

# Dynamic Source Routing Implementation

John Hashem  
John Roland  
Suleyman Muhammad  
Todd Matzelle  
Ying Jin

## Abstract

The purpose of this project is to demonstrate a practical implementation of the Dynamic Source Routing (DSR) algorithm, or variance thereof, within a real life environment. The routing program has been written in C for a Linux environment and can be run from the command line. With our implementation of the DSR protocol, we utilized Nokia N800 tablets as nodes for routing. At the beginning of the message transmission, the route discovery protocol is used to find the route; each node has a route cache in which it can cache source routes that it has learned. Message transmission following route cache works before route cache expired. If any node fails or runs out of power, the route maintenance protocol will take effect and the new route will be found through the route discovery process.



## I INTRODUCTION

In ad-hoc networks, a collection of mobile nodes within wireless transmission range of one another may form a temporary network without the aid of any infrastructure or centralized administration. With the formation of these ad-hoc networks, the exchange of information often spans across multiple nodes. As such, multi-hop routing protocols are most appropriate for use. For our project, the Dynamic Source Routing (DSR) protocol was selected and implemented as a modified version of the algorithm. This protocol is reactive and requires no periodic advertisements for the discovery or maintenance of routes. Instead, functions are performed in an on-demand manner where sending nodes only request routing information when an entry to a destination does not exist within its own cache. Whether already in cache or built in an on-demand manner, a complete sequence of nodes (network hops) is be provided. This sequence of nodes is used to forward a packet to its targeted destination. Due to the on-demand nature of requests, overhead is often largely reduced. This is especially the case when little or no significant host movement is taking place.

## II DESIGN PROCESS

The design process began with the creation of a state machine, which first depicted the traditional implementation of the DSR algorithm. Utilizing this as a basic framework for our design, each of the processes of the algorithm was designed, built, and implemented. Any variations from DSR were then mirrored in changes made to our final version of the modified DSR state machine diagram. In our design we determined that four kinds of message packets were necessary: Data, Route Request, Route Reply, and Error. The formats of these packets differ from the traditional DSR implementation, which is explained in more detail later in this document. Branches of the state machine began as pseudo-code and were then implemented in the C programming language. Code was written on the Linux operating system and compiled using the GNU C compiler (gcc). Additionally, compilation was done in the Scratchbox environment in order to simulate the N800s processor.

## Deviations from Traditional DSR Protocol

As mentioned above, during the design of our protocol modifications were made to the original DSR algorithm. These changes include; transmitting messages back to the sender with an error, having the sending node handle Route Requests resulting from errors, and changing the overall format of packets.

## Modified DSR State Machine Diagram

The state machine below illustrates the high level decision making process from the perspective of any given node within a wireless ad-hoc network. As shown below, information exchanges for nodes will always start and end within the *Wait* state. Each rectangle represents a unique state, while arrows highlight events/actions that are taken. Please note that thick lines denote the 1st events/actions taken after the *Wait* state.

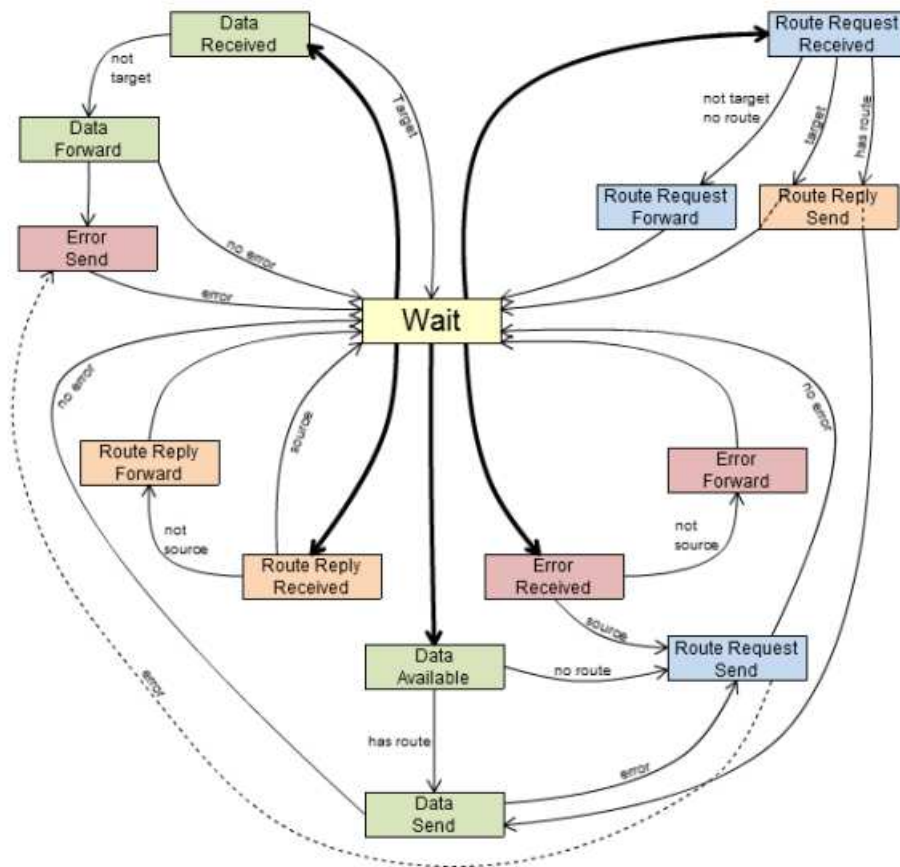


Fig. 1. DSR State Machine Diagram

## Modified DSR Path Permutations Chart

The following chart illustrates all unique permutations of events/actions and states that can occur using the modified DSR state diagram. For spatial reasons, the original chart was simplified so that it could be included within this document. If desired, the complete chart is available as an appendix to this paper.

This chart is a good example how a quality assurance engineer might attack testing our algorithm, and serves as a measurable and reportable format for communicating results back to a developer. In our case the chart serves to illustrate how many scenarios can occur, in what order, and beginning with what state and shows where certain permutations would fail if functions are incomplete. While reviewing the chart please note that all colors correlate directly with states within the state diagram.

DSR Path Permutation Chart															
Path		Wait	Data				Route Request			Route Reply			Error		
#	Type		Available	Received	Send	Forward	Received	Send	Forward	Received	Send	Forward	Received	Send	Forward
1	DATA	1, 4	2		3										
2	DATA	1, 6	2		3			4						5	
3	DATA	1, 4	2					3							
4	RRQ	1, 4								2	3				
5	RRQ	1, 3								2		3			
6	DATA	1, 3		2											
7	DATA	1, 4		2		3									
8	DATA	1, 5		2		3								4	
9	RRP	1, 3								2					
10	RRP	1, 5			3					2			4		
11	RRP	1, 4								2		3			
12	ERR	1, 3						2							
13	ERR	1, 3													2

### Notes

- All #'s in the chart represent a sequential steps for a given path permutation
- All paths correspond directly to the modified DSR state diagram

### Key

RRQ	Route Request
DATA	Data Packet
RRP	Route Reply
ERR	Error
#	Not Implemented due to time constraints

## Packet Formats

The following tables represent various packet formats utilized for communication between nodes, using our modified DSR algorithm.

### Data

Flag	Hop Sequence	Target Node	Originating Node	Message
1	hop_seq	target	originator	msg

### Route Request

Flag	Hop Sequence	Target Node	Message
2	hop_seq	target	msg

### Route Reply

Flag	New Route	Hops Remaining to Target
3	new_route	hops_left

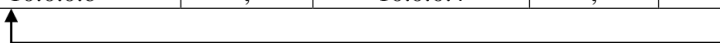
### Error

Flag	Hop Sequence to Originator	Target Node	Message
0	rev_hop_seq	target	msg

## Hop Sequence Format

The hop\_seq in the above packet formats is a semi-colon-delimited string containing all of the hops that a message must take to get to its destination. The table below illustrates this format and the process by which a hop sequence is rotated. Changes in hop sequence occur each time a packet is passed from one node to another.

1 <sup>st</sup> Hop	Delimiter	2 <sup>nd</sup> Hop	Delimiter	3 <sup>rd</sup> Hop
10.0.0.6	;	10.0.0.4	;	10.0.0.2

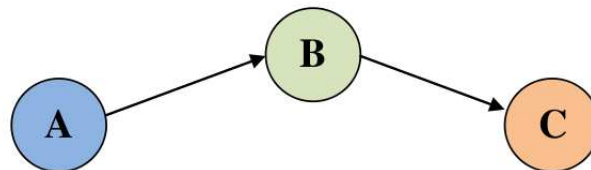

  
Hop Rotation

## Simulation Example

The following is an example of how our simulation works using the packet formats mentioned above. Please note that the colors used below do *not* correlate with the state diagram or DSR path permutations chart.

Heres a breakdown of the details surrounding this message:

- **A** wants to transmit a message to node **C**
- **A** is only within range of node **B**
- **B** is within range of nodes **A** and **C**



## Assumptions

- A has already moved from the *Wait* state to *Data Available* and is now ready to try and send data.
- A realizes that it has no route entry for a path to the target node.
- A knows that it must follow the process below in order to obtain this missing route.

## The Route Request (RRQ) & Route Reply (RRP) Process

1. The process of discovering a route is initiated by a sending node with a **RRQ** to its neighbors. As such node A broadcasts the **RRQ** and is heard by node B. As a result of this broadcast, B has learned where node A is.

Flag	Hop Sequence	Target Node	Message
2	A	C	Hello!

2. The process continues when B initiates a **RRQ** broadcast. This is heard by nodes A & C. As a result node A learns the whereabouts of B, and C learns the whereabouts of nodes A & B.

**Note:** Node C now has the full topology.

Flag	Hop Sequence	Target Node	Message
2	A;B	C	Hello!

3. When C sees that it is the target node, it responds with a **RRP**. This **RRP** is sent to B, which in turn teaches B the location of C.

**Note:** Node B now has the full topology.

Flag	New Route	Hops Remaining to Target
3	B;C	B;A

4. The **RRP** process ends when B sends to A. At this point A learns the locations of C, completing its own topology.

Flag	New Route	Hops Remaining to Target
3	B;C	A

## The Process of Sending Data

Once the **RRQ** & **RRP** processes have finished, the route cache for nodes A, B, & C are well established and the **Data** packet transmission process can begin. This process can be described as follows:

5. Since A now has a route to C within its own cache it can now begins the **Data Send** process. This process is initiated when A sends the packet shown below to B.

Flag	Hop Sequence	Target Node	Originating Node	Message
1	B;C	C	A	Never mind ;-)

6. After B receives the packet from A, it checks to see if it's the Target. Since it's not it rotates the hop sequence and forwards the packet to node C.

Flag	Hop Sequence	Target Node	Originating Node	Message
1	C;B	C	A	Never mind ;-)



7. Finally, **C** receives the packet from **B** and checks to see if it's the target node. Since it is the target it reads the message, which thereby completes the communication process.

### III PERFORMANCE EVALUATION AND RESULTS

Technical difficulties impeded our ability to properly evaluate the performance of our algorithm. This is because not all of the Nokia N800s devices were properly configured, and thus could not be utilized. It was determined that since the only reason for originally using the N800s was because of the initial desire to implement OLSR as a plug-in for the devices. Otherwise, they were simply Linux-running tablets. Therefore a Plan B for testing involved a combination of the tablets and Linux-running laptops.

Off-campus, an ad hoc network between the laptops and N800s was successfully created. The algorithm itself was verified as working. Route discovery and message transmission worked successfully. Another problem quickly became an obstacle, however; a structured topology could not be created due to the inability to use MAC filtering successfully. At this point, time became an issue in making further progress in this area.

### IV CONCLUSION

This project was an excellent project for understanding, designing, and implementing a Dynamic Source Routing algorithm. Everyone was able to actively and effectively contribute to the design of the project. This was the most important part in terms of actually understanding what DSR is and how it works. This well-planned design turned into a fundamentally functional application with demonstrated success.

Unfortunately, two key factors contributed to us having to end the project where we did. The first was the difficulties with the N800s. This includes the large amount of time spent on trying to comprehend the plug-in aspect of the tablets before ultimately dismissing that option and going for a standalone application. Afterwards, the time spent before determining that not all of the tablets were properly configured for our use. Ultimately, this led to our second and biggest issue: simple time constraint. While there was enough time to successfully design and implement a functioning DSR application, there simply was not enough time to do everything we had hoped to do. In terms of coding, all of diagram was implemented except for error handling. This did give us enough functionality to proceed with performance evaluation/testing, had it not been for the other technical issues with the devices.

In the end, however, we are still satisfied with this DSR implementation. We feel confident that our underlying algorithm works and believe that with a little more time and minus the time spent on OLSR, plugins, and technical difficulties, we would have been able to fully implement our design. All-in-all, this was an enjoyable project that we were able to learn a great deal from.