2006

Routing in Anhinga

Aakash Chauhan

Routing in Anhinga

Master's Project Report

Aakash Chauhan
Department of Computer Science,
Rochester Institute of Technology

Chairman : Prof. Hans-Peter Bischof *Date*

Reader: Prof. Alan Kaminsky *Date*

Observer: Prof. Sidney Marshall *Date*

1. Abstract

Anhinga project with Many-to-Many Protocol (M2MP) takes fundamentally different approach to resolve dynamic network topology in ad hoc networks. Instead of trying host-address based networking and routing to make work in ad hoc network environment, M2MP removes device address and groups. All messages goes to all devices within proximal area and each device decides whether and how to process the message based on message's content. [1] This project expands on this infrastructure by delivering these messages beyond sender device's broadcast range. The project implements Dynamic Source Routing (DSR) protocol based controlled flooding techniques. It also provides functionality for node-to-node communication based on IETF draft for DSR [13] in Anhinga infrastructure.

2. Overview

2.1 Anhinga [3]

The Anhinga Project is the distributed computing infrastructure designed with supporting collaborative applications running in mobile ad hoc networks.

Anhinga provides infrastructure for mobile wireless computing devices such as laptop computers, cell phones, pocket PCs that runs completely on the mobile device. Unlike traditional middleware infrastructure for networking technologies which require central servers and wired connections, anhinga runs on mobile devices and does not require central server to run collaborative applications in mobile environment.

Anhinga is built on two major components, Many-to-Many Protocol (M2MP) and Many-to-Many Invocation (M2MP). M2MP is a message broadcasting protocol designed with serverless ad hoc networks of small devices in mind. M2MI is built on top of M2MP. M2MI is object oriented abstraction of broadcast method invocation. M2MI allows a method invocation across the devices and through out the network. M2MI can simply be described as "All the devices out there who implement this interface, execute this method."

The architecture of Anhinga and M2MP & M2MI in particular is described in detail in later sections.

2.2 Why this project is needed

There is extensive amount of research done about routing in ad hoc networks. Majority of work concentrates on routing in network based on the host address (such as IP Address). Anhinga with M2MP and its broadcast mechanism eliminates the host address requirement to communicate within a network. But anhinga do not really route packets across nodes between geographically dispersed networks.

This project takes aim at combining these two. Meaning putting an existing MANET routing protocol into anhinga and make two nodes in geographically dispersed network communicate via M2MP.

2.3 Various MANET protocols

Ad hoc network is a collection of wireless mobile hosts that dynamically form a temporary network where any node can join or leave the network without any prior notification. It has no fixed infrastructure. It is formed and deformed on the fly, depending on the need of the mobile users. There is extensive research being done in routing in these Mobile Ad Hoc Networks (MANET). In such mobile networks battery power and bandwidth are very limited. The research done concentrates on minimum protocol overhead with quick adaptability to rapidly changing network topology.

Major categorization of MANET protocols is as follows:

1. **Pro-active (Table- Driven) -**

Pro-active (Table-driven) protocol tries to maintain consistent view of routes to all possible nodes from a given node. The protocol tries to maintain these routes even before they are ever needed or used. The Destination-Sequenced Distance-Vector Routing (DSDV) protocol is a table driven algorithm that modifies the Bellman-Ford routing algorithm to include timestamps that prevent loop-formation. The Wireless Routing Protocol (WRP) is a distance vector routing protocol which belongs to the class of path-finding algorithms that exchange second-to-last hop to destinations in addition to distances to destinations. Since these type of protocol try to keep track of all the changes in network topology, there is significant routing overhead is attached to this class of MANET routing protocols.

2. **Reactive (On - demand) –**

On-demand routing protocols try to reduce control overhead, thus decreasing bandwidth and power usage. These protocols limit the amount of bandwidth consumed by maintaining routes to only those destinations for which a source has data traffic. Therefore, the routing is source-initiated as opposed to table-driven routing protocols that are destination initiated. This type of protocol also avoids heartbeats between nodes since they do not try to maintain complete network topology. There are several examples of this approach (e.g., DSR, ABR, AODV, TORA, SSA, ZRP) and the routing protocols differ on the specific mechanisms used to reduce flood search packets and their responses, cache the information heard from other nodes' searches, determine the cost of a link, and determine the existence of a neighbor.

3. Hybrid (Pro-Active/Reactive) –

Hybrid Routing, is a mix of distance-vector routing and link-state routing. It tries to balance the overhead by sharing mix of local and global knowledge about network with other nodes. Enhanced Interior Gateway Routing Protocol (EIGRP) is an example of this type of protocol.

3. Proposed Protocol and Working

3.1 What is DSR Protocol

The Dynamic Source Routing protocol (DSR) is a simple and efficient on-demand (reactive) routing protocol. Unlike lot of other protocols DSR does not maintain network topology, which allows the network to be completely self-organizing and self-configuring, without the need for any existing network infrastructure or administration. There are two major mechanisms in DSR, 'Route Discovery' and 'Route Maintenance'. These two together allow nodes to discover and maintain routes to any destinations in an ad hoc network. Since this protocol works completely on-demand, the overhead scales automatically to changes in the network and nothing else. This allows overhead less protocol when all nodes are stationary in the topology and scales up with topological changes. The protocol allows multiple routes to any destination and allows each sender to select the route used in routing its packets. The route discovery mechanism guarantees loop-free routes. DSR also works in networks with unidirectional link.

The protocol was designed with mobile ad hoc networks in mind, where battery power and bandwidth are expensive. To restrict unnecessary bandwidth usage and processing power, DSR does not initiate a route discovery until it is needed and initiated by a node. Similarly once a route is found there is no effort is put in maintaining that route. Route availability or new route search is only initiated when a route error occurs for existing route and some node initiates a new route discovery process to that same destination. Unlike other reactive routing protocols like ABR or SSA, DSR does not require heart beat message between neighbors to maintain network topology.

DSR is based on the Link-State-Algorithms which mean that each node is capable to save the best way to a destination. Also if a change appears in the network topology, then the whole network will get this information by flooding.

3.2 Working Of DSR Protocol

As discussed earlier there are two major components of DSR,

1) Route Discovery, which consists of
    a. Route Request
    b. Route Reply
2) Route Maintenance

General workings of these are discussed in the following section. A detailed discussion about implementation and working of these mechanisms is done in section 4.

3.2.1 Route Discovery

When a node wants to send a message, it first checks its route cache table for a route to destination. If no route found then an automatic route discovery process is initiated. The node seeking the route sends out route request message. When a node receives a route request message, it adds itself to the source route and rebroadcasts the request. Once the request reaches the intended destination node, destination node adds itself to the route and initiates a route reply message. Section 4 discusses the detail working of route request and route reply.

For example, suppose a node A is trying to find a route to node E. The Route Discovery initiated by node A in this example would proceed as follows:

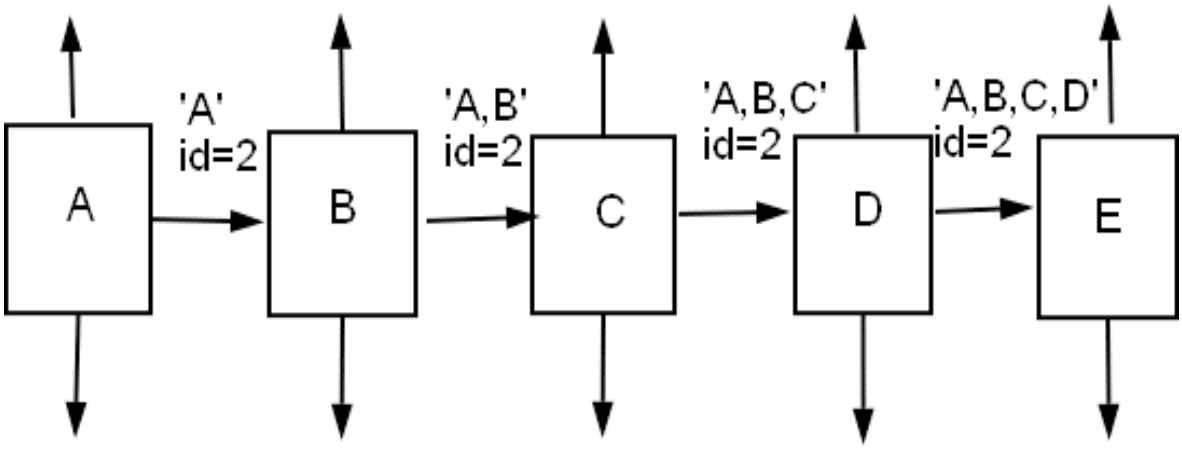


Figure 1: Route Discovery [18]

If node A has in his Route Cache a route to the destination E, this route is used. If not, the Route Discovery protocol is started:

a) Node A (initiator) sends a RouteRequest packet by flooding the network

b) If node B has recently seen another RouteRequest from the same target or if the address of node B is already listed in the Route record, then node B discards the request.

c) If node B is the target of the Route Discovery, it returns a RouteReply to the initiator. The RouteReply contains a list of the "best" path from the initiator to the target. When the initiator receives this RouteReply, it caches this route in its Route Cache for use in sending subsequent packets to this destination.

d) Otherwise node B isn't the target and it forwards the RouteRequest to his neighbors (except to the initiator).

3.2.2 Route Maintenance

In DSR every node confirms that the next hop in the Source Route receives the packet it sent. When a node can not successfully transmit a packet to its next intended neighbor after multiple tries, a Route Error message is sent to the initiator,

so that it can remove that Source Route from its Route Cache. If there is no route in the cache, a RouteRequest packet is broadcasted. Also if there was any other route in the route cache that was using exact same broken link then all those routes are invalidated. Figure 5 shows an example,

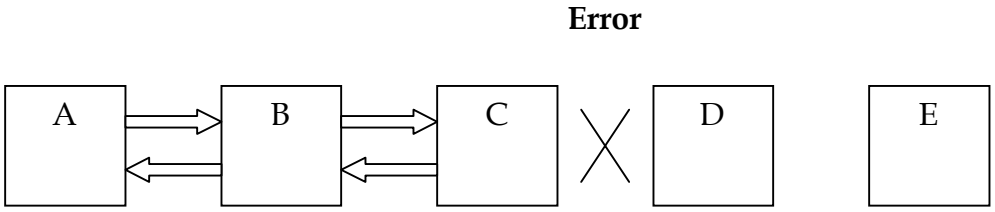


Figure 2: Route Error [18]

1. If node C does not receive an acknowledgement from node D after some number of requests, it returns a RouteError to the initiator A.

2. As soon as node receives the RouteError message, it deletes the broken-link-route from its cache. If A has another route to E, it sends the packet immediately using this new route.

3. Otherwise the initiator A is starting the Route Discovery process again.

Details about this route process are discussed in section 4.

4. Architectural Overview

4.1 Existing M2MP Architecture [3]

The Many-to-Many Protocol is designed for the wireless proximal ad hoc networking environment. Following points highlight major features of M2MP architecture:

- **There are no device addresses.** M2MP works in broadcast medium. Consequently, devices can enter and leave the network in an ad hoc fashion without having to do network configuration.
- **Messages are broadcast to all devices.** Instead of point to point communication, all the messages in M2MP are broadcasted taking inherent advantage of wireless communication medium in wireless ad hoc network.
- **A message's relevancy is determined by its contents.** A node determines if a particular message is of its interest based on its content.
- When an application on one device sends an M2MP message, the application writes a stream of bytes with the message's contents to the M2MP layer. The M2MP layer breaks the byte stream into a sequence of packets and broadcasts each packet. On the receiving side, the M2MP layer puts content of each packet back into corresponding message input stream which is used by the application to read the content. In order to receive a particular message an application needs to register a *message filter*, a fixed length byte string with the M2MP layer. When M2MP layer receives any message it matches the initial content of that message to these filters. If the initials match then the message is passed to the upper layer application that registered for that filter. The message is passed to upper layers only when they match a certain filter, otherwise the M2MP layer discards the message. An application that uses M2MP, such as M2MI, can use this filtering mechanism to weed out unwanted messages and separate itself from other M2MP based application. For further information and detailed architecture of M2MP protocol please refer to, http://www.cs.rit.edu/~anhinga/m2mp.shtml

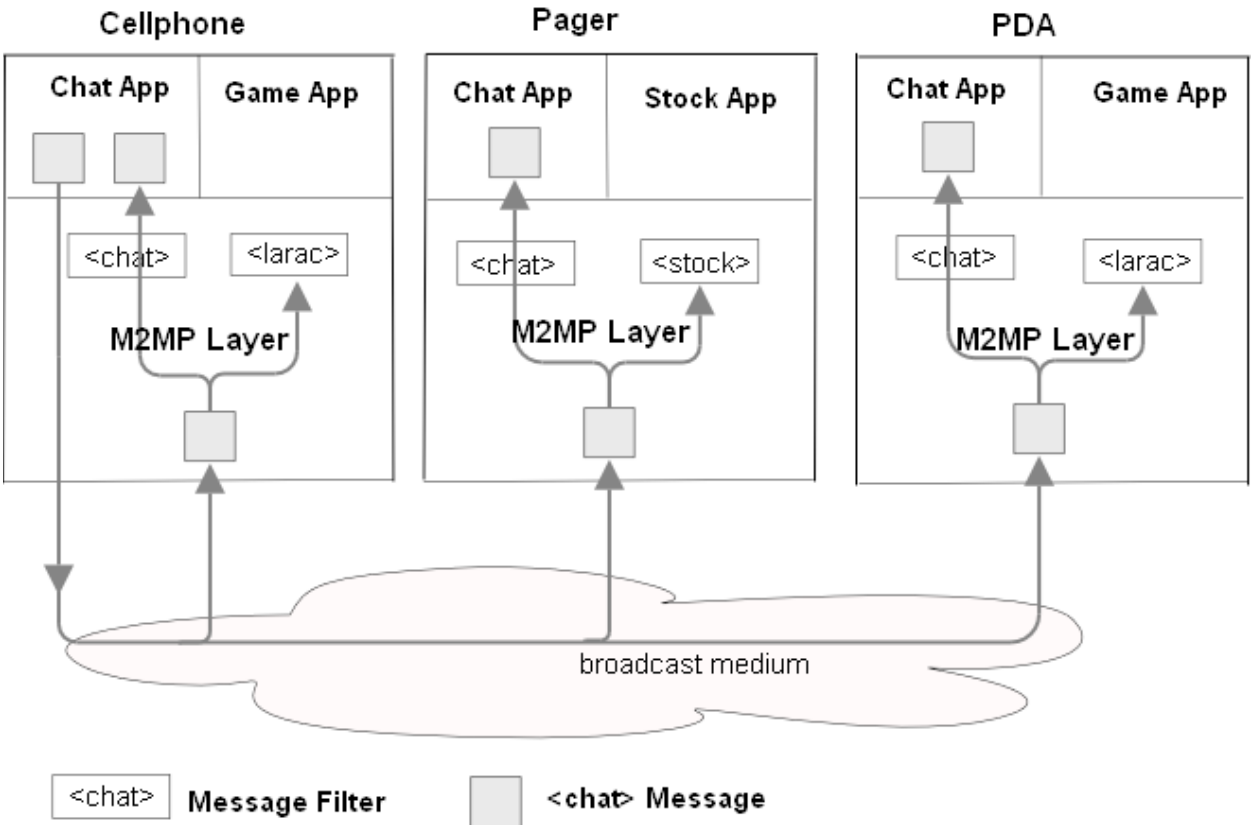


Figure 3: M2MP Architecture [3]

4.2 Existing M2MI Architecture [3]

Many-to-Many Invocation (M2MI) allows building collaborative systems that run in wireless proximal ad hoc networks of fixed and mobile computing devices. M2MI provides an object oriented method call abstraction based on broadcasting. An M2MI invocation means "Every object out there that implements this interface, call this method." An application on top of M2MI is built by defining one or more interfaces, building objects that implement those interfaces, and broadcasting method invocations to all the objects on all the devices. M2MI works on top of above discussed Many-to-Many Protocol (M2MP), which broadcasts messages to all nearby devices over wireless network instead of routing messages from device to device. M2MI builds remote method invocation proxies dynamically at run time,

eliminating the need to compile and deploy proxies ahead of time. Because of dynamic proxy, central servers are not required; which eliminates lot of network administration. Also since M2MI works in pure broadcasting environment, complicated, resource-consuming ad hoc routing protocols are not required.

M2MI provides object oriented abstraction of message broadcasting. An application can invoke a method declared inside an interface on multiple objects on multiple devices at same time using M2MI. For further information on M2MI architecture please refer to, http://www.cs.rit.edu/~anhinga/m2mi.shtml

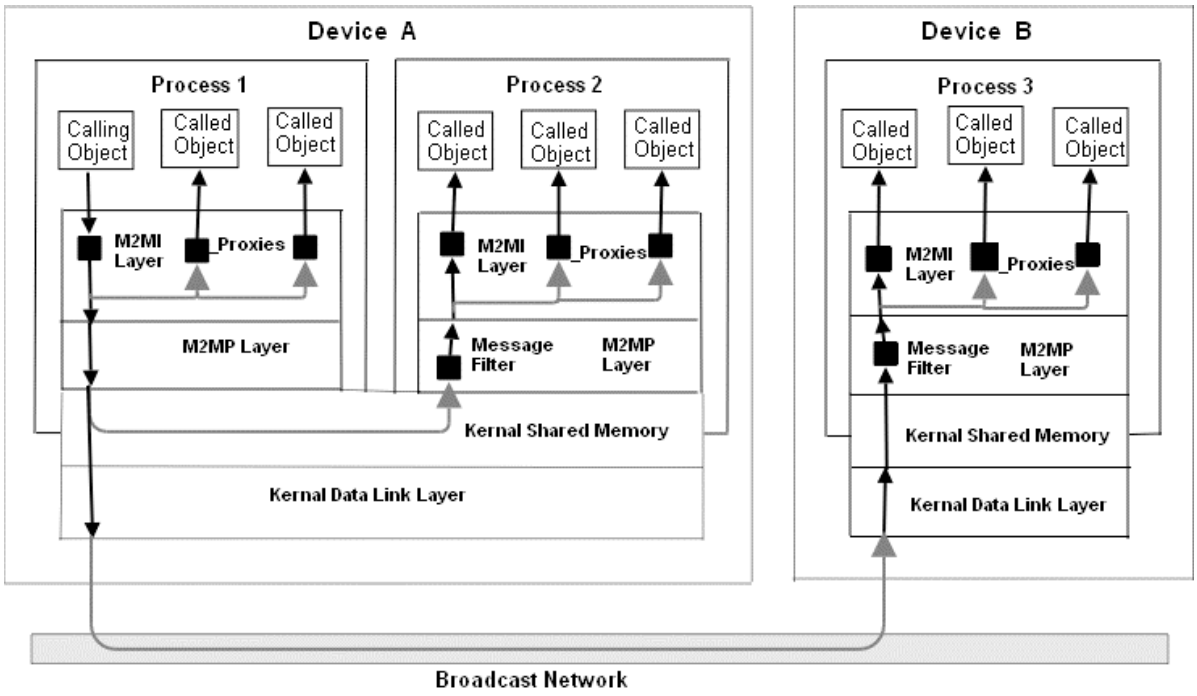


Figure 4: M2MP & M2MI Architecture [3]

4.3 Project Architecture

As discussed above, the main constraint with existing anhinga architecture is that two nodes separated by geographically dispersed network can not communicate with each other. They have to be within each others broadcast range in order to communicate. Figure 5 shows existing anhinga architecture along with the new

DSR Router process. Just like all other M2MP processes, DSR Layer receives all the messages received by this node via the M2MP Daemon Channel. Unlike a normal M2MP process, where it filters out messages based on the message prefix, the routing layer processes each message it receives. Once a message is received, it is handed off to Routing Engine.

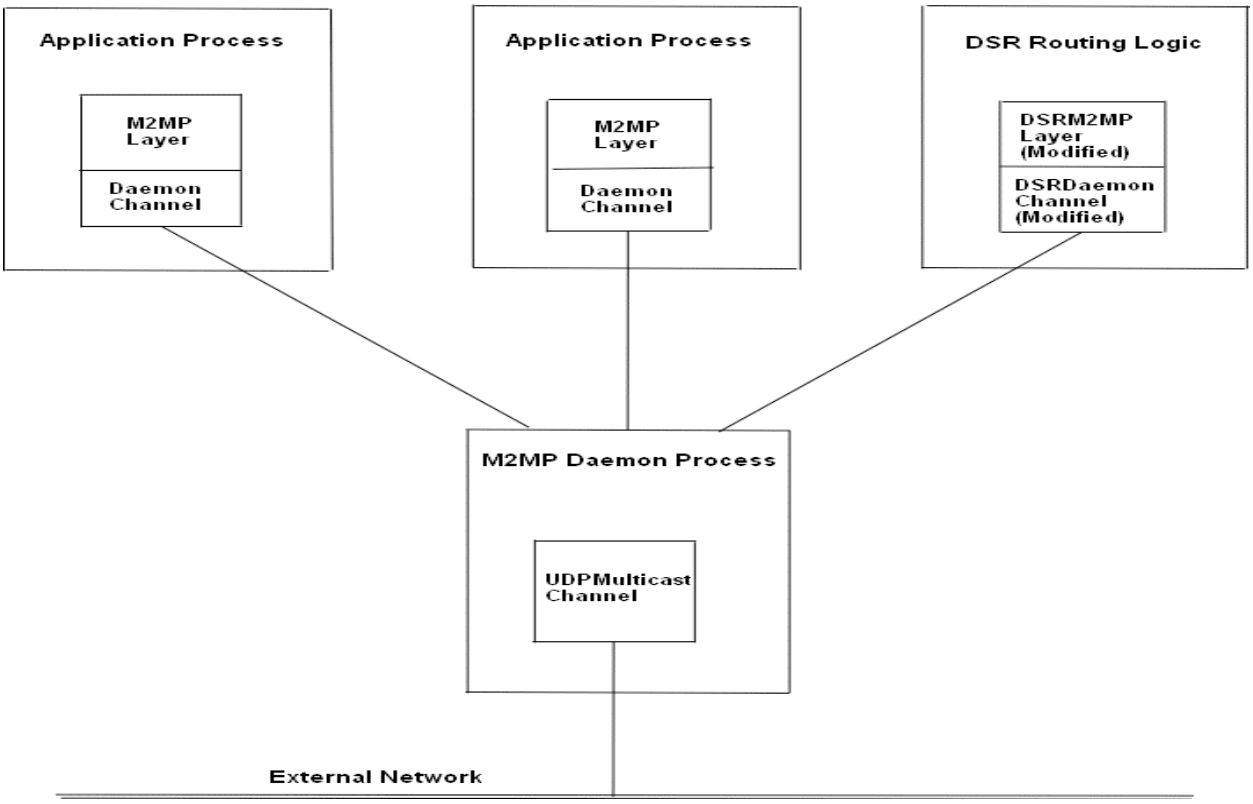


Figure 5: General Architecture of Route in Anhinga

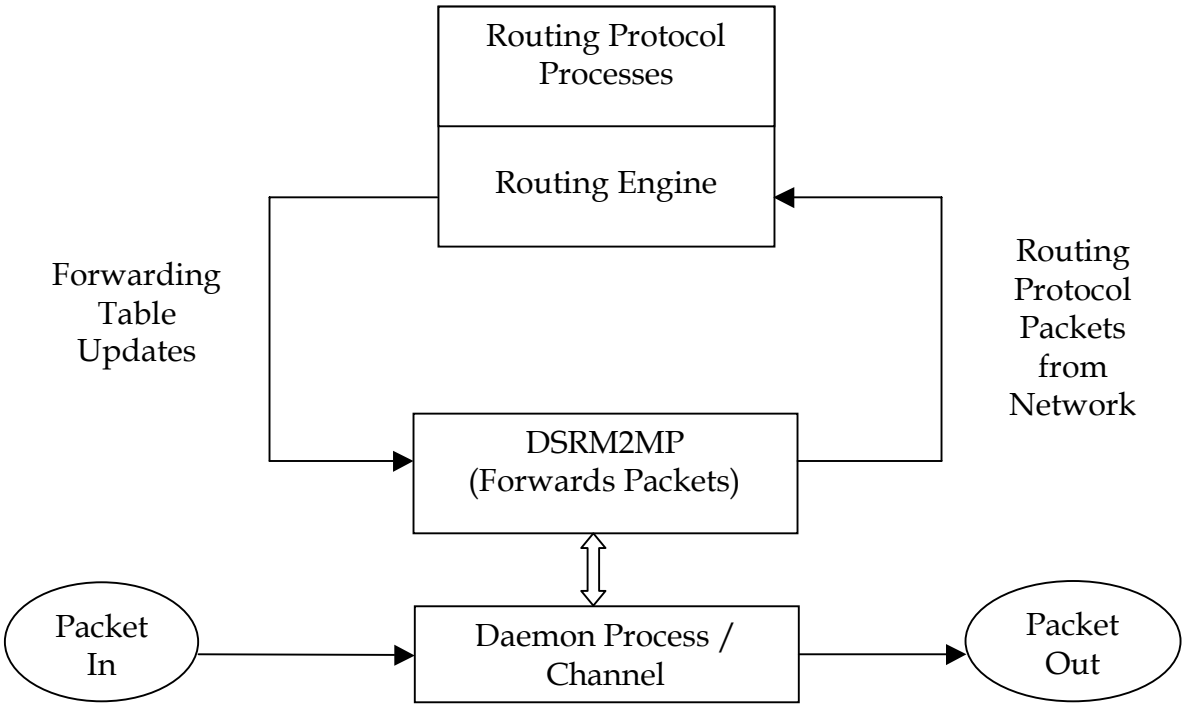


Figure 6: Routing Layer Architecture [16]

As shown in Figure 6 above, once the routing engine receives routing protocol packets from outside, it process it as per protocol rules, updates local tables and rebroadcasts them back if necessary. Figure 7 shows all the class level details of routing layer and what routing rules are implemented.

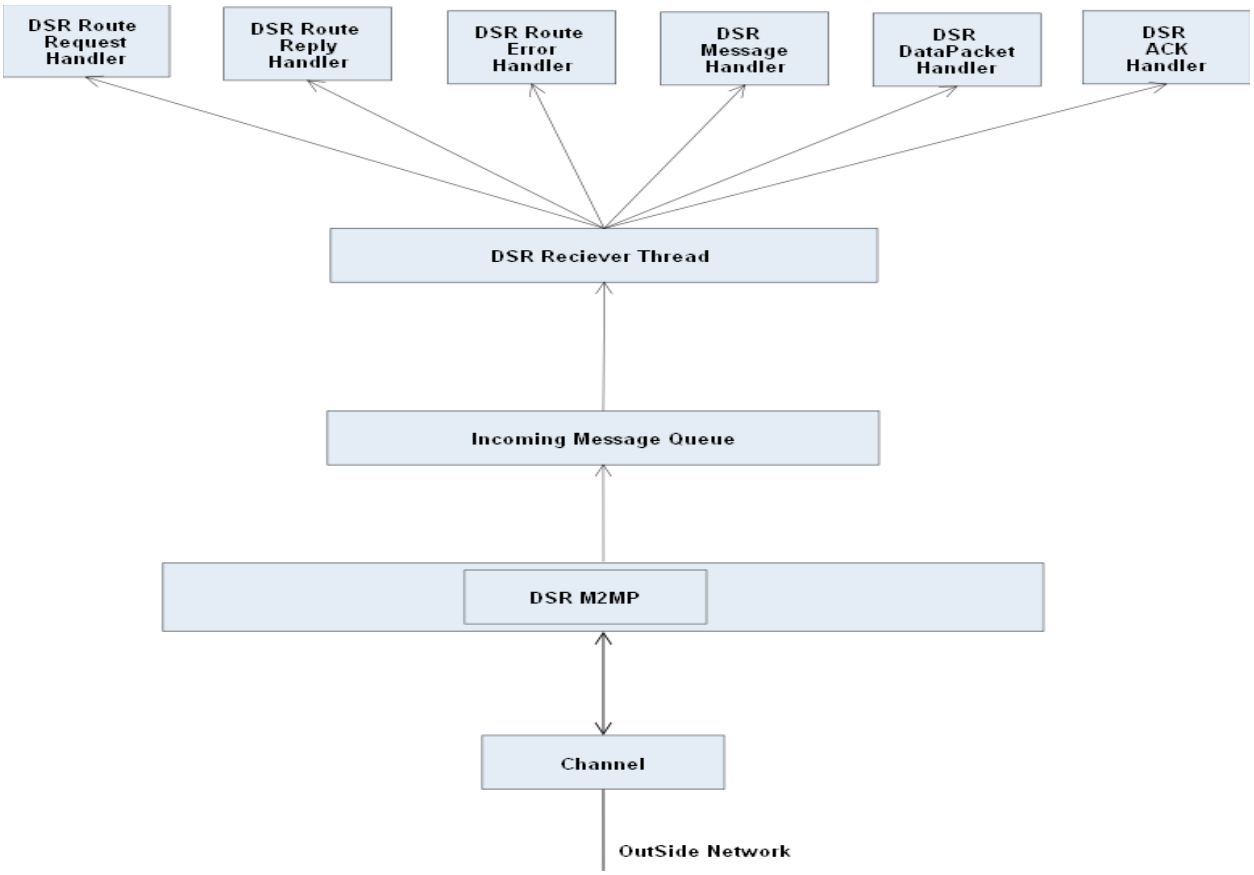
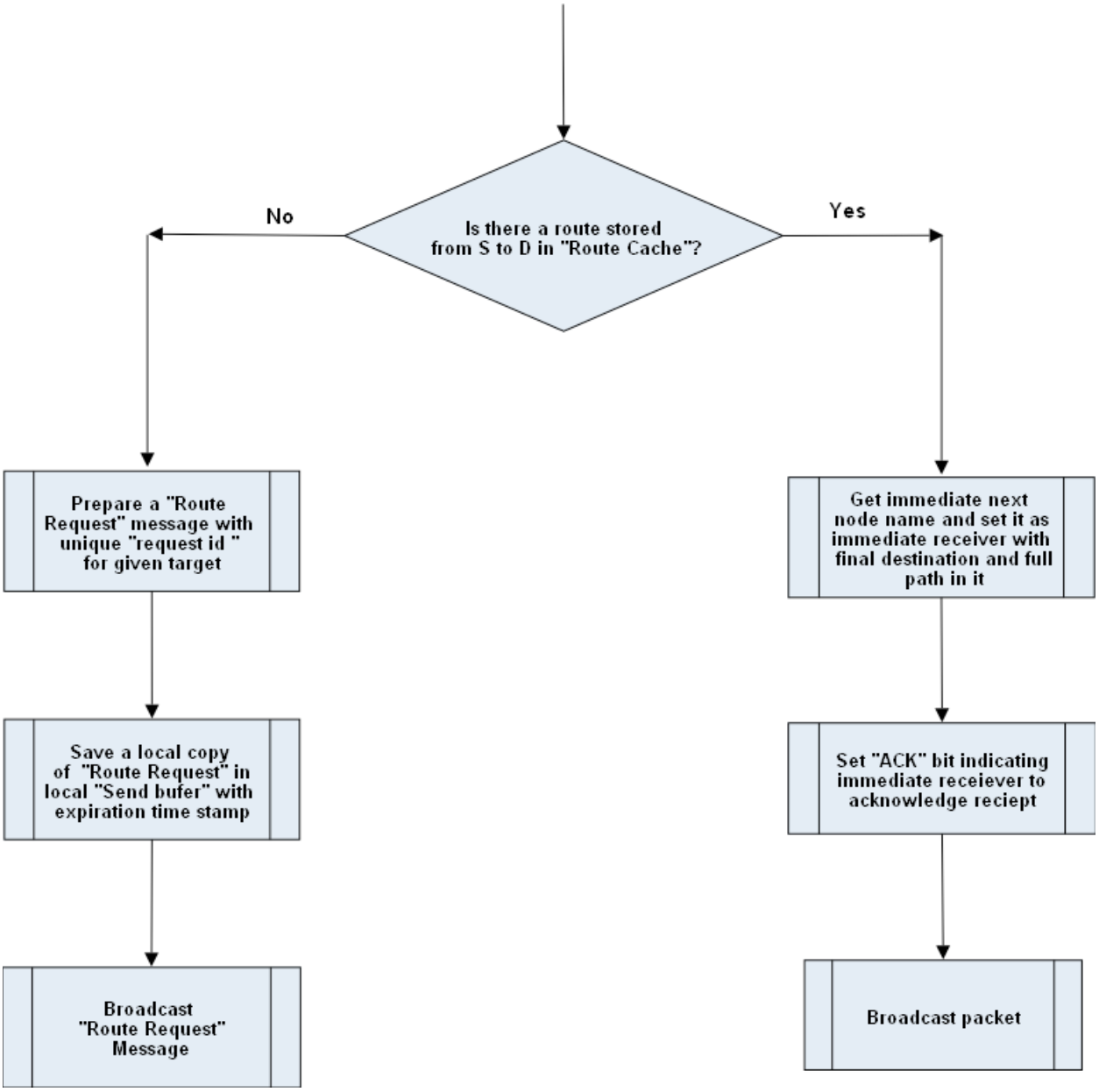


Figure 7: Class Level Details of Routing Engine

4.4 Routing Algorithm

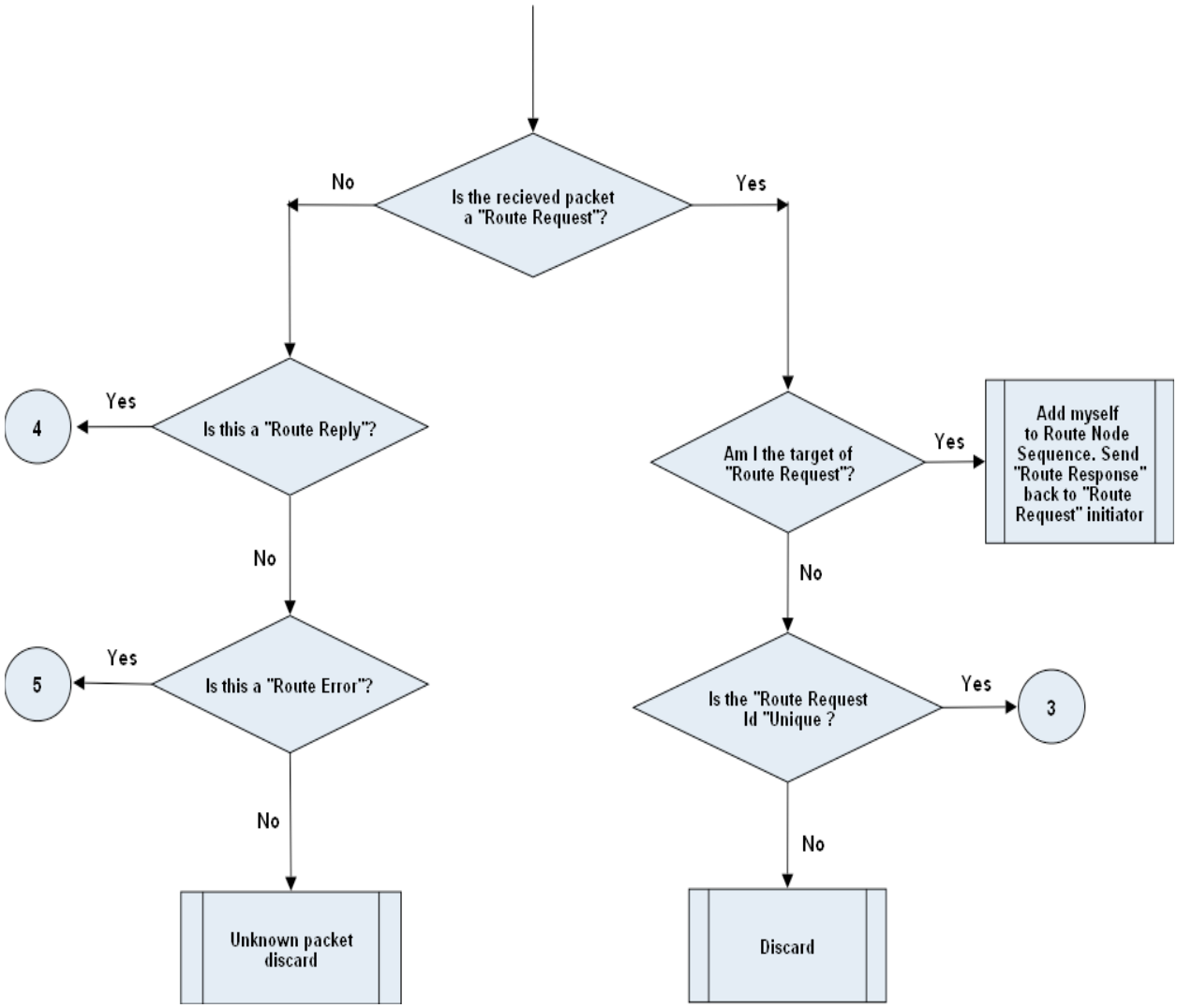
This section describes general working of the protocol in terms of flow charts. Very similar logic is programmed in the routing engine. When Source S wants to send a packet to Destination D then flow of the process to discovered a Route are as:-



In summary when a node wants to send a targeted message to a particular destination, it checks its route cache for answers. If no route found a route request is initiated.

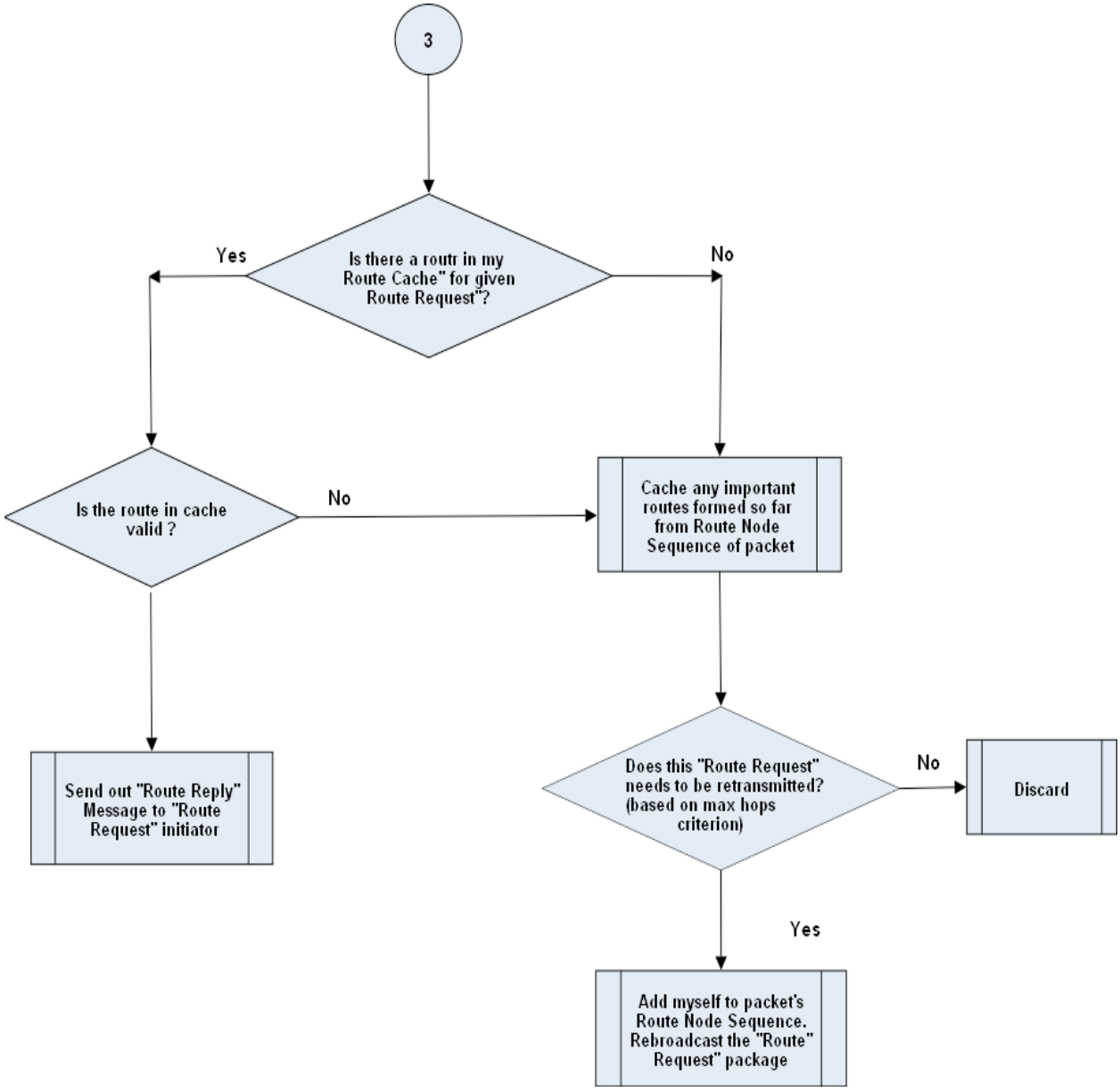
a) Decision making process about a packet received

When a node receives a packet it checks if this is old style M2MP packet. If so it checks the hop count for that message and makes a decision about rebroadcasting based on that. For all other DSR specific messages it follows the logic described below:



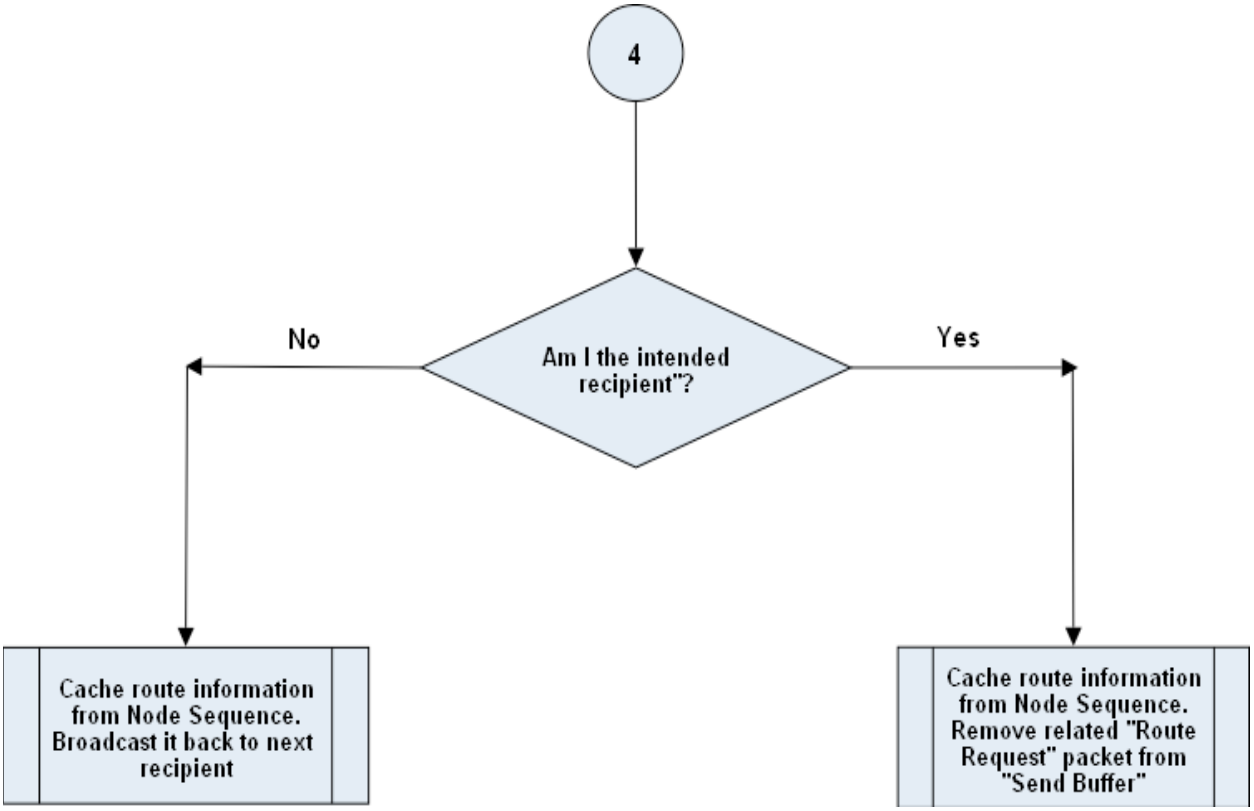
b) “Route Request” packet received

For a unique route requests received from node other then self, a route is checked in local cache. If found a route reply is sent otherwise this node adds itself to source route and rebroadcasts it based on max hop count value.



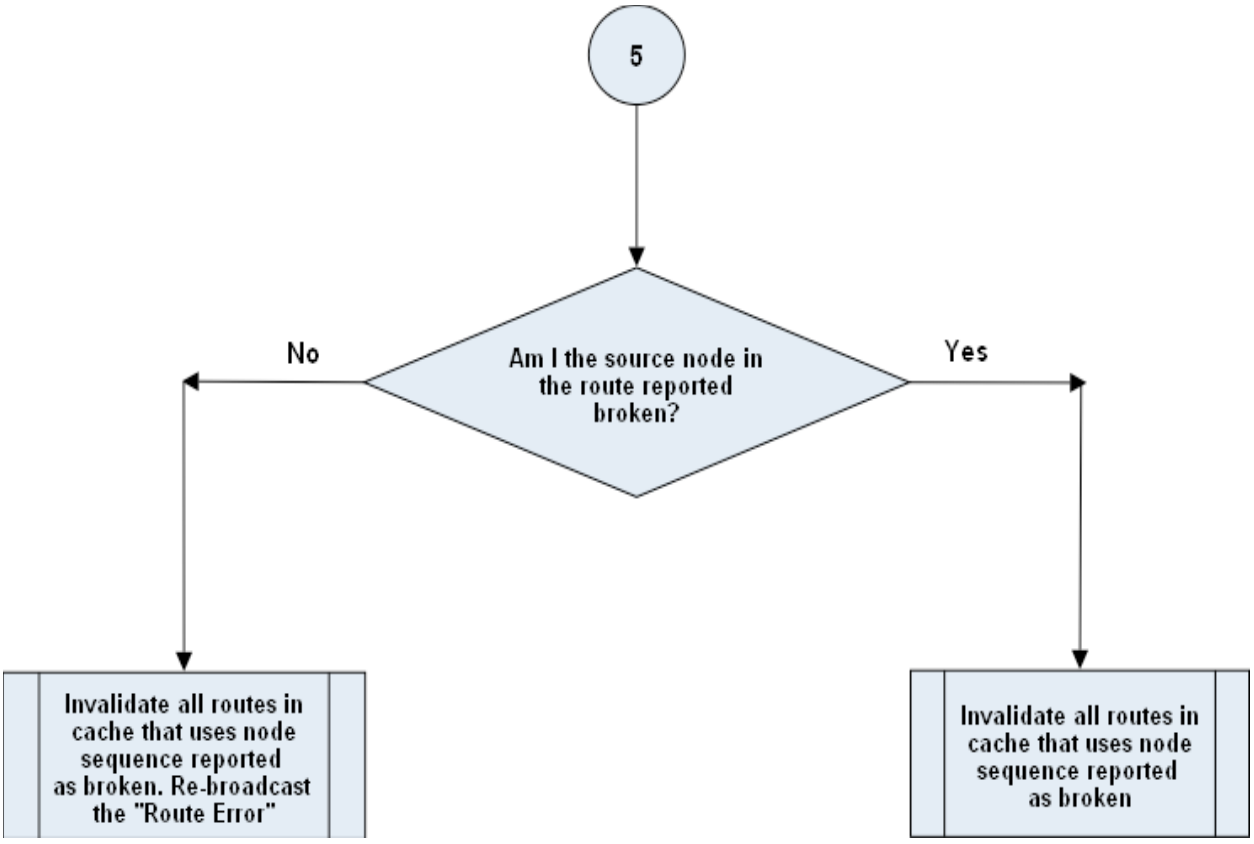
c) "Route Reply" packet received

For a new route reply message received from node other then self, the source route is extracted out of the message. After that if this node was the intermediate intended receiver of this message then it rebroadcasts it back for final destination.



d) "Route Error" packet received

Route error packets are used to clean up faulty route from cache. All the routes in route cache are checked for given sequence of broken route. Based on max hop count criterion a rebroadcast is done.


5
Am I the source node in the route reported broken?
No
Yes
Invalidate all routes in cache that uses node sequence reported as broken. Re-broadcast the "Route Error"
Invalidate all routes in cache that uses node sequence reported as broken

5. Design Specifications

This section discusses different packages in the API of the framework. As explained in the architecture there are four major layers / components in the implementation, DSR layer, M2MP layer, M2MI layer and application layer. The list of packages and their description is as follows:

a) edu.rit.m2mp.dsr – This package is the routing layer. It is the brain of routing mechanism.

b) edu.rit.m2mp – This is the original anhinga layer that provides Many-to-Many protocol implementation.

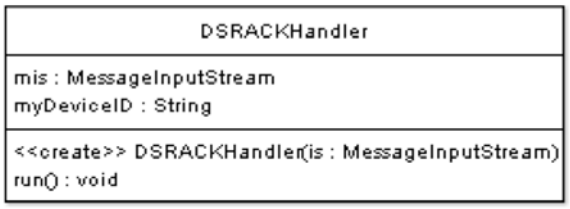
c) edu.rit.m2mi – This is the Many-to-Many invocation layer running on top of M2MP.

d) edu.rit.m2mi.chat2 – This is the test application written on top of M2MI to test the routing mechanism.

5.1 Class Description package edu.rit.m2mp.dsr

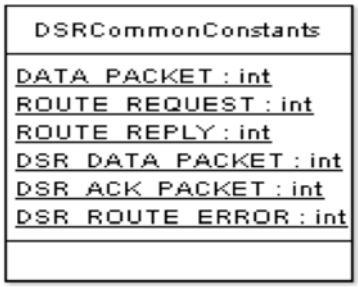
(i) class DSRACKHandler

This class processes incoming DSR ACK message. It first checks if this message ID is already processed. If yes then discards it. Otherwise adds it to recently processed messages to avoid re-processing duplicate message. If this message was broadcasted by this node itself then discard it. If this node is final intended receiver of this message, the node is processing it and this router engine do not need to worry about it. Otherwise update the hop count and rebroadcast this packet.



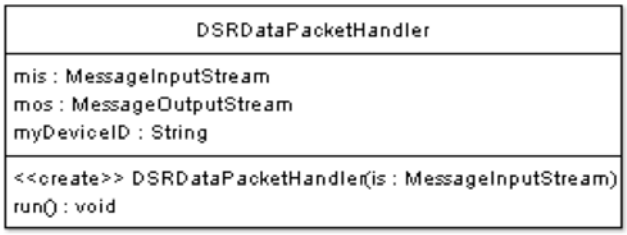
(ii) class DSRCommonConstants

This class defines common constants used throughout the DSR Router engine. It define generic old style M2MP/M2MI Data Packet constant (DATA_PACKET), DSR Route Request Packet constant (ROUTE_REQUEST), DSR Route Reply packet constant (ROUTE_REPLY), DSR Data Packet constant (DSR_DATA_PACKET), DSR Acknowledge Packet constant (DSR_ACK_PACKET), DSR Route Error Packet constant (DSR_ROUTE_ERROR).



(iii) class DSRDataPacketHandler

This class processes incoming DSR Data message. If this is a duplicate message discard it. If this node is the final destination for this data packet, send out an acknowledge packet, if requested with the data packet. If this node was the original sender of this data packet, discard it. Otherwise updates address of next hop, maximum hop count and rebroadcast it.



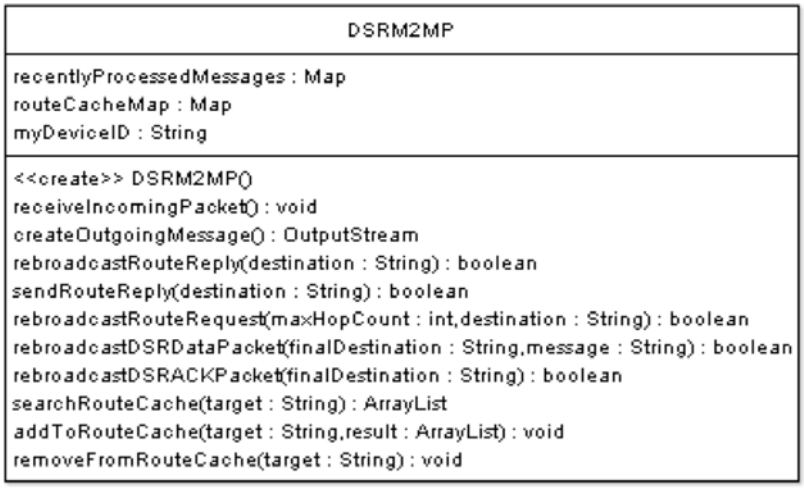
(iv) class DSRException

DSRException indicates that the DSR Layer failed because a certain Runtime values.

(v) class DSRM2MP

Class DSRM2MP does same functionality for DSR Routing engine as regular M2MP does for classical anhinga. It provides DSR M2MP Layer. It holds important data structures like Route Cache Map that contains all valid routes known to this node. It also holds recently processed messages map, which is used to eliminate duplicate messages. This class is the entry point for any incoming message. Upon receiving any incoming message this if its a new non-duplicate message, a new MessageInputStream is created and message is queued in incoming message queue. The DSRReceiverThread constantly keeps scanning this message queue and works on each incoming message.

This class also provides interface between outgoing channel and DSR routing engine. This is accomplished by exposing methods like createOutgoingMessage, rebroadcastRouteReply, sendRouteReply, rebroadcastRouteRequest, rebroadcastDSRDataPacket, rebroadcastDSRACKPacket, searchRouteCache and addToRouteCache. Each of this method is extensively used by individual message type processor like DSRRouteRequestHandler, DSRRouteReplyHandler etc.

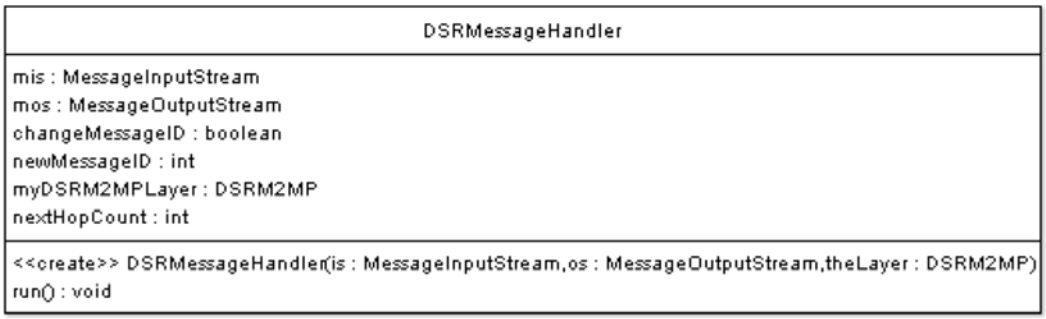


(vi) class DSRM2MPProperties

Class DSRM2MPProperties provides access to properties that are used to configure the DSR Layer.

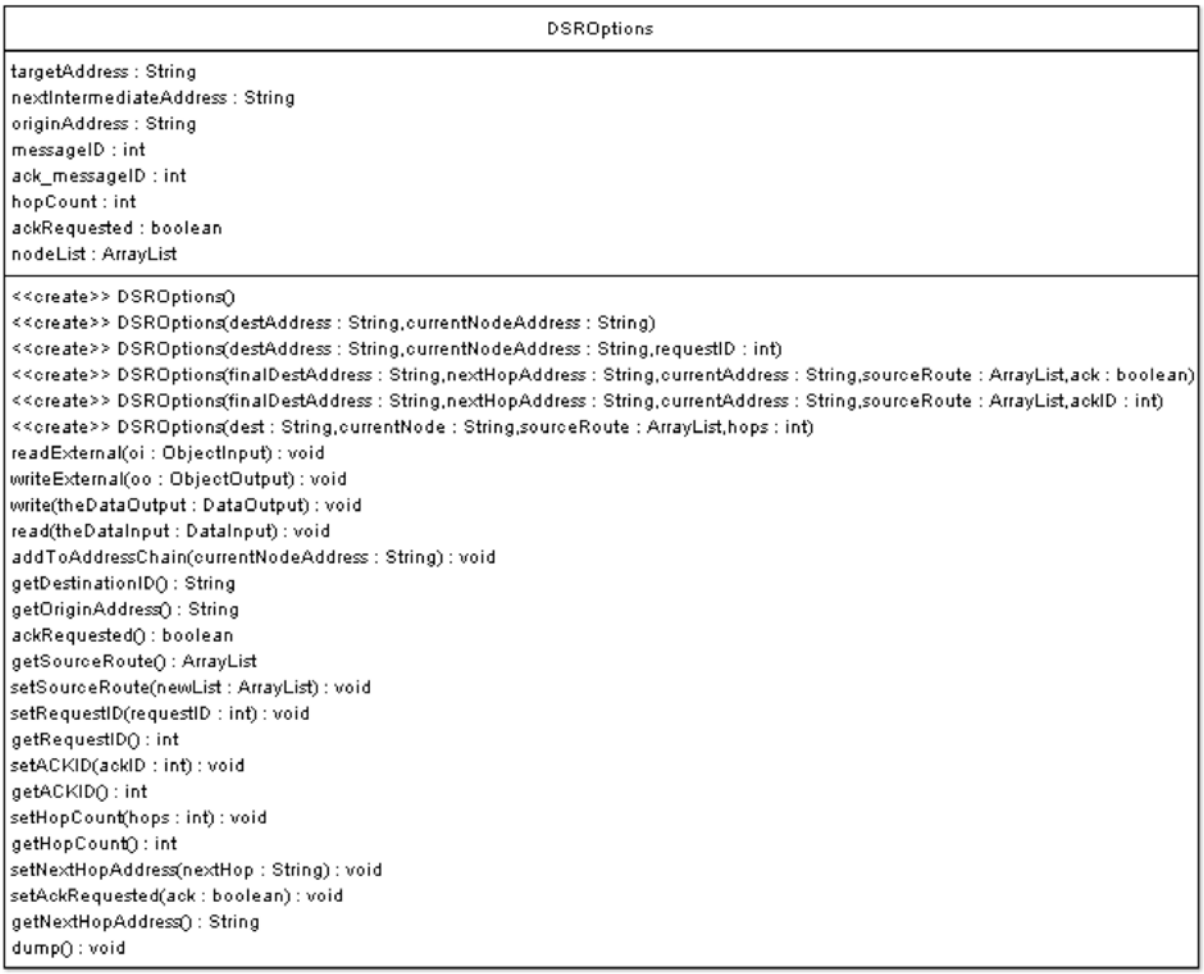
(vii) class DSRMessageHandler

This class processes traditional old style M2MP / M2MI Packets. If it is a duplicate message, discards it. Otherwise if this packet has not been re-broadcasted maximum requested time, it will increase rebroadcast count and rebroadcast the message.



(viii) class DSROptions

This class defines options attached each DSR packet. This option packet when attached to traditional Anhinga packet defines type of DSR packet. This class also provides lot of useful DSR message related functionality like, next intermediate address, final target destination, message origin address.



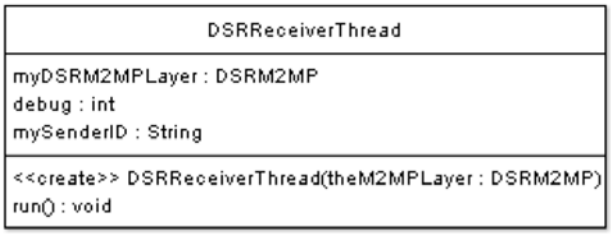
(ix) class DSRReceiverThread

Class DSRReceiverThread provides a thread for receiving and processing incoming messages. This receiver thread constantly scans incoming message queue. As soon as a message is received, this class finds out type of the message and hands it off to appropriate handler. For e.g. if an incoming message type is DSR Route Reply then a

DSRRouteReplyHandler object is created and this MessageInputStream is passed to it to process the message appropriately.

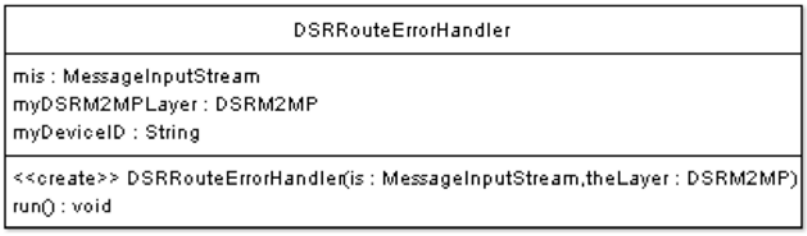
If the M2MI property "edu.rit.m2mi.debug.ReceiverThread" is 1 or higher, then whenever an exception is thrown while receiving or processing an incoming M2MI invocation message, the receiver thread will print the exception stack trace on the standard error stream. The receiver thread will continue running, however.

If the M2MI property "edu.rit.m2mi.debug.ReceiverThread" is 2 or higher, then the receiver thread will print a message on the standard error stream whenever the receiver thread receives an incoming M2MI invocation message.



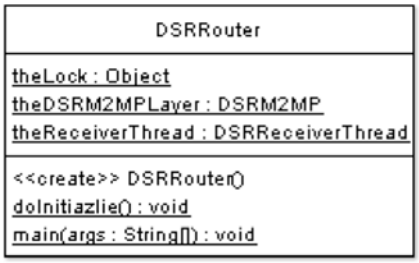
(x) class DSRRouteErrorHandler

This class processes incoming DSR Route Error message. It first checks if this message is already processed. If yes then discards it. Otherwise based on the faulty route reported, route table is updated.



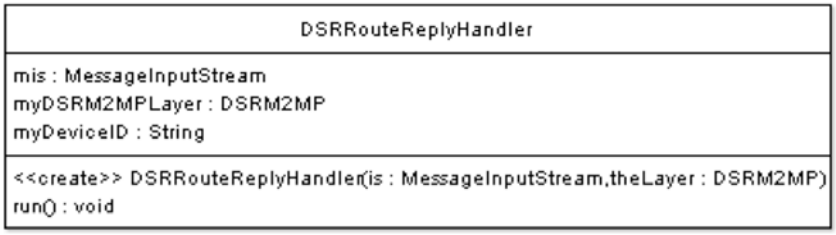
(xi) class DSRRouter

This class is the starting point of DSR Router. It initializes the DSRM2MP Layer and starts the receiver thread by passing it the DSRM2MP layer object. This class contains the main method that starts the DSR Routing engine and keeps it alive till killed.



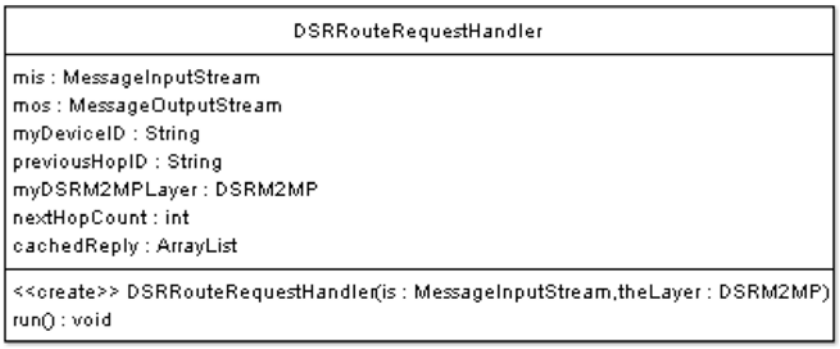
(xii) class DSRRouteReplyHandler

This class processes incoming DSR Route Reply message. For non-duplicate messages it will add the route to its own route cache if this node is part of the source route of route reply message. The route reply message is rebroadcast if this node was not original sender and the next intended receiver for this route reply was this node.



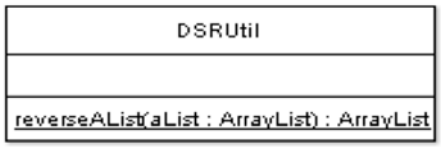
(xiii) class DSRRouteRequestHandler

This class processes incoming DSR Route Request message. For a new request, if this node is the sender then discards the request. If this node is the final destination for which the route request is initiated, then a route reply is sent. Otherwise if there is a route in local route cache to the destination, an appropriate route reply is sent. As last resort, if the maximum hop count for the route request is not exceeded then route request is rebroadcasted.



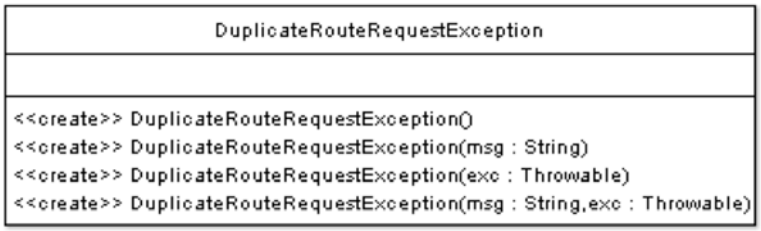
(xiv) class DSRUtil

This class provides common useful functionality across the DSR Routing layer.



(xv) class DuplicateRouteRequestException

Class DuplicateRouteRequestException indicates that the upper layer is trying to initiate a duplicate route request.



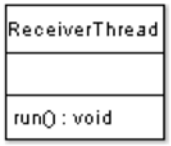
(xvi) interface DSRServices

This interface defines the contract for receiving services provided by DSR Routing layer.

5.2 Modification in package edu.rit.m2mp

(i) class M2MP

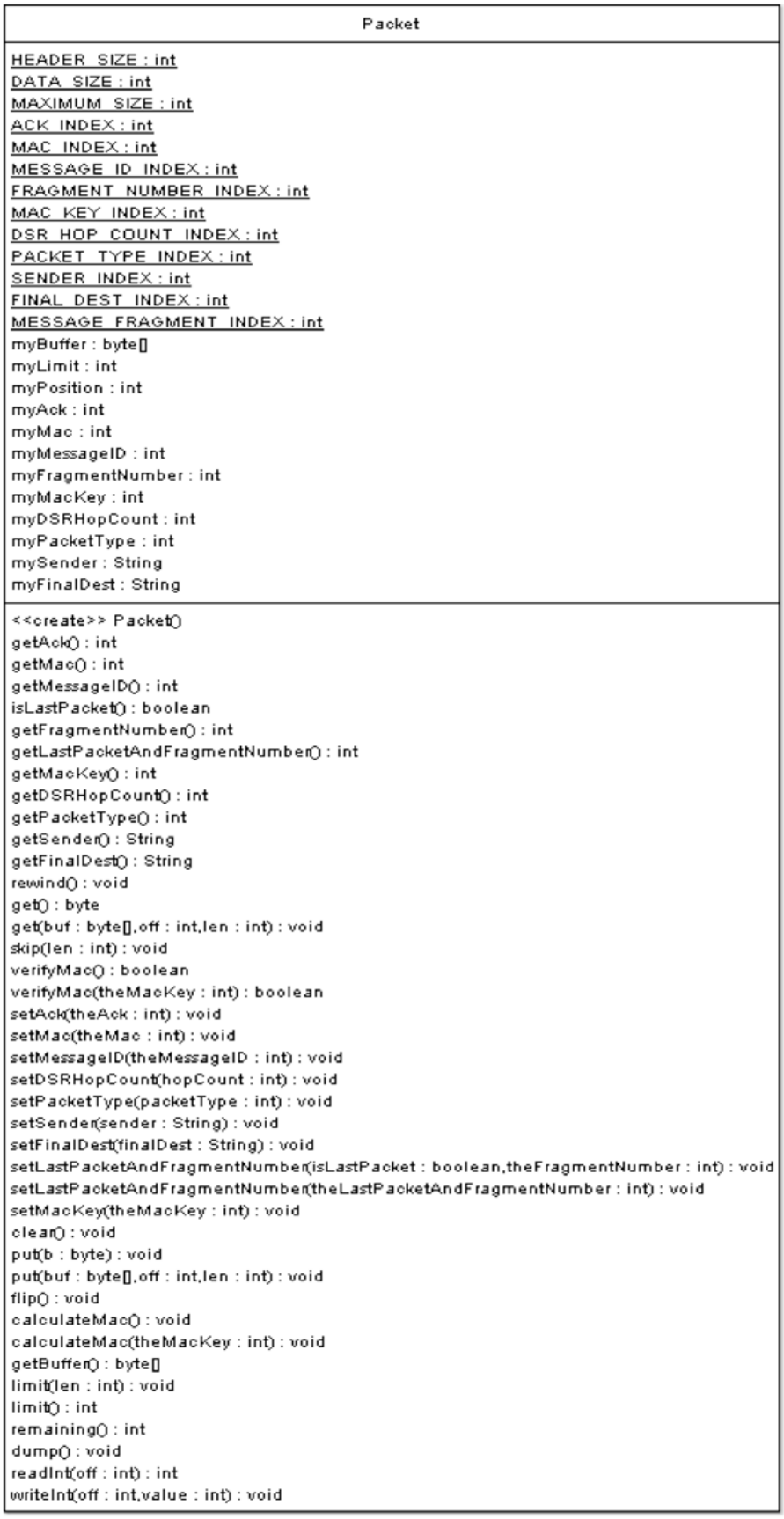
The original M2MP class has been updated to make it aware of DSR data packet aware. A method "sendRouteRequest" is added so that M2MI like upper layer can take advantage of DSR Routing mechanism and request a route to some destination. It is also modified to add a listener for DSR routing related messages by upper layer like M2MI using method "addRouteReplyReceiver". To allow upper layers to send DSR specific packets other methods like, "sendDSRDataPacket", "sendDSRACKPacket", "sendDSRErrorPacket" have also been added. Method "receiveIncomingPacket" inside M2MP class is also modified to make it aware of DSR Routing related specific message processing. This method based on the type of incoming message sets the MessageInputStream's message type which is later used by router and other layers to process a particular message.



(ii) Class Packet

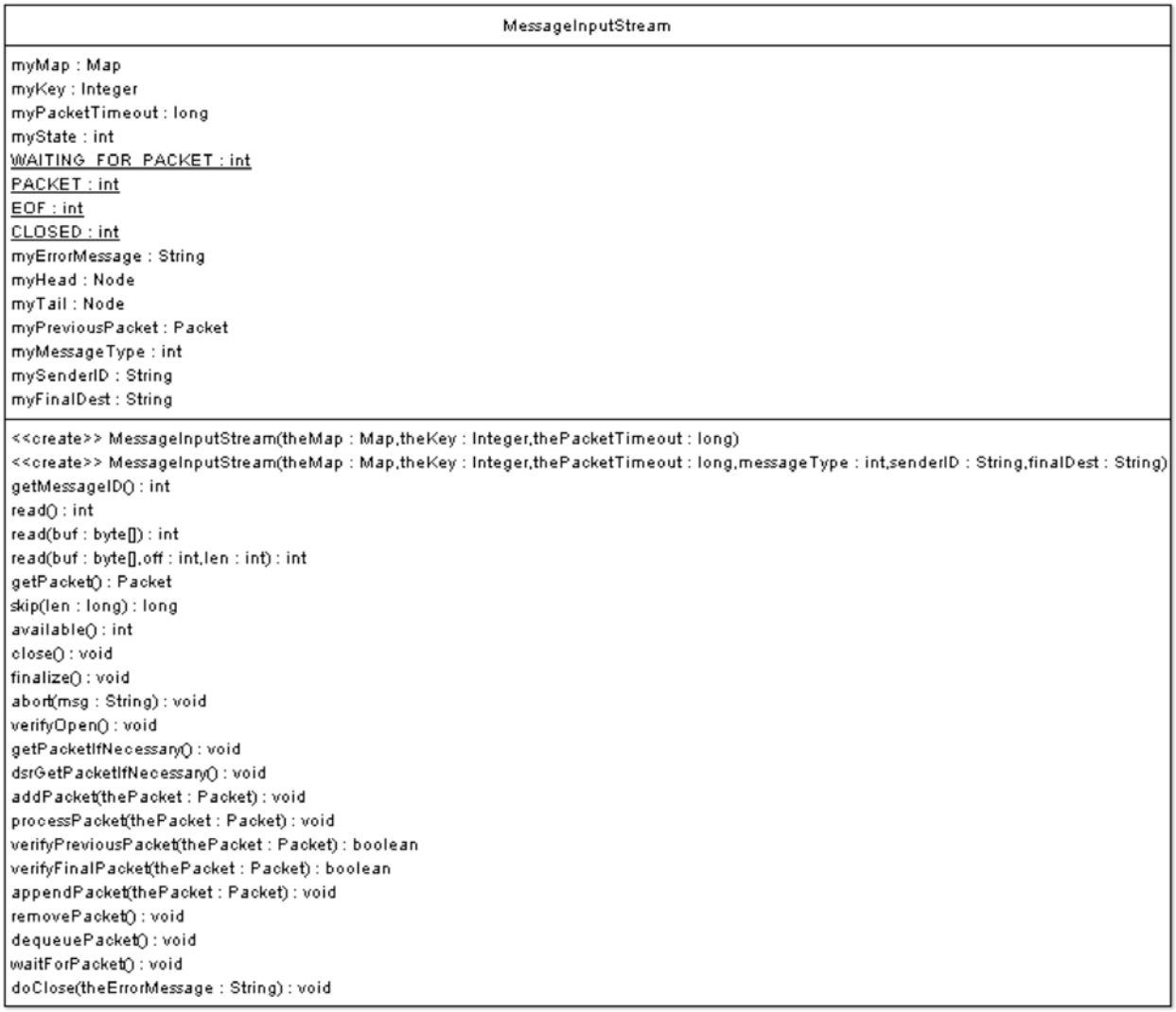
The Packet class has been updated extensively to accommodate DSR related data in each packet. Major fields added are,

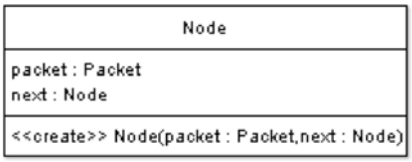
A) Packet Type: Type of the packet. Value to this is assigned from DSRCommonConstants.
B) Hop Count: Maximum # of allowed rebroadcasts for this packet.
C) Sender ID: Who originally sent out this packet
D) Target ID: Next intended receiver of this packet
E) Destination ID: Final destination for this packet


Packet
HEADER_SIZE : int
DATA_SIZE : int
MAXIMUM_SIZE : int
ACK_INDEX : int
MAC_INDEX : int
MESSAGE_ID_INDEX : int
FRAGMENT_NUMBER_INDEX : int
MAC_KEY_INDEX : int
DSR_HOP_COUNT_INDEX : int
PACKET_TYPE_INDEX : int
SENDER_INDEX : int
FINAL_DEST_INDEX : int
MESSAGE_FRAGMENT_INDEX : int
myBuffer : byte[]
myLimit : int
myPosition : int
myAck : int
myMac : int
myMessageID : int
myFragmentNumber : int
myMacKey : int
myDSRHopCount : int
myPacketType : int
mySender : String
myFinalDest : String
<<create>> Packet()
getAck() : int
getMac() : int
getMessageID() : int
isLastPacket() : boolean
getFragmentNumber() : int
getLastPacketAndFragmentNumber() : int
getMacKey() : int
getDSRHopCount() : int
getPacketType() : int
getSender() : String
getFinalDest() : String
rewind() : void
get() : byte
get(buf : byte[],off : int,len : int) : void
skip(len : int) : void
verifyMac() : boolean
verifyMac(theMacKey : int) : boolean
setAck(theAck : int) : void
setMac(theMac : int) : void
setMessageID(theMessageID : int) : void
setDSRHopCount(hopCount : int) : void
setPacketType(packetType : int) : void
setSender(sender : String) : void
setFinalDest(finalDest : String) : void
setLastPacketAndFragmentNumber(isLastPacket : boolean,theFragmentNumber : int) : void
setLastPacketAndFragmentNumber(theLastPacketAndFragmentNumber : int) : void
setMacKey(theMacKey : int) : void
clear() : void
put(b : byte) : void
put(buf : byte[],off : int,len : int) : void
flip() : void
calculateMac() : void
calculateMac(theMacKey : int) : void
getBuffer() : byte[]
limit(len : int) : void
limit() : int
remaining() : int
dump() : void
readInt(off : int) : int
writeInt(off : int,value : int) : void

(iii) class MessageInputStream

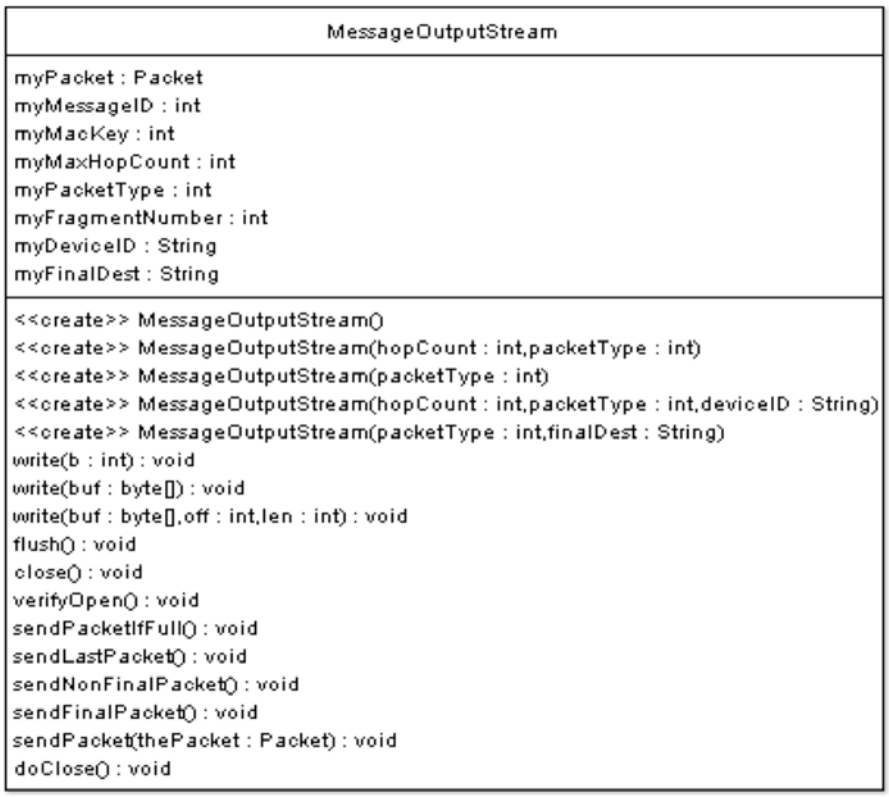
MessageInputStream class was modified to add various DSR Specific features into it, so that the processing of each message is streamlined. Each message input stream object has type of message it is carrying, which determines type of message handler object used to process this message. Various methods modified and constructors added.





(iv) class MessageOutputStream

Changes necessary based on changes in each Packet. Various methods modified to reflect changes related to format of each packet. Also various constructors added to accommodate specific type of messages.



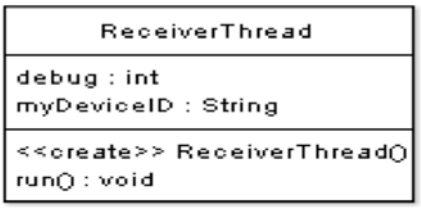
5.3 Modification in package edu.rit.m2mi

(i) class M2MI

Various methods like "dsrServiceReceiver" were added to M2MI class. This was done to allow abstraction between applications working on top of M2MI and M2MP layer. In future there can be better exposed API either from M2MI or from M2MP to address this issue more cleanly.

(ii) class ReceiverThread

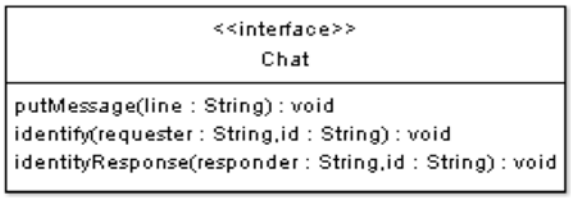
Receiver thread was modified to provide extra facilities to application layer working on top of M2MI. These changes were in regards with processing DSR specific messages.



5.4 Package edu.rit.m2mi.chat2

(i) interface Chat

Interface Chat specifies the interface for a rudimentary M2MI-based chat object. It is same as in original ChatDemo1 application except that identityResponse() is added to accommodate display of neighbors list.



(ii) interface ChatFrameListener

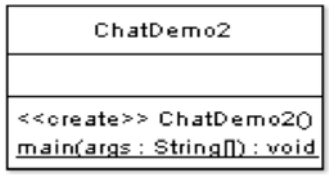
Interface ChatFrameListener specifies the interface for an object that is triggered when the user sends a line of text in a link ChatFrame

(iii) interface NeighborsListListener

Interface NeighborsListListener specifies the interface for an object that is triggered when a new neighbor ping is received.

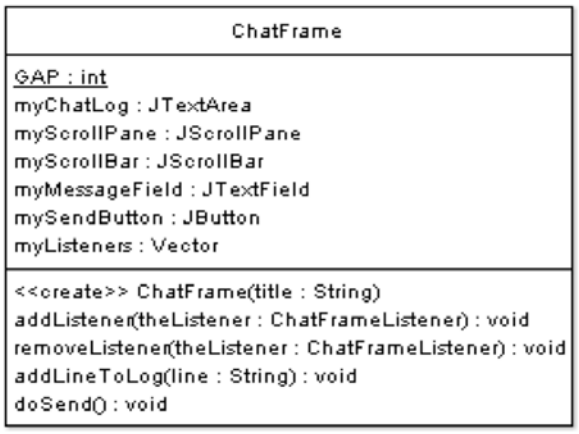
(iv) class ChatDemo2

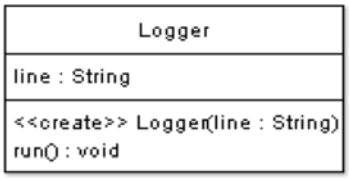
Class ChatDemo2 is a rudimentary M2MI-based chat application. The program displays a simple chat UI that implements interface. As in original ChatDemo1 application when the user sends a line of text in the UI, the line is broadcast to all the chat objects by calling putLine() on an omnihandle for interface Chat. When each chat object receives a putLine() invocation, it displays the line of text in the chat log in its UI.



(v) class ChatFrame

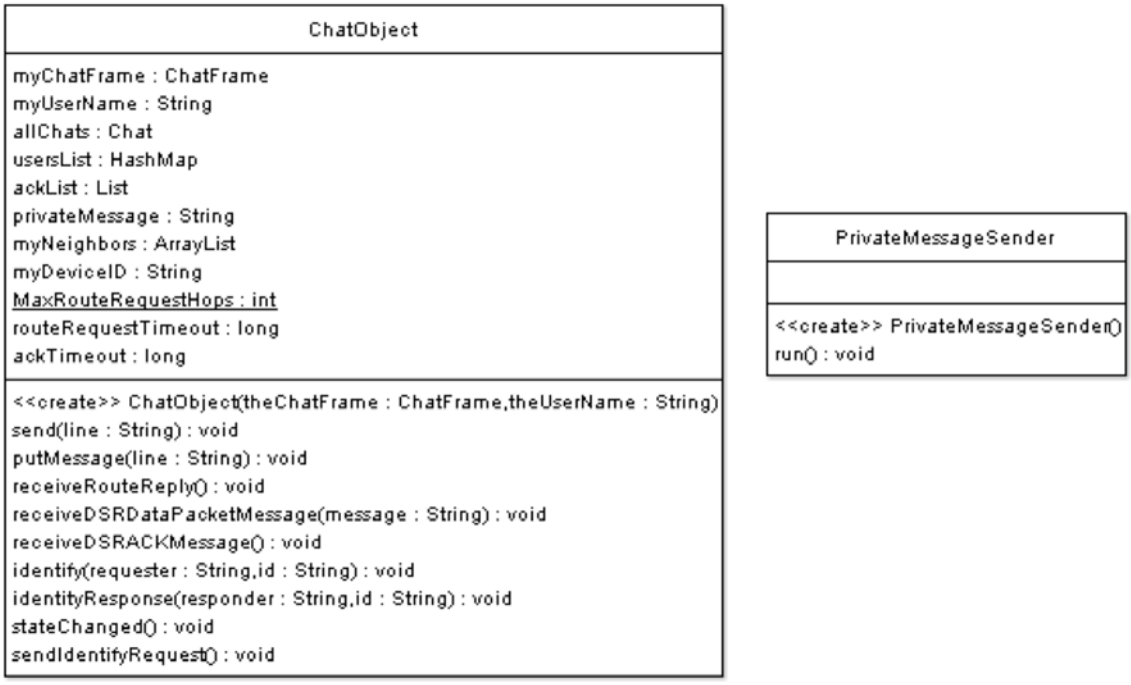
Class ChatFrame encapsulates the UI for a rudimentary M2MI-based chat application. This class also has an inner class ChatFrame.Logger, which provides an object that adds one line of text to the chat log display. It is encapsulated as a separate class so adding the line of text can be done asynchronously by the Swing UI thread. This class also handles the extra logic for private chat.





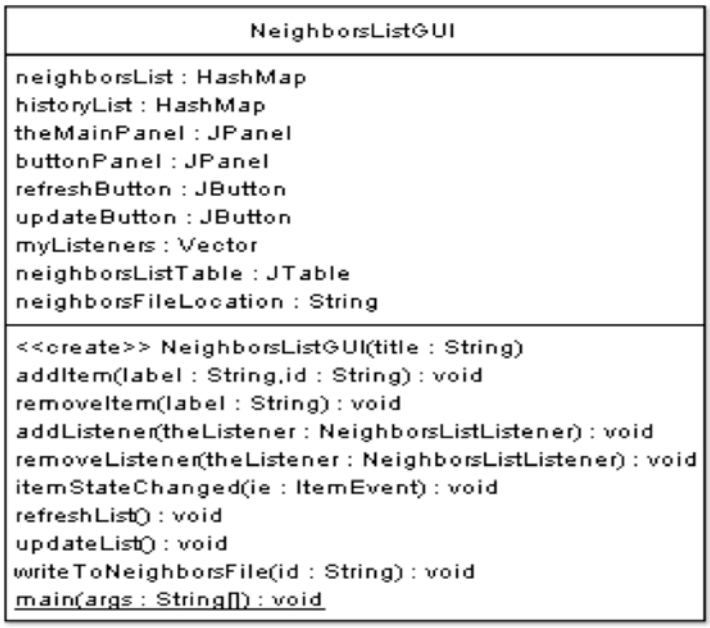
(vi) class ChatObject

Class ChatObject is exact same as in chat1 package. Except that it adds extra functions to manage route request and route discoveries received from lower levels.



(vii) class NeighborsListGUI

This class provides a frame that displays all the nodes that are alive in the network. It shows names of all the nodes to which a private message can be sent.


NeighborsListGUI
neighborsList : HashMap
historyList : HashMap
theMainPanel : JPanel
buttonPanel : JPanel
refreshButton : JButton
updateButton : JButton
myListeners : Vector
neighborsListTable : JTable
neighborsFileLocation : String
<<create>> NeighborsListGUI(title : String)
addItem(label : String,id : String) : void
removeItem(label : String) : void
addListener(theListener : NeighborsListListener) : void
removeListener(theListener : NeighborsListListener) : void
itemStateChanged(ie : ItemEvent) : void
refreshList() : void
updateList() : void
writeToNeighborsFile(id : String) : void
main(args : String[]) : void

6 User Guide

This section describes user manual for routing protocol implemented.

6.1 System Requirements

The system is tested on following configuration.

- Sun Solaris OS running on SUN SPARC architecture
- J2SDK 1.4.2
- M2MP/M2MI library version 2004/03/02

System will run on any other OS with java version 1.4.2 or higher running on it. DSR router is note tested for M2MP / M2MI APIs version later then 2004/03/02.

6.2 How to Configure

DSR protocol uses M2MP's configuration file "m2mp.properties. This is a typical M2MP configuration file that includes DSR router settings. Properties highlighted in red box are DSR specific.

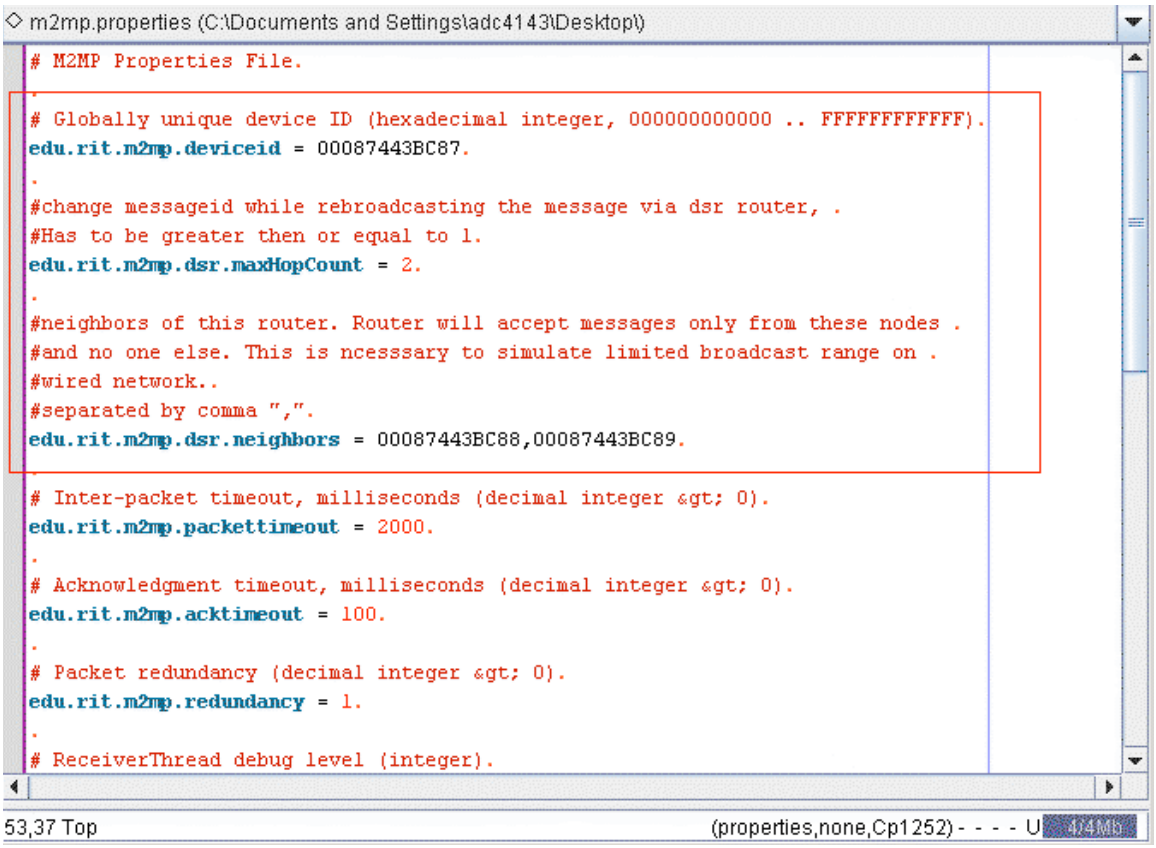


Figure 8: Configuration file (m2mp.properties)

For specifics about configuring M2MP layer please refer to respective documentation at,

http://www.cs.rit.edu/~anhinga/m2miapi20040302/doc/edu/rit/m2mp/package-summary.html#configuring

Following three properties are very important in deciding behavior of DSR router.

- **edu.rit.m2mp.deviceid –** The globally unique device ID of the device on which the M2MP Layer is running. It is same as mentioned in configuration for M2MP layer. It must be a 48-bit hexadecimal integer in the range 000000000000 through FFFFFFFFFFFF. Every device in the world must use a different device ID. The suggested value is the Ethernet MAC address of the device's network interface. This should be the same as the device ID in the M2MI properties file. *Be sure to specify a different device ID on each different machine that is running M2MP!* The Same device ID is used by DSR Router

to identify itself with neighbors and to identify its own neighbors for node-to-node communication.

- **edu.rit.m2mp.dsr.maxHopCount = 2 –** Maximum number of network hops up to which this packet will be re-broadcasted. This value is used when hop count is not mentioned programmatically when sending out a message. This is important for backward compatibility. M2MI messages written before DSR Router inclusion will use this property value as default. Value of this property should be 1 or higher.

- **edu.rit.m2mp.dsr.neighbors –** This property value defines neighbors for this given node. This host will accept packets sent by nodes that are in this list. Any packets received from nodes other then in this list are discarded. The neighbors are mentioned using their unique device ID. This property is needed to simulate dispersed network on a wired broadcast network.

6.3 How to Start

6.3.1 Running M2MP Daemon

For DSR router to run successfully, M2MP Layer must be configured to use M2MP Daemon. M2MP Daemon must be started in a separate process before running DSR Router or any other M2MP or M2MI based application. To start M2MP Daemon,

```
>java edu.rit.m2mp.Daemon
```

6.3.2 Running DSR Router

To take advantage of router, DSR Router must be started in a separate process before running any M2MP or M2MI based application. To start DSR Router,

```
>java edu.rit.m2mp.dsr.DSRRouter
```

6.3.3 Running Test Application

After starting Daemon channel and DSR Router in sequence, any M2MP or M2MI based application can be started. To start Chat Demo application, developed as part of this project to test APIs can be started as follows,

```
>java edu.rit.m2mi.chat2.ChatDemo2 <user name>
```

7 Testing and Benchmarking

System was tested in Computer Science lab on Sun Solaris operating system with Java 1.5.0.

7.1 DSR Test Results

A custom chat application was written based on Chat application from Anhinga. The chat application allowed sending messages using a GUI as well as programmatically. At startup of chat application a window shows available neighbors a node can chat with in the network.

Figure 9: Available Neighbors

To broadcast a message to all the nodes in the network, simply type the message in chat window. The message will be re-broadcasted based on the default max hop count mentioned in property file. To send a private message to a specific target, message is prefixed with "pc - <neighbor-name> - ." For private chat if a route is not known to the destination host, a route discovery message is automatically initiated. Once the route is discovered, a targeted data message is sent using the route. Cases where no route is found or a message delivery is failed appropriate messages are displayed by chat application which it receives from DSR layer. A node can receive messages only from its neighbors. These neighbors are provided from that node's configuration file. Please refer to configuration section for further details.

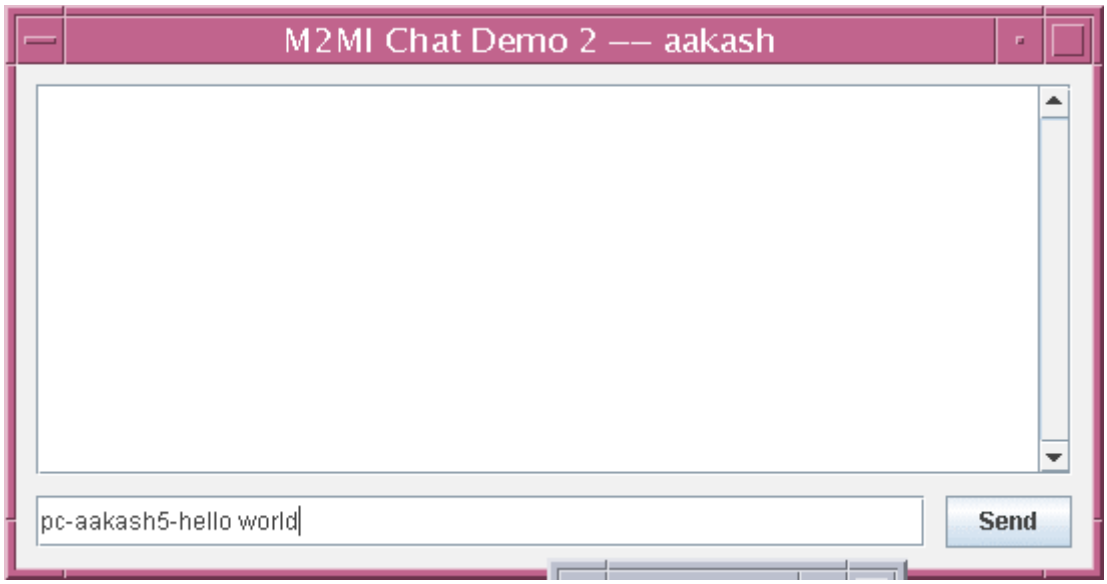


Figure 10: Private Chat

The application was tested on 2 different network topologies. One is the diamond like network topology as shown in Fig 18. Second was linear network as shown in Fig. 19.

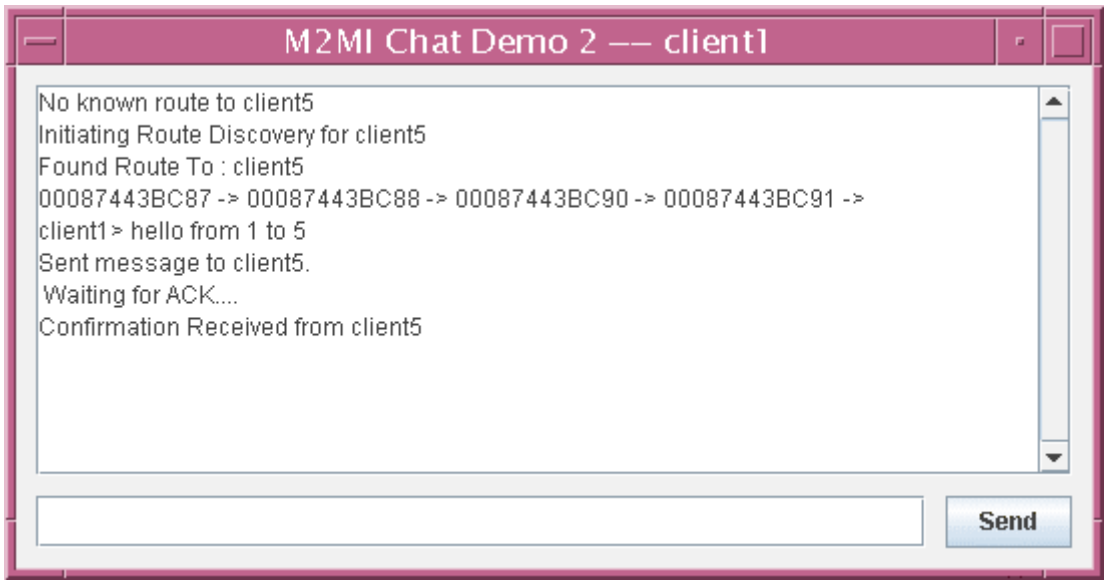


Figure 11: Route Discovery

7.1.1 Diamond Configuration

Figure 15 shows the network configuration of test performed.

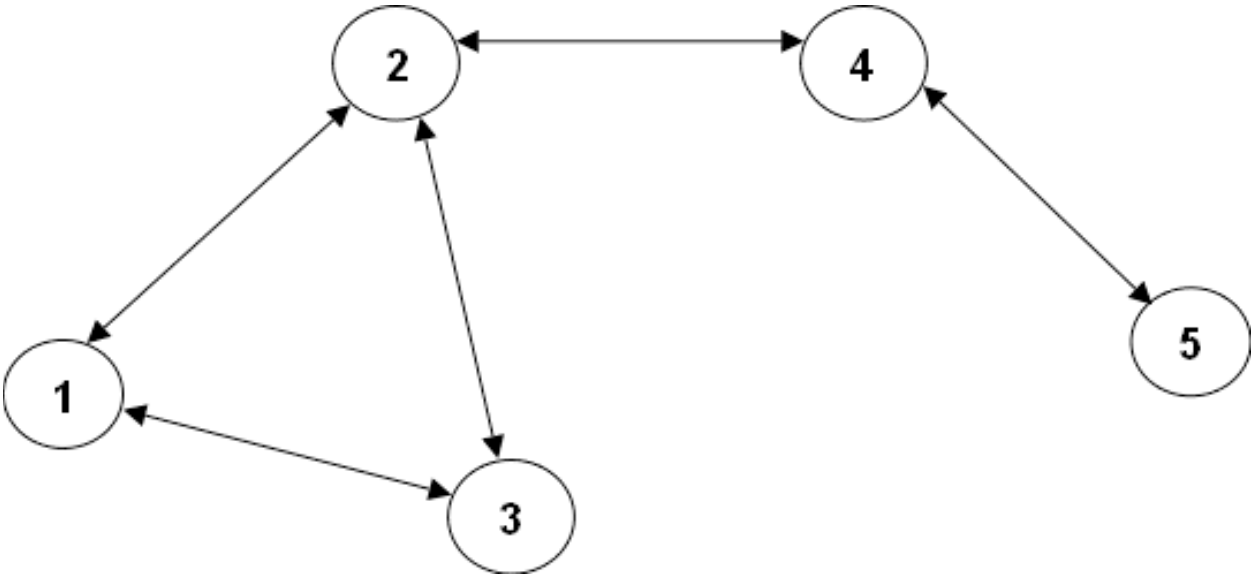


Figure 12: Diamond Configuration

Node #	Node Address	Neighbors	Neighbors Address
1	00087443BC87	2, 3	00087443BC88, 00087443BC89
2	00087443BC88	1,3,4	00087443BC87, 00087443BC89, 00087443BC90
3	00087443BC89	1,2	00087443BC87, 00087443BC88
4	00087443BC90	2,5	00087443BC88, 00087443BC91
5	00087443BC91	4	00087443BC90

Table 1: Network Configuration

Following table shows the test results for diamond like network configuration. It shows time it took for a node to find a route to the destination node in milliseconds. It also shows time it took to send data to that destination and receiving an acknowledgement back from there. Last columns show content of the route cache table for each of client at

the time of message being sent. The route cache is incremental and is valid through out the table from first message up to the last message.

Diamond Like Network

Sender Of Message : Client 1		**Cache content of Each Client at End of Route Request**				
To Client 5		**Client 1**	**Client 2**	**Client 3**	**Client 4**	**Client 5**
Route Search Time (Avg)	5291 ms	87->88->90->91	88->90->91			
Data Transfer Time (Avg)	647 ms					
To Client 4						
Route Search Time (Avg)	355 ms	87->88->90				
Data Transfer Time (Avg)	361 ms					
To, Client 3						
Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	254 ms					
To, Client 2						
Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	168 ms					
Sender Of Message : Client 2						
To Client 5						
Route Search Time (Avg)	0 ms		Used From Cache			
Data Transfer Time (Avg)	344 ms					
To Client 4						
Route Search Time (Avg)	0 ms		Used From			

		Cache				
Data Transfer Time (Avg)	138 ms					
To, **Client 3** Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	121 ms					
To, **Client 1** Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	158 ms					
Sender Of Message : Client 3						
To **Client 5**						
Route Search Time (Avg)	571 ms	Replies From Cache	89->88->90->91			
Data Transfer Time (Avg)	600 ms					
To **Client 4**		Replies From Cache	89->88->90			
Route Search Time (Avg)	401 ms					
Data Transfer Time (Avg)	138 ms					
To, **Client 2** Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	128 ms					
To, **Client 1** Route Search Time (Avg)	0 ms					

Data Transfer Time (Avg)	139 ms					
Sender Of Message : Client 4						
To Client 5						
Route Search Time (Avg)	0 ms				Used From Cache	
Data Transfer Time (Avg)	161 ms					
To Client 3						
Route Search Time (Avg)	338 ms		Replies From Cache		90->88->89	
Data Transfer Time (Avg)	346 ms					
To, Client 2						
Route Search Time (Avg)						
Data Transfer Time (Avg)	122 ms					
To, Client 1						
Route Search Time (Avg)	386 ms		Replies From Cache		90->88->87	
Data Transfer Time (Avg)	388 ms					
Sender Of Message : Client 5						
To Client 4						
Route Search Time (Avg)	0 ms					
Data Transfer Time (Avg)	125 ms					
To Client 3						
Route Search Time (Avg)	9005 ms				Replies From Cache	91->90->88->89

Data Transfer Time (Avg)	346 ms					
To, Client 2						
Route Search Time (Avg)	334 ms				Replies From Cache	91->90->88
Data Transfer Time (Avg)	371 ms					
To, Client 1						
Route Search Time (Avg)	9008 ms				Replies From Cache	91->90->88->87
Data Transfer Time (Avg)	9009 ms					

Table 2: Test Results for Diamond topology

7.1.2 Linear Configuration

Second topology that was used to test the routing protocol was the simple linear network topology.

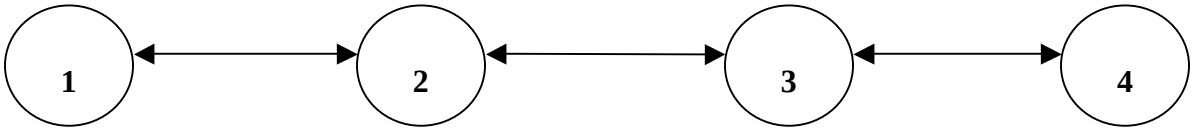


Figure 13: Linear Configuration

Table here shows the configuration details of this linear network.

Node	Node	Neighbors	Neighbors Address

#	Address		
1	00087443BC87	2	00087443BC88
2	00087443BC88	1,3	00087443BC87, 00087443BC90
3	00087443BC90	2,4	00087443BC88, 00087443BC91
4	00087443BC91	3	00087443BC89

Table 3: Linear Configuration

Following table shows test results for the linear network topology. As in the previous test, the time it takes to find a route to a destination and the time it takes to send data packet to that destination and receiving confirmation is measured.

Linear Network

Sender Of Message : Client 1		**Cache content of Each Client at End of Route Request**			
To		**Client 1**	**Client 2**	**Client 3**	**Client 4**
Client 4					
Route Search Time (Avg)	7135 ms	87->88->90->91	88->90->91		
Data Transfer Time (Avg)	611 ms				
To					
Client 3					
Route Search Time (Avg)	337 ms	87->88->90			
Data Transfer Time (Avg)	437 ms				
To,					
Client 2					
Route Search Time (Avg)	0 ms				
Data Transfer Time (Avg)	132 ms				
Sender Of Message : Client 2					
To,					
Client 4					
Route Search Time (Avg)	0 ms				
Data Transfer Time (Avg)	353 ms				
To,					
Client 3					
Route Search Time (Avg)	0 ms				

Data Transfer Time (Avg) 140 ms					
To, **Client 1** Route Search Time (Avg) 0 ms Data Transfer Time (Avg) 205 ms					
Sender Of Message : Client 3					
To, **Client 4** Route Search Time (Avg) 0 ms Data Transfer Time (Avg) 136 ms					
To, **Client 2** Route Search Time (Avg) 0 ms Data Transfer Time (Avg) 125 ms					
To, **Client 1** Route Search Time (Avg) 348 ms Data Transfer Time (Avg) 400 ms				90->88->87	
Sender Of Message : Client 4					
To, **Client 3** Route Search Time (Avg) 0 ms Data Transfer Time (Avg) 126 ms					
To, **Client 2** Route Search Time (Avg) 353 ms					91->90->88

Data Transfer Time (Avg)	125 ms				
To, Client 1					
Route Search Time (Avg)	614 ms			Replied from Cache	91->90->88->87
Data Transfer Time (Avg)	656 ms				

Table 4: Test Results for Linear Topology

8 Conclusion

Project successfully provides routing implementation. Default anhinga at M2MP level works only in broadcast mechanism. This implementation provides point to point communication facility with routing mechanism which can be helpful for upper layer like M2MI to concentrate on. Scalability and over head is concern. Heuristics required to optimize. This protocol has low routing overhead compared to other protocol that maintains network topology. Among main reason for better performance is the protocol implementation derives from a completely on-demand DSR protocol. The protocol requires no periodic dedicated activities to maintain the routes. When all nodes are stationary and all the routes needed for communication are discovered there is no periodic overhead to maintain unchanged network topology. Routing related overhead scales as network topology changes. For broadcast and multicast communication with controlled flooding, client nodes may get hogged with packets. Time complexity to detect changes in network is also inversely related to activity within network. With high communication between nodes, any changes in topology are detected faster.

9 Future Works

This project was designed and developed with enabling communication between nodes in dispersed network in mind. Even though this implementation provides APIs for application on top of it to use, a great deal can be improved to provide infrastructure for applications on top to use its core functionality. Another major changes can be done is the way in which peer-to-peer messages are transmitted. Currently all the messages are broadcasted in local network rather then being uni-casted or multi-casted. This change may affect the performance of the system in a major way. Following are some other points on which project can be enhanced in future.

1) **"Salvaging Packets":** When broadcasting a "Route Error" packet, if a node has alternate path to broken link in its cache; it piggy backs the replacement route with "Route Error" page.

2) **"Automatic Route Shortening":** A route can be automatically shortened if a node over hears a packet carrying a source route, it examines node's unused route, and if it finds itself in there, that means that the node immediately before it in the route is not needed. In such case this node will return "gratuitous Route Reply" to source node with shortened route.

3) **Increased spreading of "Route Error" message:** Piggy backs a "Route Error" packet to corresponding "Route Request" packet so that neighbor nodes do not reply with old route from their cache. And they update their own route cache.

   Source node caches negative information. Meaning the node that requested "Route Request" based on previous "Route Error" will cache information of broken link from "Route Error" for certain amount of time. This will allow the node to discard any "Route Reply" based on stale cache.

10 Reference

[1] Alan Kaminsky and Hans-Peter Bischof. Many-to-Many Invocation: A new object oriented paradigm for ad hoc collaborative systems. 17th Annual ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA 2002), Onward track, Seattle, Washington, USA, November 2002, http://www.cs.rit.edu/~anhinga/publications/m2mi20020716.pdf

[2] AD HOC Networking, Charles E. Perkins, editor, Boston, MA: Addison-Wesley, 2001.

[3] The Anhinga Project. http://www.cs.rit.edu/~anhinga.

[4] Routing Protocols in Mobile Ad Hoc Networks. Aakash Chauhan, Research/Review paper for Data Communication and Networks - 1. RIT. Winter 2002. http://www.rit.edu/~adc0467/dcn1Paper.pdf

[5] J.J. Garcia-Luna-Aceves and Marcelo Spohn "Source-Tree Routing in Wireless Networks" In Proceedings of IEEE ICNP 99: 7th International Conference on Network Protocols, Toronto, Canada, October 31-November 3, 1999.

[6] Zygmunt J. Haas and Marc R. Pearlman. "ZRP: A Hybrid Framework for Routing in Ad Hoc Networks." In Charles E. Perkins, editor, Ad Hoc Networking (Boston, MA: Addison-Wesley, 2001). Pages 221-253.

[7] C. E. Perkins and E. M. Royer, "Ad hoc on-Demand Distance Vector Routing." In Proceedings of 2nd IEEE workshop on Mobile Computing System and Applications, February 1999.

[8] J. Broch et al. A performance comparison of multi-hop wireless ad hoc network routing protocols. Proc. ACM MOBICOM 98, October 1998.

[9] David B Johnson, David A Maltz and Josh Broch. "Dynamic Source Routing Protocol for Multihop wireless Ad Hoc Networks." In Charles E. Perkins, Ad Hoc Networking pg. 139-172.

[10] Sung J Lee, William Su and Mario Gerla. "On-Demand Multicast Routing Protocol (ODMRP) in Multihop Wireless Mobile Networks."
http://citeseer.nj.nec.com/cache/papers/cs/20405/http:zSzzSzwww.cs.ucla.eduzSzNRLzSzwirelesszSzPAPERzSzsjlee_monet.pdf/lee01demand.pdf

[11] Java On-Demand Multicasting Routing Protocol (JOMP)
http://homepages.cs.ncl.ac.uk/einar.vollset/home.formal/jomp.html

[12] DSR Routing Simulator: An exploration into ad-hoc Routing
http://www.cs.rit.edu/~ark/543/teams/Aquafina/

[13] IETF DSR Draft
http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-10.txt

[14] HF-DSR: Dynamic Source Routing for High Frequency Radio Networks
http://www.cs.rit.edu:8080/ms/static/ark/2005/3/mds1761/index.html

[15] http://wiki.uni.lu/secan-lab/Dynamic+Source+Routing.html

[16] http://www.juniper.net/techpubs/hardware/m10i/m10i-hwguide/architecture-re.html#fig-architecture-re

[17] http://www.cs.cmu.edu/~dmaltz/dsr.html

[18] http://wiki.uni.lu/secan-lab/Dynamic+Source+Routing.html