Wireless and Mobile Networks

Project 1

Group 1 Report

Seth Denney, Albert Wallace

Introduction

The goal of Project 1 is to design and implement a Vehicular Ad-Hoc Network for large trucks on the highway. This VANET should allow the trucks to communicate with each other through UDP packets and, through use of the data received through these packets, form platoons. These platoons will consist of a leader truck and zero or more follower trucks in the rightmost lane of a highway, with each following truck maintaining a trailing distance behind its immediate leader within a specified allowable range.

The design approach that we took is Object-Oriented, using Python. Two classes were implemented, one pair of which is used for each truck (represented by a different computer/port pair).

Class Design

The first class is TruckNode, which represents the truck itself. TruckNode stores all location and kinematic information about itself and any known neighbors, and also holds all control logic for avoiding collisions, joining platoons, and updating its own data. The second class is VanetController, which provides all network-level services required by the application to TruckNode. VanetController is used by TruckNode, and has external methods for sending a TruckNode's data out to a known neighbor, and it also has internal mechanisms to control packet forwarding, probabilistic packet dropping, and neighbor-packet sequence control.

Control Structure

The control structure of the program begins with an initialization script, init.py, that builds a specified TruckNode from the configuration file and then passes control to the TruckNode. The TruckNode class itself is multithreaded, such that two distinct threads are running simultaneously, a Control Thread and a Listener Thread. The Control Thread runs in real time, updating its own information, broadcasting to its neighbors, and writing its state out to a log file at an interval of 10 milliseconds. The Listener Thread utilizes the data broadcasting functionalities of VanetController. The Listener Thread runs asynchronously with the Control Thread, constantly listening for broadcasts on its own port and updating its cached knowledge about its neighbors as it receives new information. The Listener Thread utilizes the data reception functionalities of VanetController.

Platoon Logic

Each TruckNode follows certain local rules that allow the composite system to function properly. A partial list is given here (certain low-level rules are ignored for clarity), in the order of precedence:

1. If there is a TruckNode in front of you, do not get too close.
2. If you are not in a platoon, try to join the closest platoon that is not behind you.
3. If you are not in a platoon, and no joinable platoons are available, form your own platoon that others may then join.
4. If you are trying to switch lanes to join a platoon, do not switch until there are no TruckNodes obstructing your range.
5. If you are a platoon leader, join any platoon in front of you.
6. Platoons may only exist in the rightmost lane.
7. If you are a platoon leader, accelerate or decelerate to the ideal platoon speed.

With these rules, no collisions will occur, and the maximum amount of TruckNodes that can form platoons will form the minimum number of platoons.

VANET Implementation

--Broadcasting

Each TruckNode has a unique ID, which is synonymous with its unique port number. Each TruckNode is also run on a separate tux machine, although this is not necessary with our implementation, which makes no assumptions regarding IP addresses other than the requirement that all TruckNodes be running on the same local subnet (for broadcasts). As explained above, each TruckNode broadcasts its information to each of its neighbors at a regular interval, currently 10 ms. The data packets that are sent are composed with the following structure:

1. 16-bit Sequence number, incremented with each broadcast
2. 16-bit Source address (ID = port number)
3. 16-bit Previous hop (self ID = self port number)
4. 16-bit platoon ID (ID of platoon leader, or self ID if not in platoon)
5. 16-bits each for IDs of immediately leading/trailing vehicles (or 0 if none)
6. 32-bits each for Xx, Xy, Vx, Vy, Ax, Ay

Totaling to 2 + 2 + 2 + 2 + 2*2 + 6*4 = 36 bytes per packet

--Receiving

On the receiving side, whenever a packet is received from another TruckNode (or rather, VanetController used by a TruckNode), the VanetController first checks to see if the packet is well-formed. If so, it checks its packet reception table, which maps source IDs to the latest received sequence numbers from those IDs. For example:

The VanetController receives a packet that originated from TruckNode 11111 with sequence number 98. It goes to its packet reception table and checks the entry for 11111 (assume it already exists), and sees that the last received packet originating from 11111 had a sequence number of 99. Thus, the packet is discarded, as it is an old packet, probably forwarded from some far-off neighbor.

If the VanetController finds that the packet is newer than its previous packet from the source TruckNode (from the above lookup table), it proceeds to simulate a probabilistic drop rate. This drop

rate is proportional to the square of the distance between the sending (NOTE: previous hop, not source) TruckNode and the receiving TruckNode. The proportionality constant used in the drop pdf is such that the packet will drop with a probability of 20% when the two TruckNodes are 100 meters apart. If the packet is valid, new, and not dropped, it is returned back to the calling TruckNode (specifically its Listener Thread) for cache updating.

--Forwarding

Each packet that is successfully received and passed back to the calling TruckNode is added to a queue of packets that have yet to be forwarded. Each time that the TruckNode broadcasts to its neighbors, each packet in the VanetController's forwarding queue is broadcasted as well, and the queue is cleared.

--Periodic Reading/Writing of Config File

In the project specification, it is stated that each TruckNode must periodically update the config file with its current position, and must also read the config file periodically. The motivation for this requirement stems from a specific use case: If there exist two connected (read: within range and communicating) sets of at least one TruckNode, where no TruckNode in the first set is within communication range of any TruckNode in the second set, then neither set is aware of the existence of the other set. This would not be a problem if the two sets could never meet, but this is not guaranteed. In the case where the two non-communicant sets do meet, they would still never become aware of each other due to the neighbors-only broadcast rules, and crashes would be imminent. By checking the periodically-updated config file, a node can acquire new neighbors that even its current neighbors do not know about, and the problematic scenario mentioned above will be prevented.

Unfortunately, this requires much concurrent file IO, which is risky, slow, and not true to the Ad-Hoc Network nature of the system, which in reality would not have the luxury of a shared Network File Server with common config files, so a method for acquiring new neighbors that is more in line with the goals of this application is desired.

The solution to this problem lies within periodic full-spectrum broadcasts. Every 100 ms, instead of reading and writing to the shared config file, each TruckNode will broadcast its information to the entire spectrum of available ports, and not only those of its current neighbors. These packets will be received by any node that is within range of the sending node, regardless of pre-existing neighbor connections. Note that, due to the quadratic packet drop rate, far away nodes will most likely not hear the blanket broadcasts, which remains true to the desired realities of the simulation and achieves the same ends as the initial config file method.

Time Synchronization

The issue of time synchronization across all TruckNodes involves two primary problems: differing CPU execution speed and differing simulation start times. The issue of differing CPU speeds can be resolved by running in real time, rather than using simulated 10ms timesteps. While different CPUs might process virtual time faster or slower than others, all CPUs process real time at the same rate. The issue of differing start times stems from the fact that each TruckNode that is initialized on a different machine must be initialized manually, and consequently each TruckNode will be initialized at a different time. This can be solved by having each TruckNode await a global start signal after initialization. In the current implementation, each TruckNode uses its VanetController to listen for an initial UDP broadcast on its port, and when it is received, the TruckNode begins its simulation. Using a standalone script that broadcasts over all allocated ports, the user can signal all initialized TruckNodes to start simultaneously.

Additionally, the VANET utilizes a TDMA scheme to avoid receiver collisions (which leads to dropped packets). Each transmission period (10ms) is divided evenly among the possible ports (10), and each slot is assigned with the formula ID % 10.

Run Instructions

To run the simulation, open a console window for each node, preferably, but not necessarily on separate computers. Navigate to the directory containing init.py, TruckNode.py, and VanetController.py. Initialize the node by typing the following:

>> python init.py (truck ID) [initial velocity [configuration file]]

Remember that the truck ID is the port number for the current node to use. If no velocity is specified, it will be generated with the preprogrammed distribution. If no config file is specified, it will attempt to use ‘config.txt’. A manual config file cannot be specified if no starting velocity is specified.

Each line of the configuration file must follow the following format:

(nodeID) (initX) (initY) links [linkID]*

Where nodeID is the node’s ID/port, initX and initY are the starting X and Y coordinates, and linkID is the ID/port of a node that is linked to by the node. There can be multiple links, separated by a space. A sample configuration file is provided, named config.txt.

Once all nodes have been initialized, open a final console, navigate to the directory containing start.py, and run the following command:

>> python start.py

At the end of the simulation, there will be an output file for each node, named with the format ‘TruckID.log’. This can be used as outlined in Testing above to view the simulation details.

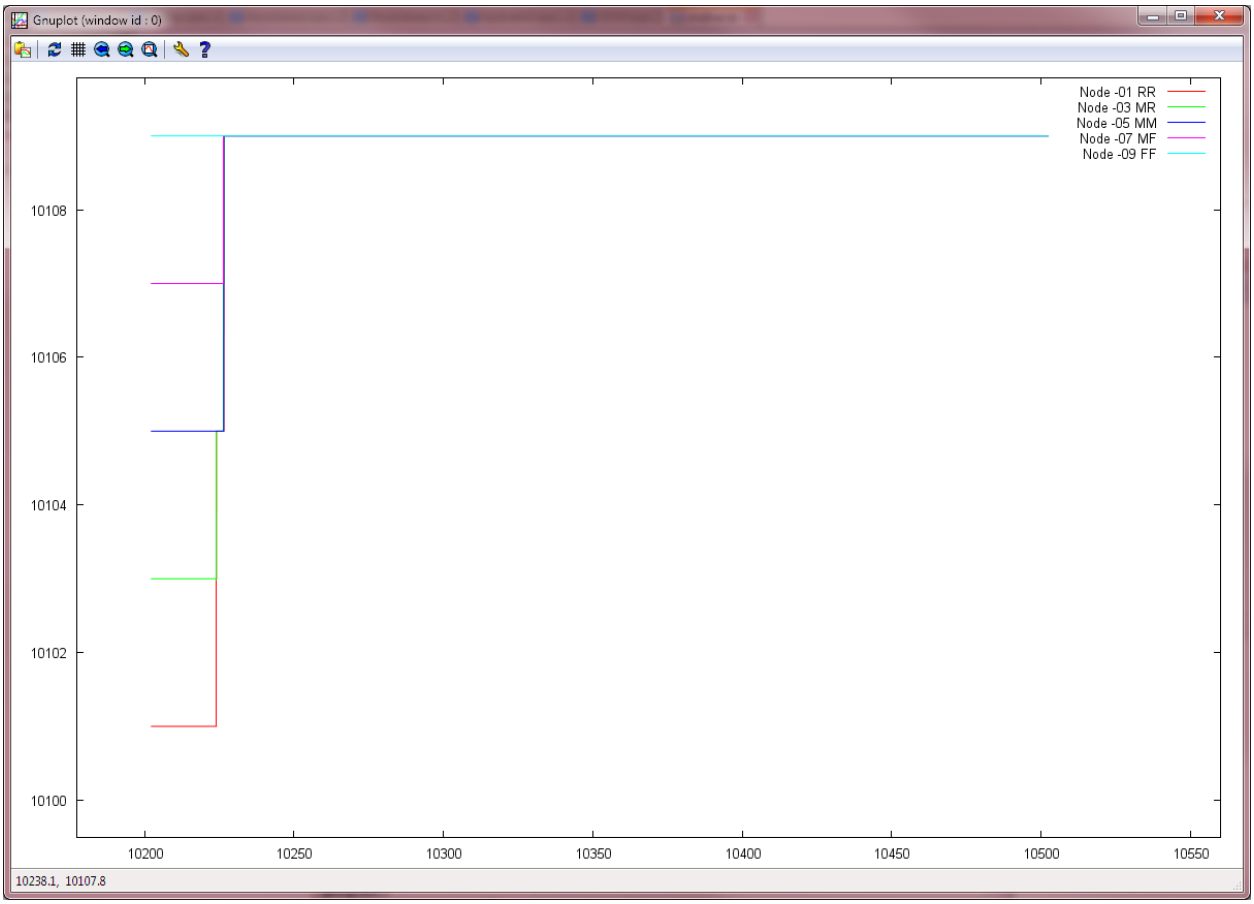
Testing

--Simulation Logging

To test the simulation, each TruckNode writes to a unique log file every time it broadcasts its data (roughly every 10ms). Each line in this log file contains information about time, position, velocity, acceleration, platoon ID, and immediate leader and follower. This information can be plotted using a data-plotting application such as GNUPlot to show the paths taken and many other metrics about the simulation. Additionally, these log files note the decisions made for each timestep, which is extremely helpful when trying to understand how the logic affected the simulation.

--Sample Test Results

From Test 15 [E3], Platoon ID [Y-Axis] versus Time [X-Axis]:

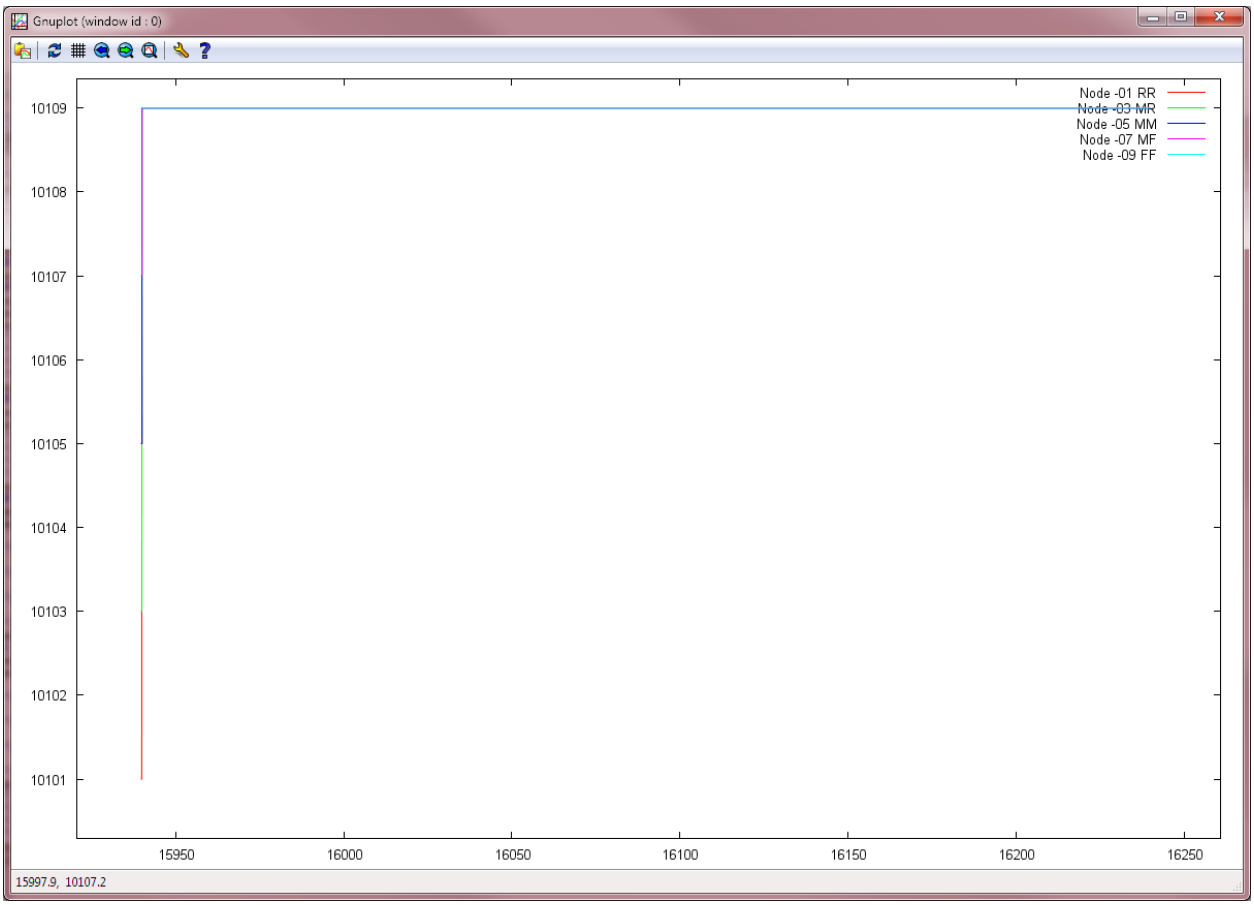


This graph represents a platoon with five (5) trucks, or 5 nodes. To begin, each node assigns its own ID (its port number, in this case) as its platoon ID, effectively making each node its own platoon. Once the nodes can successfully communicate with neighbors and negotiate a platoon plan, they rather quickly form the platoon and the ID of the platoon becomes that of the leader. So early on, Truck 10101 joined up with 10103, but in virtually no time at all, they joined Truck 10105 to form Platoon 10105. Eventually, all trucks settled out and found themselves in Platoon 10109. This scenario occurs during a 5 minute simulation, with the following start velocities, X positions, and lane configurations:

- 10101 (rearmost truck): 26 m/s, 0 m, right
- 10103: 25 m/s, 98 m, right
- 10105: 24 m/s, 196 m, right
- 10107: 23 m/s, 294 m, right
- 10109 (frontmost truck): 22 m/s, 392 m, right

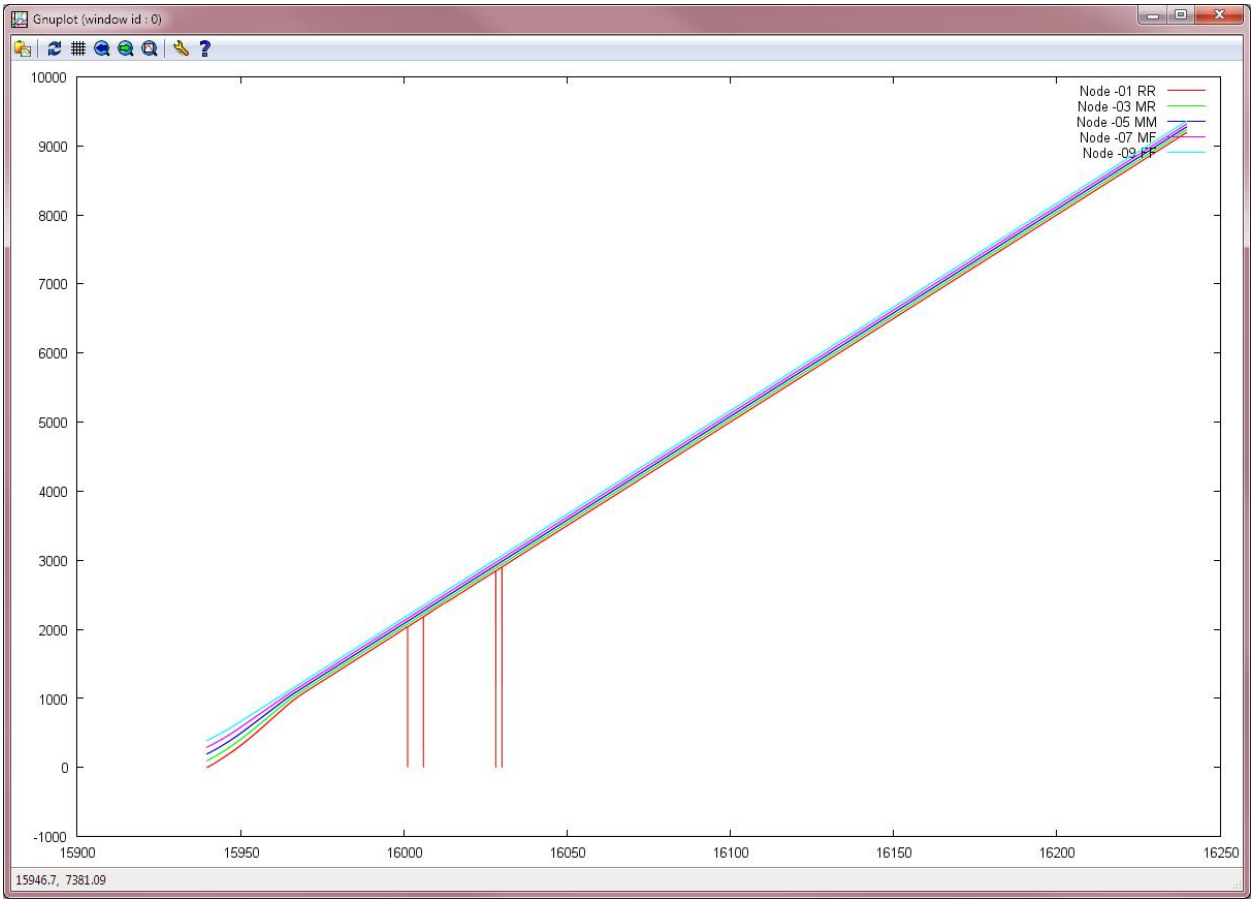
Comparatively, if the configuration has all trucks in the left lane, a more immediate switch occurs:

From Test 19 [G2-E1], Platoon ID [Y-Axis] versus Time [X-Axis]:



This difference in efficiency is by design; while in the left lane, the trucks are allow to be more aggressive/active at forming a platoon. With them all in the right lane, they are assumed to be cruising in a platoon, with a bias to remaining in that (potentially) smaller platoon rather than trying to hurry to form a (potentially) larger platoon. This particular graph came about as a revisit to the specified conditions; in the initial test of this configuration, due to their positioning in the left lane, when the trucks in the rear formed a platoon and hit a cruising speed, the trucks in the front continued a steady acceleration from their relatively slow starting speed until hitting maximum speed. This put the forward trucks out of communication with one another, and prevented communication with the multi-truck platoon that had formed. The revision prevented the loss of range while solid communication was occurring; the front truck was asked to keep its slow speed, while the trailing trucks were allowed to speed up, adjust their position and signal strength, and form the platoon. This can be seen in the following graph:

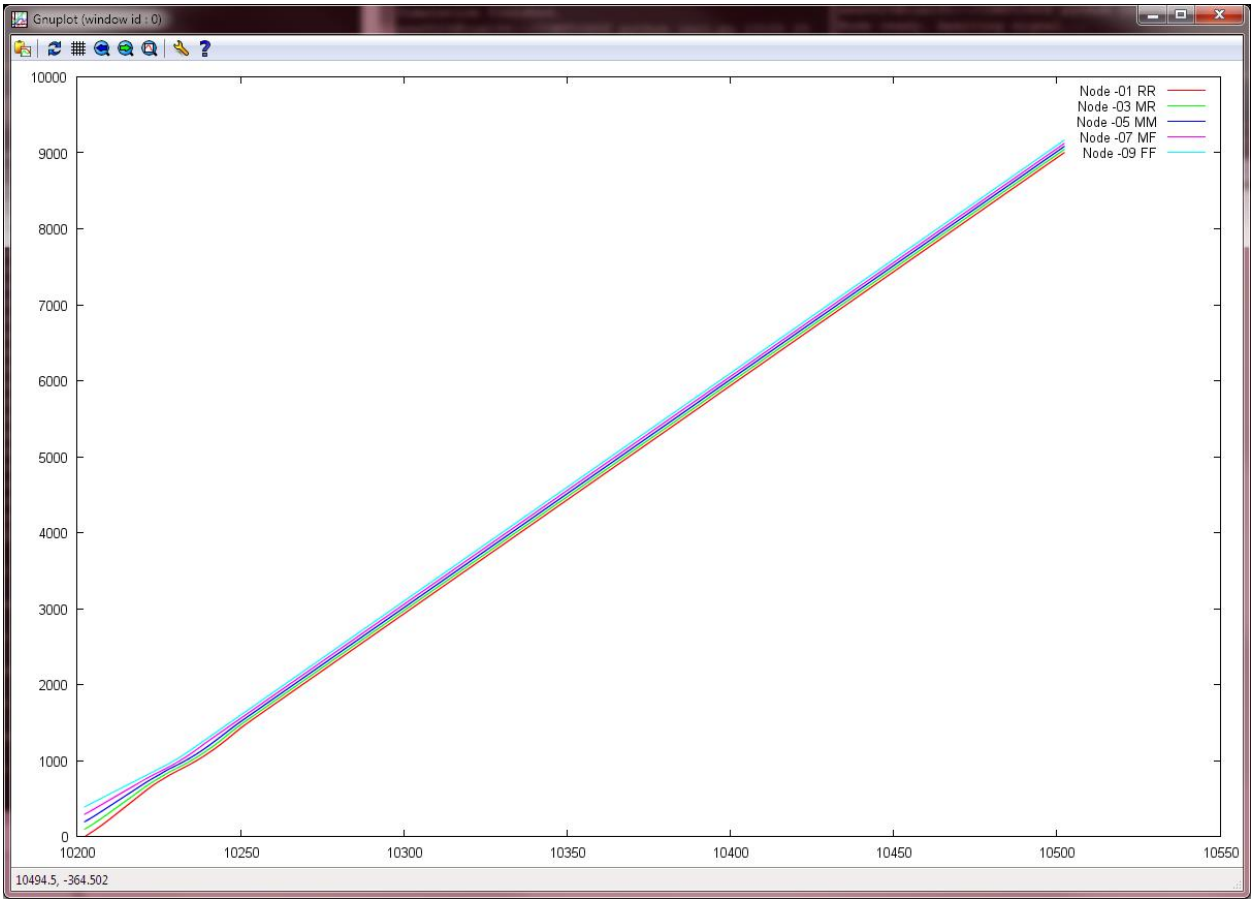
From Test 19 [G2-E1], Longitudinal position on highway [Y-Axis] versus Time [X-Axis]:



There is an initial curve most readily apparent in the rearmost node 10101, whereas the frontmost node 10109 appears to never need to accelerate. The downward dips in the graph do not indicate a flaw; rather, it is error correction. The truck was attempting to check/update its cache, and had to do a refresh of its internal data. This resulted in a glitch being written out to the log, where old information is discarded internally but still exists in the log (information is written out immediately, in a serial fashion, during processing, in case any severe issues occur), and one line of log data was appended to the end of another (incomplete) line of data. Communication with the platoon was not affected, the truck remained in the platoon, and such glitches occur so infrequently (4 entries out of nearly 30,000 lines of the log) that reliability is not compromised. The rest of the log writing

This occurs in random spots, and may not occur at all, as shown in the right-lane equivalent of the test:

From Test 15 [E3], Longitudinal position on highway [Y-Axis] versus Time [X-Axis]:



No issues to be found for this run; all trucks had a relatively easy time communicating with one another, they did not have any data transfer conflicts, and they remained in their platoon once formed.

--Network Metrics

Separate from this testing was testing to gather information on the throughput. As it stands, throughput is reported in terms of packets. We chose to run separate tests for 3, 4, and 5-node systems, with randomly assigned values for speed, and random values for longitudinal positioning, to get a real-world range. With the knowledge that each packet is 36 bytes, we can determine throughput for a variety of runs, as reported by each individual node:

Test 1 (3-unit platoon) (over 90 seconds)	Unit #	Drop Rate	Throughput (packets/sec)	Average throughput: 491.83 pkts/s Average end-end delay: 0.017 sec/pkt
	10101	3.3%	465.4	
	10103	1.4%	497.03	
	10105	9.1%	513.07	

Test 2 (4-unit platoon) (over 90 seconds)	Unit #	Drop Rate	Throughput (packets/sec)	Average throughput: 184.0375 pkts/s Average end-end delay: 0.025 sec/pkt
	10101	41.9%	85.23	
	10103	5.39%	97.0	
	10105	2.20%	294.2	
	10107	2.47%	259.72	

Test 3 (5-unit platoon) (over 90 seconds)	Unit #	Drop Rate	Throughput (packets/sec)	Average throughput: 457.215 pkts/s Average end-end delay: 0.032 sec/pkt
	10101	1.67%	425.37	
	10103	11.48%	363.16	
	10105	4.69%	601.45	
	10107	3.46%	546.84	
	10109	3.61%	349.24	

With this limited collection of data, we have an average end-to-end latency time of 0.025 seconds per packet, with a max (during heavy congestion & fairly large amounts of dropped packets on at least one node) of 0.032 seconds per packet. Our average throughput, on the other hand, ranged from 80 packets/sec at a slow/distant node, to 601 packets/sec at a well-linked/fast node. It is likely that we did not hit the ceiling of maximum performance, but given that we ideally would need only a few packets per second for our system to happily work, our system seems to scale fairly well even at the most strained of times.