Wireless and Mobile Networks

Project 2

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 of packets**

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
7. 16-bit placeholders for up to 10 nodes to indicate that they have seen the packet

Totaling to 2 + 2 + 2 + 2 + 2\*2 + 6\*4 + 10\*2 = 56 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/Sending**

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.

*Project 2 builds on this simple premise with an MPR algorithm:*

Rather than have a set number of MPR nodes, we decided to do sort of an objective, dynamic MPR setup. What this means: none of the nodes considers itself strictly as an MPR or a normal node; rather, each node sort of acts as an MPR for its neighbors depending on whether it is the best node for sending packets to a given neighbor. If it determines that it is the best node, then it will forward packets. Given that our algorithm does not require packets to be specifically addressed to any other nodes, we just automatically forward packets accordingly, and each node decides if it cares about the packet (if it is a new packet, or if it a packet that the node has already seen).

All packets are used to update information about speed, velocity, and neighbors, the latter of which is new for this project. As such, packets now feature an area containing hybrid information: of neighbors of the source are stored here, and nodes that are likely to have been visited by the packet are also stored here. There are ten slots in this area of the packet, allowing hard-coded support for up to 10 nodes during simulation. Each possible node ID has its own slot (wherein the first slot is for ID 10100, and the last slot is for 10109) to allow for a direct search of whether a packet has visited a node.

To keep this information up-to-date, the source node enters its known neighbors in the packet as it adds its position and velocity information, and then passes the packet to at least one of its neighbors. When the packet reaches another node and is forwarded, that secondary node also enters its neighbors as a way of indicating exactly who has seen the packet, without clearing any of the previously stored neighbors. By the end of the chain, the packet should have just about every node being used in the simulation listed.

With this new information, nodes can now take an educated guess as to who the neighbors of other nodes are. This enables nodes to guess if they are the best chance to send the packet to a given node; if there is a stronger link assumed to be available, then the node will skip sending/forwarding the packet directly, instead relying on a known neighbor to pass the packet on.

In a roundabout way, this ensures that nodes alternate exactly who acts as an MPR. Take a given scenario, with a platoon of all ten trucks, already as in-range as possible. With this setup, we can assume that each truck can reach two trucks ahead and two trucks behind, at the very least. Assume that each truck already knows its neighbors (determined by a robust function present in project 1). Now:

* The frontmost truck wants to send a packet. It sends to all of its neighbors, so it updates its neighbors, the second and third truck.
* The second and third trucks successfully receive the packet. The second truck has 1, 3, and 4 as neighbors; truck 3 has 1, 2, 4, and 5. This means that truck two will try to send to 3, looks at the previously visited list, and determines 3 has probably seen it, but agrees to send because it is closer than node 1, so it is a more robust connection. However, it skips sending to 4, as it knows 3 is closer.
* This kind of pattern continues until the end, with some nodes only receiving the packet once, while others receive it multiple times.

With this setup, we make sure that packets are eventually sent. If a node receives a duplicate packet, it simply drops the packet and incurs only a tiny performance penalty; in that way, nodes has a decent amount of free time comparable to a more hard-coded MPR setup, without having to tell each node that it is indeed an MPR. Plus, during testing, we saw a range of throughput, from 700 packets/second to 1700 packets/second and greater (depending on the number of trucks), indicating that each node technically has the capability to handle such a dynamic setup without failing. A double benefit is that, should a node fail in the middle, the platoon algorithm (carried over from project 1) can continue on uninterrupted, without the need to reassign MPR nodes.

With only minor tweaks, we could choose to reduce packet redundancy by passing not only which nodes are likely to have been visited by a packet, but the chance that the packet has reached a node. This would allow the nodes to more intelligently perform their routing operations, but through fears that control packets would occasionally be lost (due to a non-zero drop rate), we prefer to take the safe route that is a bit less efficient in larger platoons but provides various improvements overall and retains acceptable levels of safety.

**--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 and Results**

**--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.

**--Some basic scenarios tested:**

1. *Scenario: Trucks start off in range, small number of trucks (~5 trucks):*

It takes a smaller number of trucks, potentially closer together, to difference of a dynamic MPR setup in terms of packets sent. As expected, with trucks 1 through 5 (where 1 is front and 5 is rear), the middle-most truck 3 had a noticeable increase in the number of packets it had to handle: with the MPR setup, it has to handle between 50 to 200 packets more (with between 1500 and 1600 packets total). Similarly, there is a trend that trucks 2 and 4, right outside the centermost truck, saw a decrease in packets with the MPR setup: those nodes saw 200 to 700 fewer packets in general, within a range of 1000 to 1700 packets total. Oddly enough, the performance of the nodes at the extreme ends – nodes 1 and 5 – remained fairly similar, with no remarkable difference. This is acceptable, as the overall number of packets handled must be the same, and the ends understandably had to deal with the lowest amount of routing in the first place.

1. *Scenario: Trucks start off in range, large number of trucks (~10 trucks):*

When we move to a full 10-truck platoon, results become more muddied. At high distances between trucks, the drop rate (percent of packets dropped due to excessive distance) decreased between 1% and 5%, which is an improvement (and reflects nodes being more proactive about transmitting data only if they are sure they are the best choice for connecting to neighbor nodes). At closer distances, the MPR-enabled platoon saw a decrease in the required throughput with a similarly low drop rate.

1. *Trucks start off out of range:*

Unfortunately, it was nearly impossible to get repeatable, predictable results when all trucks started out of range. It required more or less all but one truck to be in range, and then they have to join together fairly quickly. Other configurations were left untested for this round, due to success with the 5 truck and 10 truck tests above.

*Secondary thoughts:*

Control (broadcast) packets: With the tweaked algorithm, there exists a higher potential for broadcast packets to reach nodes farther down the chain of the platoon. In the old flooding algorithm, nodes would spread the control packets to their neighbors, and each neighbor would freely re-distribute the packets to anyone in range, and the packets would eventually make it to the end. Now, the initial broadcast still occurs, but each node will ideally only receive a duplicate copy of that broadcast once or twice, and as with all types of packets, the control packets are sent with greater confidence; nodes try to communicate only with close nodes if possible.

Note that during testing, with no forwarding of control packets across the platoon, a single platoon happily split into multiple separate platoons. This would occur across a range of distances, between 50 and 100 meters. Once re-enabled, trucks could be ~99 meters apart before they began to frequently lose synchronization; even then, thanks to the more directional path, the communication efficiently kept the trucks together.

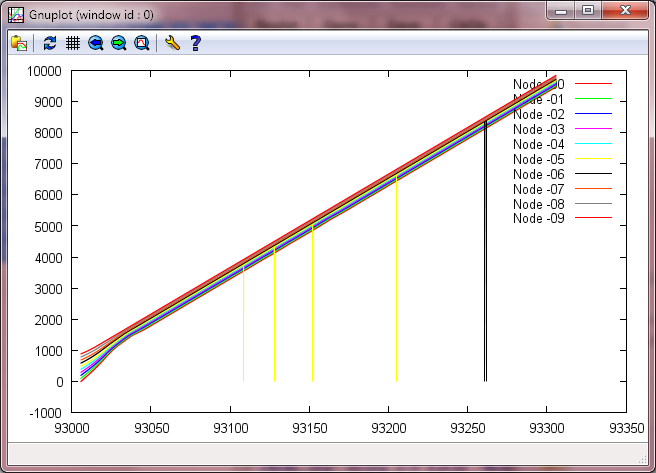
One downside of sorts is that the minimum front-to-back packet transmission time increases in certain scenarios. Communication from truck to truck otherwise remains stable, and this results in a slightly decreased short-range communication time. Plus, the lack of dropped packets means that a packet has a higher chance of reaching the end of the platoon from the front of the platoon.

*What about platooning?*

Thankfully, normal platooning still works as expected, verified in the following graphs:

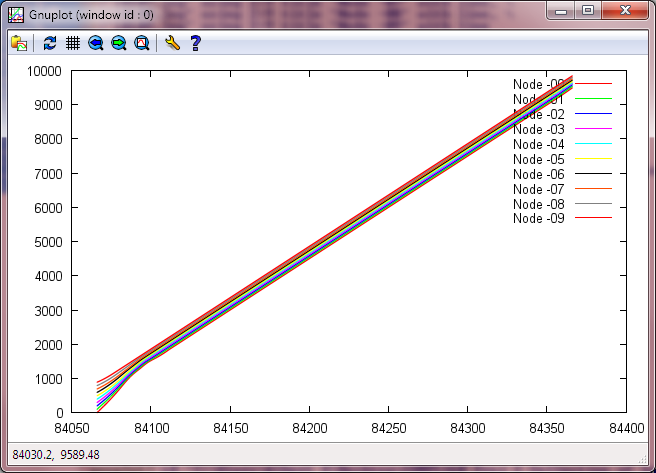
Old algorithm, 10 trucks,

~99 m starting distance between nodes,

Time versus position: 

New algorithm, 10 trucks,

~99 m starting distance between nodes,

Time versus position: 

In both cases, we see that the trucks manage to join and remain in a platoon with no issue.

**Automated Control**

**--Pexpect Module**

These simulations previously required that we open one terminal for each TruckNode that will be run, as well as an additional terminal for the start signal. For simulations involving 10 TruckNodes, it is easy to see that this will not be practical for larger systems. Therefore, we made use of the Pexpect module in Python. This module can automate ssh-based processes through Python, and allows the use of a single entry point from the operator’s point of view. With a script written with the Pexpect module, a user can instantiate any number of TruckNodes, each on a separate tux machine and using a different port within a specified range (such as the range allotted to a given group). This script uses a combination of three primary functions: spawn, expect, and sendline.

1. spawn() – creates a new subprocess with a certain command, in our case starting with an ssh command into tux
2. expect() – tells the child process to expect a response matching a given regular expression
3. sendline() – sends a command to the child process

Through the use of these three methods in the Pexpect module, it is not very difficult to automate the process of spawning 10 processes, each of which initializes a separate TruckNode on a separate machine.