CPE 400 Project

Dynamic Routing Mechanism Design in Faulty Network

Kripash Shrestha, Erik Miannay

Project overview:

Mobile Ad Hoc Networks

In mobile ad hoc networks, nodes communicate with each other by forwarding data between other connected nodes. In such networks, a node may be connected directly with another node or indirectly through other nodes' connections. So, data navigates through the network from node-to-node. Consequently, it's the job of the nodes within a network to figure out how to get data from one node to another. This differs from a router-centric network, where, end-hosts are connected to routers and routers control the passing of data. And in the context of wireless networking, there is a base station that connects wireless hosts to a network. But this type of network tends to be fixed, while mobile ad hoc networks are characterized by rapidly changing network topology. So, in order to accommodate for this, a special type of reactive routing protocol is needed.

Dynamic Source Routing Protocol

To facilitate data exchanges between nodes in a mobile ad hoc network, the Dynamic Source Routing protocol (to be called 'DSR' herein) can be applied. To offer a human analogy to describe the concept of DSR's routing algorithm, take someone's network of friends at school as an example. If a person wants to get in touch with someone at their school in which they do not have direct contact with, that person can ask each of their friends to see if they know them. If their friends don't know them, they can then ask their friends—ad infinitum. And thanks to the phenomena of 'six degrees of separation,' and assuming everyone in a given school is friends with someone else in that school, the person that is trying to be reached will eventually know that the original requestor is attempting to contact them (so, a path has been established). More so, friendship networks tend to change rapidly too! Thus, the path between any two people at school is subjected to change.

Likewise, DSR works analogously by one node asking its neighbors—a neighbor being another directly connected node—if they are connected to the wanted node. If those neighbors aren't connected, then they ask their neighbors. This recursive asking of neighbors occurs until every node in the entire network has been asked. If the desired node is found, then the chain of asks is retraced back to the original node so that the original node now knows a valid path to the desired node.

DSR Implementation:

Project's Implementation of DSR

This project implements DSR by simulating how paths are determined in a highly volatile network (highly volatile network is described in next section). In DSR, a route request (to be called 'RREQ' herein) most propagate throughout an entire network, and if the requested node receives the request, a route reply (to be called 'RREP' herein) must travel back along the traversed path that the request took. More so, to prevent redundant RREQs, each node in the network must keep record of RREQs that it has already forwarded. DSR also includes caching, where nodes in the network have the ability to save paths it comes across through RREQs and RREPs. For this project, however, caching in not implemented. So, neither is notifying other nodes of a route error (a RERR) since there are no route caches to invalidate. This is because this project wants to simulate a highly volatile network, which makes caching very inefficient.

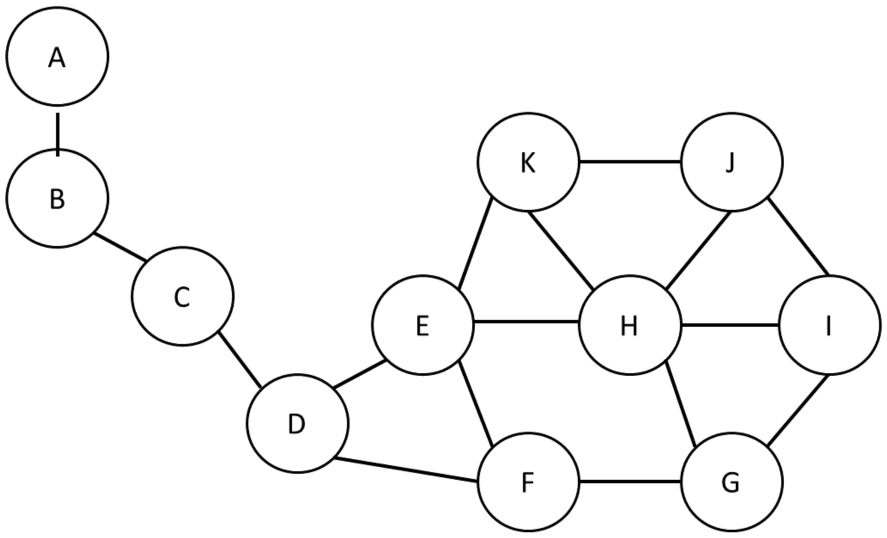
Simulating A Highly Volatile Network

We want to implement a highly volatile network to achieve a novel simulation case of DSR. We define a highly volatile network to be a network where links between connections are created and destroyed in a rapid fashion. This sort of network can be imagined by thinking of a future scenario where self-driving cars come in and out of contact with other self-driving continuously. (A car turns a corner—leaving one network and joining another.) So, to simulate this sort-of scenario, our program creates and destroys links at random in a 'thread' function (a simulation of a thread), while the main function attempts to determine routes between nodes.

Example Network

Our program begins with the example network shown in Figure 1. The links between nodes are equally weighted; though, due to how our program simulates RREQ propagation, RREQs are sent first by nodes whose letter values come first. To clarify, consider node 'E' in Figure 1. Node 'E' will be forwarded an RREQ from node 'D' before node 'F'; so, in the next wave of RREQ forwarding, node 'E' gets to send out an RREQ before node 'F'. While in the end, all nodes that are connected will eventually get a RREQ, this does affect how RREQs are spread throughout the network in terms of precedence even though links are equally weighted. This is merely a byproduct of how the simulation is implemented, since we're attempting to simulate independent nodes in a sequential process.

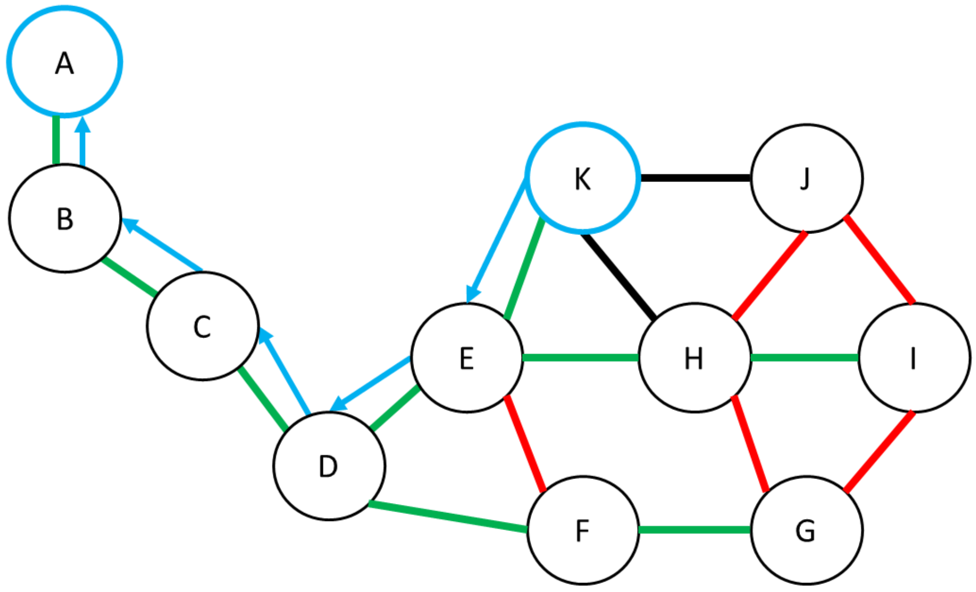
Figure 1:



Example of RREQ Propagation and RREP Traceback

If node ‘A’ wants to determine a path to node ‘K’ (which is fully demonstrated in ‘Simulation Example’ section later as the ‘Fixed Network Example’), the propagation of ‘A’s RREQ throughout the network is shown in Figure 2. A green link means the RREQ is accepted by the receiving node, a black link means that node never forwarded the RREQ (since that is the desired node), and a red link means the RREQ is ignored (since that receiving node has already forward the RREQ). The blue in Figure 2 is the path that the RREP takes during the traceback. So, the path would be: [ABCDEK].

Figure 2:



Link Failures and Connections Changing

Our program generalizes link failures, downed nodes, and changing connections all as links being removed between nodes. Removal is done by randomly selecting two connected nodes to be disconnected. All links have an equal chance of being removed. More so, we’re not just simulating this type of change, but link creation as well by adding connections between nodes.

When removing links, several cases arise; these cases are described in Table 1.

Table 1:

Case	**Consequence**
Node on edge of network loses only connection to network	The node no longer has ability to contact other nodes in the network; until a new connection is established.
Node that is acting on a bridge is lost	Two new sub-networks are created, wherein both sub-nets are now isolated from each other; until a new bridge is established.
Node in dense area is lost	Simple rerouting.
Nodes are in an unreachable island	These nodes can contact each other, but not the rest of the network; until a new bridge is established.

Simulator Examples:

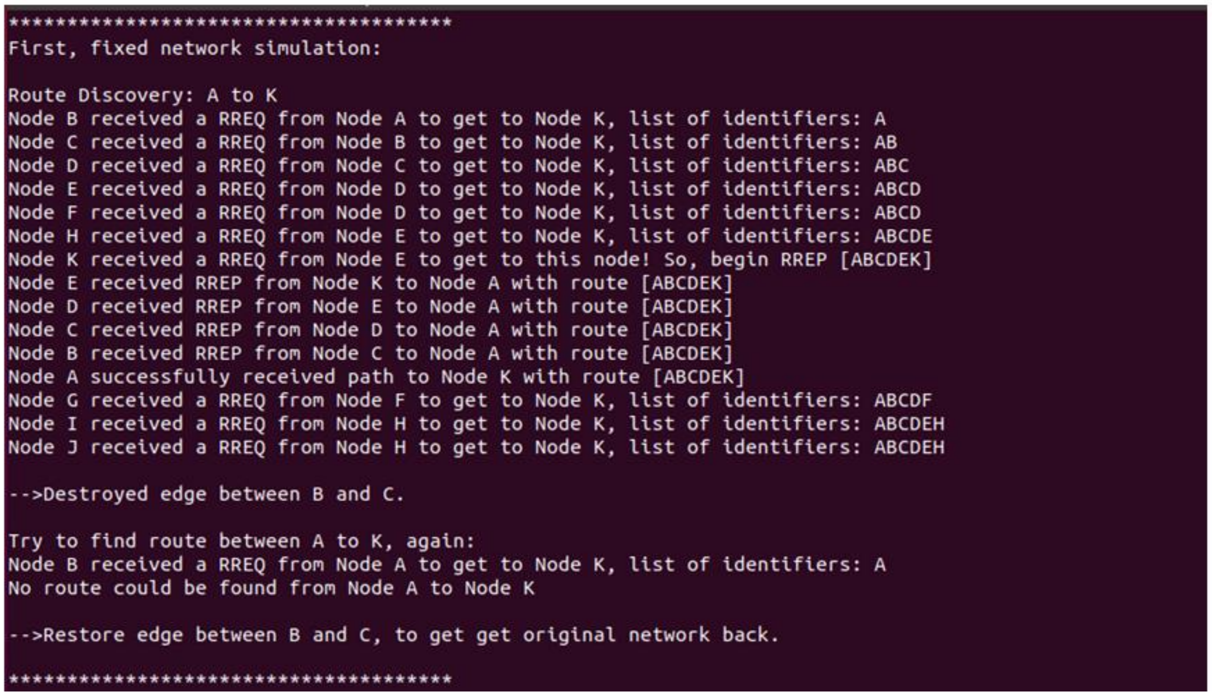
When running the program there are two examples: fixed network—to demonstrate basic DSR functionality—and highly volatile example--to demonstrate rapid destruction/creation of links between nodes.

In either case, the program sends a RREQ between two nodes and the propagation of the RREQ throughout the network is printed out to the command line every time a node receives a RREQ. The same is true for the corresponding RREP, assuming a path was found. If no path found, the program times out after 5.0 seconds—which is how the requesting node interprets the event of not being able to reach a node.

When running the program: it first does the fixed network example, then five iterations of the highly volatile example, and lastly, redoing the fixed network example (to see if the path changed after many links being created/ destroyed).

Fixed Network Example:

Printout 1:



```
**************************************
First, fixed network simulation:

Route Discovery: A to K
Node B received a RREQ from Node A to get to Node K, list of identifiers: A
Node C received a RREQ from Node B to get to Node K, list of identifiers: AB
Node D received a RREQ from Node C to get to Node K, list of identifiers: ABC
Node E received a RREQ from Node D to get to Node K, list of identifiers: ABCD
Node F received a RREQ from Node D to get to Node K, list of identifiers: ABCD
Node H received a RREQ from Node E to get to Node K, list of identifiers: ABCDE
Node K received a RREQ from Node E to get to this node! So, begin RREP [ABCDEK]
Node E received RREP from Node K to Node A with route [ABCDEK]
Node D received RREP from Node E to Node A with route [ABCDEK]
Node C received RREP from Node D to Node A with route [ABCDEK]
Node B received RREP from Node C to Node A with route [ABCDEK]
Node A successfully received path to Node K with route [ABCDEK]
Node G received a RREQ from Node F to get to Node K, list of identifiers: ABCDF
Node I received a RREQ from Node H to get to Node K, list of identifiers: ABCDEH
Node J received a RREQ from Node H to get to Node K, list of identifiers: ABCDEH

-->Destroyed edge between B and C.

Try to find route between A to K, again:
Node B received a RREQ from Node A to get to Node K, list of identifiers: A
No route could be found from Node A to Node K

-->Restore edge between B and C, to get get original network back.

**************************************
```

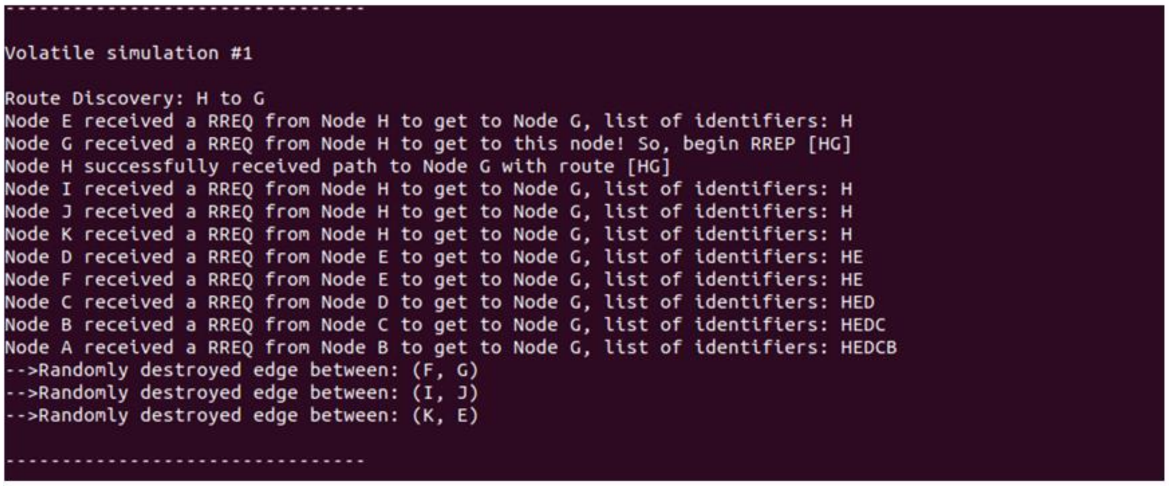
For Printout 1, the path determined in this example was already calculated in Figure 2—where the path between node ‘A’ to ‘K’ is [ABCDEK]. Thus, the simulator outputs the expected path.

A few things to note:

- Each node connected in the example network in Figure 2 (A through K) printed a response saying it received an RREQ; so, the entire network received a RREQ.
- Only nodes in the path determined printed a response saying it received a RREP.
- When the edge between B and C is destroyed—which causes A and B to become an island—A and B are no longer connected to the rest of the network, so they can only send a RREP to each other; thus, no path is found when tried again.

Highly Volatile Example:

Printout 2:



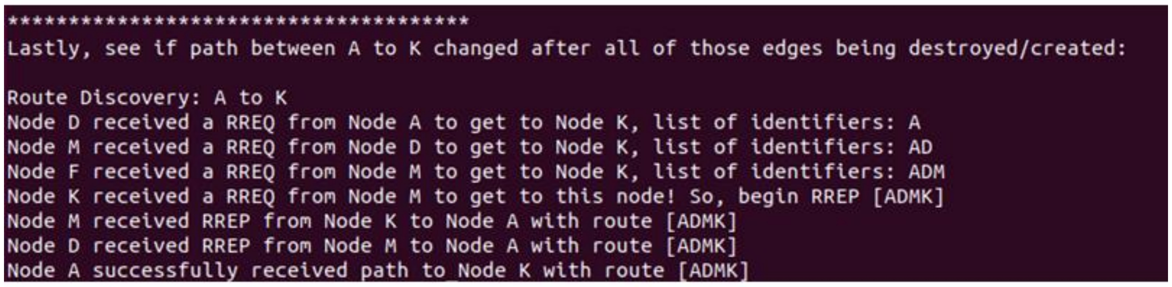
```
-------------------------------

Volatile simulation #1

Route Discovery: H to G
Node E received a RREQ from Node H to get to Node G, list of identifiers: H
Node G received a RREQ from Node H to get to this node! So, begin RREP [HG]
Node H successfully received path to Node G with route [HG]
Node I received a RREQ from Node H to get to Node G, list of identifiers: H
Node J received a RREQ from Node H to get to Node G, list of identifiers: H
Node K received a RREQ from Node H to get to Node G, list of identifiers: H
Node D received a RREQ from Node E to get to Node G, list of identifiers: HE
Node F received a RREQ from Node E to get to Node G, list of identifiers: HE
Node C received a RREQ from Node D to get to Node G, list of identifiers: HED
Node B received a RREQ from Node C to get to Node G, list of identifiers: HEDC
Node A received a RREQ from Node B to get to Node G, list of identifiers: HEDCB
-->Randomly destroyed edge between: (F, G)
-->Randomly destroyed edge between: (I, J)
-->Randomly destroyed edge between: (K, E)

-------------------------------
```

Printout 3:



```
***************************************
Lastly, see if path between A to K changed after all of those edges being destroyed/created:

Route Discovery: A to K
Node D received a RREQ from Node A to get to Node K, list of identifiers: A
Node M received a RREQ from Node D to get to Node K, list of identifiers: AD
Node F received a RREQ from Node M to get to Node K, list of identifiers: ADM
Node K received a RREQ from Node M to get to this node! So, begin RREP [ADMK]
Node M received RREP from Node K to Node A with route [ADMK]
Node D received RREP from Node M to Node A with route [ADMK]
Node A successfully received path to Node K with route [ADMK]
```

For Printout 2, the volatile example is first selecting two nodes at random to see if they can connect, and then links are created/destroyed before going on to the next iteration.

A few things to note:

- The simulation runs the volatile example 5 times.
- After the last volatile example, the fixed example is re-ran (and shown in Printout 3) to show that the path from A to K is no longer [ABCDEK] but [ADMK]
- The 'thread' function that is destroying/creating links is evident by its print outs. Specifically, it prints which links are being destroyed and what links are being created.
- Since links are destroyed/created at random, running the simulation again will produce different results then the example captured in Printout 2 and 3.

Conclusion

In mobile ad hoc networks, the DSR protocol can be deployed as a way to create valid paths of communication between nodes that are only connected to other nodes. Since this type of network can be volatile—especially in the one imagined in this project—traditional proactive routing algorithms aren't feasible. As demonstrated in the project, even in frequently changing network topologies, can communication paths still be determined. DSR is a notable protocol to study as mobile ad hoc networks become more prevalent in applications like self-driving cars.