The University of Western Australia

#insert\_uwa\_logo

CITS4419: Mobile Wireless and Computing

**Group Project**

**DSR Documentation**

Ash Tyndall, Asra Alsahib, 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

# The project

The goal of the project is to implement a Dynamic Source Routing (DSR) protocol. To achieve this, the DSR was written in Python 3.3.2.

# Simulator

There are two types of simulator involved. Firstly, network simulator is used for testing whether DSR successfully sending and receiving messages between a fixed number of nodes. Secondly, CNet simulator is used for the actual simulation of the network routing.

## Network Simulator

Network layer is responsible to simulate communication between the nodes in DSR. At the moment, network simulator generated 5 virtual nodes. Network simulator is also generating a default talk matrix to represent the communication topology between the nodes. The number of attempts of sending and receiving packets is set to 10 times. Simulator stored all the communication details in the log files. Each node has its own log file. In this case, network simulator generated 5 log files when it is running. Each log file is updated every 0.5 second. Log files stop updating when network simulator and/or DSR terminated.

### Constants and Attributes

|  |  |
| --- | --- |
| **Constants** | **Descriptions** |
| NUM\_NODES = 5 |  |
| CAN\_TALK = [(0, [ [0, 1, 1, 1, 1],  [1, 0, 1, 1, 1],  [1, 1, 0, 1, 1],  [1, 1, 1, 0, 1],  [1, 1, 1, 1, 0] ]),  (3, [ [0, 1, 1, 1, 1],  [1, 0, 1, 1, 1],  [1, 1, 0, 1, 1],  [1, 1, 1, 0, 1],  [1, 1, 1, 1, 0] ]),  (6, [ [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0],  [0, 0, 0, 0, 0] ]),  (9, [ [0, 1, 1, 1, 1],  [1, 0, 1, 1, 1],  [1, 1, 0, 1, 1],  [1, 1, 1, 0, 1],  [1, 1, 1, 1, 0] ])] |  |
| RUN\_LOOPS = 10 |  |
| COMM\_LOC = ‘logs/’ |  |

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| num\_nodes |  |
| processes |  |
| out\_pipes |  |
| In\_pipes |  |
| comm\_loc |  |
| loops |  |
| start\_time |  |

#### Network simulator methods

node\_simulation(q, log\_to, node\_addr, other\_nodes, loops): This method

can\_talk(a, b): Firstly, this method calculates the time difference between packet’s sending time and the current time. If the time difference is larger than time in the talk matrix then

 time\_diff **=** int(time**.**time() **-** self**.**start\_time)

    prev\_m **=** CAN\_TALK[0][1]

**for** t, m **in** CAN\_TALK:

**if** time\_diff **>** t:

          prev\_m **=** m

**if** time\_diff **<** t:

**break**

**return** prev\_m[a][b] **==** 1

start(): This method

main(): Main function created simulator object and set the number of nodes, talk matrix, number of loops and log files destination folder. Then, main function calls method start to begin simulation operation.

## Cnet simulator

# Packet

Messages sent to each DSR node are embedded in the form of packet. The type of messages are summarised in Table 1. This section will describe the packet class (*dsr\_packet.py*).

|  |  |
| --- | --- |
| **Message Type** | **ID** |
| REQUEST | 1 |
| REPLY | 2 |
| ERROR | 3 |
| SEND | 4 |
| ACK | 5 |

Table 1: 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 that the packet uses to reach its destination |
| 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. This is -1 if the packet is being broadcasted. |
| FromID | The identifier of the DSR source node. |
| OriginatorID | The ID of the original packet. |
| toID | The identifier of the DSR destination node. |

## 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* with each node ID separated by ‘>’**| ***Content*** | ***ID*** | ***fromID*** | ***originatorID*** | ***toID***

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

## Constants and Attributes

|  |  |
| --- | --- |
| **Constants** | **Descriptions** |
| MAX\_transmissions = 2 | The maximum number of transmissions taken before sending route error. |
| MAX\_time\_between\_ack = 1 | The maximum time between a specific acknowledgement sent. |
| MAX\_time\_between\_request = 1 | The maximum time between route request sent. |

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| Network | The network layer that connect the DSR node. *(see* 2*. Network)* |
| ID | The identifier of the DSR node. This ID is the node address that is required for creating a DSR object [i.e. dsr1 = DSR(*node\_addr*)]. |
| Next\_packet\_id | The id of next packet, initialized to be 0. |
| Received\_queue | A list of the packets which are ready to be received by the DSR. DSR will process the packets in the *received\_queue* in a first in first out basis. |
| Send\_queue | A list of packets which are ready to be sent by the DSR. DSR will process the packets in the *send\_queue* in a first in first out basis. |
| Send\_buffer | Buffer that stores a list of sent route request packet, waiting for the route reply. Each item in the buffer is represented as (*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 stores a list of packets for which the DSR node is the destination node. |
| Outbox | A list of packets in string format which are ready to be sent on the network. Each item in the outbox is represented as (*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. *(see* 5*. Route Cache)* |
| Seen | A set of (*id*, *fromID*) tuples representing the packet’s *id* and fromID for which the DSR node had already seen. |

## Constructing packet

Make\_packet(type, path, contents): The method construct a packet by taking in packet’s *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): Similar to *make\_packet*, but with the originatorID as an additional parameter.

## Network methods

Network\_broadcast(pkt): Packet *pkt* is broadcast to other the neighbour nodes. The broadcasted message are characterised by the *pkt*’s *fromID* = -1 and *pkt*’s toId = -1. This *pkt* is added to the outbox queue.

Network\_sendto(pkt, toID): Firstly, the packet *pkt*’s *fromID* and *toID* are updated to the node’s *ID* and *toID* respectively. If the *pkt* is not *acknowledgement* message, then *add\_to\_ack\_buffer(pkt)* is called. Then the *pkt* that is ready to be sent through the network will be packed into a string format, and appended to the *outbox*.

## Routing methods

Route\_request(msg): This method will check two attributes of the route request message *msg*: 1) *ms*g’s *contents*; and 2) *msg’s path*. If *msg* is directed for itself, it will send a route reply by reversing the *msg’s path*. However, if that is not the case, and that the *msg’s* *path* contains the node’s *ID*, then the DSR node will do nothing to avoid cycles. If the two conditions are not met, then the DSR node will broadcast the request to its neighbour nodes.

Route\_reply(msg): This method will check if the route reply message *msg* is directed for the DSR node.

Route \_error(msg):

Route \_send(msg):

Route \_discover(msg, toID):

## Send packet

Msg\_acknowledgement(msg):

Send\_message(contents, toID):

## Receive packet

Receive\_packet(pkt):This method will parse the string packet *pkt* that is received through the network and appended to *receive\_queue*.

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 loop through the acknowledgements *ack* in the *awaiting\_acknowledgement\_buffer*. 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. If the *ack* did not exceed the *MAX\_transmissions* duration, the node will whether the time taken to received 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():

Check\_send\_buffer():This method will loop through the sent packet *send* in the *send\_buffer*.

Update(): Each DSR node will constantly be updating its queue and buffer. The nodes will first check and process the *ack\_buffer* and *send\_buffer* via *check\_ack\_buffer* and *check\_send\_buffer* respectively. After checking the buffer, the DSR node will process all the messages in the *receive\_queue* and *send\_queue*.

For each message *msg* in the *receive\_queue,* the DSR node will ignore any messages that are from it, and there are five scenarios for each message type:

* Request: run route\_request(msg)
* Reply: run route\_reply(msg)
* Error: run route\_error(msg)
* Send: run route\_send(msg)
* Ack: run msg\_acknowledgement(msg)

# Route Cache

## Constant and Attributes

|  |  |
| --- | --- |
| **Constant** | **Descriptions** |
| MAX\_DELTA= 1000 |  |

|  |  |
| --- | --- |
| **Attributes** | **Descriptions** |
| Edge\_list |  |
| Edge\_age |  |
| me | The ID of the DSR node which the route cache belongs to. |

## Routing

offer\_route(route):

add\_link(fromID, toID):

remove\_link(fromID, toID):

get\_shortest\_path(toID):