

CITS4419: Mobile Wireless and Computing

**Group Project**

**DSR Report**

Ash Tyndall, Asra Alshabib, Bo Chuen Chung, Dayang Abang Mordian, Hui Li Leow, Max Ward, Raphael Byrne, Timothy Raphael, Vincent Sun, Zhiqiang (Cody) Qiu

Unit coordinator: Professor Amitava Datta

Semester 2, 2013

Table of Contents

[1. The project 3](#_Toc372648248)

[2. DSR Architecture 3](#_Toc372648249)

[2.1. Components 3](#_Toc372648250)

[2.2. Optimisations 3](#_Toc372648251)

[3. Documentation 3](#_Toc372648252)

[3.1. DSR Packet 3](#_Toc372648253)

[3.1.1. Attributes 4](#_Toc372648254)

[3.1.2. Packet representation 4](#_Toc372648255)

[3.2. DSR Routing 4](#_Toc372648256)

[3.2.1. Constants and Attributes 4](#_Toc372648257)

[3.2.2. Packet construction 5](#_Toc372648258)

[3.2.3. Network methods 6](#_Toc372648259)

[3.2.4. Routing methods 6](#_Toc372648260)

[3.2.5. Acknowledgement 7](#_Toc372648261)

[3.2.6. Send Packet 7](#_Toc372648262)

[3.2.7. Receive packet 7](#_Toc372648263)

[3.2.8. DSR Updates 8](#_Toc372648264)

[3.3. Route Cache 9](#_Toc372648265)

[3.3.1. Constant and Attributes 9](#_Toc372648266)

[3.3.2. Routing 9](#_Toc372648267)

[4. Simulation 10](#_Toc372648268)

[4.1. Network Simulation 10](#_Toc372648269)

[4.2. Real-time simulation 10](#_Toc372648270)

[5. Results 10](#_Toc372648271)

[6. Discussions 10](#_Toc372648272)

# The project

The goal of the project is to implement a Dynamic Source Routing (DSR) protocol. The DSR implementation (described in Section 2) 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 Requirements

# DSR Architecture

This section will cover the components

## Components

## Optimizations

# Documentation

The DSR algorithm is implemented in python. This section will cover the documentation of the python implementation for the project.

## DSR Packet

Messages sent to each DSR node are embedded in the form of packet. The type of messages are summarized in Table 1. 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 2: Packet’s message type

### Attributes

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| Type | Packet’s message type represented by its corresponding ID as shown in Table 1. |
| 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). |

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

Table 3: Packet’s attribute

### 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

### 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. |

Table 4: Constants in DSR class

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

Table 5: Parameter in DSR class

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| ID | The identifier of the 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 which are ready to be received by the DSR node on a first in first out basis. |
| Send\_queue | Keeps track of packets which are ready to be sent by the 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*, *originatorID*, *start*, *counter*), where *start* is the starting time for which the packet *pkt* is processed, and *counter* keep tracks of the number of times the packet has been sent. |
| Done\_buffer | Buffer that keeps track of packets that are sent to the 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*), where *start* is the starting time for which the packet *pkt* is being processed, and *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* 3.3*. 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 the DSR node. |
| Already\_received\_msgs | A set of |

Table 6: Attributes in DSR class

### 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): Similar to *make\_packet*, but here, 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 wants to send *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 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*, the node will make a route reply packet *make\_packet\_o(REPLY, rev\_path, toID, msg’s originatorID)*, 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 that the *this*.*ID* is one of the addresses in *msg*.*path*, then the DSR node will do nothing to avoid cycles.
3. If the first two conditions are not met, then this node will *broadcast* a route *request* packet using *make\_packet\_o(REQUEST, msg.path, msg.contents, msg.originatorID)*.

Route\_request\_with\_error(msg): This method will process route request *msg* that contains piggybacked error. The DSR node will remove the *brokenLink* from its route cache.

Route\_reply(msg): This method will check the *msg.contents* which is used to identify the recipient’s node. There are two cases here:

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, the 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 the node will make a reply packet using make\_packet\_o(REPLY, msg.path, msg.contents, msg.originatorID) and forward it to the next address in the path after itself.

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

1. If the destination of *msg* is *this.ID*, then the node will add the *msg* to the *done\_buffer.*
2. If condition 1 is not met, then the node will packet 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(msg, toID): The node will look at its route cache for *path* to *toID* via *get\_shortest\_path(toID)* – (Refer to Section 3.3 Route Cache). There are two cases here:

1. If *path* to *toID* is in the route cache, then the 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, the node will make a request packet *pkt* via *make\_packet(REQUEST, path, toID)*, where *path* is a list containing *this.ID*. The node will broadcast the request packet and record the *time* of the broadcast, and append *(msg, pkt.originatorID, time, 1)*  to the *send\_buffer*.

Route\_discover\_with\_error(msg, toID, brokenLink):

### Acknowledgement

Msg\_acknowledgement(msg): 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): (*contents, toID*) is appended to the *send\_queue*.

### Receive packet

Receive\_packet(pkt):This method will parse the string packet *pkt* which is then added to the *receive\_queue* if and only if the node is the recipient of *pkt*. Additionally, the 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 3.3 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 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 return the message and remove it 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 *ack* exceed the the *MAX\_transmissions* duration, the DSR node will broadcast the error message to its neighbours about the broken link. The DSR node will do a route discovery in an attempt to fix the routing.
2. If the *ack* did not exceed the *MAX\_transmissions* duration, the node will whether the time taken to receive the *ack* exceed the *MAX\_time\_between\_ack* interval. If the time exceeded, then the node will send the packet again.

Add\_to\_ack\_buffer(pkt): This method will add packet *pkt* to *awaiting\_acknowledgement\_buffer.* 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 items *send\_buffer*. It will loop through the sent item (*msg*, *originatorID*, *start*, *counter*) in the *send\_buffer* to check whether it exceeds the sent duration, *MAX\_time\_between\_request\*counter*. If this condition is met by an item, it will be removed from the *send\_buffer*. A new item *(msg, originatorID, new\_start, counter+1)* is then appended to the buffer, where *new\_start* is the new time of packet retransmission.

Update(): This method will update the attributes of the DSR node. When the method is called, the 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)
   * ERREQ: run route\_request\_with\_error(msg)
   * REPLY: run route\_reply(msg)
   * ERROR: run route\_error(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 route cache is represented using graph.

### Constant and Attributes

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

Table 7: Constant in route cache

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

Table 8: Parameter in 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 9: Attributes in route cache

### Routing

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 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

We perform real-time simulation using Raspberry Pi device as the mobile nodes. 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, real-time simulation is adapting to the dynamic changes (i.e. the number of mobile nodes participate in the network change and the communication links between certain nodes may broken at any time)

# Results

DSR performance result is documented in the form of test cases. Our test cases cover the general aspects of DSR. These include:

* Route Discovery
* Route Request
* Route Cache
* Route Reply
* Route Maintenance
* Others

**DSR TEST CASES**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Description** | **Expected Results** | **Actual Results** | |
| **Network Simulation** | **Real-time Simulation** |
| **Route Discovery** | Automatic route discovery | DSR successfully performs automatic route discovery. | **Successful**  Successful automatic route discovery. |  |
| On-Demand route discovery | DSR successfully performs route discovery only on demand. No periodic broadcasting involved. | **Successful**  Successful on-demand route discovery |  |
| **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**  Successful sending route request. |  |
| 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**  Successful with route request waiting. |  |
| 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. |  |
| 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). |  |  |
| **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. |  |
| Route cache updates | DSR successfully maintains an 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 an 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. |  |
| 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. | Unknown |  |
| **Route Reply** | Complete route discovery | DSR successfully sends route reply message from destination node that has route to destination in its route cache. | **Successful** |  |
| Partial route discovery | DSR successfully sends route reply message from intermediate node that has route to destination in its route cache |  |  |
| 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. |  |  |
| Accumulated route reply | DSR successfully sends route reply message back to the source node using accumulated route (the nodes through which it has passed). |  |  |
| 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). |  |  |
| **Route Maintenance** | Alternative route for broken route | DSR successfully uses alternative route stored in route cache when priority route is broken. |  |  |
| New route discovery for broken route | DSR successfully discovers new route when priority route is broken and route cache has no alternative route stored. |  |  |
| Active acknowledgment | DSR successfully retransmits packet for a fixed number of times if no acknowledgement received. |  |  |
| 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. |  |  |
| Spreading route Error Message | DSR successfully sends error message if the nodes do not receive any acknowledgement after retransmit message for a fixed number of times. |  |  |
| On-Demand route maintenance | DSR successfully operates route maintenance only on demand. No periodic broadcasting involved. |  |  |
| 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. |  |  |
| 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. |  |  |
| Automatic route shortening | DSR successfully performs automatic route shortening.  Destination node informs source node that it can ignore several intermediate nodes |  |  |
| **Others** | Packet Size | DSR successfully sends smaller size of overhead packets (DSR is designed to send smaller packet). |  |  |
| 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. |  |  |
| Energy-efficient | DSR successfully shows more energy-efficient and does not congest the network with too many control messages. |  |  |
| Unique ID | DSR successfully assigns unique ID for each node. |  |  |
| Promiscuous mode of operation | DSR successfully allows each node to overhear or not to overhear other nodes' transmission. |  |  |

Table 10: DSR test cases

# 

# Discussions