BATMANSim

A Better Approach To Mobile Ad hoc Networking Simulator

Brittany McGarr

CPE400 – 1001

Fall 2015

Introduction

With the rising popularity of the Internet infiltrating every aspect of our daily lives, connection protocols have become vital for extending and maintaining the flow of data between users and devices. In particular, the humble cell phone has evolved into an icon of personal computing: the smartphone. Almost every system a person will interact with has some stream of data flowing to them through various protocols.

Mobile Ad hoc Networks (MANETs) are a new field of networking that take the short-range communication capabilities of many mobile devices to transport information in an ad hoc network. This means that devices can opt in to share in the transaction of data, further extending a bridge of neighboring devices that can cover great distances where hardware-supported WiFi or cell networks may not be able. For a team of rescue workers, this could mean extending critical networking features into blackout zones.

Optimized Link State Routing Protocol (OLSR), is a proactive protocol in which link-state tables are maintained in each node (device) about all participants in the mesh-structure of the network. This was effective for a small group on the same network, but as users opt in to the group and extend the table, the time it takes to update and maintain the link states grows exponentially. For a daily use, such as city-wide networking, this may be unrealistic without many supporting nodes hooked to a traditional WiFi or Ethernet gateway.

B.A.T.M.A.N. seeks to reduce this exponential cost by maintaining only those neighbors it can swiftly and effectively communicate with, or “shorter-hops.” These short hops allow B.A.T.M.A.N. to maintain a much smaller table of localized networks with neighboring networks joining in through those closest.

To maintain these connections, B.A.T.M.A.N. nodes send out Originator Messages (OGMs) periodically through the network. The OGMs are compact data packets of 52-bytes that contain the originator's address, the forwarding sender's address, a Time To Live (TTL), and a sequence number. The sequence number is chosen by the originator node (usually 0) and is updated with each OGM. Those OGM messages are used to update the connections known to the recipient node, where the sender's address is replaced with the current node and forwarded out again. The sequence number helps to determine which connections may be more stable with lower sequences indicating rapid, secure connections, and larger or missing sequences indicating lost or delayed OGMs.

Overall, B.A.T.M.A.N. seems to offer fantastic glimpses into a provider-less networking scheme that is practical for areas otherwise unsuited to tower or satellite-based networking, such as rescue efforts in remote locations. With such a protocol in place, many rescue workers could collaborate and transmit data critical to assessing relief efforts in spite of having no infrastructure, such as mobile WiFi ports or cellular data. These networks could be established on the fly with users extending the reach of other connectivity in the interim.

This project, in particular, aims to demonstrate the adaptable qualities of the BATMAN protocol through a simulation. The user may create connections and nodes as if through a network administrator, run various time simulations, and even pass around messages to view how transport is achieved.

Contribution

The BATMANSim simulator allows the user to interact with a BATMAN-based system to view the effects of creating users, adding connections, and running the protocol for analyzing the effects.

As detailed in the introduction, the BATMAN protocol relies on OGM broadcasting from one node to its nearest neighbors. These OGMs are received at the neighbors, stored in their OGM collections, and retransmitted through the system. At each hop, the sender's IP is changed to the recipient's so that neighbors know which direction to send messages (actually packets, but we will use message to indicate payload-carrying, non-OGMs).

For example, Alice, Bob, and Cathy are a network. Alice knows Bob, and Bob knows Cathy, but Cathy and Alice do not know each other directly. Alice would like to network in the future, so she sends a message (in keeping with this analogy, we will use business cards as OGMs) to Bob. Bob writes his name on the business card, too. Bob eventually meets with Cathy and hands off the business card. The business card has Alice's name, but not her phone number. Cathy would like to contact Alice, so she gets a hold of Bob, who she knows was the last person to hold the card because he signed his name on it. Bob eagerly forwards the contact back to Alice.

In the BATMAN protocol, these exchanges are performed every minute or so to propagate the information to the outlying nodes. In a two-hop scenario such as the example described, the time for Cathy to receive the business card would be the time it takes Alice to create it (she can only order so many at a time and must wait for those to be printed), the time it takes Alice to transport the card to Bob, the time for Bob to write his name, and the time for the business card to be successfully handed off to Cathy. In a real BATMAN scenario, this propagation is in the order of hundreds of milliseconds, so the simulator is fitted to this model for the time increments.

For each simulation run time, a number of actions must occur: the system performs maintenance (decrementing time to live values to simulate time spent in the system), each node must generate OGMs if their time has arrived to send out a fresh wave, each node must receive an item from their queue, and the system must transport an item from each node to its next-hop.

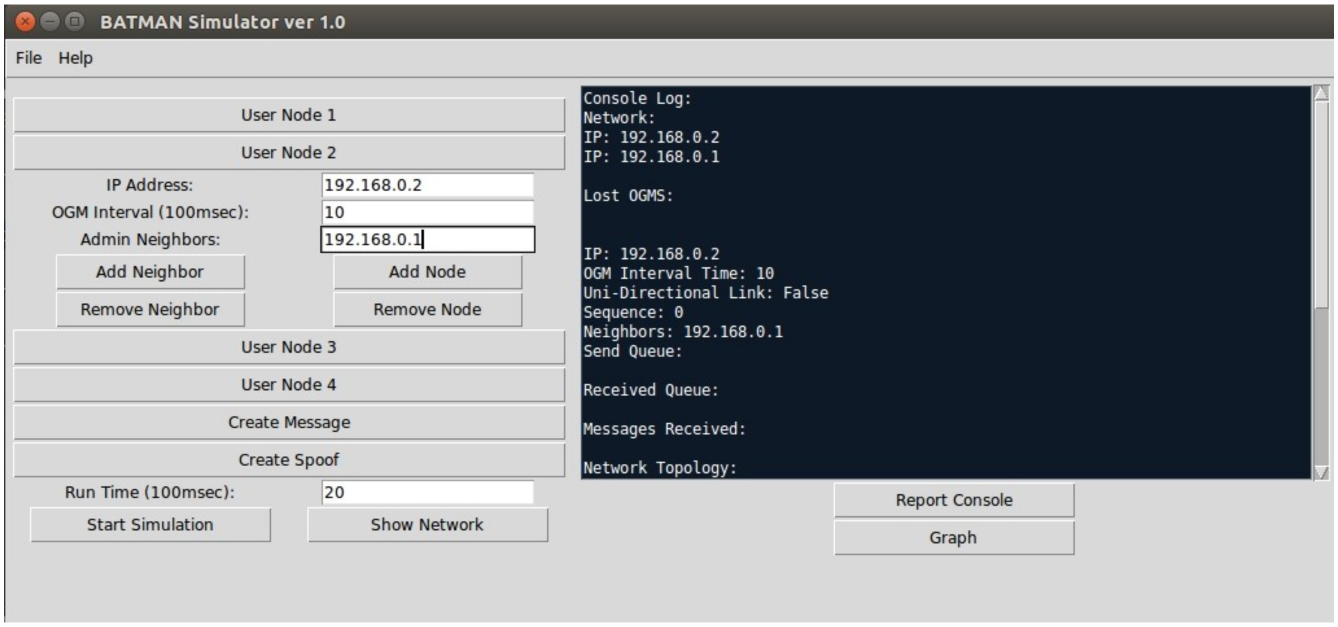
In practice, OGMs have a TTL or Time-To-Live value that gets steadily decremented to limit the maximum number of hops allowed in the system. Because the network may be extensive, keeping nodal topology limited to reasonable distances keeps it from being impractical and slow. In the simulator, the TTL also helps to phase out users that have left the network, removing them when the OGM has expired.

Generating OGMs is the network's primary method of updating its routing schemes. Because these OGMs are propagated through the system, they must be periodically updated and marked with sequence numbers to indicate which copies are the freshest. Higher sequence numbers indicate more recent copies. Supposing a node received two OGMs from the same originator; how, then, would it know which was closer? One way to check would be to see which OGM has the higher sequence number, indicating that the copy is the newest sent from the originator, and therefore, faster to arrive than the delayed copy received through another node. This would be the more desirable route to take, so the recipient will store the higher-order OGM and propagate that copy.

Upon receiving an item from the queue, the protocol must determine whether the node comes from a neighbor (having both originator IP and sender IP identical) and update the OGMs collected to indicate this latest information. Additionally, the user may need to forward an OGM to its nearest neighbors, spreading awareness of the sending node to the rest of the system. Finally, if the message is not an OGM and intended for a destination node, this destination should be compared for the next-hop path as this message must be propagated through the system until the destination is found. To do this, the receiving node checks the destination IP against the received OGMs in its table. If an OGM has come to the node, that means it has some sort of path through the system. The recipient then appends the next-hop node address given from the sender IP in the OGM and forwards the packet through the system just as Cathy contacted Bob to forward a message to Alice.

While the workings may seem like a lot of information to keep a hold of, BATMAN is actually a very simple system in practice, simply passing messages along without any need for route calculation at each step. However, its blithe simplicity can also be its downfall. As nodes exit the system, that information takes a while to propagate. Each OGM must expire at the neighbor nodes, going through the system and taking an average of 180 seconds before any node notices something has gone wrong. In that time, messages could be passed in next-hop fashion, creating a loop as the message is never delivered.

The BATMANSim works in a similar fashion. In the scenario, the user of the simulator acts as the network administrator, phasing in nodes or removing them. Simulations are run in discrete time intervals (a recommended minimum of 4-5 time slots accounts for each step of the system updates). The network simulator then runs these time slots with the protocol, forwarding and receiving OGMs, and the resultant network is displayed for the user.



IP Address: 192.168.0.2
OGM Interval (100msec): 10
Admin Neighbors: 192.168.0.1
Run Time (100msec): 20

```
Console Log:
Network:
IP: 192.168.0.2
IP: 192.168.0.1

Lost OGMS:


IP: 192.168.0.2
OGM Interval Time: 10
Uni-Directional Link: False
Sequence: 0
Neighbors: 192.168.0.1
Send Queue:

Received Queue:

Messages Received:

Network Topology:
```

Figure 1: The BATMANSim main screen.

Initially, the user poses as a network administrator. User Nodes 1-4 open to drop down entry fields that can be populated with data including the node's IP address, OGM broadcast interval, and administrator-provided neighbors for initial configuration. Populate the fields with the desired configuration. Add Neighbor or Remove Neighbor require valid entries in the field, so populate those fields after the initial nodes have been constructed. The Console Log to the right displays the network's present configuration and is updated when Show Network is clicked or after a simulation run. Report Console writes the Console Log to a file for viewing later, and Graph draws a graph given the known nodes and edges of the network and prints the graph to a file.

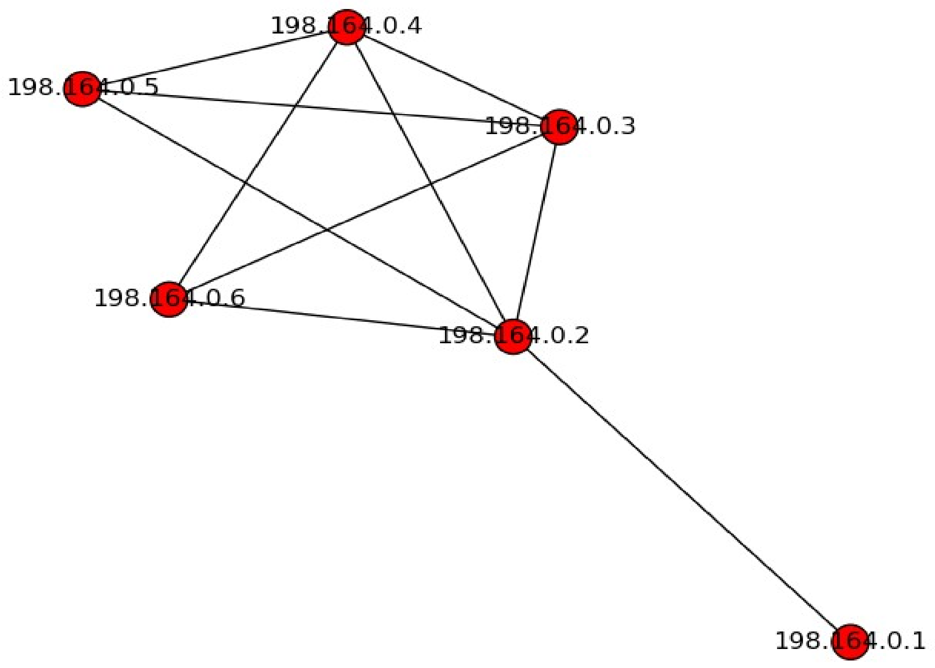
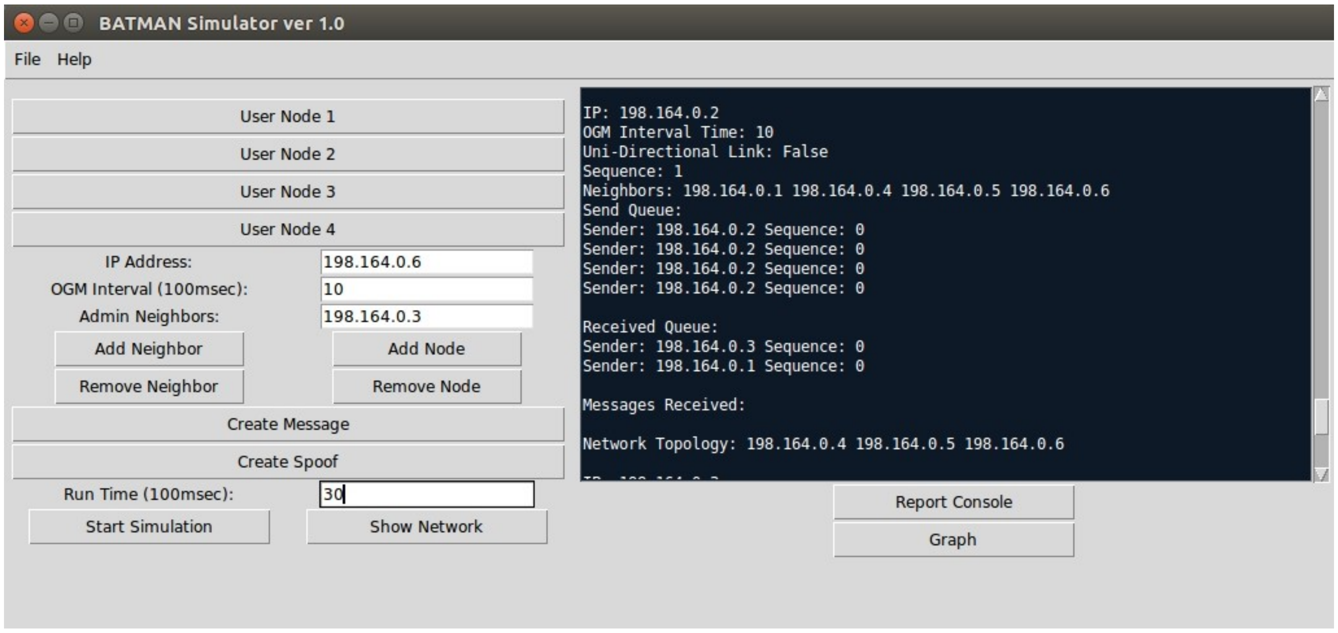


Figure 2: A sample graph from populated data.

Figure 2 shows a possible configuration of an initial network. Notice that more than four nodes may be populated in the network. Simply reuse the User Node entries by creating a new IP Address field. The user can go back to editing a node by repopulating the IP Address field with the target node's IP Address.

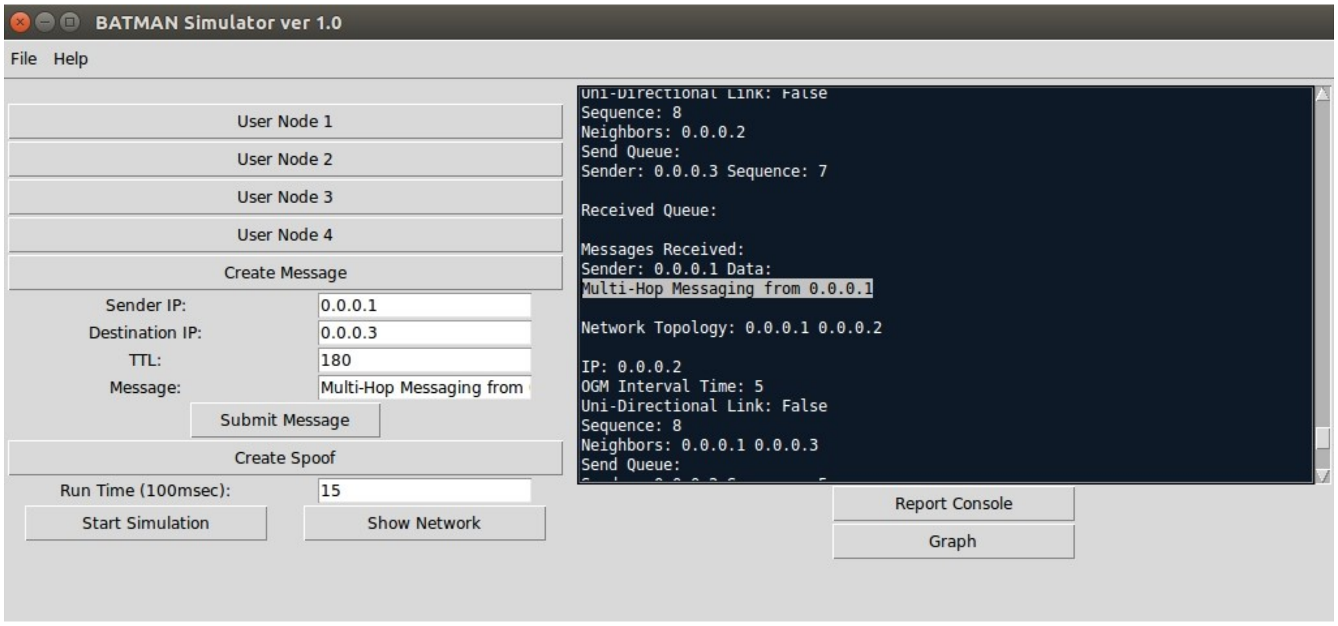


IP Address: 198.164.0.6
OGM Interval (100msec): 10
Admin Neighbors: 198.164.0.3
Run Time (100msec): 30

```
IP: 198.164.0.2
OGM Interval Time: 10
Uni-Directional Link: False
Sequence: 1
Neighbors: 198.164.0.1 198.164.0.4 198.164.0.5 198.164.0.6
Send Queue:
Sender: 198.164.0.2 Sequence: 0
Sender: 198.164.0.2 Sequence: 0
Sender: 198.164.0.2 Sequence: 0
Sender: 198.164.0.2 Sequence: 0

Received Queue:
Sender: 198.164.0.3 Sequence: 0
Sender: 198.164.0.1 Sequence: 0

Messages Received:

Network Topology: 198.164.0.4 198.164.0.5 198.164.0.6
```

Figure 3: The network has been run for 30 time intervals.

Figure 3 shows the network from Figure 2 run through 30 time intervals. As a result, the displayed node has received four confirmed next-hop “Neighbors;” three of which have been confirmed through OGMs. “Network Topology” is the collection of received OGMs that are most up to date in the system. The GUI only displays the originator IP addresses. Full reports of the contained OGMs are available in report printouts through “Save” in the “File” menu option. Send and Received Queues also display only sender IP and sequence number for compact reference in the GUI display but also contain the full OGM packet in report printing.

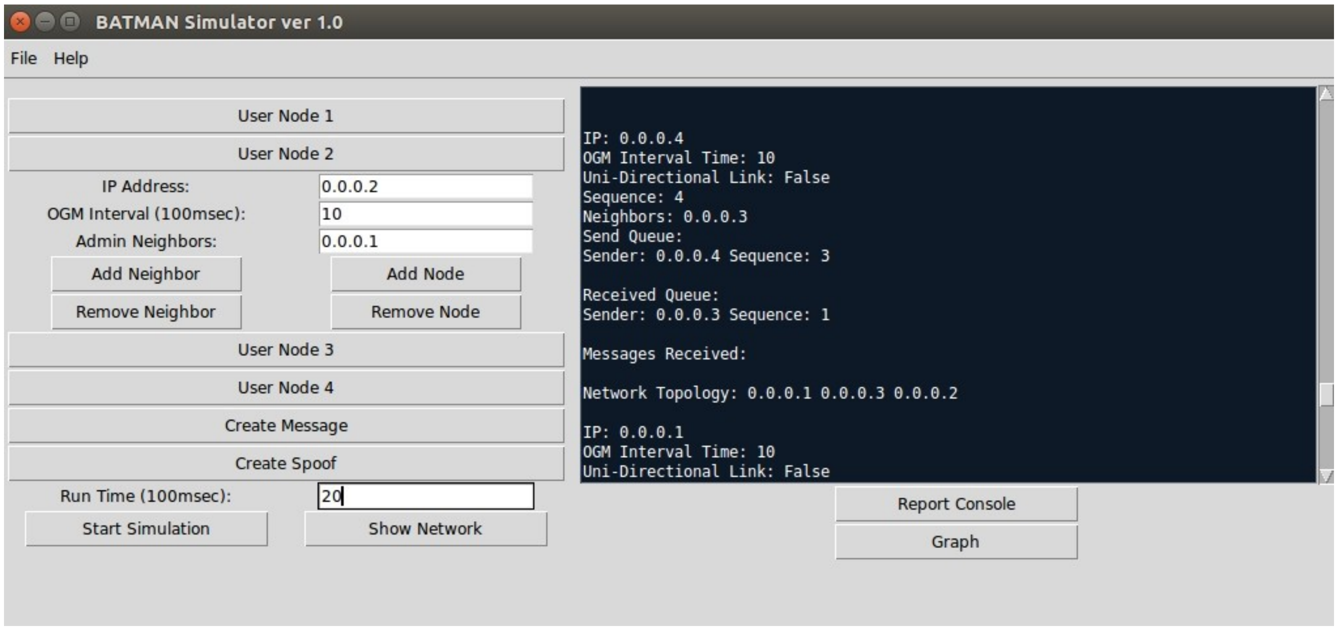


Sender IP: 0.0.0.1
Destination IP: 0.0.0.3
TTL: 180
Message: Multi-Hop Messaging from
Run Time (100msec): 15

```
Uni-Directional Link: False
Sequence: 8
Neighbors: 0.0.0.2
Send Queue:
Sender: 0.0.0.3 Sequence: 7

Received Queue:

Messages Received:
Sender: 0.0.0.1 Data:
Multi-Hop Messaging from 0.0.0.1

Network Topology: 0.0.0.1 0.0.0.2

IP: 0.0.0.2
OGM Interval Time: 5
Uni-Directional Link: False
Sequence: 8
Neighbors: 0.0.0.1 0.0.0.3
Send Queue:
```

Figure 4: Message creation and receiving through the Create Message drop down menu.

In addition to simply running the network and detecting next-hop and network information, users can create messages to simulate passing through the network. In Figure 4, a message was created from node 0.0.0.1 to node 0.0.0.3 with a payload. The message passed through 0.0.0.2 to get to 0.0.0.3 when the originator learned of the nearest hop in its network topology.

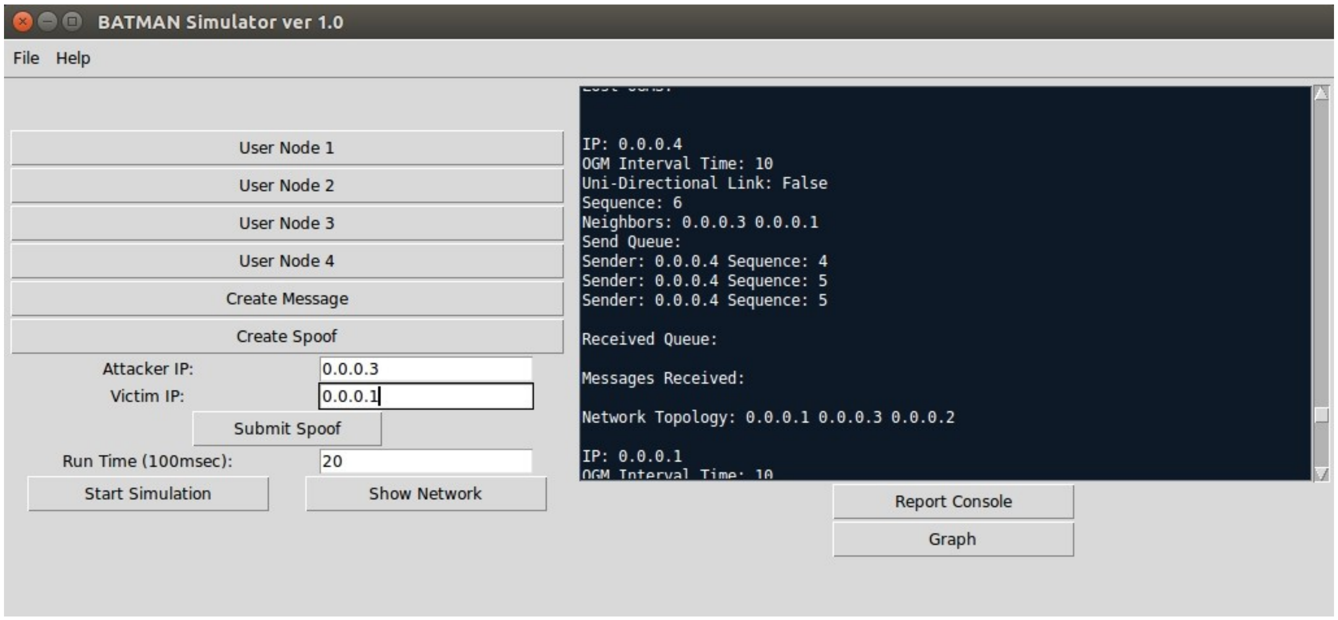


IP Address: 0.0.0.2
OGM Interval (100msec): 10
Admin Neighbors: 0.0.0.1
Run Time (100msec): 20

```
IP: 0.0.0.4
OGM Interval Time: 10
Uni-Directional Link: False
Sequence: 4
Neighbors: 0.0.0.3
Send Queue:
Sender: 0.0.0.4 Sequence: 3

Received Queue:
Sender: 0.0.0.3 Sequence: 1

Messages Received:

Network Topology: 0.0.0.1 0.0.0.3 0.0.0.2

IP: 0.0.0.1
OGM Interval Time: 10
Uni-Directional Link: False
```

Figure 5: A simple, linear network after being run long enough to discover every node.

Figure 5 shows a linear network where each node is connected one after the other. (0.0.0.1 ← 0.0.0.2 ← 0.0.0.3 ← 0.0.0.4) Here, 0.0.0.4 is properly only aware of 0.0.0.3 as its nearest neighbor and has been given time to know that the other nodes are reachable through OGMs.



Attacker IP: 0.0.0.3
Victim IP: 0.0.0.1
Run Time (100msec): 20

```
IP: 0.0.0.4
OGM Interval Time: 10
Uni-Directional Link: False
Sequence: 6
Neighbors: 0.0.0.3 0.0.0.1
Send Queue:
Sender: 0.0.0.4 Sequence: 4
Sender: 0.0.0.4 Sequence: 5
Sender: 0.0.0.4 Sequence: 5

Received Queue:

Messages Received:

Network Topology: 0.0.0.1 0.0.0.3 0.0.0.2

IP: 0.0.0.1
OGM Interval Time: 10
```

Figure 6: An IP Spoof simulation from 0.0.0.3 emulating 0.0.0.1.

Figure 6 shows the results of creating a spoof and running the network. Here, 0.0.0.3 is broadcasting its IP address and OGM messages as 0.0.0.1. Even though no real connection exists directly between 0.0.0.4 and 0.0.0.1, 0.0.0.3 has tricked 0.0.0.4 into believing 0.0.0.1 is a next-hop neighbor. Spoofs can be created for any node on the network. Attacker IP must be a real node to populate the data, but then, any value can be broadcast as the Victim IP, even nodes not on the network.

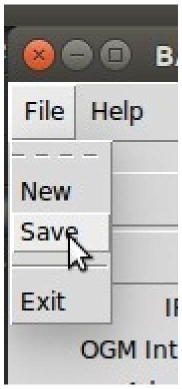


Figure 7: The Save option from the File menu.

Because printing the entire contents of the user nodes, ogms, and messages would be cumbersome on the Console Log, the user can opt to Save the contents of the network to a file. The file will contain a detailed report of each node in the system, all queued OGMs and messages, and all discovered OGMs from each node's perspective. The file is labeled “fullReport_xxxxxxxxxxxx.txt” where the x's denote the day, month, year, hour and minute of the generated report. Similar dating schemes exist for the topographical graphs of the system and console log reports.

Should the user wish to create a new network, the New option will clear the network. The user can then opt into creating more nodes or Exit the system.

Code Explanation

The BATMANSim is written in Python 2.7, so a system supporting this version should be compatible. In addition, several libraries are required for the GUI: Python Tkinter, Python Networkx, and Matplotlib for Python. The system was constructed on Ubuntu 14.04, similar to those found in the Engineering Computing lab at the University of Nevada. While other Linux distributions hold similar configurations, this is the only operating system the program has been tested on. Administration privileges may be necessary to install the additional libraries.

After downloading the code repository, extract the files and run the program in the directory with: python applicationUI.py

The program architecture follows a model, view, controller structure. OGMs are created with the ogm.py class, stored in the User class (user.py), and directed by the Controller (controller.py). The Controller serves as the network backbone, providing network functions such as transport and coordination. It is responsible for passing the OGMs between the User instances through the tick(deltaTime) function, called at intervals given by the system client.

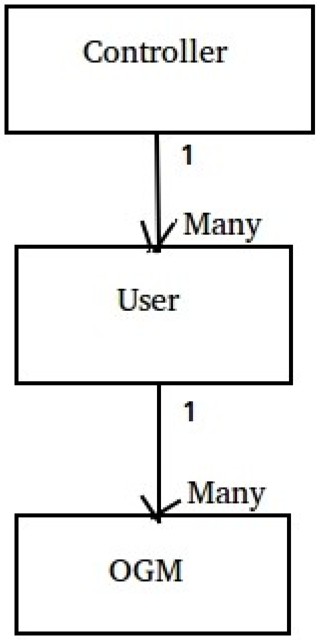


Figure 8: A block diagram of the code structure indicating the relationships of the modules.

Because the controller serves as the head of the structure, all commands should be given through functions in the controller, but creation of nodes may be handled elsewhere as long as the controller's addUser(userNode) function is called to ensure unique entries into the system. Due to Python's public nature in class members, all attributes of each system are considered exposed, so utilizing some functions are necessary to ensure safe and correct usage.

Within the system, the controller establishes a one-to-many relationship with the user nodes. Multiple users are stored in the controller with their transportation and time services called based on function calls. The controller also stores “lost” OGMs in the system for reporting purposes; however, only Users should be responsible for the creation and maintenance of OGMs between Users.

Users are responsible for the actual creation and routing of OGMs. Users are representative of end systems or routers. They maintain sending and receiving queues, create packets, and maintain the network state by holding OGMs passed in from the receiving queue. The User.tick(deltaTime) function maintains the TTL values of the known OGMs in order to reduce overhead of storing the system. In reality, BATMAN OGMs are simply decremented at each hop in order to reduce the TTL, but as our system relies on discrete time steps, this mechanism serves to put a "lifetime" on a smaller network for the simulation and maintain simplicity.

OGMs also serve as data messages for the structure of the program to eliminate the need to have multiple classes with varying overheads for the experiment. OGMs have the classic BATMAN structure: sender IP, originator IP, sequence number, and TTL. However, additional class members have been added for the purpose of reporting and tracking the passage of the OGMs. NextHop helps the controller with forwarding the packets, and traceroute maintains a list of IP addresses through the paths the OGMs took for the experiment. As OGM also serves as a packet for messaging, payload and destination IP have been added for this end. A final parameter, directional, was added to offer extensibility testing between BATMAN 0.1 and BATMAN 0.2. BATMAN 0.1 was oblivious to uni-directional links that may report as being neighboring nodes even though nothing can be sent through them. Version 0.2 mitigated this routing-loop danger by only allowing two-way links to direct traffic. This simulator runs on the assumption that links are bi-directional, but may be further implemented with 0.1 logic for educational purposes.

ApplicationUI is the GUI interface for the program. It defers to the Controller for the system's creation and maintenance and exists as a wrapper for the Controller. Simple queries are performed to ensure safe input of data and reporting functions.

Results

Results will vary based on individual configuration of the system, but the following test scenarios will illustrate the behavior of the BATMAN protocol in a general sense. These test cases are stored in the testdata folder included in the submission and are labeled ending in "_testX" referring to their testing scenarios described below.

Test set 1 is the simplest configuration of the system. Two nodes, 0.0.0.1 and 0.0.0.2 (referred to as A and B, following), are initialized in the system. B has been configured with A as its nearest neighbor, but A has no knowledge of B. The system is run for 3 time intervals. During that time, an OGM is sent to A informing it of B, and A updates its nearest neighbors information and populates the OGM in its network topology. The system is run for 3 more intervals, giving A a chance to broadcast its response back to B, and B updates its table, establishing that the link to A is bi-directional. The system successfully reveals the full topology on both ends.

Test set 2 extends the route-discovery to three nodes connected in sequence: A, B, and C. This time, all are aware of their nearest neighbors but must discover the full topology of the system. Here, the simulator was run for 5 intervals. During that time, B managed to confirm paths from B to A and B to C, but no OGMs have run the full span. The simulator is run for 5 more intervals, and the full topologies are discovered.

Test set 3 uses the same 3-node structure of test 2. Time is given to establish the topologies, and at the third run of 5 intervals, a message is constructed from A to C. The message passes from A to B, following the next-hop path located in the OGM last received from C. From B, the message is indexed again and forwarded to C where it is successfully delivered. Inspecting the fullReport file for test 3 reveals that C did indeed receive this message by traversing A → B → C. (An unusual error occurs during message acceptance wherein the report mistakenly advertises a different IP than the recipient node as the title for the received messages queue, but this is strictly an error in the reporting string as the message does report the correct path and values associated. It is unknown at this time why this occurs.)

Test set 4 features the IP spoofing effects. When an attacking node enters the system, it broadcasts an erroneous IP address in its OGMs, tricking other nodes into thinking neighbors are nearby when they are malicious nodes. The network in this scenario consists of a loop of four nodes knowing only their adjoining neighbors. Node A becomes an attacker, disrupting messages by spoofing B's address to D, making D think that B is a neighbor. The attack is successfully disruptive, and D starts to reference B as its neighbor. The network topology changes from the loop into the incorrect configuration in "_test4_good.png" and "_test4_bad.png" respectively.

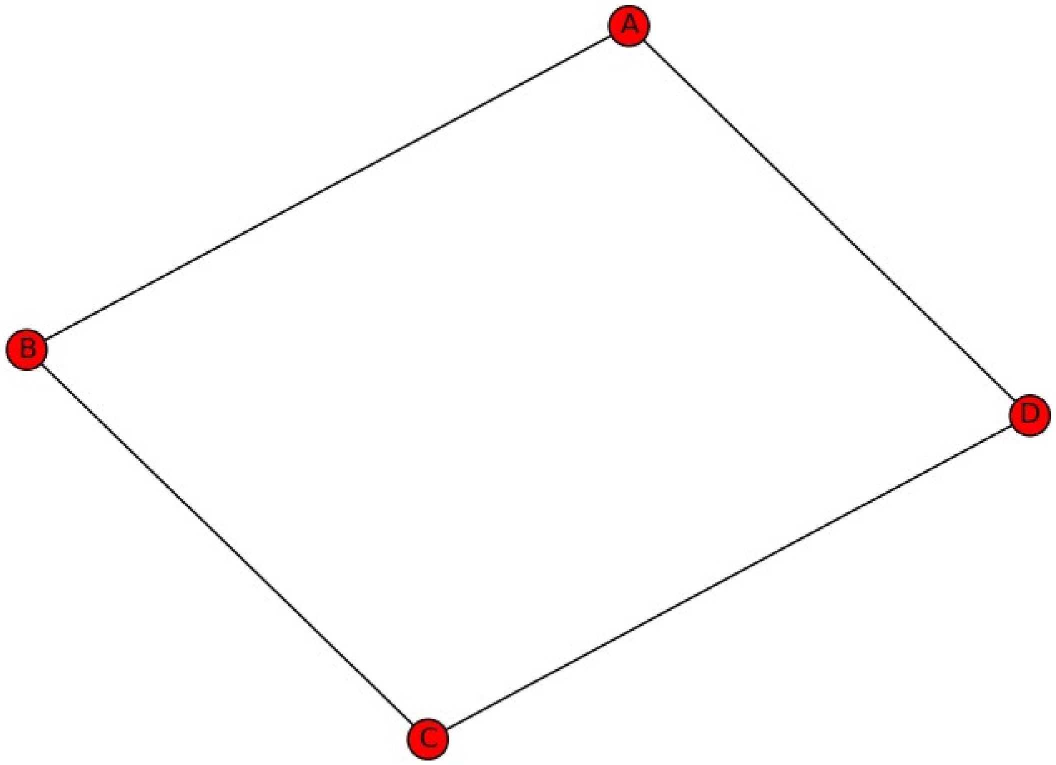


Figure 9: The original configuration of Test 4. Note that D has no direct hop to B and must go through neighbors A or C.

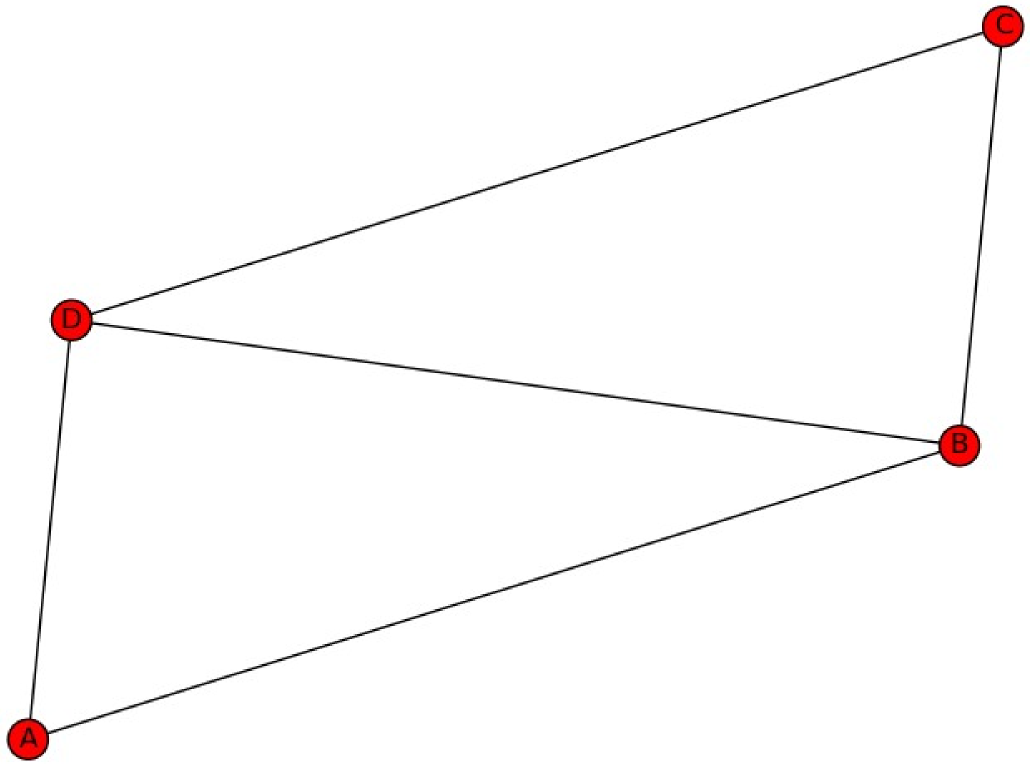


Figure 10: Node A has IP-spoofed B's address, leading D to believe it is connected to B as a next-hop link in Test 4.

Test set 5 illustrates the time delay that occurs when a node is dropped from the system. Here, three nodes are configured in a line as in tests 2 and 3. After establishing the topologies, B drops out of the system, and A and C cannot forward messages to each other. Time intervals up to 100 occur before the nodes forget about B altogether, and the Lost OGMs report fills. Additionally, the graph reports only two, separated nodes. This is reflective of real-world BATMAN scenarios where dropped nodes take long intervals to propagate through the system, and critical nodes, such as B, take longer.

In addition to these scenarios, custom options can be input by the user through the interface. Refer to the Contribution section for instructions and interface screen captures. A typical narrative involves inputing the node's IP address and OGM Broadcast Time, clicking Add Node, populating the neighbor IP based on already added nodes, and clicking Add Neighbor. This system has been tested on up to 12 nodes using this narrative, but system memory limitations should be considered for large network simulations.

Conclusion and Future Direction

Unlike link-state-based routing protocols, BATMAN provides a routing through discrete steps by OGM passing. This saves much in the overhead of storing and configuring network topologies based on detected information from every node in the system. BATMAN's OGMs also provide self-healing networking when neighbors are shared between nodes, offering alternative routes when a node exits the network.

With rapid route-discovery and self-healing, simplistic protocol in place, BATMAN is poised to take over mobile ad hoc networks as more mechanisms are created to ensure its safe operation. Already, BATMAN has been implemented as a standard protocol in the Linux kernel, enabling devices as small as the Raspberry Pi to participate in BATMAN protocols, and whole towns are starting to benefit from the protocol. Increasingly wireless networks and cellular services could benefit from the additional coverage BATMAN may provide to locations where classic telecommunication may not be feasible.

In particular, rescue workers such as firefighters and forest rangers may benefit from ad hoc networks implemented on Android U-Beam (Near Field Communication or NFC) to convey important information for assessing rescue efforts where traditional telecommunication is impossible. Unlike cellular networks or satellite communications, BATMAN could operate within materials that would usually impede cell services. By relaying information in the hop-to-hop manner, a chain of smartphone carrying rescue workers could serve as nodes in the system, passing information down the line like a baton pass in a relay race. With Google expanding the capabilities of Android U-Beam libraries, photos, messages, and even short videos may be in the very near future for headless connection.