrix. Note that, the main purpose of the network simulator is only to test DSR functionalities in a fixed network topology environment.

## Real-time simulation

The purpose of this simulation is to test DSR functionalities in a dynamic network topology changes. This simulation mimics the network simulator, however, rather than using a fixed number of mobile nodes and a fixed communication matrix between the nodes, the real-time simulation is adapting to the dynamic changes (i.e. the number of mobile nodes participate in the network change dynamically and the communication links between certain nodes may broken at any time).

In this simulation, we setup 6 Raspberry PIs and a number of laptops to participate in the network as mobile nodes. Raspberry PIs are known to have low power mobile device whilst laptops are known to have high power mobile device. Therefore, using both devices to simulate the network will enable us to identify the performance of DSR in a different power level of the mobile nodes.

The simulation involved connecting all the nodes together in the network and try sending messages between each other. Our first testing is to put the nodes together at one place where all the nodes are within each other’s communication range and try sending messages between each other. This is to ensure that our network setup is successful and DSR is doing what it supposed to do. Our second testing is to try changing the network topology by moving the mobile nodes around far from each other. This testing is essential to test whether DSR successfully supports route discovery when the links are broken during the network change and when the nodes are not within each other’s communication range.

# Results

Our current implementation of DSR is focusing on the basic functionalities of DSR. Both simulation results are documented in Appendix: DSR Test Cases. We summarized the results in the following paragraph:

Route Discovery

Our results indicate that both network and real-time simulation successfully perform DSR on-demand route discovery. This implies that our DSR implementation is able to support a fixed and dynamic network topology.

Route Request

Our results indicate that both network and real-time simulation successfully implemented DSR route request functionalities. The only limitation is that, our implementation does not cover the restricted propagation of route request to increase hop count when the nodes are receiving no route reply. The communication is executes only when the nodes fall into each other’s communication range (which is the basic DSR requirements).

Route Cache

Our results indicate that both network and real-time simulation successfully implemented DSR route cache functionalities. The mobile nodes are able to overhear the other nodes communication to learn the path and store it in their route cache, and, use updated route cache information to route the packet to destination node.

Route Reply

Our results indicate that both network and real-time simulation successfully implemented DSR route reply functionalities. The mobile nodes are able send route reply messages and avoid congesting the network with unnecessary replies.

Route Maintenance

In DSR route maintenance, we cover only the general aspects of maintenance such as route shortening and broken links. Our results indicate that both network and real-time simulation successfully support both route shortening and broken links.

Others

Other aspects in our testing include packets dropping, energy efficiency, unique ID and promiscuous operation. Our results indicate that both network and real-time simulation successfully support all the other attributes of DSR accept energy efficiency. We are unable to determine energy efficiency in network simulation because the output is in the text format whilst real-time simulation requires more depth testing to determine the energy efficiency.

# Discussions

Despite the good performance from the results, we found two limitations of the DSR protocol.

1. We assume that the link between any two nodes has weight of 1, which is not very useful if there is a variation in distance between nodes.
2. When sending a packet to a destination node, the source node will give priority to the path that is available in the route cache (if and only if it exists) instead initiating a new route discovery. As node move around, if the destination is near the source, it could have send packet directly to the destination. However in our case, the source will continue to use the path its route cache without initiating a route discovery.

There are a lot of mechanisms that can be extended to our DSR protocol. One extension could be using a distance-based DSR network, which can exploit the other algorithms to find the shorter path. Another extension could be initiating a route request with hop limit when sending a packet, before using the path in the route cache.

Overall, we have successfully implemented the basic DSR functions for this project.

“If there’s a will, there is a way” ☺

# Appendix: DSR Test Cases

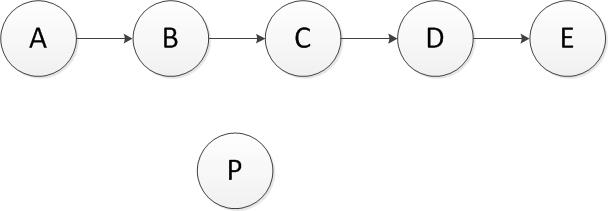


Figure : Topology

For this project, we have designed a set of test cases as tabulated in the following table. This table is based on the sample network topology in Figure 1. The results for the test cases are also included in the table below.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Description** | **Expected Results** | **Actual Results** | |
| **Network Simulation** | **Real-time Simulation** |
| **Route Discovery** | Automatic route discovery | DSR successfully performs automatic route discovery. | **Successful**  The next simulation step showing that the source node automatically performs route discovery to destination node. | **Successful**  Automatic route discovery executes every time the nodes are within each other’s communication range. |
| On-Demand route discovery | DSR successfully performs route discovery only on demand. No periodic broadcasting involved. | **Successful**  The next simulation step showing that the source node automatically performs route discovery to destination node and no other periodic broadcasting identified. | **Successful**  Automatic route discovery executes every time the nodes are within each other’s communication range and no other periodic broadcasting identified. |
| **Route Request** | Sending route request | DSR successfully allows nodes firstly search their own route cache to see whether they have stored route to destination node.  If nodes have the route, then nodes successfully sends that route to source node, else, nodes broadcast the route request message to their neighbors and attach their own ID to the route request message. | **Successful**  The upcoming simulation steps showing that the source node is using route cache information to determine the path to destination node. | **Successful**  The mobile nodes showing that the source node is using route cache information to determine the path to destination node. |
| Route request frequency | DSR successfully allows source node to wait for a fixed amount of time before initiating another route request to avoid flooding the network. | **Successful**  Waiting time for route request is identified. | **Successful**  Waiting time for route request is identified. |
| Hop limit | DSR successfully controls the route request to neighborhood area to avoid unnecessary route reply message. | **Successful**  The simulator defined a default communication matrix to determine which node can talk to each other at each time instance. This avoids unnecessary route reply message. | **Successful**  The route request executes based on node discovery. DSR does not send route request if the neighborhood nodes are outside the communication range |
| Restricted Propagation of Route Request | DSR successfully increases hop count proportionately when receiving no route reply messages to avoid network congestion an unnecessary route reply messages (i.e. start with distance 2, then when no reply message, increase distance to 3 and so on). | **N/A**  The communication between mobile nodes is determined by the communication matrices. The route request executes only when the communication matrix is true. | **N/A**  The communication between mobile nodes is based on the communication range. If the nodes fall into each others’ communication range, route request executes. |
| **Route Cache** | Maintaining route cache information | Each node successfully maintains a route cache and remember the routes that it has learnt about | **Successful**  The next simulation step which has similar source and destination nodes with previous simulation step showing that the source node is able to search its own route cache to destination node. | **Successful**  The same mobile nodes which try to communicate with each other again showing that the source node is able to search its own route cache to destination node. |
| Route cache updates | DSR successfully maintains up-to-date information in the route cache for each node. | **Successful**  The next simulation step which has similar source and destination nodes with previous simulation step showing that the source node is able to use up-to-date route cache information. | **Successful**  The same mobile nodes which try to communicate with each other again showing that the source node is able to use up-to-date route cache information . |
| Route cache has no available information | DSR successfully initiates a new route discovery when no route cache information available. | **Successful**  The first simulation step showing that the source node initiate route discovery when it is unable to obtain any information from its own route cache. | **Successful**  The first communication between mobile nodes showing that the source node initiate route discovery when it is unable to obtain any information from its own route cache. |
| Caching overhead routing information | DSR successfully allows node P to store overhear routing information in its route cache from node B to node C. DSR is also successfully use this information when node P receive route request from node B to node C. | **Successful**  The next simulation steps are able to use the route to destination node that is identified by the other nodes in the previous simulation step. | **Successful**  The mobile nodes are able to use the route to destination node that is identified by the other nodes in the previous communication. |
| **Route Reply** | Complete route discovery | DSR successfully sends route reply message from destination node that has route to destination in its route cache. | **Successful**  Identified. | **Successful**  Identified. |
| Partial route discovery | DSR successfully sends route reply message from intermediate node that has route to destination in its route cache. | **Successful**  Identified. | **Successful**  Identified. |
| Waiting before reply | All nodes successfully wait for a random amount of time and listen to the traffic before sending route reply message to avoid network congestion and packet collisions. | **Successful**  Waiting time before route reply is identified. | **Successful**  Waiting time before route reply is identified. |
| Accumulated route reply | DSR successfully sends route reply message back to the source node using accumulated route (the nodes through which it has passed). | **Successful**  Accumulated route reply is identified in the packet details. | **Successful**  Accumulated route reply is identified in the packet details. |
| Route reply storm | DSR successfully avoids route reply storm (many nodes try to send route reply for the same destination which may flood the network). | **Successful**  Waiting time before route reply is identified. | **Successful**  Waiting time before route reply is identified. |
| **Route Maintenance** | Alternative route for broken route | DSR successfully uses alternative route stored in route cache when priority route is broken. | **Successful**  Identified. | **Successful**  Identified. |
| New route discovery for broken route | DSR successfully discovers new route when priority route is broken and route cache has no alternative route stored. | **Successful**  When the route is broken and route cache is empty, the next simulation steps showing that the source node no longer has priority route stored in its route cache and it starts a new route discovery | **Successful**  When the route is broken and route cache is empty, the mobile nodes showing that the source node no longer has priority route stored in its route cache and it starts a new route discovery |
| Active acknowledgment | DSR successfully retransmits packet for a fixed number of times if no acknowledgement received. | **Successful**  The simulation output showing that DSR keep transmitting | **Successful**  Re-transmitting is identified on the output when the source node failed to send message to destination node |
| Passive acknowledgement | DSR successfully sends acknowledgement to node A when node A overhear the forwarding of the packet to node B and knows that node B successfully received the packet. | **Successful**  Identified. | **Successful**  Identified. |
| Spreading route Error Message | DSR successfully sends error message if the nodes do not receive any acknowledge-ment after retransmit message for a fixed number of times. | **Successful**  Identified. | **Successful**  Identified. |
| On-Demand route maintenance | DSR successfully operates route maintenance only on demand and no periodic broadcasting involved. | **Successful**  Retransmitting packets occurs only when the source node failed to reach destination node. No periodic broadcasting identified. | **Successful**  Retransmitting packets occurs only when the source node failed to reach destination node. No periodic broadcasting identified. |
| Changing in communication pattern | Number of overhead packets increases and DSR is successfully performs new route discovery and new route discovery packets are the overhead packets. | **Successful**  Identified. | **Successful**  Identified. |
| Packet salvaging | DSR successfully indicates to the other nodes that the packet sent has been salvaged when the node receives route error message and re-send the packet that cause route error. | **Successful**  Packets are attached with broken link information to let the other nodes know that the packet has been salvaged. | **Successful**  Packets are attached with broken link information to let the other nodes know that the packet has been salvaged. |
| Automatic route shortening | DSR successfully performs automatic route shortening.  Destination node informs source node that it can ignore several intermediate nodes. | **Successful**  Identified. | **Successful**  Identified. |
| **Others** | Drop packets when nodes are static | DSR successfully drops the number of overhead packets to zero when the nodes are static and all routes have been discovered. | **Successful**  Identified. | **Successful**  Identified. |
| Energy-efficient | DSR successfully shows more energy-efficient and does not congest the network with too many control messages. | **N/A**  Unable to determine the energy efficiency because the simulation output is in the text format. | **N/A**  Unable to determine the energy efficiency of mobile nodes at the moment. Longer and depth testing are needed to determine this attribute. |
| Unique ID | DSR successfully assigns unique ID for each node. | **Successful**  Route reply messages contain unique ID of the source and destination node | **Successful**  Route reply messages contain unique ID of the source and destination node |
| Promiscuous mode of operation | DSR successfully allows each node to overhear or not to overhear other nodes' transmission. | **Successful**  The next simulation steps are able to use the route to destination node that is identified by the other nodes in the previous simulation step. | **Successful**  The mobile nodes are able to use the route to destination node that is identified by the other nodes in the previous communication. |