2022

VANET Broadcast Protocol: A Multi-Hop Routing Framework for Vehicular Networks in ns-3

William M. Bjorndahl
Loyola Marymount University, wmbjorndahl@gmail.com

VANET Broadcast Protocol:
A Multi-Hop Routing Framework for Vehicular Networks in ns-3

by

William M. Bjorndahl

A thesis presented to the

Faculty of the Department of
Electrical Engineering
Loyola Marymount University

In partial fulfillment of the
Requirements for the Degree
Master of Science in Electrical Engineering

July 18, 2022

Advisor: Gustavo Vejarano, Ph.D.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

Abstract

Vehicles are more frequently being built with hardware that supports wireless communication capabilities. Dedicated short-range communications (DSRC) is a standard that enables the hardware on vehicles to communicate with one another directly rather than through external infrastructure such as a cellular tower. With DSRC supporting small-range communications, multi-hop routing is utilized when a packet needs to reach a long-range destination. A vehicular ad-hoc network (VANET) broadcast protocol (VBP) was developed. This thesis introduces VBP, an open-source framework for simulating multi-hop routing on mobile and wireless vehicular networks. VBP is built for the routing layer of the network simulation tool called network simulator 3 (ns-3) and contains a custom protocol that adapts to various traffic conditions on a roadway. To test VBP we ran six simulations across three traffic levels. Results confirm that VBP successfully routes packets or queues packets when a first or next hop is not available. The development process of VBP is documented to help researchers who are trying to create a custom routing protocol for ns-3.

I. Introduction

A. Vehicular Ad-hoc Networks (VANETs)

To support the development of intelligent transportation systems in the United States the Federal Communications Commission allocated the 5.9 GHz band for DSRC-based operations[1]. With other countries setting similar standards, opportunities exist to create applications that may benefit driving safety such as:

- Emergency vehicles indicating direction of approach and alerting downstream drivers to clear a lane
- Police polling a specific section of highway for a hijacked vehicle
- Vehicles notifying another vehicle when it is in their blind spot

Without DSRC technology, drivers rely on sounds such as sirens or horns to communicate with other motorists. Figure 1 illustrates an ambulance dispatched to an emergency location and highlights the inefficiencies of using sirens to alert surrounding drivers to clear a lane. A siren may not be heard because the noise is reduced as it enters the cabin of a vehicle. Further distractions such as loud music or a phone conversation reduce the driver's ability to perceive the oncoming emergency vehicle. This causes lane congestion and leads to longer ambulance response times.

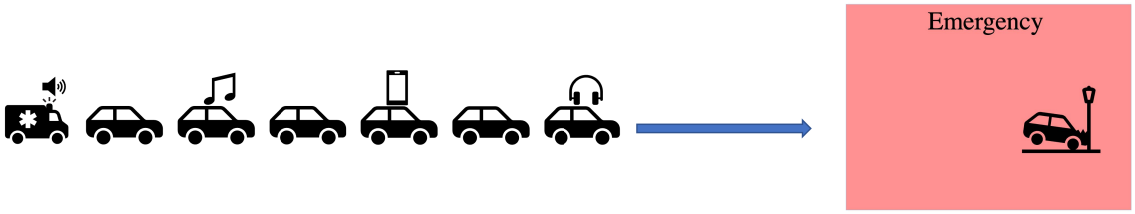

Fig. 1. Without DSRC technology

One way to circumvent the issue described in Figure 1 is to treat each vehicle as a node on a wireless and mobile network. Considering the road environment as a VANET allows us to use routing protocols to send messages between vehicles. In our VANET a custom routing protocol facilitates the transfer of data packets from a source vehicle to a destination vehicle. Other VANETs may support packet exchange between a vehicle and infrastructure along a highway, called vehicle-to-infrastructure or vehicle-to-everything.

B. Contributions

To the best of our knowledge this thesis identifies three unique contributions.

1) Contribution 1: Develop a Multi-Hop Routing Protocol on a VANET for ns-3: There are many network simulators available for researchers. These tools have built-in functionality for generic routing protocols that work on networks such as a group of computers connected via ethernet. Some available simulators are made specifically for VANETs[2] but require a pre-defined route to be set before the simulation starts. Not supported in these simulators are multi-hop routing and routes that update based on traffic condition. VBP, described in this thesis, provides multi-hop capabilities that produce a small end-to-end delay using dynamic routing based on the traffic environment. VBP is created for ns-3 at the routing layer and expands on the work of a lab member[3]. The lab member created routing algorithms for various traffic conditions and VBP implements these algorithms on the routing layer of the TCP/IP model.

A component of VBP is called the targeted broadcast area (TBA). This is the region of highway that we want the broadcast to reach. A vehicle that contains the data packet will broadcast it to all other vehicles within communication range. The two parameters for the TBA are location and expiration time. These are defined in our framework and discussed in later sections.

2) Contribution 2: Provide Simulation Parameters and Setup in ns-3 for a VANET: We simulate VANETs in ns-3 to verify the functionality of our routing protocol, VBP. Our simulation models contain all the necessary components for a researcher to easily test our protocol. The simulation scripts we provide can be adjusted for different mobility models such as the starting position and velocity of each vehicle. The simulation infrastructure we provide can be easily modified and debugged.

3) Contribution 3: Document the development of a routing protocol for ns-3: A brief overview of routing is provided in the documentation of ns-3[4]. While a researcher can explore ns-3's codebase for implemented routing protocols, the process to develop a custom routing protocol is ambiguous. No documentation or guide details this procedure, leading to the third contribution of this thesis: document our creation of a custom routing protocol for ns-3. Also provided is the open-source code of VBP that researchers can use for their projects.

C. Broadcasting in VANETs

VBP uses two types of packets: control and data. Control packets are sent and received by every vehicle within communication range. They provide information about each neighboring vehicle. More details about control packets are provided in Section V. Data packets are routed by algorithms to travel from a source to a destination. More details about these packets are provided in Section VI. Vehicles in our simulations are configured to send packets a maximum distance of 250m. A data packet can 'hop' every 250m until it reaches the TBA. The goal of a data packet is to reach the TBA with a small end-to-end delay.

1) Example: Ambulance: Figure 2 shows how the problems presented in Figure 1 can be alleviated through VANETs. Safety is increased because vehicles have more time to pull over and the ambulance will have less impeding traffic when approaching the emergency.

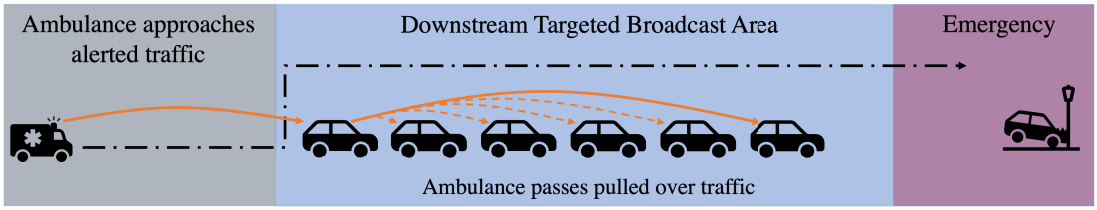

Fig. 2. With DSRC technology

2) Example: Police Probing: Another application that benefits from VANETs is surveying a road for a specific vehicle. Figure 3 illustrates how packets can hop among vehicles on a road until the vehicle of interest is found. This application can relay to the police the location of a specific vehicle, which is useful to detect hijacked cars.

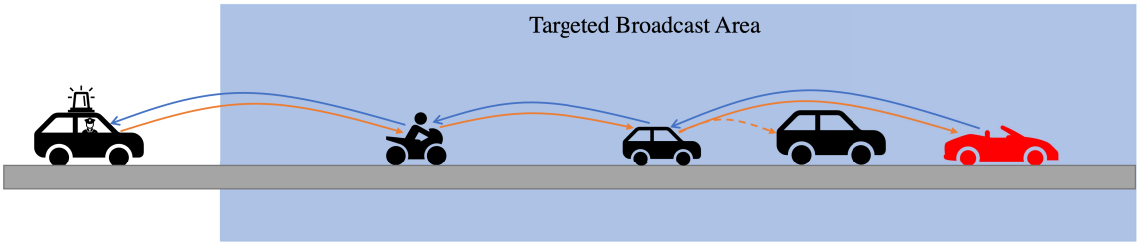

Fig. 3. Police probing

D. Routing Data Packets Towards the Targeted Broadcast Area

No central controller is used to make routing decisions, meaning vehicles only know about other vehicles through packet exchange. Each node is responsible for the forwarding and reception of all data packets. Routing is used to find a path for data packets so they reach the destination.

We utilize the routing layer so each vehicle can perform the following:

- Generate and transmit control packets
- Maintain a neighbor list based on received control packets from other nodes
- Determine the optimal next-hop in various traffic conditions for data packets
- Queue data packets if no hop found
- Identify location of the TBA relative to vehicle position

The following examples demonstrate some behaviors of data packet routing in VBP traffic simulations. This shows how VANET can be used in real-world scenarios.

1) Example: Ambulance Approaching Caravan Downstream: This example shows a fast moving ambulance approaching a constant velocity caravan of cars. The ambulance transmits a data packet once it is in communication range (250m) of a downstream vehicle heading towards the TBA, signifying the 'first-hop'. Routing algorithms implemented in VBP find the next available hop so the packet reaches the TBA in minimal time. Our protocol detects five potential next-hops since any vehicle can receive and forward data packets. VBP chooses the head of the caravan as the 'second-hop' since it is closest to the TBA and moving towards the TBA, illustrated in Figure 4.

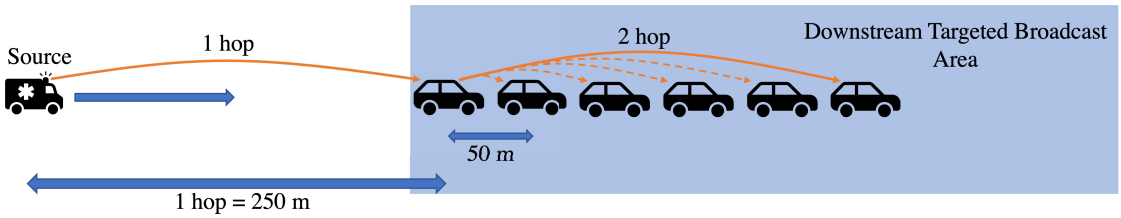

Fig. 4. Downstream example

2) Example: Upstream Traffic Alerts: This example presents an ambulance blocking the road to help a crashed vehicle. Directly behind the ambulance is stopped traffic while traffic further beyond that is still moving at normal highway speeds. The ambulance sends a data packet upstream to alert incoming traffic to prepare to stop.

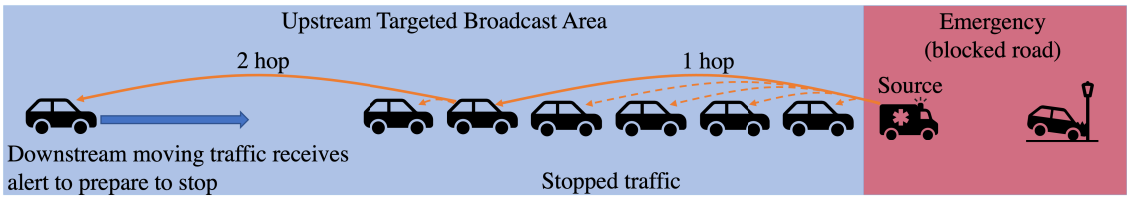

Fig. 5. Upstream example

E. Related Work

OMNeT++ is an open-source network event simulation library and framework similar to ns-3. While OMNeT++ has built-in routing capabilities, its feature set is not comparable to the functionality of the VBP framework built for ns-3 at the routing layer. OMNeT++ is limited to static routing, where a route is pre-configured[5]. In VBP each node monitors their list of neighbors and uses this information to calculate the first hop and intermediate hops of a route.

II. Routing Algorithms

Routing algorithms determine the route for a data packet traveling from a source to a destination vehicle. Routes consist of two types of hops: the first hop and intermediate hops. Each vehicle can receive and forward data packets while the source node has the additional capability to generate and send these packets. A data packet will be queued on a vehicle until a next hop is available.

For the first and next hops, we use the following criteria to select the algorithm of the hop so the data packet reaches the destination with minimal end-to-end delay:

- Center of TBA location relative to a vehicle position
- Vehicle direction
- Traffic level

The data packet contains the TBA location and expiration time and this information is accessible by the node upon reception of a packet. To determine if an vehicle is moving towards or away from the TBA, VBP checks the velocity of the vehicle, V. Also used by our algorithms is the vehicle's distance to the center of the TBA, X_{TBA}

Every node uses and maintains information about itself and its neighbors to sense its traffic environment. A node knows its neighbor information, such as the distance of a node within one hop, X_N and the IP address of vehicles ahead and behind. VBP checks if a packet comes from behind (traveling downstream) or from ahead (traveling upstream). We also formulate if a vehicle is expected to reach the center of the TBA before the zone expires through the message delivery time, T_{MD}. These calculations involve the current vehicle position, the center position of the TBA and the neighborhood speed. Table I presents the first and intermediate hop algorithms.

TABLE I
First and next hop algorithms

Traffic Levels			
Upstream Towards TBA	Upstream Away TBA	Downstream Towards TBA	Algorithm of First/Next Hop
High			$\max X_{TBA}$
Medium			$\max \sqrt{T_{MD}^2 + X_N^2}$
Low			$\max T_{MD}$
	High		$\min X_{TBA}$
	Medium		$\min \sqrt{V^2 + X_{TBA}^2}$
	Low		$\min V$
		High	$\min X_{TBA}$
		Medium	$\max \sqrt{V^2 + X_N^2}$
		Low	$\min T_{MD}$

A. First Hop Determination

The first hop occurs after the source vehicle generates a packet. We describe the process to find the first hop in Figure 6. As an example, consider the case in Figure 2 where an emergency vehicle alerts downstream traffic to pull over to the side of the road. The emergency vehicle generates a data packet that travels downstream to vehicles a distance of 250m. The first hop will have to be accomplished so the alert can broadcast to surrounding vehicles and hop to the next forwarding node. The algorithms that determine the first hop are narrowed down to the following:

$$\begin{aligned} \text{High Traffic} &= \min X_{TBA} \\ \text{Medium Traffic} &= \max \sqrt{V^2 + X_N^2} \\ \text{Low Traffic} &= \min T_{MD} \end{aligned} \tag{1}$$

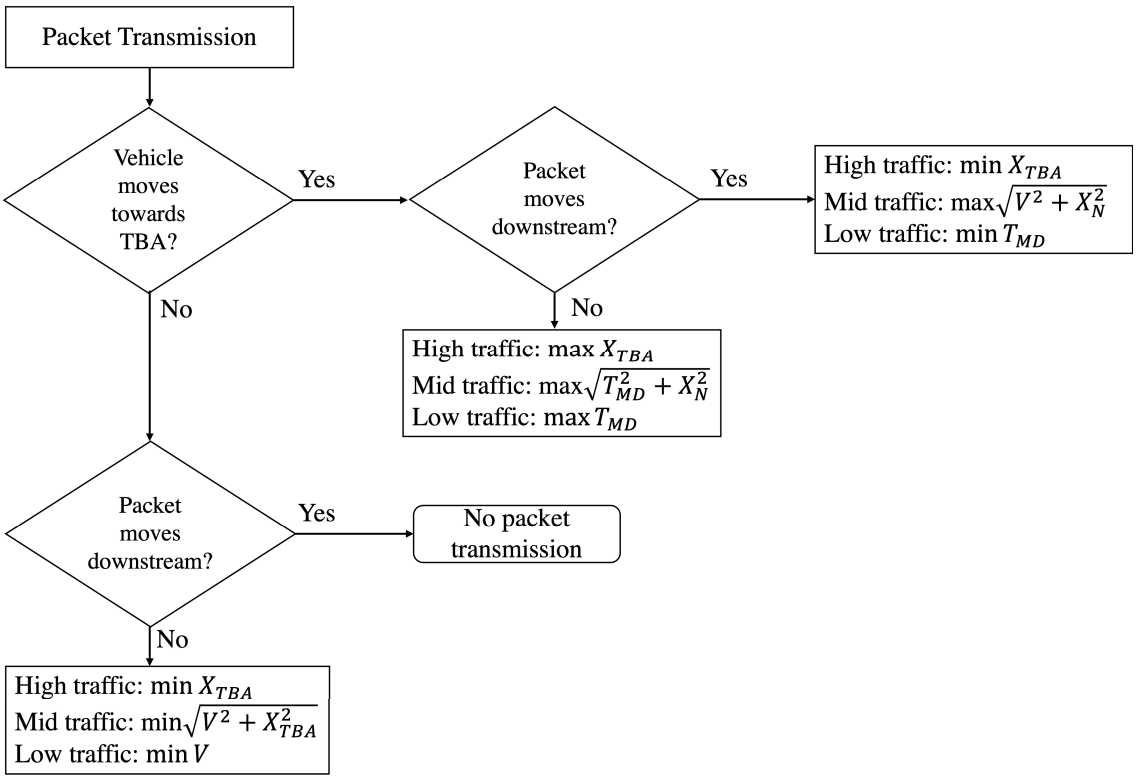

Fig. 6. Packet transmission flowchart

B. Intermediate Hops

An intermediate hop occurs when a node forwards a packet to the next node. Upon a node receiving a first or next hop, VBP decides if the packet forwards to the next hop vehicle. These intermediate hops occur until no next hop is available, where a packet is queued on the most recent recipient node. Using Figure 2 as an example, the first hop vehicle forwards the data packet to the next hop which is farther downstream than other vehicles in the traffic. Our algorithm chooses the vehicle to forward to based on Figure 7.

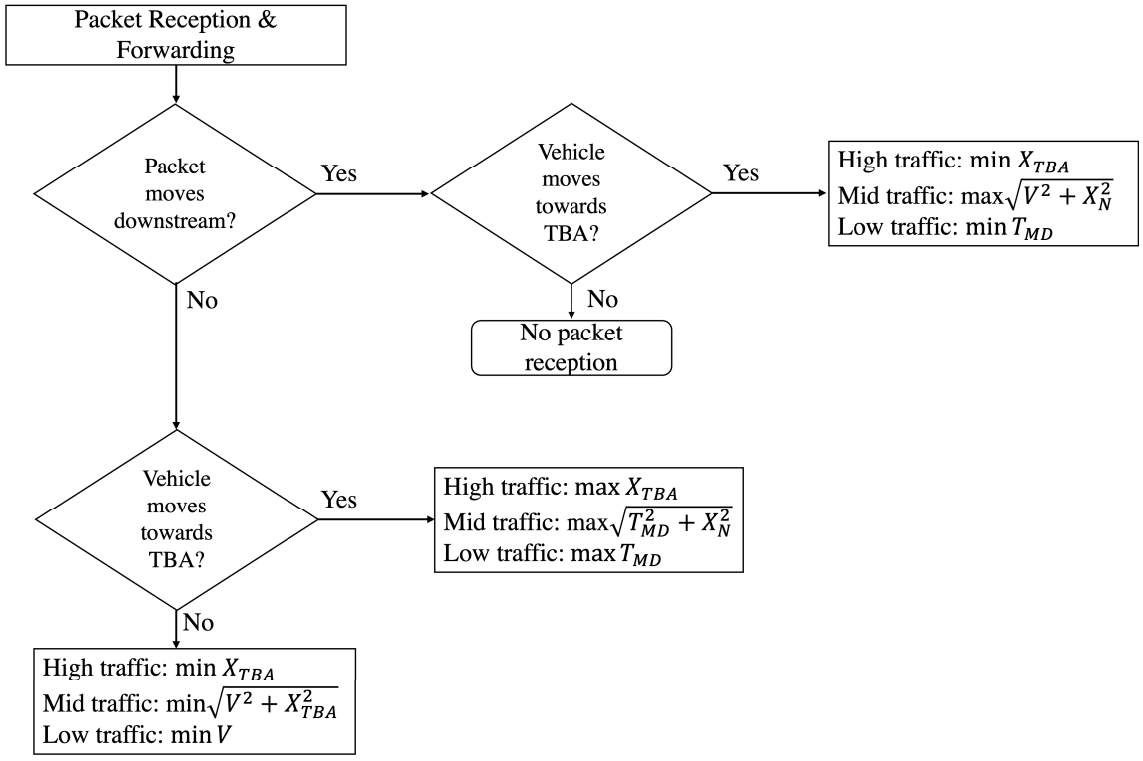

Fig. 7. Packet reception & forwarding flowchart

C. Limitations

1) Limitation 1: Packets are not broadcast to vehicles in between a hop: While VBP supports multi-hop capabilities, packets are not broadcast to the vehicles in between a hop. The Internet layer of the TCP/IP model is used to route packets on our protocol. To transmit a packet we set a destination MAC address to specify the vehicle to receive the packet. On our route we specify the destination IP address as a broadcast, allowing

all vehicles within transmission range to see the packet at the Internet layer. Since we set one MAC address as the recipient vehicle, nodes without a matching MAC address cannot take delivery of the packet.

To fix this we should specify a broadcast address rather than a specific vehicle address when setting the next hop. To limit every node from forwarding the broadcast packet, we should update our header data fields to include next hop ahead IP address and next hop behind IP address. These fields will be used during reception of a data packet, confirming a specific vehicle IP address is the next hop. When a vehicle IP address is the next hop, VBP will allow the node to forward and deliver the packet. Otherwise, the node can only locally deliver the broadcast packet. This ensures we keep the multi-hop capabilities while broadcasting the packet to vehicles in between hops.

2) Limitation 2: The UDP header is added on a data packet at the incorrect layer: VBP is a framework for ns-3 that is developed at the Internet layer of the TCP/IP model. At this layer we do not have access to modify the headers of an upper layer, the Transport layer, where UDP headers reside. This means UDP headers cannot be added or removed at a lower level such as the Internet layer. We noticed when a node receives a packet (on the Internet layer) a UDP header (Transport layer) is not found. We add the UDP header manually on the Internet layer as a patch, which is incorrect conceptually.

3) Limitation 3: VBP only supports the transmission of VBP data packets: When a packet is transmitted to the first hop we specifically check for VBP data packets. This means only VBP data packets can be routed using our framework. If researchers wish to use our framework for different types of data, they will have to modify this function to accept their packets.

4) Limitation 4: Simulations fail in complex road designs: The calculations of this protocol will fail if the forwarding vehicle makes a U-turn. Our routing protocol works on straight roads with minor changes in direction.

III. Routing Architecture

In VBP a data packet is sent from a source to a destination. This process is called unicast routing. Our framework consists of simulation scripts, a helper and a routing module. The helper and routing module are partitioned into the following C++ classes:

- VanetBroadcastHelper
- RoutingProtocol
- VanetRoutingHeader
- VbpHelloHeader
- VbpQueue
- VbpNeighbors
- RoutingTableEntry
- RoutingTable

} VBP routing module

The files that contain the implementation of each class can be found here.

A. Routing Protocol

`Class RoutingProtocol` handles the routing capabilities of VBP and inherits `class Ipv4RoutingProtocol` from ns-3. Eight functions originating in `Ipv4RoutingProtocol` are expected to be implemented:

- RouteOutput()
- RouteInput()
- NotifyInterfaceUp()
- NotifyInterfaceDown()
- NotifyAddAddress()
- NotifyRemoveAddress()
- PrintRoutingTable()
- SetIpv4()

A routing overview diagram is provided by ns-3[4], shown in Figure 8. The main classes in the diagram are `Ipv4RoutingProtocol` and `Ipv4L3Protocol` because it handles routing and forwarding of data packets. VBP lies within the `Ipv4RoutingProtocol` space and can access connected objects as displayed in the figure. Notably we use:

- UDP layer to execute RouteOutput()
- Sockets to interface between the routing protocol and applications
- Ipv4L3Protocol to send data packets

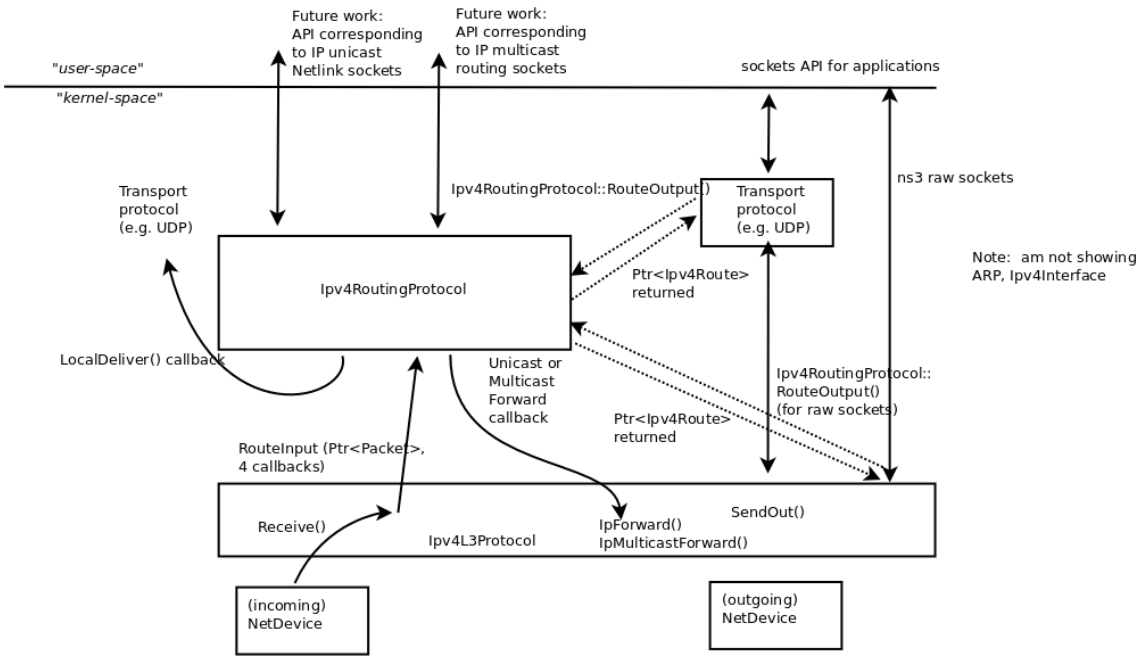

Fig. 8. ns-3 routing overview

The routing diagram for VBP is shown in Figure 9. `VBP Neighbors` is accessed for both neighborhood and routing purposes. `VanetBroadcastHelper` is used by the simulation, which is outside of our routing protocol environment. Objects (boxed) and functions (highlighted) are organized based on its process in the routing protocol in VBP. Items highlighted green are public functions and highlighted blue are private functions.

The VBP routing protocol is split between these processes:

- Initializing VBP in the simulation
- Maintaining and accessing neighborhood information
- Routing data packets

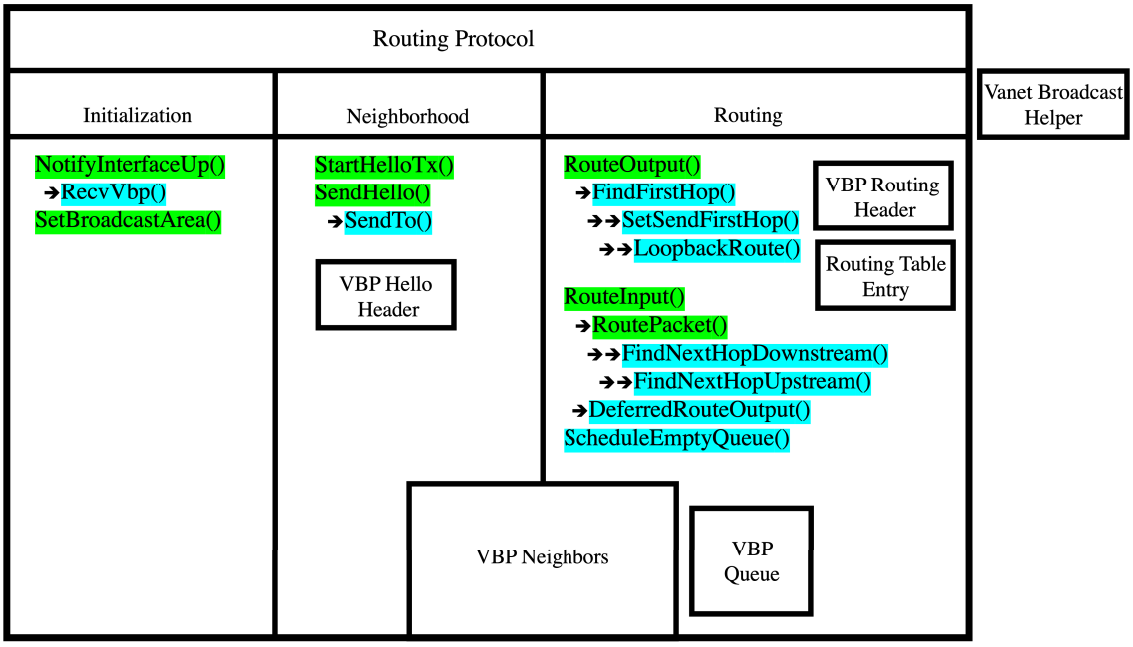

Fig. 9. VBP Routing Protocol

B. VBP Neighbors

Each vehicle starts the simulation with an empty neighbor list. As the simulation runs, control packets are exchanged between vehicles within a one hop distance. A node updates its neighbor list with the information found in these exchanged packets. Our routing protocol uses this neighbor list for calculations that help determine both the first and next hop of a data packet. Figure 10 exhibits the functions used by a node to maintain its neighbor list and obtain neighbor information.

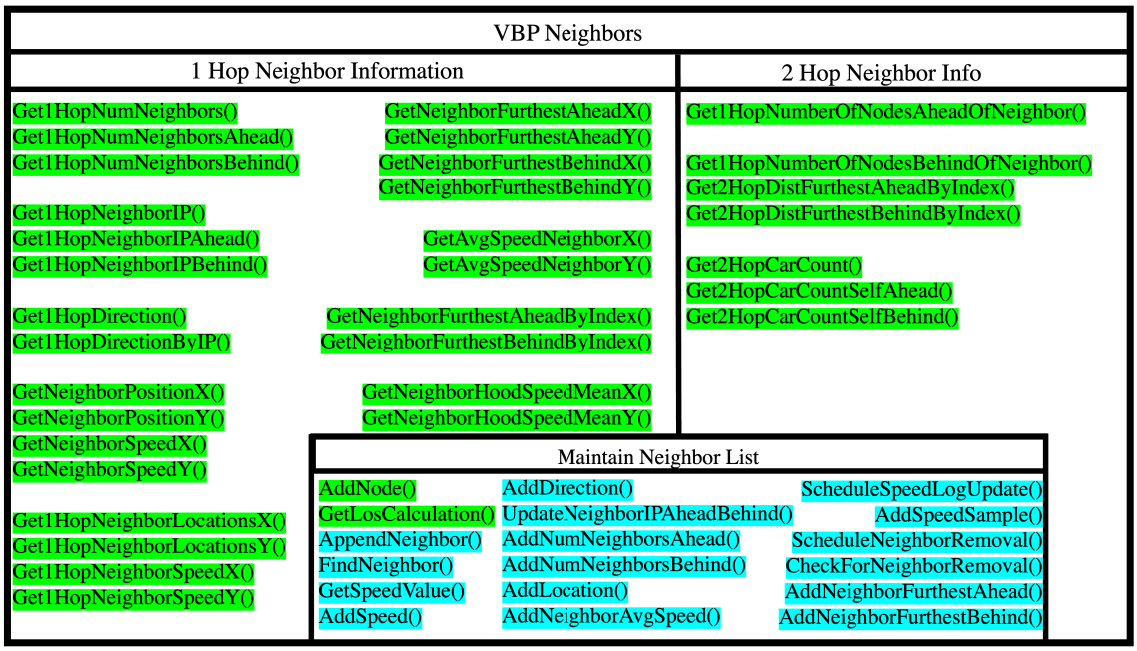

Fig. 10. VBP Neighbors

C. *VBP Queue*

VBP queues packets on a node when no next hop is found. The functions used to create the queue are indicated in Figure 11.

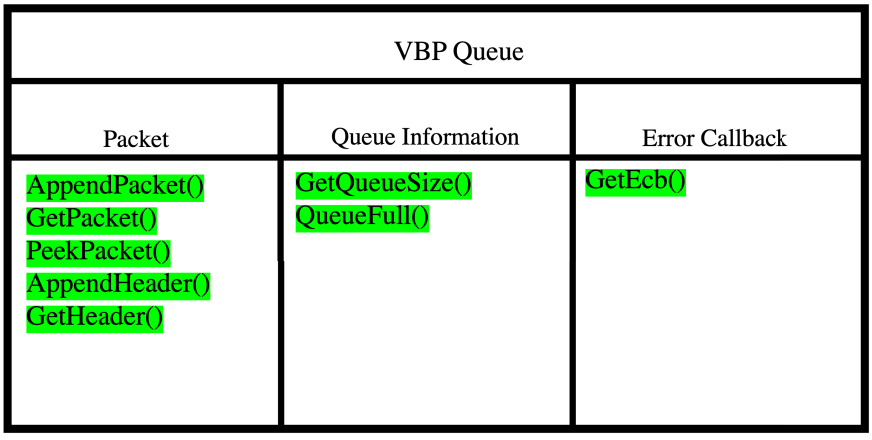

Fig. 11. VBP Queue

D. Vanet Broadcast Helper

A `Helper` bridges the simulation to the routing module by installing routing and internet capabilities on the nodes. Our helper `class VanetBroadcastHelper` inherits from `class Ipv4RoutingHelper` found in the internet module of ns-3. The functions used to create the helper are specified in Figure 12.

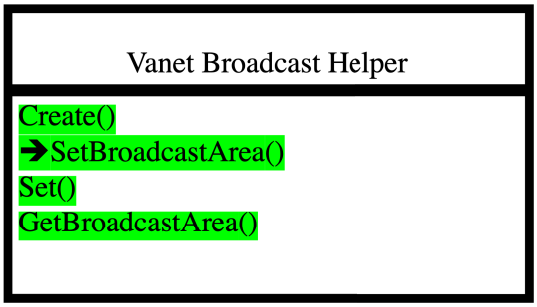

Fig. 12. Vanet Broadcast Helper

The helper contains a function called `Create()` that initializes VBP (Listing 1 Line 6 - Line 7) and internet capabilities (Line 8 - Line 9) on the simulation vehicles. We instruct nodes to begin transmitting control packets (Line 10) and for the simulation to set the TBA (Line 11).

Listing 1. Create()

```
1  Ptr<Ipv4RoutingProtocol>
2  VanetBroadcastHelper::Create(Ptr<Node> node) const
3  {
4      NS_ASSERT_MSG(!(isnan(m_broadcastArea[0])  isnan(m_broadcastArea[1])  isnan(
       m_broadcastArea[2])  isnan(m_broadcastArea[3])),"Need to set coordinates of broadcast
        area with VanetBroadcastHelper");
5
6      Ptr<vbp::RoutingProtocol> agent = m_agentFactory.Create<vbp::RoutingProtocol>();
7      node->AggregateObject(agent);
8      Ptr<Ipv4> ipv4 = node->GetObject<Ipv4>();
9      agent->SetIpv4(ipv4);
10     agent->StartHelloTx();
11     agent->SetBroadcastArea(m_broadcastArea);
12     return agent;
13 }
```

IV. Simulation Setup

Table II introduces the parameters used by our simulations. Some items are used for the network setup while others are used for the Friis Propagation Loss Model, which models the line of sight path loss in free space[6].

TABLE II
SIMULATION PARAMETERS

Simulation Parameter	Value
Net Address	10.1.1.0
Net Mask Address	255.255.255.0
Net Broadcast Address	10.1.1.255
UDP Port	8080
VBP Port	8081
Source Start Time [s]	1
Pk Interarrival Time [s]	1
Vehicle Distance [m]	5
Vehicle Speed [$\frac{m}{s}$]	15
Frequency [Hz]	2.4e9
Sys Loss	1
Min Loss [dB]	0
Tx Power [dBm]	110
Tx Gain [dB]	0
Rx Gain [dB]	0
Rx Sensitivity [dBm]	0

A. Create Nodes & Set Mobility

The simulation script initializes the vehicular network we wish to model by creating nodes (Listing 2 Line 4 - Line 6) and setting mobility (position and velocity) on the nodes (Line 8 - Line 12).

Listing 2. Mobility models

```
1  #include "ns3/mobility-module.h"
2  #include "ns3/constant-velocity-mobility-model.h"
3
4  uint32_t NumNodes = 5;
5  NodeContainer nodes;
6  nodes.Create(NumNodes);
7
8  float vehicleDistance = 50;
9  float vehicleSpeed = 5; //low 5, medium 15, high 30
10 for (int i = 0; i < int(NumNodes); i++) {
11    nodes.Get(i)->GetObject<ConstantVelocityMobilityModel>()->SetPosition(vehicleDistance*
        i, 0, 0));
12    nodes.Get(i)->GetObject<ConstantVelocityMobilityModel>()->SetVelocity(vehicleSpeed, 0,
        0));}
```

It is necessary to import the ns-3 modules that set position and velocity of nodes (Line 1 - Line 2). The simulation is set for each vehicle to start 50 meters apart (Line 8) and move at a velocity of 15 meters per second (Line 9). Traffic levels of

the simulation are adjusted by modifying these values. The simulation described by Listing 2 is illustrated through the network animator, illustrated in Figure 13.

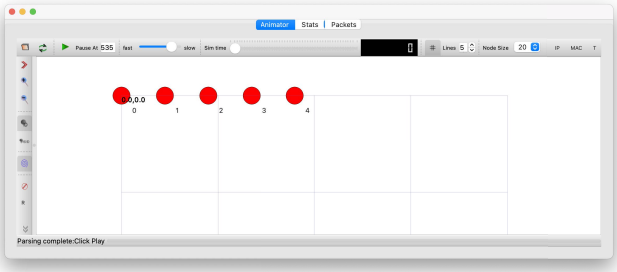

Fig. 13. Network animator example

B. *VanetBroadcastHelper*

`VanetBroadcastHelper` is called to install VBP on the simulation nodes (Listing 3 Line 5 & Line 7). Also required is `InternetStackHelper` which installs internet stack capabilities such as IP, TCP and UDP on each node (Line 4 & Line 8).

Listing 3. Install Routing

```
1 #include "vanet-broadcast-helper.h"
2 #include "ns3/internet-module.h"
3
4 InternetStackHelper stack;
5 VanetBroadcastHelper vbp;
6 vbp.SetBroadcastArea({100000, -10, 100050, 10});
7 stack.SetRoutingHelper(vbp);
8 stack.Install(nodes);
```

The user specifies the rectangular TBA coordinates in meters relative to the origin (Line 6). The coordinates follow the set $\{x_1, y_1, x_2, y_2\}$, where x_1 and y_1 represents the x-coordinate and y-coordinate of the upper left corner of the TBA and x_2 and y_2 represents the x-coordinate and y-coordinate of the bottom right corner of the TBA.

The coordinates set on Line 6 are exemplified in Figure 14. Positive 'y' is in the downward direction which follows the design principles set in other modules of ns-3.

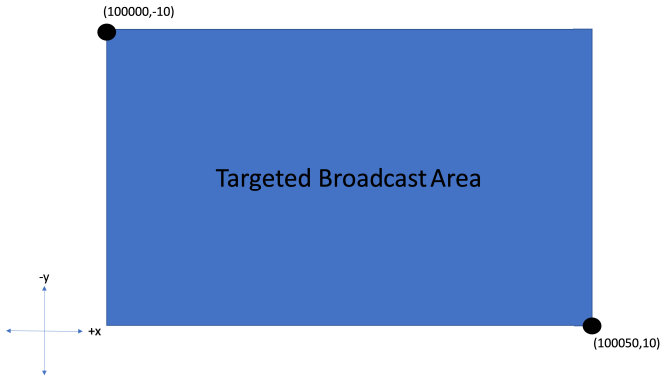

Fig. 14. The TBA mapped by the coordinates set in Listing 3 Line 6.

C. Applications

VBP routes data packets that are generated at the application layer of the TCP/IP model (Figure 15). For example, a dispatched emergency vehicle has a source application to create alert messages that will be sent to far-away traffic. All vehicles have a sink application, allowing it to receive packets generated from the source application.

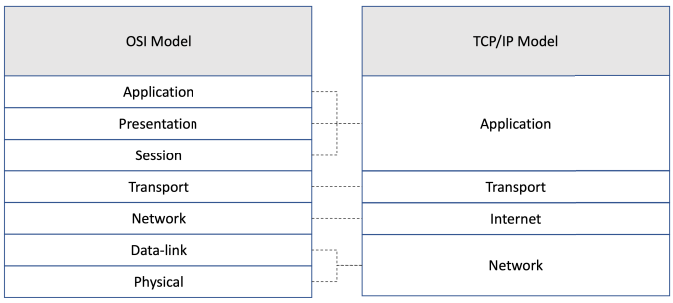

Fig. 15. TCP/IP model

The simulation script accesses the application layer to install both the source and sink applications. A socket is created to access the source node (Listing 4 Line 3) and initialize the app (Line 5 & Line 6) two seconds into the simulation (Line 7).

Listing 4. Install source application

```
1 #include "VbpApp.h"
2
3 Ptr<Socket> udpSourceSocket = Socket::CreateSocket(nodes.Get(0), UdpSocketFactory::
      GetTypeId());
4 Ptr<MyRandomExpTrafficApp> udpSourceAppPtr = CreateObject<MyRandomExpTrafficApp>();
5 udpSourceAppPtr->Setup(udpSourceSocket, Ipv4Address(NET_BROADCAST_ADDRESS), VBP_PORT,
      PacketSize, DataRate(AppDataRate), PRNGRunNumber);
6 nodes.Get(0)->AddApplication(udpSourceAppPtr);
7 udpSourceAppPtr->SetStartTime(Seconds(2));
```

Sink applications allow a node to receive packets generated by source applications. Every node should have this ability, explaining the loop process (Listing 5 Line 3 - Line 11). The set up process is similar to the source application (Line 5 - Line 10).

Listing 5. Install sink application

```
1  #include "MyRandomExpTrafficApp.h"
2
3  for(uint32_t i = 0; i < NumNodes; i++)
4  {
5     Ptr<Socket> vbpSocket = Socket::CreateSocket(nodes.Get(i), UdpSocketFactory::GetTypeId
       ());
6     Ptr<VbpApp> vbpAppPtr = CreateObject<VbpApp>();
7     vbpAppPtr->Setup(vbpSocket, VBP_PORT);
8     nodes.Get(i)->AddApplication(vbpAppPtr);
9     vbpSocket->SetRecvCallback(MakeCallback(&ReceivePacket));
10    vbpAppPtr->SetStartTime(Seconds(SOURCE_START_TIME));
11 }
```

V. Control Packets

Control packets are assigned port number 655 and packet type 'h' in our framework. They generate and transmit from each vehicle ten times every second. Also known as hello packets, their purpose is to exchange information between two nodes within communication range. The parameters of a hello packet header are presented in Table III.

TABLE III
CONTROL DATA

Control Parameter	Example	Description
Packet type	h	Signifies control 'hello' packets
Position X $[m]$	376.949	X-coordinate of current node
Position Y $[m]$	0	Y-coordinate of current node
Speed X $[\frac{m}{s}]$	5.007	Speed of node in x-direction
Speed Y $[\frac{m}{s}]$	0	Speed of node in y-direction
Neighbors ahead	5	number of one-hop neighbors downstream
Neighbors behind	5	number of one-hop neighbors upstream
Neighbor furthest ahead X-Pos $[m]$	625.651	neighboring node furthest ahead in the x-plane
Neighbor furthest ahead Y-Pos $[m]$	0	neighboring node furthest ahead in the y-plane
Neighbor furthest behind X-Pos $[m]$	126.466	neighboring node furthest behind in the x-plane
Neighbor furthest behind Y-Pos $[m]$	0	neighboring node furthest behind in the y-plane
Average Speed X $[\frac{m}{s}]$	5.007	Average x-plane speed of neighboring nodes
Average Speed Y $[\frac{m}{s}]$	0	Average y-plane speed of neighboring nodes

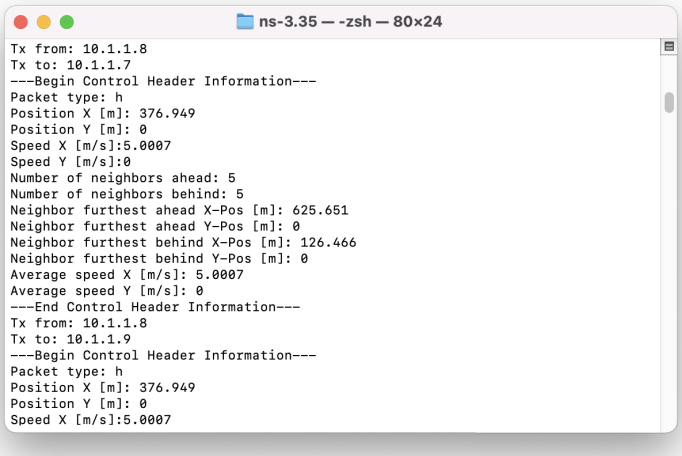

Fig. 16. Console output showing control packet

A node is considered a neighbor of another node if a hello packet is exchanged between them. Each node has a list of neighbors that initially starts empty at the beginning of a simulation. This list updates as a node either maintains contact with a neighbor or establishes contact with a new node. After thirty seconds of not receiving a hello packet from a neighbor, its entry on the neighbor list is dropped. The VBP routing module uses the neighbor list entries when calculating the optimal route of a data packet.

A. Generating & Sending

1) SendHello(): Nodes generate and send control packets every one-hundred milliseconds. In the VBP these tasks are completed by `RoutingProtocol::SendHello()` and `RoutingProtocol::StartHelloTx()`.

`SendHello()` creates the hello packet (Listing 6 Line 9) and calculates the control data (Line 14 through Line 28) that is added to the packet's header (Line 44).

The socket is used to calculate the position (Line 14) and velocity (Line 15) of a node. We call functions of class `VbpNeighbors()` to gather information about the furthest neighbor ahead (Line 17 - Line 21), the furthest neighbor behind (Line 23 - Line 27). We set this data in the hello header (Line 29 - Line 41). With the data set, a node broadcasts control packets to its neighbors (Line 47 - Line 58).

Listing 6. SendHello()

```
1  void
2  RoutingProtocol::SendHello()
3  {
4     NS_LOG_FUNCTION(this);
5     for (std::map<Ptr<Socket>, Ipv4InterfaceAddress>::const_iterator j = m_socketAddresses
        .begin(); j != m_socketAddresses.end(); ++j)
6     {
7        Ptr<Socket> socket = j->first;
8        Ipv4InterfaceAddress iface = j->second;
9        Ptr<Packet> packet = Create<Packet>();
10       // create header here
11       VbpHelloHeader HelloHeader;
12
13       // get info needed in packet from sockets
14       Vector pos = m_thisNode->GetObject<MobilityModel>()->GetPosition();
15       Vector vel = m_thisNode->GetObject<MobilityModel>()->GetVelocity();
16       // set dummy values to header setData (pass hardcoded values)
17       Vector furthestAhead = Vector3D(NAN, NAN, 0);
18       int furthestIdxAhead = m_neighborsListPointer->GetObject<VbpNeighbors>()->
       GetNeighborFurthestAheadByIndex(pos);
19       if (furthestIdxAhead >= 0)
20       {
21       furthestAhead = Vector3D(m_neighborsListPointer->GetObject<VbpNeighbors>()->
       GetNeighborPositionX(furthestIdxAhead), m_neighborsListPointer->GetObject<
       VbpNeighbors>()->GetNeighborPositionY(furthestIdxAhead), 0);
22       }
23       Vector furthestBehind = Vector3D(NAN, NAN, 0);
24       int furthestIdxBehind = m_neighborsListPointer->GetObject<VbpNeighbors>()->
       GetNeighborFurthestBehindByIndex(pos);
25       if (furthestIdxBehind >= 0)
26       {
27       furthestBehind = Vector3D(m_neighborsListPointer->GetObject<VbpNeighbors>()->
       GetNeighborPositionX(furthestIdxBehind), m_neighborsListPointer->GetObject<
       VbpNeighbors>()->GetNeighborPositionY(furthestIdxBehind), 0);
```

```
28        }
29        HelloHeader.SetData(m_helloPacketType,
30                            pos.x,
31                            pos.y,
32                            vel.x,
33                            vel.y,
34                            m_neighborsListPointer->GetObject<VbpNeighbors>()->
      Get1HopNumNeighborsAhead(),
35                            m_neighborsListPointer->GetObject<VbpNeighbors>()->
      Get1HopNumNeighborsBehind(),
36                            furthestAhead.x,
37                            furthestAhead.y,
38                            furthestBehind.x,
39                            furthestBehind.y,
40                            m_neighborsListPointer->GetObject<VbpNeighbors>()->
      GetAvgSpeedNeighborX(vel.x),
41                            m_neighborsListPointer->GetObject<VbpNeighbors>()->
      GetAvgSpeedNeighborY(vel.y));
42
43        // add header to packet
44        packet->AddHeader(HelloHeader);
45
46        // Send to all-hosts broadcast if on /32 addr, subnet-directed otherwise
47        Ipv4Address destination;
48        if (iface.GetMask() == Ipv4Mask::GetOnes())
49        {
50        destination = Ipv4Address("255.255.255.255");
51        }
52        else
53        {
54        destination = iface.GetBroadcast();
55        }
56        Time jitter = Time(MilliSeconds(Period_HelloTx + m_uniformRandomVariable->
      GetInteger(0, Jitter_HelloTx)));
57        Simulator::Schedule(jitter, &RoutingProtocol::SendHello, this);
58        SendTo(socket, packet, destination);
59    }
60  }
```

2) SendTo: After setting the data of the hello header `RoutingProtocol::SendTo()` sends the control packet to the destination socket (Listing 7 Line 4).

Listing 7. SendTo()

```
1  void
2  RoutingProtocol::SendTo(Ptr<Socket> socket, Ptr<Packet> packet, Ipv4Address destination)
3  {
4     socket->SendTo(packet, 0, InetSocketAddress(destination, VBP_HELLO_PORT));
5  }
```

3) StartHelloTx(): `StartHelloTx()` calls `SendHello()`, beginning the transmission of hello packets in our simulation. A variable called `jitter` is used to slightly deviate from a true period of a hundred milliseconds, modeling a realistic design in communications. (Listing 8 Line 5).

Listing 8. StartHelloTx()

```
1  void
2  RoutingProtocol::StartHelloTx()
3  {
4     m_uniformRandomVariable = CreateObject<UniformRandomVariable>();
5     Time jitter = Time(MilliSeconds(Period_HelloTx + m_uniformRandomVariable->GetInteger
        (0, Jitter_HelloTx)));
6     Simulator::Schedule(jitter, &RoutingProtocol::SendHello, this);
7  }
```

B. Receiving

Nodes receive hello packets from all nodes within transmission range. It is expected that a control packet comes from a known socket, created in `NotifyInterfaceUp()`. A receiving node receives and processes hello packets using `RoutingProtocol::RecvVbp()` and `RoutingProtocol::RecvHello()`.

1) NotifyInterfaceUp(): `NotifyInterfaceUp` says if a node's interface is active, the vehicle is findable by other nodes through control packets. We create two sockets for control packets on each node, one for local discovery and one for external discovery (Listing 9 Line 24 and Line 34).

Listing 9. NotifyInterfaceUp()

```
1  void
2  RoutingProtocol::NotifyInterfaceUp(uint32_t interface)
3  {
4     /*
5     Protocols are expected to implement this method to be notified of the state change of
       an interface in a node.
6     */
7     NS_LOG_FUNCTION(this);
8     if (interface > 1)
9     {
10       NS_LOG_WARN("VBP does not work with more then one interface.");
11    }
12    Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol>();
13    if (l3->GetNAddresses(interface) > 1)
14    {
15       NS_LOG_WARN("VBP does not work with more then one address per each interface.");
16    }
17    Ipv4InterfaceAddress iface = l3->GetAddress(interface, 0);
18    if (iface.GetLocal() == Ipv4Address("127.0.0.1"))
19    {
20       return;
21    }
22    // include check that m_socketaddresses is empty and m_socketSubnetBroadcastAddresses
       is empty. Print out message only one interface is allowed if check fails
23    //  Create a socket to listen only on this interface
24    Ptr<Socket> socket = Socket::CreateSocket(GetObject<Node>(), UdpSocketFactory::
       GetTypeId());
25    NS_ASSERT(socket != 0);
26    socket->SetRecvCallback(MakeCallback(&RoutingProtocol::RecvVbp, this));
27    socket->BindToNetDevice(l3->GetNetDevice(interface));
28    socket->Bind(InetSocketAddress(iface.GetLocal(), VBP_HELLO_PORT));
29    socket->SetAllowBroadcast(true);
30    socket->SetIpRecvTtl(true);
31    m_socketAddresses.insert(std::make_pair(socket, iface));
32
33    // create also a subnet broadcast socket
34    socket = Socket::CreateSocket(GetObject<Node>(), UdpSocketFactory::GetTypeId());
35    NS_ASSERT(socket != 0);
36    socket->SetRecvCallback(MakeCallback(&RoutingProtocol::RecvVbp, this));
37    socket->BindToNetDevice(l3->GetNetDevice(interface));
38    socket->Bind(InetSocketAddress(iface.GetBroadcast(), VBP_HELLO_PORT));
39    socket->SetAllowBroadcast(true);
40    socket->SetIpRecvTtl(true);
41    m_socketSubnetBroadcastAddresses.insert(std::make_pair(socket, iface));
42
43                                            << "NotifyInterfaceUp "
44                                            << "--- " << m_ipv4->GetNInterfaces() <<
       " Interfaces");
45
46    m_thisNode = socket->GetNode();
```

```
47    m_neighborsListPointer->GetObject<VbpNeighbors>()->SetThisNode(m_thisNode);
48  }
```

2) RecvVbp(): `RecvVbp()` verifies that received packets are from known sockets for either a local interface (Listing 10 Line 13) or broadcast interface (Line 17). Control packets are passed to `RecvHello()` to further process hello packets (Line 39).

We recommend debugging hello header information from the log functions in `RecvVbp()` (Line 28 - Line 36).

Listing 10. RecvVbp()

```
1   #include "vbp-hello-packet-header.h"
2
3   void
4   RoutingProtocol::RecvVbp(Ptr<Socket> socket)
5   {
6     NS_LOG_FUNCTION(this);
7     Address sourceAddress;
8     Ptr<Packet> packet = socket->RecvFrom(sourceAddress);
9     InetSocketAddress inetSourceAddr = InetSocketAddress::ConvertFrom(sourceAddress);
10    Ipv4Address sender = inetSourceAddr.GetIpv4();
11    Ipv4Address receiver;
12
13    if (m_socketAddresses.find(socket) != m_socketAddresses.end())
14    {
15      receiver = m_socketAddresses[socket].GetLocal();
16    }
17    else if (m_socketSubnetBroadcastAddresses.find(socket) !=
      m_socketSubnetBroadcastAddresses.end())
18    {
19      receiver = m_socketSubnetBroadcastAddresses[socket].GetLocal();
20    }
21    else
22    {
23      NS_ASSERT_MSG(false, "Received a packet from an unknown socket");
24    }
25    // remove the header from the packet:
26    VbpHelloHeader destinationHeader;
27    packet->PeekHeader(destinationHeader);
28    NS_LOG_LOGIC("---Tx From --- " << sender);
29    NS_LOG_LOGIC("---Tx To --- " << receiver);
30    NS_LOG_LOGIC("---Begin Header Information --- ");
31    NS_LOG_LOGIC("Packet Type: " << destinationHeader.GetPacketType());
32    NS_LOG_LOGIC("Position X: " << destinationHeader.GetPositionX());
33    NS_LOG_LOGIC("Position Y: " << destinationHeader.GetPositionY());
34    NS_LOG_LOGIC("Speed X: " << destinationHeader.GetSpeedX());
35    NS_LOG_LOGIC("Speed Y: " << destinationHeader.GetSpeedY());
36    NS_LOG_LOGIC("---End Header Information --- ");
37    if (destinationHeader.GetPacketType() == m_helloPacketType)
38    {
39      RecvHello(packet, receiver, sender);
40      NS_LOG_LOGIC("Neighbors List: " << "Receiver " << receiver << " Sender " << sender
      << " Packet type: " << destinationHeader.GetPacketType());
41    }
42  }
```

3) RecvHello(): `RecvHello()` determines if the node receiving the control packet is ahead or behind the sender (Listing 11 Line 7 - Line 37). Then the sending node is added to the receiving node's list of neighbors (Line 41 - Line 54).

Listing 11. RecvHello()

```
1   void
```

```
2     RoutingProtocol::RecvHello(Ptr<Packet> p, Ipv4Address receiver, Ipv4Address sender)
3     {
4       VbpHelloHeader helloHeader;
5       p->PeekHeader(helloHeader);
6       // determine if forwarding node is ahead=1 or behind=0 by using dot product
7       float dotProduct;
8       float dotProductVel;
9       Vector receiveNodePos = m_thisNode->GetObject<MobilityModel>()->GetPosition();
10      Vector diff = Vector3D(helloHeader.GetPositionX(), helloHeader.GetPositionY(), 0) -
       receiveNodePos; // vector pointing from receiving node to forwarding node
11      Vector receiveNodeVelocity = m_thisNode->GetObject<MobilityModel>()->GetVelocity();
12
13      if (receiveNodeVelocity.GetLength() == 0)
14      {
15        // if receiving node not moving, don't process anymore
16        return;
17      }
18
19      dotProductVel = receiveNodeVelocity.x * helloHeader.GetSpeedX() +
       receiveNodeVelocity.y * helloHeader.GetSpeedY();
20      if (dotProductVel <= 0)
21      {
22        // if velocity vectors do not align, don't process because neighbor moving in
       opposite direction
23        return;
24      }
25
26      dotProduct = receiveNodeVelocity.x * diff.x + receiveNodeVelocity.y * diff.y;
27      uint16_t direction; // 0 = behind, 1 = ahead
28      if (dotProduct >= 0)
29      {
30        // if dot product positive, then ahead
31        direction = 1;
32      }
33      else
34      {
35        // if dot product negative, then behind
36        direction = 0;
37      }
38
39      // use received packet to update neighbors information in object neighbors
40      // will add node as new neighbor or update information for that neighbor
41      m_neighborsListPointer->GetObject<VbpNeighbors>()->AddNode(sender,
42       direction,
43       helloHeader.GetNumNeighborsAhead(),
44       helloHeader.GetNumNeighborsBehind(),
45       helloHeader.GetPositionX(),
46       helloHeader.GetPositionY(),
47       helloHeader.GetSpeedX(),
48       helloHeader.GetSpeedY(),
49       helloHeader.GetNeighborFurthestAheadX(),
50       helloHeader.GetNeighborFurthestAheadY(),
51       helloHeader.GetNeighborFurthestBehindX(),
52       helloHeader.GetNeighborFurthestBehindY(),
53       helloHeader.GetAvgSpeedX(),
54       helloHeader.GetAvgSpeedY());
55    }
```

VI. Data Packets

Data packets are assigned port number 8081 and packet type 'd' in our framework. Also referred to as VBP packets, they are generated by the application on the source node (Listing 4) and routed using the routing layer of the TCP/IP model (Figure 15). Table IV introduces the parameters contained in the header of the data packets.

TABLE IV
VBP DATA

Parameter	Example	Description
Packet type	d	Signifies VBP data packet
Previous Hop IP	10.1.1.1	IPv4 Address of previous hop node
TBA Pos 1 X $[m]$	100000	X-coordinate of upper left TBA point
TBA Pos 1 Y $[m]$	-10	Y-coordinate of upper left TBA point
TBA Pos 2 X $[m]$	100050	X-coordinate of bottom right TBA point
TBA Pos 2 Y $[m]$	10	Y-coordinate of bottom right TBA point
TBA Expiration Time $[s]$	1e+09	Time

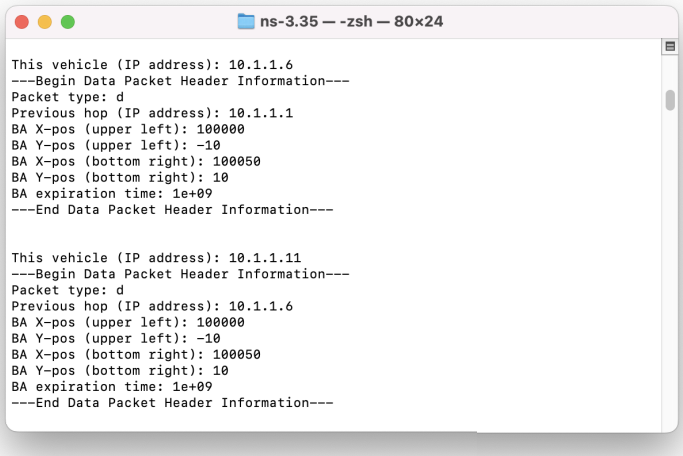

Fig. 17. Console output showing data packet information

A. *RouteOutput()*

`RoutingProtocol::RouteOutput()` deals with outgoing data packets from the source node by facilitating the packet's first hop. This occurs after the packet is generated by the source application. The input parameters are

- Ptr<Packet> packet, the packet to be routed by VBP
- const Ipv4Header &header, used as an input parameter
- Ptr<NetDevice> oif, the output interface netdevice. May be zero, or may be bound via socket options to a particular output interface.
- sockerr, Output parameter for error regarding sockets

Prior to routing `RouteOutput()` checks for available socket interfaces (Listing 12 Line 7 - Line 12). Then the data of the packet is set (Line 22) and added to the VBP header (Line 23). We confirm that a first hop is available with `RoutingProtocol::FindFirstHop()` (Line 28) and if so, call `RoutingProtocol::SetSendFirstHop()` (Line 32). If no first hop is currently available a loopback route is returned (Line 45).

Listing 12. RouteOutput()

```
1  Ptr<Ipv4Route>
2  RoutingProtocol::RouteOutput(Ptr<Packet> p, const Ipv4Header &header, Ptr<NetDevice> oif,
       Socket::SocketErrno &sockerr)
3      {
4        NS_LOG_FUNCTION(this);
5        Ptr<Ipv4Route> route;
6
7        if (m_socketAddresses.empty())
8        {
9          sockerr = Socket::ERROR_NOROUTETOHOST;
10         NS_LOG_LOGIC("No vbp interfaces");
11         return route;
12       }
13
14       sockerr = Socket::ERROR_NOTERROR;
15       VbpRoutingHeader routingHeader;
16       RoutingTableEntry rt;
17       Ipv4Address dst = header.GetDestination();
18       Ipv4Address src = header.GetSource();
19       Ipv4InterfaceAddress iface = m_socketAddresses.begin()->second;
20       Ipv4Address origin = iface.GetAddress();
21       Ptr<NetDevice> dev = m_ipv4->GetNetDevice(m_ipv4->GetInterfaceForAddress(iface.
     GetLocal()));
22       routingHeader.SetData(m_dataPacketType, origin, m_broadcastArea[0], m_broadcastArea
     [1], m_broadcastArea[2], m_broadcastArea[3], m_BroadcastTime);
23       p->AddHeader(routingHeader);
24       Ipv4Address nextHopAhead;
25       Ipv4Address nextHopBehind;
26       NS_LOG_LOGIC("Route Output Packet Type: " << routingHeader.GetPacketType());
27
28       if (FindFirstHop(&nextHopAhead, &nextHopBehind)) // find next hop
29       {
30         NS_LOG_LOGIC("Sender FindFirstHop: " << iface.GetAddress());
31         NS_LOG_LOGIC("Send First Hop - Ahead " << nextHopAhead << " Behind " <<
     nextHopBehind);
32         SetSendFirstHop(&nextHopAhead, &nextHopBehind, p, dev, iface, src, dst);
33       }
34       else
35       {
36         NS_LOG_LOGIC("Else Case: Send First Hop - Ahead " << nextHopAhead << " Behind "
     << nextHopBehind);
37         // Valid route not found, return loopback
38         uint32_t iif = (oif ? m_ipv4->GetInterfaceForDevice(oif) : -1);
39         DeferredRouteOutputTag tag(iif);
40         NS_LOG_DEBUG("Valid Route not found");
41         if (!p->PeekPacketTag(tag))
42         {
```

```
43            p->AddPacketTag(tag);
44          }
45          route = LoopbackRoute(header, oif);
46        }
47        return route;
48      }
```

B. *FindFirstHop()*

`FindFirstHop()` returns true when a valid IP address is found within transmission range of the source node. We first calculate and check if a vehicle is moving towards the TBA (Listing 13 Line 12 - Line 13). If an IP address that belongs to a node is returned our algorithm knows there is an available hop downstream (ahead) (Line 15). Next, we look for vehicular IP addresses upstream (behind) the source (Line 17). If a node is found either upstream or downstream of the source node, a valid IP address is within transmission range and `FindFirstHop()` returns true (Line 22) . If no nodes are found in either direction, signified by invalid IP addresses of 102.102.102.102 (Line 18), then no first hop is available and the function returns false (Line 20).

Listing 13. FindFirstHop()

```
1  bool
2  RoutingProtocol::FindFirstHop(Ipv4Address *nextHopAheadPtr, Ipv4Address *nextHopBehindPtr
       )
3  {
4       Vector vehiclePos = m_thisNode->GetObject<MobilityModel>()->GetPosition();
5       Vector vehicleVel = m_thisNode->GetObject<MobilityModel>()->GetVelocity();
6       float upperLeftBA_x = m_broadcastArea[0];
7       float upperLeftBA_y = m_broadcastArea[1];
8       float lowerRightBA_x = m_broadcastArea[2];
9       float lowerRightBA_y = m_broadcastArea[3];
10      Vector centerBA = Vector3D((upperLeftBA_x + lowerRightBA_x) / 2, (upperLeftBA_y +
       lowerRightBA_y) / 2, 0);
11      Vector vehicleToBA = centerBA - vehiclePos;
12      bool movingToBA = (vehicleVel.x * vehicleToBA.x + vehicleVel.y * vehicleToBA.y) >=
       0; // true if moving towards TBA
13      if (movingToBA)
14      {
15        nextHopAheadPtr->Set(FindNextHopDownstream(centerBA, movingToBA).Get());
16      }
17      nextHopBehindPtr->Set(FindNextHopUpstream(centerBA, movingToBA).Get());
18      if (*nextHopAheadPtr == Ipv4Address("102.102.102.102") && *nextHopBehindPtr ==
       Ipv4Address("102.102.102.102"))
19      {
20        return false;
21      }
22      return true;
23 }
```

C. *SetSendFirstHop()*

`SetSendFirstHop()` sets the route and sends the packet from `RouteOutput()`. A module from ns-3 called `ns3::Ipv4L3Protocol()` is used to send the data packet via the routing layer. VBP splits the potential routes of the packet send into three cases:

- Case 1: Both downstream and upstream
- Case 2: Downstream only
- Case 3: Upstream Only

We consider case 1 when a next hop is found both ahead and behind of the source vehicle (Listing 14 Line 8 - Line 21). Since a packet can only hop in one-direction, we copy the packet (Line 16) and set it on the opposing route. This allows us to set the route and send the original packet downstream (Line 11 - Line 14) and the copied packet upstream (Line 17 - Line 20).

Case 2 occurs when a hop is found only ahead of the source node. A downstream route is set and one packet is sent (Line 24 - Line 27). Case 3 happens when a hop is found only behind of the source node. An upstream route is set and one packet is sent (Line 31 - Line 34).

Listing 14. SetSendFirstHop()

```
1  void
2  RoutingProtocol::SetSendFirstHop(Ipv4Address *nextHopAheadPtr, Ipv4Address *
       nextHopBehindPtr, Ptr<Packet> p, Ptr<NetDevice> dev, Ipv4InterfaceAddress iface,
       Ipv4Address src, Ipv4Address dst)
3  {
4       RoutingTableEntry rt;
5       rt.SetOutputDevice(dev);
6       rt.SetInterface(iface);
7       Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol>();
8       if (*nextHopAheadPtr != Ipv4Address("102.102.102.102") && *nextHopBehindPtr !=
       Ipv4Address("102.102.102.102")) // case: hops both ahead and behind
9       {
10        // next hop ahead
11        Ptr<Ipv4Route> routeDownstream;
12        rt.SetNextHop(*nextHopAheadPtr);
13        routeDownstream = rt.GetRoute();
14        l3->Send(p, src, dst, PROT_NUMBER, routeDownstream);
15        // next hop behind
16        Ptr<Packet> q = p->Copy();
17        Ptr<Ipv4Route> routeUpstream;
18        rt.SetNextHop(*nextHopBehindPtr);
19        routeUpstream = rt.GetRoute();
20        l3->Send(q, src, dst, PROT_NUMBER, routeUpstream);
21      }
22      else if (*nextHopAheadPtr != Ipv4Address("102.102.102.102"))
23      {
24        Ptr<Ipv4Route> routeDownstream;
25        rt.SetNextHop(*nextHopAheadPtr);
26        routeDownstream = rt.GetRoute();
27        l3->Send(p, src, dst, PROT_NUMBER, routeDownstream);
28      }
29      else
30      {
31        Ptr<Ipv4Route> routeUpstream;
32        rt.SetNextHop(*nextHopBehindPtr);
33        routeUpstream = rt.GetRoute();
34        l3->Send(p, src, dst, PROT_NUMBER, routeUpstream);
35      }
36 }
```

D. *RouteInput()*

`RoutingProtocol::RouteInput()` is called when a node receives a packet. This function decides if a packet should be forwarded to the next hop node or locally delivered to the current node, and returns true in both scenarios. The input parameters are:

- Ptr<const Packet> p, received packet
- const Ipv4Header &header, input parameter used to form a search key for a route
- Ptr<const NetDevice> idev, pointer to ingress network device
- UnicastForwardCallback ucb, callback for the case in which the packet is to be forwarded as unicast
- MulticastForwardCallback mcb, callback for the case in which the packet is to be forwarded as multicast
- LocalDeliverCallback lcb, callback for the case in which the packet is to be locally delivered
- ErrorCallback ecb, callback to call if there is an error in forwarding

We first check for available sockets, IP addresses and packets (Listing 15 Line 6 - Line 13. A loopback device is used to defer routes and queue the packet until a route is found (Line 18 - Line 26). VBP is unicast routing and will not forward or deliver a packet if the destination is multicast (Line 28 - Line 31). We then determine whether the address and interface of a received packet can be accepted for local delivery (Line 34). Upon this confirmation we check for an available local callback (Line 36) and determine a node can accept packets for delivery if an available callback is found. Data packets are local delivered (Line 61) when `RoutePacket()` returns true (Line 43) while all control packets are local delivered (Line 65).

Listing 15. RouteInput()

```
1  bool
2  RoutingProtocol::RouteInput(Ptr<const Packet> p, const Ipv4Header &header,
3                              Ptr<const NetDevice> idev, UnicastForwardCallback ucb,
4                              MulticastForwardCallback mcb, LocalDeliverCallback lcb,
       ErrorCallback ecb)
5  {
6       if (m_socketAddresses.empty())
7       {
8         return false;
9       }
10      NS_ASSERT(m_ipv4 != 0);
11      NS_ASSERT(p != 0);
12      // Check if input device supports IP
13      NS_ASSERT(m_ipv4->GetInterfaceForDevice(idev) >= 0);
14      int32_t iif = m_ipv4->GetInterfaceForDevice(idev);
15      Ipv4Address dst = header.GetDestination();
16      Ipv4Address src = header.GetSource();
17      // Deferred route request
18      if (idev == m_lo)
19      {
20        DeferredRouteOutputTag tag;
21        if (p->PeekPacketTag(tag))
22        {
23          DeferredRouteOutput(p, header, ucb, ecb);
```

```
24          return false;
25        }
26      }
27      // VBP is not a multicast routing protocol
28      if (dst.IsMulticast())
29      {
30        return false;
31      }
32      Ipv4InterfaceAddress iface = m_socketAddresses.begin()->second;
33      // Unicast local delivery
34      if (m_ipv4->IsDestinationAddress(dst, iif))
35      {
36        if (lcb.IsNull() == false)
37        {
38          uint8_t protocol_numb = header.GetProtocol();
39          if (protocol_numb == PROT_NUMBER)
40          {
41            bool packetSentIndicator = false;
42            Ptr<Packet> q = p->Copy();
43            bool lcbIndicator = RoutePacket(q, dst, src, &packetSentIndicator); // true
    lcb. false no lcb
44            if (lcbIndicator)
45            {
46              UdpHeader udpHead;
47              udpHead.SetDestinationPort(VBP_DATA_PORT); //8081
48              udpHead.SetSourcePort(VBP_DATA_PORT); //8081
49              udpHead.InitializeChecksum(header.GetSource(), header.GetDestination(),
    PROT_NUMBER);
50              q->AddHeader(udpHead);
51              Ipv4Header headerCopy;
52              headerCopy.SetDestination(header.GetDestination());
53              headerCopy.SetDscp(header.GetDscp());
54              headerCopy.SetEcn(header.GetEcn());
55              headerCopy.SetIdentification(header.GetIdentification());
56              headerCopy.SetPayloadSize(header.GetPayloadSize());
57              headerCopy.SetProtocol(UDP_PROT_NUMBER); //17
58              headerCopy.SetSource(header.GetSource());
59              headerCopy.SetTos(header.GetTos());
60              headerCopy.SetTtl(header.GetTtl());
61              lcb(q, headerCopy, iif);
62            }
63            return true;
64          }
65          lcb(p, header, iif);
66        }
67        else
68        {
69          NS_LOG_ERROR("Unable to deliver packet locally due to null callback " << p->
    GetUid() << " from " << src);
70          ecb(p, header, Socket::ERROR_NOROUTETOHOST);
71        }
72        return true;
73      }
74      NS_LOG_ERROR("Unable to forward packet due to not being a VANET Broadcast Protocol
    data packet " << p->GetUid() << " from " << src);
75      ecb(p, header, Socket::ERROR_NOROUTETOHOST);
76      return false;
77  }
```

E. RoutePacket()

`RoutingProtocol::RoutePacket()` is used in `RouteInput()` as an indicator for local callback. Two cases trigger local delivery of the data packet:

- Case 1: the vehicle that contains the packet is already in the TBA area (Listing 16 Line 6 - Line 16)
- Case 2: the vehicle that contains the packet is currently not in the TBA and is expected to reach the TBA before expiration (Line 18 - Line 34)

The first case occurs when a vehicle is already in the TBA so this function returns true, indicating a node takes local delivery of a packet at the application layer of the TCP/IP model (Line 14). For the second case, calculations that consider a vehicle's velocity (Line 32) and position compared to the TBA center (Line 33) are used to determine if a vehicle will reach the TBA before expiration time. If true a packet local delivers to the application layer. When Case 2 occurs, the packet is forwarded to the next hop (Line 35 - Line 74). The packet is sent using `ns3::Ipv4L3Protocol()` to the next hop based on VBP's routing algorithm, giving our framework multi-hop functionality. Packets are queued if our algorithm indicates there is no first or next hop available (Line 68 - Line 73).

Listing 16. RoutePacket()

```
1   bool
2   RoutingProtocol::RoutePacket(Ptr<Packet> p, Ipv4Address dst, Ipv4Address src, bool *
     packetSentIndicator){
3      *packetSentIndicator = false;
4      Vector vehiclePos = m_thisNode->GetObject<MobilityModel>()->GetPosition();
5      Ptr<VbpNeighbors> neighborsList = m_neighborsListPointer->GetObject<VbpNeighbors>();
6      // case 1: vehicle already in targeted broadcast area
7      VbpRoutingHeader routingHeader;
8      p->PeekHeader(routingHeader);
9      if ((routingHeader.GetPosition1X() <= vehiclePos.x) && (vehiclePos.x <=
     routingHeader.GetPosition2X()))
10     {
11       if ((routingHeader.GetPosition1Y() <= vehiclePos.y) && (vehiclePos.y <=
     routingHeader.GetPosition2Y()))
12       {
13         p->RemoveHeader(routingHeader);
14         return true; // true = lcb
15       }
16     }
17     // case 2: vehicle not in TBA and may reach TBA before expiration
18     Vector BA1 = Vector3D(routingHeader.GetPosition1X(), routingHeader.GetPosition1Y(),
     0); // for targeted broadcast area point one
19     Vector BA2 = Vector3D(routingHeader.GetPosition2X(), routingHeader.GetPosition2Y(),
     0); // for targeted broadcast area point two
20     Vector centerBA = Vector3D((BA1.x + BA2.x) / 2, (BA1.y + BA2.y) / 2, 0);
21     float neighborhoodSpeed = Vector3D(neighborsList->GetNeighborHoodSpeedMeanX(),
     neighborsList->GetNeighborHoodSpeedMeanY(), 0).GetLength();
22     float currentMDT = CalculateDistance(vehiclePos, centerBA) / neighborhoodSpeed;
23     bool closeToBA = false;
24     Ptr<Packet> q = p->Copy();
25     if ((Simulator::Now() / 1e9 + Seconds(currentMDT)) <= Seconds(m_BroadcastTime))
26     {
27       p->RemoveHeader(routingHeader);
28       closeToBA = true; // will need to return closeToBA after forwarding. return true
29     }
30     bool enqueuePacketIndicator = false;
31     Ipv4Address prevHopIP = routingHeader.GetPrevHopIP();
32     Vector vehicleVel = m_thisNode->GetObject<MobilityModel>()->GetVelocity();
33     Vector vehicleToBA = centerBA - vehiclePos;
34     bool movingToBA = (vehicleVel.x * vehicleToBA.x + vehicleVel.y * vehicleToBA.y) >=
     0; // true if moving towards TBA
35     Ipv4Address nextHopAhead;
```

```
36       Ipv4Address nextHopBehind;
37       // forward packet for case 2
38       if (FindNextHop(&nextHopAhead, &nextHopBehind, centerBA, movingToBA, closeToBA, &
         enqueuePacketIndicator, prevHopIP))
39       {
40         Ipv4InterfaceAddress iface = m_socketAddresses.begin()->second;
41         Ptr<NetDevice> dev = m_ipv4->GetNetDevice(m_ipv4->GetInterfaceForAddress(iface.
         GetLocal()));
42         RoutingTableEntry rt;
43         rt.SetOutputDevice(dev);
44         rt.SetInterface(iface);
45         Ptr<Ipv4L3Protocol> l3 = m_ipv4->GetObject<Ipv4L3Protocol>();
46         Ipv4Address thisVehicleIP = iface.GetAddress();
47         q->RemoveHeader(routingHeader);
48         routingHeader.SetData(m_dataPacketType, thisVehicleIP, routingHeader.GetPosition1X
         (), routingHeader.GetPosition1Y(), routingHeader.GetPosition2X(), routingHeader.
         GetPosition2Y(), routingHeader.GetBroadcastingTime());
49         q->AddHeader(routingHeader);
50         if (nextHopAhead != Ipv4Address("102.102.102.102"))
51         {
52           Ptr<Ipv4Route> routeDownstream;
53           rt.SetNextHop(nextHopAhead);
54           routeDownstream = rt.GetRoute();
55           l3->Send(q, src, dst, PROT_NUMBER, routeDownstream);
56           *packetSentIndicator = true;
57         }
58         else
59         {
60           Ptr<Ipv4Route> routeUpstream;
61           rt.SetNextHop(nextHopBehind);
62           routeUpstream = rt.GetRoute();
63           l3->Send(q, src, dst, PROT_NUMBER, routeUpstream);
64           *packetSentIndicator = true;
65         }
66       }
67       Ipv4InterfaceAddress iface = m_socketAddresses.begin()->second;
68       if (enqueuePacketIndicator)
69       {
70         m_queuePointer->GetObject<VbpQueue>()->AppendPacket(q);
71         Ipv4Header header;
72         m_queuePointer->GetObject<VbpQueue>()->AppendHeader(header);
73       }
74       return closeToBA;
75     }
```

VII. Tests & Results

We use multiple traffic levels across four simulations to verify VBP operates the following ways:

- Data packets hop towards the direction of the TBA
- Data packets queue on a vehicle when no next hop is available

Each test simulates a caravan of 21 vehicles, where vehicles are placed every 50m starting at the origin and set to move at the same constant velocity. An IP address is assigned to each vehicle. The caravan vehicle closest to the origin has IP address 10.1.1.1 and the caravan vehicle farthest from the origin has IP address 10.1.1.21. Vehicles in between have their own IP address. For example, the sixth caravan vehicle from the origin has IP address 10.1.1.6. Dependent on the simulation, the targeted broadcast area is placed 10,000m either to the left or right of the caravan. The TBA has a long enough expiration time to ensure all vehicles will reach the center of it. In total we run 18 simulations and each passes our tests.

A. Experiment 1: Caravan Moving Towards Targeted Broadcast Area

This simulation tests which node on our VANET is chosen as a first or intermediate hop, and which node of the caravan is the last to be the recipient of a forwarded data packet. The TBA is to the right of the caravan that is moving towards the right. Within this experiment we run two tests across three traffic levels each. One test places the source application on the left most vehicle of the caravan (vehicle with IP address 10.1.1.1) and the second test places the source application on the middle vehicle of the caravan (vehicle with IP address 10.1.1.11). All tests results are successful and show that the packet is forwarded towards the TBA using multi-hop routing. The last recipient of a forwarded data packet is the right most vehicle of the caravan (vehicle with IP address 10.1.1.21). The packet queue grows on this vehicle only.

TABLE V
Experiment 1 results

Test Parameters				
Source (IP Address)	Caravan Movement Relative to TBA	TBA Location Relative to Caravan	Intermediate Hops (IP Address)	Destination (IP Address)
10.1.1.1	Towards	Right	10.1.1.6 10.1.1.11 10.1.1.16	10.1.1.21
10.1.1.11	Towards	Right	10.1.1.16	10.1.1.21

B. Experiment 2: Caravan Moving Away from Targeted Broadcast Area

This simulation tests which node on our VANET is chosen as a first or intermediate hop, and which node of the caravan is the last to be the recipient of a forwarded data packet. The TBA is to the left of the caravan that is moving towards the right. Within this experiment we run two tests across three traffic levels each. One test places the source application on the right most vehicle of the caravan (vehicle with IP address 10.1.1.21) and the second test places the source application on the middle vehicle of the caravan (vehicle with IP address 10.1.1.11). All tests results are successful and show that the packet is forwarded towards the TBA using multi-hop routing. The last recipient of a forwarded data packet is the right most vehicle of the caravan (vehicle with IP address 10.1.1.1). The packet queue grows on this vehicle only.

TABLE VI
EXPERIMENT 2 RESULTS

Test Parameters				
Source (IP Address)	Caravan Movement Relative to TBA	TBA Location Relative to Caravan	Intermediate Hops (IP Address)	Destination (IP Address)
10.1.1.21	Away	Left	10.1.1.16 10.1.1.11 10.1.1.6	10.1.1.1
10.1.1.11	Away	Left	10.1.1.6	10.1.1.1

C. Experiment 3: Queuing When Caravan Moves Towards Targeted Broadcast Area

This simulation introduces an additional node to the VANET (vehicle with IP address 10.1.1.22) that is placed between the caravan and the TBA. The TBA is to the right of the caravan that is moving towards the right. Node 10.1.1.22 travels slower than the caravan and comes into contact range of the caravan halfway through the simulation time. Within the caravan, the packet performs multiple hops as it travels from vehicle 10.1.1.1 to vehicle 10.1.1.21 (shown in Subsection VII-A). The packets are queued on node 10.1.1.21 until it comes into communication range of vehicle 10.1.1.22. The queue transfers to node 10.1.1.22, as we expect in a successful test result.

TABLE VII
EXPERIMENT 3 RESULTS

Test Parameters					
Source (IP Address)	Caravan Movement Relative to TBA	TBA Location Relative to Caravan	Intermediate Hops (IP Address)	Destination (IP Address)	Empty Queue to (IP Address)
10.1.1.1	Towards	Right	10.1.1.6 10.1.1.11 10.1.1.16	10.1.1.21	10.1.1.22

D. Experiment 4: Queuing When Caravan Moves Away from Targeted Broadcast Area

This simulation introduces an additional node to the VANET (vehicle with IP address 10.1.1.22) that is placed between the caravan and the TBA. The TBA is to the left of the caravan that is moving towards the right. Node 10.1.1.22 travels faster than the caravan and comes into contact range of the caravan halfway through the simulation time. Within the caravan, the packet performs multiple hops as it travels from vehicle 10.1.1.21 to vehicle 10.1.1.1 (shown in Subsection VII-B). The packets are queued on node 10.1.1.1 until it comes into communication range of vehicle 10.1.1.22. The queue transfers to node 10.1.1.22, as we expect in a successful test result.

TABLE VIII
EXPERIMENT 4 RESULTS

Test Parameters					
Source (IP Address)	Caravan Movement Relative to TBA	TBA Location Relative to Caravan	Intermediate Hops (IP Address)	Destination (IP Address)	Empty Queue to (IP Address)
10.1.1.21	Away	Left	10.1.1.16 10.1.1.11 10.1.1.6	10.1.1.21	10.1.1.22

VIII. Conclusions

This thesis presents the work done to develop VBP, a routing protocol for ns-3. This routing protocol has applications for improving driver awareness and road safety. We extend the capabilities of other simulators by allowing multi-hop capability and dynamic routing. Algorithms in VBP determine the first and next hop of a packet based on traffic conditions such as vehicle direction of travel and traffic levels. Future improvements of VBP include developing more complex algorithms that consider road conditions such as curves and intersections.

Simulations were conducted to test our protocol. We ran six simulations across three different traffic levels, each producing a successful result. Packets showed correct behavior when multi hopping between nodes on our VANET and queuing when no first or next hop is available. The simulation scripts used to test VBP are included so researchers can easily run experiments using our protocol. Components of these scripts, such as mobility, are modifiable so a user can customize starting location, velocity and number of vehicles. Upon future improvements to the development of VBP, more complicated simulations can be run that consider real-world road conditions.

Documentation was created to detail the process of creating a custom routing protocol for ns-3. To the best of our knowledge there are limited resources available that explain the purpose and code inside the various ns-3 routing components. We diagrammed the routing architecture (Section III) and explained classes used to send and receive control (Section V) and data (Section VI) packets. This work should help future researchers in this area as they create their own custom routing protocol.

IX. Appendix: Calculate Traffic Levels

The traffic levels of a simulation are based on a qualitative measurement called the level of service (LOS). VBP calculates the volume to capacity of a road and compares it to lookup tables provided by the Transport Research Board[7]. We provide an example LOS lookup table in Table IX, showing the different categories of traffic.

TABLE IX
Level of service

LOS	Description	Vehicle Spacing $[m]$	Density $[pc/mi/ln]$
A	Free flow and unimpeded maneuverability	167	$\leq$ 10
B	Reasonable free flow restricted maneuverability	100	$\leq$ 16
C	Stable flow	67	$\leq$ 24
D	Approaching unstable flow flow	50	$\leq$ 32
E	Unstable flow	37	$\leq$ 36.7/39.7
F	Breakdown flow (operating at capacity)	$<$ 37	variable

X. Table of Acronyms

Targeted broadcast area	TBA
Message delivery time	T_{MD}
Distance to neighbor	X_N
Distance to targeted broadcast area	X_{TBA}
Vehicle speed	V
Level of service	LOS
Vehicular Ad-hoc Network	VANET

REFERENCES

[1] "Dedicated short range communications (dsrc) service," Apr 2019. [Online]. Available: https://www.fcc.gov/wireless/bureau-divisions/mobility-division/dedicated-short-range-communications-dsrc-service

[2] "Veins - vehicles in network simulations." [Online]. Available: https://veins.car2x.org/documentation/

[3] R. Ventura, "Targeted broadcasting in vehicular ad-hoc networks," Master's thesis, Loyola Marymount University, 2022 [Unpublished].

[4] "ns-3 routing overview." [Online]. Available: https://www.nsnam.org/docs/models/html/routing-overview.html#:~:text=ns%2D3%20is%20intended%20to,research%20into%20unorthodox%20routing%20techniques.

[5] "Omnet++ - fully automatic static routing table configuration." [Online]. Available: https://inet.omnetpp.org/docs/tutorials/configurator/doc/step4.html

[6] "Propagation - model library." [Online]. Available: https://www.nsnam.org/docs/models/html/propagation.html

[7] "Traffic level of service calculation methods."