An Accessible, Open-Source, Realtime AODV Simulation in MATLAB

Stuart Miller
Department of Electrical and
Computer Engineering
Missouri University of Science & Technology
Rolla, Missouri 65409
Email: mailto:sm67c@mst.edu
Web: http://web.mst.edu/~sm67c

***Abstract*—It is often necessary to provide a base-level teaching tool when explaining concepts to new learners, or for the engineer performing initial basic prototyping. Understanding routing schemes in wireless ad-hoc networks is a central concept for any new learner. Since the subject is best taught through hands-on experience, many learners find themselves at a disadvantage when exposed to the confusing and highly specialized tools used in wireless network simulation. Therefore, it is proposed to use the familiar and accessible environment of MATLAB to create base-level accessible, open-source, realtime ad-hoc routing scheme simulations, here targeting the ad-hoc on-demand distance vector (AODV) routing scheme.**

I. Introduction

One of the key advantages to using MATLAB as an environment is its natural support for high quality user interfaces. The user interface is easy to understand and can be updated with simulation data in real-time. A major drawback to many preeminent network simulation tools is their reliance on written scripts and their inability to allow real time analysis and modification. MATLAB allows real-time input/output interaction and visualization. Additionally, it provides a much better debugging environment.

Perhaps most importantly, this work can provide a much more accessible intro to wireless routing schemes, as most engineers are already familiar with MATLAB functionality. Most network simulation scripting environments provide a significant barrier to enter due the their learning curve and often extensive environment setup time. MATLAB requires no setup beyond installation of the base program and is easy to understand. It follows that Stack Overflow listed MATLAB as one the least disliked languages of 2017 [1] and that 4.3% of all contributors used MATLAB as their primary language [2] (although undoubtedly more are familiar with it causally).

For the purposes of this initial work, the focus will be on the ad-hoc on-demand distance vector (AODV) routing scheme; however it is easy to see how the concepts here can be extended to additional routing methods.

II. Literature Survey

A. AODV History

Although AODV was developed almost 15 years ago [3], it still remains highly studied in the current state of the art. AODV is a reactive routing protocal that aims to reduces overhead by requiring few network-wide broadcasts. There is no centrally kept routing information and each node discovers routes only as necessary. Only in the event that a node does not know a path to it's target destination, does it initiate a network-wide broadcast. This is known as flooding and involves a series of route request messages that propagate through the network, landing once on each connected node until that node either is the destination, or knows a path to the destination. As flooding is the single most expensive action within AODV (a notion affirmed in section V), it has been the subject of much research. Saeed, et. al. [4] propose a novel flooding procedure known as VDM-SL that aims to greatly reduce overhead by targeting route request messages at nodes based on the type of traffic they are sending in order to determine who is most likely to know a route to the destination. Alternatively, Feng, suggests a cluster-based method as a way of reducing route request overhead [5]. Their method involves selecting clusterheads to propagate the requests across large networks.

Another key feature of AODV is it's elegant error handling. In the case that a node cannot complete a link stored in it's route table (i.e. the stored link references a next hop node that's not currently connected), a route error message must be sent. Whenever this happens during a data send transmission, the transmission must be halted and a route error message propagated all the way back to the source node. While this does eliminate the need for a network wide broadcast initially, this could still be a significant cause for additional overhead. After the propagation back to the source, flooding must occur again to determine new route. Devi and Sikamani, [6] propose a route error handling scheme which does not require full propagation back to the source, but instead allows each intermediate node to attempt to handle the error themselves. Such a strategy covers cases of temporary interference causing link failures (simply trying again may resolve the issue) as well as routes which can be easily re-

routed one link up the path. In much the same fashion, Ejmaa, et. al. [7] propose a similar modification resulting in a new AODV-based routing protocol they call Dynamic Connectivity Factor routing Protocol (DCFP). This protocol also monitors local nodes and attempts to use additional local parameters to resolve route errors, thereby creating less overhead.

Nodes determine who their connected neighbors are by regular hello messages, as well as attempted and failed transmissions. One proposed improvement on this is Mittal's 2017 paper [8] targeting VANET networks which enhances hello messages to share additional neighboring node information and route info, a method which they prove can help in congested vehicular scenarios.

The recent body of work shows much interest on the subject of AODV as a routing protocol. Several recent comparative analyses of routing protocols in MANET or VANET networks feature AODV: Ferrnato, et. al. in their protocol analysis with regards to urban vehicular scenarios [9] and Abuashour and Kadoch in their study of clustered VANET scenarios [10]. A novel idea proposed by Yadav and Hussain is to introduce authentication to AODV [11], thereby preventing man-in-the-middle style attacks from malicious masquerading nodes. The abundance of work in this area serves as assurance that AODV still remains a central protocol to the field and will serve as a relevant target protocol for this project.

B. Network Simulators History

While MATLAB does already provide a comprehensive WLAN systems toolbox [12], it is solely targeted at IEEE 802.11 standards-compliant LAN-based networks and provides little-to-no support for ad-hoc or sensor networks. The selection of specified ad-hoc routing schemes and modification of node parameters is not present in the toolset provided. Unfortunately, this is far from sufficient for the goals of this endeavor.

The existing body of work does, however, show several instances of using MATLAB to test out highly specialized ad-hoc routing schemes. Gaur, et. al. show a novel routing scheme for optimizing battery life of the mica2 motes using MATLAB as a simluation environment [13]. Navya and Deepalakshmi utiltize MATLAB so show the feasibility of their proposed M-TSIMPLE routing protocal, an energy-efficient routing scheme for wireless body area networks [14]. While both of these approaches do prove the feasibility of MATLAB as a simulation environment, each is far too specialized for the generalized application needed here.

Habib, et. al. describe another approach to the usage of MATLAB in this area: to simulate path-finding algorithms in various routing schemes, and to provide a comparison among networks of varied size [15]. Their analysis focuses more on the mathematics of pathfinding and as such, is also insufficient for our purposes.

While there is significant work being produced using MATLAB as an environment, much of the groudbreaking work is being done in other simulation tools as well. Network Simulator 3 (ns-3) a popular tool for such simulations; it provides a discrete-event network simluator/emulator and scripting language based on C++ or Python code [16]. Ns-3 is targeted primarily for research and educational use. While popular, is has received significant criticism. Patel and Kmaboj criticized ns-3 harshly for it's lack of a GUI and noted it's complexity and lack of hosted work and support [17]. That is not to say it is not without significant work in the academic community. Mai, et. al. used ns-3 to simulate their version of AODV, modified for a better congestion control scheme [18]. While Network Simulator 2 [19] is now deprecated, it too, was and still is the subject of a significant number of publications.

Another noteworthy platform is NetSim [20]. Netsim Academic is the tool that most closely mirrors the intentions of this project as it is targeted at teaching/learning applications and provides adundant examples and support. Unfortunately, it is neither free nor open-source and requires a large license purchase to use. It is not without usage in the academic community though; numerous papers have cited NetSim as their primary simulation environment. Saifuddin, et. al. utilized NetSim in their analysis of alternative channel allocations for wireless networks [21]. Nayak and Sinha provided an analysis of various mobility models for routing protocols (including AODV) using NetSim [22]. Figure 1 shows an excerpt from their paper featuring their AODV simulation. A setup similar to the is the target for this MATAB implementation.

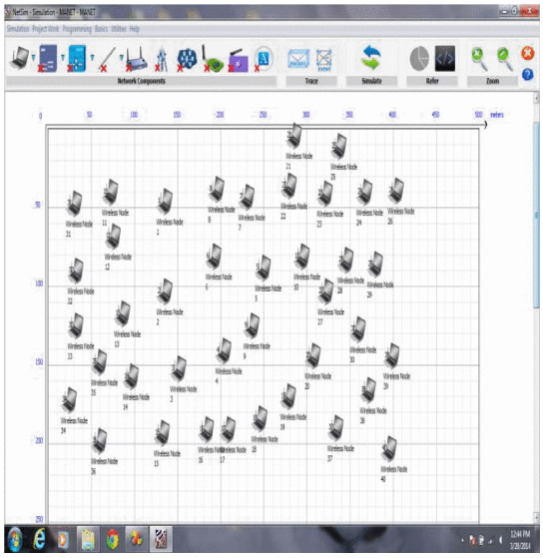

Fig. 1. Nayak and Sinha's AODV simulation window [22]

Finally, OMNeT++ is yet another discrete-event library used frequency used to build network simulators [23]. OMNeT sees quite a bit of use in the academic comnmunity as well, Chadha, et. al. used it in their simulation of energy consumption of nodes in a 3D environment [24].Thomas and Irvine investigate bandwidth usage in LTE sensor networks using OMNeT as a simulation tool [25]. Unfortunately, OmNeT falls into the same category as ns-3 in that it is overly complex and features far too steep of a learning curve to serve as a useful comparison for the objectives here.

In addition to the formal program mentioned here, several scholars have published works of their own centered around creating custom, special-purpose simulators. Xie, et. al. [26] proposed a discrete-time queuing network simulator. Driven by a need to detect abnormalities in their own layered networks to prevent DDoS attacks, their simulator provided the perfect development environment without needing to stage real attacks on live networks. Additionally, Aatish, Pooja, and Singh [27] developed a network simulation tool aimed at testing strategies unique to IPv6. Motivated by the lack of adequate existing IPv6 infrastructure, they created their own simulator to present their future-forward research into the emerging protocol.

III. Methodology

A. Overview

All of the aforementioned pre-established simulation tools fall short in that they require a large learning curve; an engineering student or learner wishing to construct a prototype in any of these environments must first spend significant time learning the intricacies and methods of each. Furthermore, none are realtime and most provide little or no real support for visualization. MATLAB as an environment provides these features out of the box. Uluisik and Sevgi summarize this sentiment quite appropriately in their paper developing a MATLAB GUI for mapping complex functions [28]; after summarizing the existing body of work, they determined that an inherently simple and easy-to-operate MATLAB package was the best means to demonstrate the concepts they wished to teach.

MATLAB provides a single high fidelity environment that facilitates both easy plotting of data and quickly creating robust user interfaces. The AODV simulation utilizes multiple figure windows that can simultaneously display different aspects of the algorithm's progress. It will also have the capability to plot statistical data afterwards.

The AODV simulation will start by allowing a layout scheme to be chosen and a set number of nodes to be plotted on a position grid (this project will focus on only 2-dimensional positional layouts). The user can add, remove, and manipulate nodes as desired and to create unique contrived network scenarios to test edge cases. Each node will then be able to "flood" its nearest neighbors and build a local routing table. The user can then request packets be sent between two nodes and observe the path in real time.

As the intent of the project is to provide a basic level teaching and learning tool; the simulation in its current state will skip some aspects of wireless network simulation. Notably, this simulation does not implement queuing. Interference, sources of time delay, and power utilization are all removed so that the learner can focus more on the basic principles at play.

B. Process Flow

Due to the lack a a sufficient pointer structure in MATLAB, the decision was made to process the path propagation hierarchically, rather than individually. This has the added benefit of

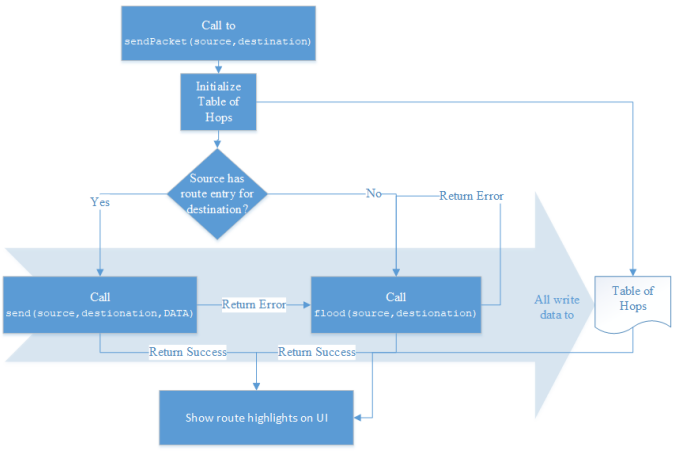

Fig. 2. Process flow for sendPacket()

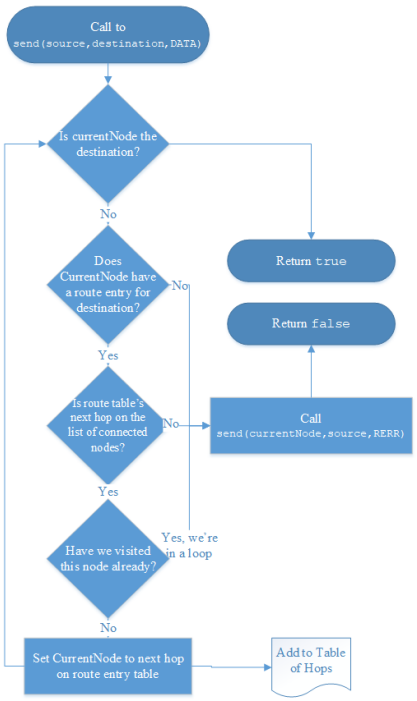

Fig. 3. Process flow for send()

providing simulations that are repeatable as variations in clock speed, processing time, etc. do not delay transmissions and possibly cause them to be received in different orders across different experimental runs. Therefore, no race conditions or deadlocks are possible scenario. This also reduces individual processing slightly by removing much of the burden of repeated calculations at the node level.

Tha main logic of the program is handled beneath the user interface. Once the user interface receives a request to send a packet, the program flow is as follows 2. A table data structure stores all the messages sent throughout the network as part of this interaction. It contains ¡node, nextNode, messageType, depth¿, where depth is the iteration that it will be highlighted

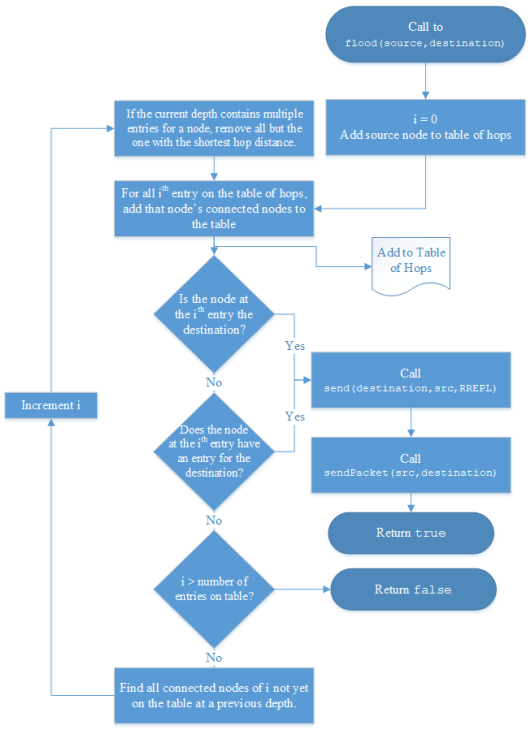

Fig. 4. Process flow for flood()

on when being diaplayed on the user interface. If the source node has a valid route to the destination, a call to send() is atttempted. If the call to send fails (i.e. encounters a route error), then flood() is called instead. Both subroutines write their changes to the table of hops. Flood may be called repeatedly, if necessary. Once the computation is complete, the table of hops is parsed and the routes are iteratively highlighted on the user interface to simulate data flow. See Appendix VIII-A for complete commented code.

The call to send() is relatively straightforward 3. Starting at the source and going until the current node is the destination, it verifies that the current node has a route to the destination that is still reachable and repeats for the next node according to the route table. In case of failure, a route error is send backwards and the subroutine returns false. See Appendix VIII-B for complete commented code.

The call to flood() is a little more involved 4. The source node is added to the table of hops. The algorithm walks down the table reading and adding more entries as it goes. For each entry, that node's connected nodes not already on the table are added to the table at depth + 1. Upon incrementing, the subroutine checks to see if multiple instances of the same node was added at the current depth. It removes all but the node with the shortest hop distance. If a node is the destination or knows a path to the destination, it instead reaturn a route reply before continuing. The subroutine terminates when the entire table has been traversed, only returning as true if a reply was sent at some point. See Appendix VIII-C for complete commented code.

Only after all subroutine calls is the user interface updated. The nodes' saved route tables are updated as the hops are displayed on the node graph.

IV. Results

A. Example 1 (Figures 5 & 6)

The simplest scenario involves a node sending a transmission across the network with no prior knowledge whatsoever. Take the node layout in figure 5. Lines show the available links to each node's nearest neighbors. A packet will be sent from node D to node G. D has no route to G so it emits a route request (RREQ), which is forwarded by each node that receives it and thus propagates throughout the network via flooding (notated by the cyan links). Along the way, each node sets up reverse route entries detailing the next node, hop count, and sequence number in order to send to the source of the route request. Upon receiving the request, node G replies with a route reply (RREPL)(notated by the blue links), which the intermediate nodes know how to direct thanks to the reverse routes that were just set up. Finally, now that D has a valid route to G, data is sent (notated by the green links). Subsequent transmissions to G from D will continue to use this route and require no new overhead.

As an aside, this simulation does not visualize any RREQ messages that are not acknowledged. In reality, each node is sending out RREQs in all directions, including the direction that it just received it's own RREQ from, but that node will discard it upon receipt.

Upon completion, each node's route table can be ovserved in the figure 6. Note that nodes contain a reverse route entry for D thanks to the flooding. Nodes A, D, and E are along the data path and contain routes to G.

B. Example 2 (Figures 7 & 8)

In Example 2, the previously stored route tables (figure 7 are retained. In this case, node C will try sending to node G based on the information already known. Since C contains no entry for G in its route table, it must flood; however, the intermediate nodes A, D, and E all do have entries for G. Instead of propagating the RREQ message onward, they respond with a RREPL. C updates its route table upon receipt of each RREPL, choosing the route with the lowest hop count to G each time. Eventually, C is able to send out to E, who the continues to send to G via its route entry established in Example 1.

Again, the route table is shown in figure 8. Note the additional entries in A, B, D, E, and F.

C. Example 3 (Figures 9 & 10)

One of the most beneficial features of AODV is its elegant handling of link failures. A link failure occurs when a pre-established route is no longer possible due to node movement or interference. AODV handles this by sending route error (RERR) messages, as shown in figure 9. In this example, node D once again tries to send to node G, but intermediate node E has moved, breaking the route. D starts off by trying to send normally, but A must reply with a RERR when it realizes it

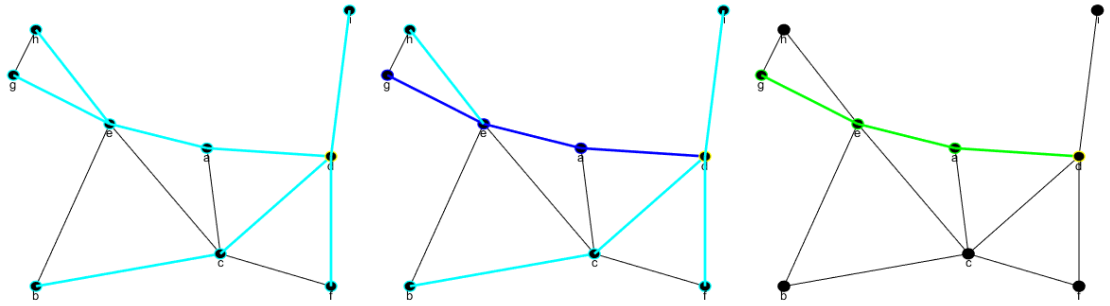

Fig. 5. Simple route request and reply, route messages RREQ, RREPL, and data

AODV Sim - Table View

Node a — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	g	e	2	1	2

Node b — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	c	2	1	1

Node c — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1

Node d — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	g	a	3	1	2

Node e — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	a	2	1	1
2	g	g	1	1	2

Node f — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1

Node g — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node h — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node i — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1

Fig. 6. Simple route request and reply, node route tables

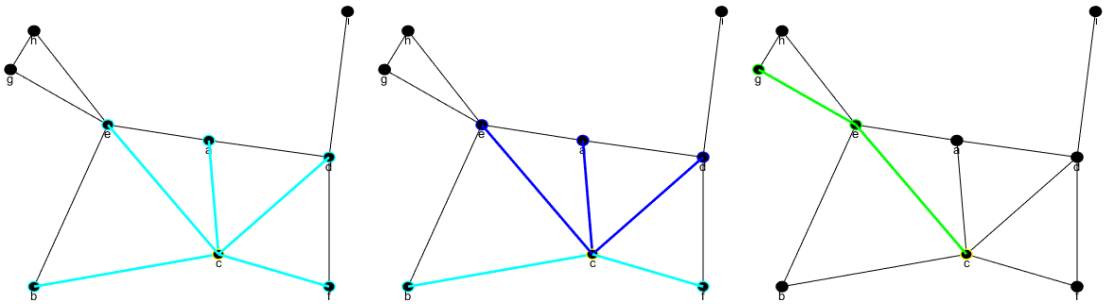

Fig. 7. Route request and reply from intermediate nodes, route messages RREQ, RREPL, and data

AODV Sim - Table View

Node a — SeqNum: 7

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	g	e	2	1	3
3	c	c	1	1	1

Node b — SeqNum: 3

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	c	2	1	1
2	c	c	1	1	1

Node c — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	g	e	2	1	2

Node d — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	g	a	3	1	3
2	c	c	1	1	1

Node e — SeqNum: 3

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	a	2	1	1
2	g	g	1	1	4
3	c	c	1	1	1

Node f — SeqNum: 5

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	c	c	1	1	1

Node g — SeqNum: 3

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node h — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node i — SeqNum: 5

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1

Fig. 8. Route request and reply from intermediate nodes, node route tables

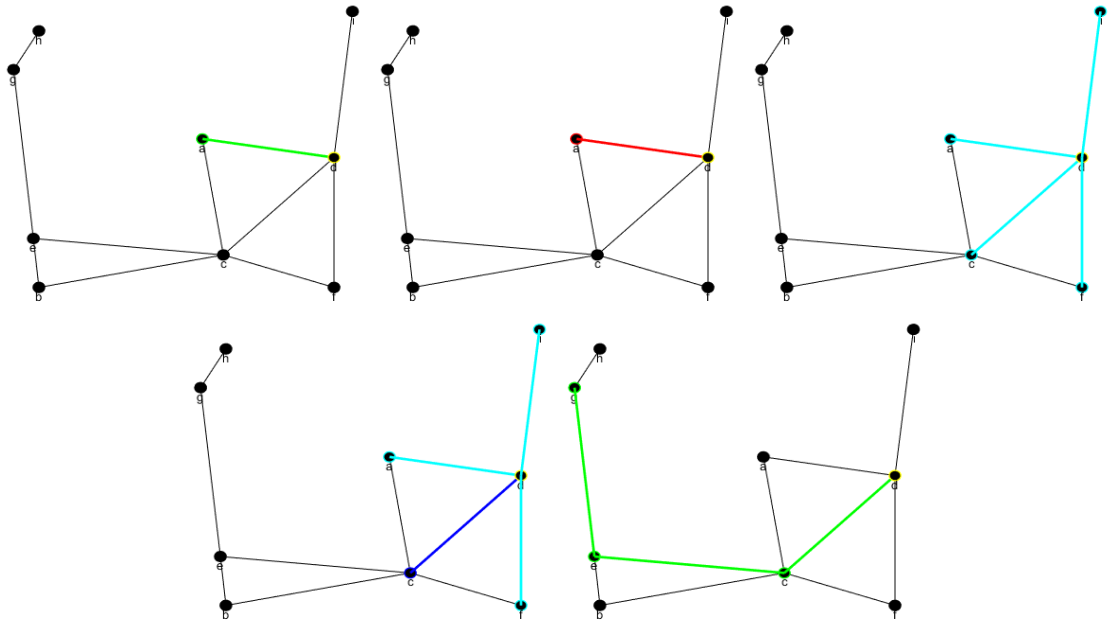

Fig. 9. Link breakage, route messages RREQ, RREPL, and data

AODV Sim - Table View

Node a — SeqNum: 2

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	2
2	c	c	1	1	1

Node b — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	c	2	1	1
2	c	c	1	1	1

Node c — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	g	e	2	1	3

Node d — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	c	c	1	1	1
2	g	c	3	1	2

Node e — SeqNum: 2

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	a	2	1	1
2	g	g	1	1	4
3	c	c	1	1	1

Node f — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1
2	c	c	1	1	1

Node g — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node h — SeqNum: 2

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	e	3	1	1

Node i — SeqNum: 1

	dest	nextHop	hopCnt	seqNum	lifeTime
1	d	d	1	1	1

Fig. 10. Link breakage, node route tables

can't reach E. This message propagates back up to D and both nodes cancel out their routes to G. D must once again flood by sending out RREQ packets. C replies that it knows a route (the one established in example 2) and D then proceeds to send to G via C.

The subsequent routing table is shown in figure 10. The sequence number have increased several times. Each time the nodes notice a change in their local network topology the increase their sequence numbers. All nodes connected or disconnected from E have been incremented to 2.

While this is an exceptional method of handling a single link failure or small change in topology, it starts to become rather burdensome when multiple route failures occur. One can imagine a contrived highly scenario where the topology changes in such a way as to allow every intermediate node to obtain a false route entry for the desired destination. In such a case, the source node must first attempt to send and receive a reply. It then floods but receives multiple replies. Because AODV provides no method of maintaining a node's past activity, it must attempt sending again, receive an error again, flood again, attempt to send again, etc. It can only cancel out one intermediate node's route destination entry at a time, potentially causing enormous overhead. Although not reproduced here, this can be seen in the simulator in real time by mnually moving all the nodes in the network once valid routing paths have been established.

V. Analysis

To investigate the problem of quantifying exactly how much overhead various network topologies generate, statistics from several repeated transmissions were aggregated. In each simulation, the network had it's nodes reset to an established starting position and the route tables for all nodes were cleared. Nodes were set in motion and updated their position regularly. Each node had a pre-established step size and direction. Nodes reflected off the boundaries of the simulation area in order to

ensure that they stayed relatively close to each other and did not go out of bounds. Random traffic was generated and sent through the network. The random number generator used a static seed to ensure that all variables were consistent from experiment to experiment.

To begin with, traffic was evaluated on a static network (figure 11). Here, the number of RREQ messages steadily increases until around the 50th packet, at which time nodes have most of the possible connections mapped out. The data rate climbs steadily since no transmissions fail and the RERR rate is accordingly zero.

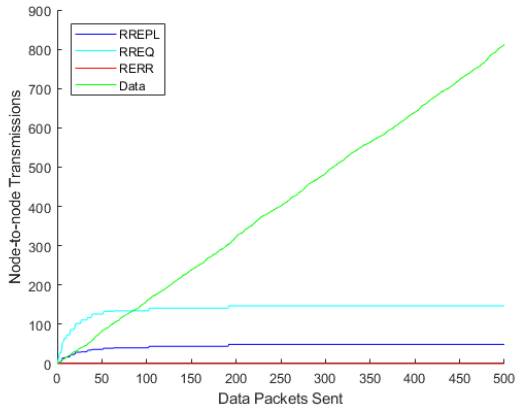

Fig. 11. Network overhead with static nodes

A more realistic scenario may involve more frequent movement. Figure 12 shows overhead with movement every 50 packets and figure 13 with movement every 10 packets. The frequent movement in figure 13 causes the route request rate to finally surpass the data rate and would likely be enough to overwhelm the controllers on most realistic nodes. In such a high mobility case, a specialized routing algorithm would likely need to be utilized as AODV's performance is lacking here.

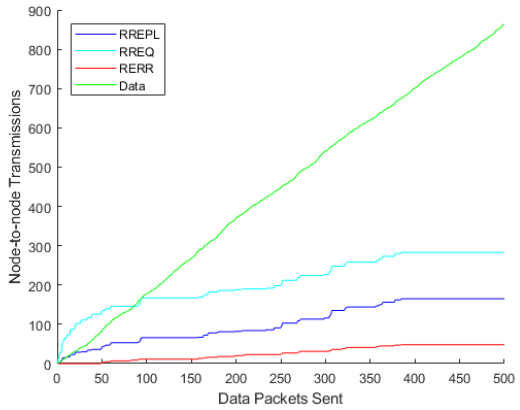

Fig. 12. Network overhead with nodes moving every 50 packets sent

Figure 14 shows the number of hops required for each transmission. Although the numbers are not initially as dissimilar

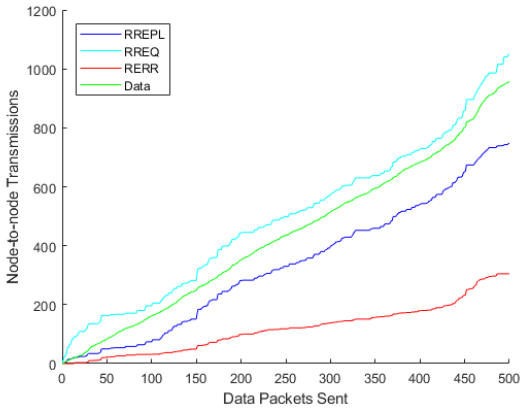

Fig. 13. Network overhead with nodes moving every 10 packets sent

as might be expected, it is clear that the "movement every 10 packets" data set had consistently higher transmissions counts towards the upper end. The "no movement" data set also had a few towards the high end as well. This represents the initial flooding, which requires all nodes in the network to send messages. This flooding is required, to some degree, at all levels and is what evens out the hop count statistics here.

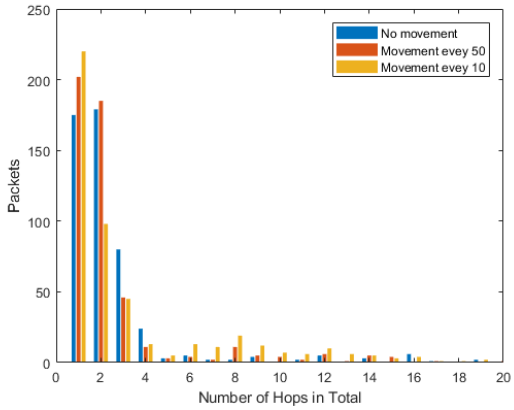

Fig. 14. Number of hops for various types of motion

VI. Future Work

Although this simulation provides an excellent base-level teaching tool, it falls short of accurate analysis in several areas. The foremost improvement that could be made would be to implement node queuing. While this would require a large burden of additional work, it would garner some interesting insights into node mechanics and provide a more realistic reflection of how ad-hoc networks truly perform.

Additionally, to enhance the usefulness of this tool, it would be beneficial to be able to compare AODV to another routing protocol like destination-sequenced distance-vector (DSDV) [29] or dynamic source routing (DSR) [30]. Being able to compare and contrast the differences and weigh the benefits of AODV would be a valuable opportunity.

VII. Conclusions

The MATLAB-based ad-hoc on-demand distance vector (AODV) simulation presented here provides a meaningful method of demonstrating basic routing concepts quickly and easily, circumventing the learning curve of more specialized network simulation tools and facilitating visual learning. The MATLAB environment provides for easy inspection and expansion into additional analysis with the provided functionality. The examples and analysis presented serve as sufficient proof-of-concept of this endeavor to emulate AODV.

The complete code, as well as usage documentation can be found on the author's Github at https://github.com/stewythe1st/AODV-Matlab

References

[1] D. Robinson, "What are the most disliked programming languages?" Tech. Rep., 2017. [Online]. Available: https://stackoverflow.blog/2017/10/31/disliked-programming-languages/

[2] S. Overflow, "Developer survey results 2017," Tech. Rep., October 2017. [Online]. Available: https://stackoverflow.blog/2017/10/31/disliked-programming-languages/

[3] Y. Rekhter and T. Li, "Ad hoc on-demand distance vector (aodv) routing," Internet Requests for Comments, RFC Editor, RFC 3561, July 2003. [Online]. Available: https://tools.ietf.org/html/rfc3561

[4] T. Saeed, Z. Iqbal, H. Afzaal, and N. A. Zafar, "Formal modeling of traffic based flooding procedure of aodv for mobile ad hoc networks," in *2016 International Conference on Emerging Technologies (ICET)*, Oct 2016, pp. 1–6.

[5] Y. Feng, B. Zhang, S. Chai, L. Cui, and Q. Li, "An optimized aodv protocol based on clustering for wsns," in *2017 6th International Conference on Computer Science and Network Technology (ICCSNT)*, Oct 2017, pp. 410–414.

[6] S. S. Devi and K. T. Sikamani, "Improved route error tolerant mechanism for aodv routing protocol in manet," in *2013 International Conference on Current Trends in Engineering and Technology (ICCTET)*, July 2013, pp. 187–190.

[7] A. M. E. Ejmaa, S. Subramaniam, Z. A. Zukarnain, and Z. M. Hanapi, "Neighbor-based dynamic connectivity factor routing protocol for mobile ad hoc network," *IEEE Access*, vol. 4, pp. 8053–8064, 2016.

[8] S. Mittal, S. Bisht, K. C. Purohit, and A. Joshi, "Improvising-aodv routing protocol by modifying route discovery mechanism in vanet," in *2017 3rd International Conference on Advances in Computing,Communication Automation (ICACCA) (Fall)*, Sept 2017, pp. 1–5.

[9] J. J. Ferronato and M. A. S. Trentin, "Analysis of routing protocols olsr, aodv and zrp in real urban vehicular scenario with density variation," *IEEE Latin America Transactions*, vol. 15, no. 9, pp. 1727–1734, 2017.

[10] A. Abuashour and M. Kadoch, "Performance improvement of cluster-based routing protocol in vanet," *IEEE Access*, vol. 5, pp. 15 354–15 371, 2017.

[11] P. Yadav and M. Hussain, "A secure aodv routing protocol with node authentication," in *2017 International conference of Electronics, Communication and Aerospace Technology (ICECA)*, vol. 1, April 2017, pp. 489–493.

[12] MATLAB, "Wlan system toolbox," Tech. Rep., 2018.

[13] P. k. Gaur, B. S. Dhaliwal, and A. Seehra, "Analysis of power saving multicasting routing applications for mica2 motes-an event based matlab implementation," in *2009 International Multimedia, Signal Processing and Communication Technologies*, March 2009, pp. 109–112.

[14] V. Navya and P. Deepalakshmi, "Mobility supported threshold based stability increased throughput to sink using multihop routing protocol for link efficiency in wireless body area networks (m-tsimple)," in *2017 IEEE International Conference on Intelligent Techniques in Control, Optimization and Signal Processing (INCOS)*, March 2017, pp. 1–7.

[15] I. Habib, N. Badruddin, and M. Drieberg, "A comparison of load-distributive path routing and shortest path routing in wireless ad hoc networks," in *2014 5th International Conference on Intelligent and Advanced Systems (ICIAS)*, June 2014, pp. 1–5.

[16] ns 3, "What is ns-3?" Tech. Rep., 2018. [Online]. Available: https://www.nsnam.org/overview/what-is-ns-3/

[17] J. Patel and P. Kamboj, "Investigation of network simulation tools and comparison study: Ns3 vs ns2," *Journal of Network Communications and Emerging Technologies*, vol. 5, no. 2, pp. 137–142, December 2015.

[18] Y. Mai, F. M. Rodriguez, and N. Wang, "Cc-adov: An effective multiple paths congestion control aodv," in *2018 IEEE 8th Annual Computing and Communication Workshop and Conference (CCWC)*, Jan 2018, pp. 1000–1004.

[19] ns 2, "The network simulator - ns-2," Tech. Rep., 2018. [Online]. Available: https://www.isi.edu/nsnam/ns/

[20] Tetcos, "Netsim academic," Tech. Rep., 2018.

[21] K. M. Saifuddin, A. S. Ahmed, K. F. Reza, S. S. Alam, and S. Rahman, "Performance analysis of cognitive radio: Netsim viewpoint," in *2017 3rd International Conference on Electrical Information and Communication Technology (EICT)*, Dec 2017, pp. 1–6.

[22] P. Nayak and P. Sinha, "Analysis of random way point and random walk mobility model for reactive routing protocols for manet using netsim simulator," in *2015 3rd International Conference on Artificial Intelligence, Modelling and Simulation (AIMS)*, Dec 2015, pp. 427–432.

[23] A. Varga, "What is omnet++?" Tech. Rep., 2018.

[24] E. R. Chadha, L. Kumar, and E. J. Singh, "3 dimensional wsn: An energy efficient network," in *2017 2nd International Conference for Convergence in Technology (I2CT)*, April 2017, pp. 1169–1175.

[25] D. Thomas and J. Irvine, "Connection and resource allocation of iot sensors to cellular technology-lte," in *2015 11th Conference on Ph.D. Research in Microelectronics and Electronics (PRIME)*, June 2015, pp. 365–368.

[26] J. Xie, C. H. Y. Wan, K. Mills, and C. Dabrowski, "A layered and aggregated queuing network simulator for detection of abnormalities," in *2017 Winter Simulation Conference (WSC)*, Dec 2017, pp. 1073–1084.

[27] C. Aatish, R. Pooja, and S. M. Singh, "An ipv6 e-learning tool with integrated network configuration simulator," in *2017 Third International Conference on Sensing, Signal Processing and Security (ICSSS)*, May 2017, pp. 259–263.

[28] C. Uluisik and L. Sevgi, "A matlab-based visualization package for complex functions, and their mappings and integrals," *IEEE Antennas and Propagation Magazine*, vol. 54, no. 1, pp. 243–253, Feb 2012.

[29] C. E. Perkins and P. Bhagwat, "Highly dynamic destination-sequenced distance-vector routing (dsdv) for mobile computers," in *ACM SIGCOMM computer communication review*, vol. 24, no. 4. ACM, 1994, pp. 234–244.

[30] D. Johnson, Y. Hu, and D. Maltz, "The dynamic source routing protocol (dsr) for mobile ad hoc networks for ipv4," Internet Requests for Comments, RFC Editor, RFC 4728, February 2007. [Online]. Available: https://tools.ietf.org/html/rfc4728

VIII. Appendix

A. sendPacket()

```
function [myTable] = sendPacket(src,dest,quickMode)

    % Check args
    if(nargin <= 2)
        quickMode = false;
    end
    if(nargin > 3 || src == dest)
        return
    end

    % Bring globals into scope
    global nodes colors routeLifetime;

    % Persistant variables
    requestDepth = 0;
    depth = 0;

    % Get sequence number
    idx = find(nodes(src).routeTable.dest==dest)';
    if(isempty(idx))
        seqNum = 0;
    else
        seqNum = nodes(src).routeTable.seqNum(idx);
    end

    % Initialize our table and add the start node to it
    myTable = table(0,0,src,src,colors.Src);
    myTable.Properties.VariableNames = {'Depth','HopCnt','Node','From','Color'};

    % If this node has a route entry for our dest, try sending normally
    tryAgain = false;
    if(any(idx))
        if(~send(src,dest,colors.Data))
            tryAgain = true;
            flood(src,dest);
        end
    else
        tryAgain = true;
        flood(src,dest)
    end

    % Delete any routes marked by RERR
    for route = find(myTable.Color == colors.RERR)'
        idx = find(nodes(myTable(route,:).Node).routeTable.dest == dest);
        nodes(myTable(route,:).Node).routeTable(idx,:) = [];
    end

    if(quickMode)
        quickUpdate();
    else

        % Make a timer to iteratively light up paths
        colorTimer = timer('Name','colorTimer',...
                           'ExecutionMode','fixedDelay',...
                           'Period',0.5,...
                           'TimerFcn',@setColor);
        depth = 0;
        start(colorTimer)
    end

    % Remove route with expired lifetimes
    for i=1:numel(nodes)
        for j = 1:size(nodes(i).routeTable,1)
            expired = find(nodes(i).routeTable.lifeTime > routeLifetime);
            nodes(i).routeTable(expired,:) = [];
        end
    end
```

B. send()

```
    function [success] = send(sendSrc,sendDest,color)
        success = false;

        % Check if we already have a route entry for this
        currentNode = sendSrc;
        visitedNodes = [];
        while true
```

```
            depth = depth + 1;

            % Look in currentNode's routeTable for the nextHop node
            nextNode = find(nodes(currentNode).routeTable.dest==sendDest);

            % Exit if no node was found
            if(~any(nextNode))
                % If we were expecting to have a valid path, send a RERR
                if(color == colors.Data)
                    replyTable = floodReply(sendSrc,currentNode,colors.Data,colors.RERR);
                    myTable = [myTable;replyTable];
                    success = true;
                    tryAgain = true;
                    return
                end
                break;
            end

            % Update lifetime field now that we've used this route
            nodes(currentNode).routeTable(nextNode,:).lifeTime = ...
                nodes(currentNode).routeTable(nextNode,:).lifeTime + 1;

            % Convert from index in routeTable to actual node index
            nextNode = nextNode(1);
            nextNode = nodes(currentNode).routeTable(nextNode,:).nextHop;

            % Exit if this nextNode is unreachable
            % Send a RERR back to source
            if(~any(find(nodes(currentNode).connectedNodes==nextNode)))
%                 myTable = [myTable;{depth,depth,currentNode,currentNode,colors.RERR}];
%                 send(currentNode,sendSrc,colors.RERR);
%                 myTable = [myTable;{depth,depth,sendSrc,sendSrc,colors.RERR}];
                if(currentNode == sendSrc)
                    myTable = [myTable;{depth,depth,sendSrc,sendSrc,colors.RERR}];
                else
                    replyTable = floodReply(sendSrc,currentNode,colors.Data,colors.RERR);
                    myTable = [myTable;replyTable];
                end
                tryAgain = true;
                success = true;
                return
            end

            % Exit if we're in a loop
            if(any(find(visitedNodes==currentNode)))
                replyTable = floodReply(sendSrc,currentNode,colors.Data,colors.RERR);
                myTable = [myTable;replyTable];
                tryAgain = true;
                success = true;
                return
            end
            visitedNodes = [visitedNodes,currentNode];

            % Update our path table
            myTable = [myTable;{depth,depth,nextNode,currentNode,color}];
            currentNode = nextNode;

            % Exit when we've reached the destination
            % Set success to true
            if(currentNode == sendDest)
                success = true;
                break
            end
        end
    end
```

C. *flood()*

```
    function [] = flood(floodSrc,floodDest)

        % Walk down table rows and add connected nodes breadth-first
        i = 0;
        success = false;
        replyNodes = [];
        while true
            i = i + 1;

            % Get connected nodes here
            currentNode = myTable.Node(i);
            connectedNodes = nodes(currentNode).connectedNodes;
            depth = myTable.Depth(i)+1;
```

```
        % Remove duplicates when we've finished at this depth
        if(depth > myTable.Depth(end))
            % For each duplicated value in myTable.Node
            for j = find(hist(myTable.Node,unique(myTable.Node))>1)
                % Find the distance between Node and From for all
                % occurrances of this duplicated node
                dist = [];
                duplicates = find(myTable.Node==j & myTable.Color == colors.RREQ)';
                for k = duplicates
                    dist = [dist,sqrt((nodes(myTable.Node(k)).x - nodes(myTable.From(k)).x)^2 ...
                                + (nodes(myTable.Node(k)).y - nodes(myTable.From(k)).y)^2)];
                end
                % Remove occurrances but the one with the min distance
                [~,idx] = min(dist);
                duplicates(idx) = [];
                myTable(duplicates,:) = [];
                i = i - numel(find(duplicates<=i));
            end
        end

        % If this node happens to have a valid entry on the route
        % table, go ahead and send normally from here out
        if(any(find(nodes(currentNode).routeTable.dest==floodDest)) || currentNode == floodDest)
            replyNodes = [replyNodes;currentNode];
            success = true;
        else

            % Add each of this node's connected nodes unless its already
            % on the table before this depth
            for connectedNode = connectedNodes
                if(currentNode ~= floodDest)
                    if(~any(find(myTable.Node==connectedNode & myTable.Depth < depth)))
                        myTable = [myTable;{depth,depth,connectedNode,currentNode,colors.RREQ}];
                    end
                else
                    success = true;
                end
            end
        end

        % Check for termination
        if(i >= size(myTable,1))
            if(isempty(replyNodes))
                tryAgain = false;
            end
            break
        end
    end
    requestDepth = depth-1;

    if(success)
        tempDepth = depth;
        for reply = replyNodes'
            depth = tempDepth;
            replyTable = floodReply(floodSrc,reply,colors.RREQ,colors.RREPL);
            myTable = [myTable;replyTable];
        end
    end

end
```