GUILHERME DÂNRLEY SILVA HANAUER

UMA ANÁLISE COMPARATIVA DOS SIMULADORES PARA REDES VEICULARES VEINS E AIRPLUG

LAVRAS – MG

2021

GUILHERME DÂNRLEY SILVA HANAUER

UMA ANÁLISE COMPARATIVA DOS SIMULADORES PARA REDES VEICULARES VEINS E AIRPLUG

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para obtenção do título de Bacharel.

Prof. Dr. Hermes Pimenta de Moraes Junior
Orientador

LAVRAS – MG

2021

Ficha catalográfica elaborada pela Coordenadoria de Processos Técnicos

da Biblioteca Universitária da UFLA

Hanauer, Guilherme Dânrley Silva

Uma análise comparativa dos simuladores para redes veiculares Veins e Airplug / . 2^a ed. rev., atual. e ampl. – Lavras : UFLA, 2021.

85 p. : il.

Monografia–Universidade Federal de Lavras, 2021.

Orientador: Prof. Dr. Hermes Pimenta de Moraes Junior.

Bibliografia.

1. TCC. 2. Monografia. 3. Dissertação. 4. Tese. 5. Trabalho Científico – Normas. I. Universidade Federal de Lavras. II. Título.

CDD-808.066

GUILHERME DÂNRLEY SILVA HANAUER

UMA ANÁLISE COMPARATIVA DOS SIMULADORES PARA REDES VEICULARES VEINS E AIRPLUG
A COMPARATIVE ANALYSIS OF THE VEHICULAR NETWORK SIMULATORS VEINS AND AIRPLUG

Monografia apresentada à Universidade Federal de Lavras, como parte das exigências do curso de Ciência da Computação, para obtenção do título de Bacharel.

APROVADA em 16 de Novembro de 2021.

Prof. PhD. Luiz Henrique Andrade Correia	UFLA
Prof. Dr. Neumar Costa Malheiros	UFLA
Prof. PhD. Hermes Pimenta de Moraes Junior	UFLA

Prof. Dr. Hermes Pimenta de Moraes Junior
Orientador

LAVRAS – MG
2021

Dedico este Trabalho de Conclusão de Curso aos meus pais e a meu irmão.

AGRADECIMENTOS

Sou grato a meus pais por não terem desistido de mim nesse longo percurso da graduação. Agradeço também, a todas as pessoas que conheci nessa jornada, pois, elas me tornaram o que sou hoje. Muito Obrigado.

RESUMO

Este trabalho analisou o impacto que os simuladores de redes veiculares têm nos algoritmos. Para isso foram escolhidos dois simuladores: *Airplug* e *framework Veins*. Foi realizado um levantamento bibliográfico sobre esses dois simuladores e como fonte de comparação adicional, o algoritmo AND (Adaptative Neighbor Discovery) já implementado no simulador *Airplug* foi implementado no *framework Veins* para que uma comparação de desempenho desse algoritmo nos dois simuladores pudesse ser feita. Foi descoberto que os simuladores tem propostas diferentes. Airplug sendo um simulador leve e que permite bastante liberdade aos desenvolvedores ao custo do realismo das simulações. O *framework Veins* por outro lado é um simulador voltado ao realismo, portanto, apresenta modelos rígidos e complexos de simulação. Por fim, apesar das diferenças, a adaptação do algoritmo AND se mostrou satisfatória, com os resultados obtidos no simulador *Veins* muito próximos aos resultados obtidos no Airplug.

Palavras-chave: Redes veiculares; Simuladores; Análise comparativa; Modelagem; Algoritmos.

ABSTRACT

This work aimed to analyze the impact that vehicular network simulators have on the algorithms that are tested in them. To achieve this, two simulators were chosen: *Airplug* and the *Veins framework*. A bibliographic survey was carried out on these two simulators and as an additional source of comparison, the AND (Adaptive Neighbor Discovery) algorithm was implemented in the *Veins framework* so that a performance comparison of this algorithm in the two simulators could be made. It was found that simulators have different proposals. Airplug being a lightweight simulator that allows developers a lot of freedom at the cost of the realism of the simulations. The *Veins framework* on the other hand is a simulator focused on realism, therefore, it presents rigid and complex simulation models. Finally, despite the differences, the adaptation of the AND algorithm proved to be satisfactory, with the results obtained in the *Veins* simulator very close to the results obtained in Airplug.

Keywords: Vehicular networks; Simulators; Comparative analysis; Algorithms; Modeling.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

O desenvolvimento tecnológico alcançado nas últimas décadas proporcionou a inclusão de novas tecnologias nos mais diversos dispositivos. Os veículos sendo um destes dispositivos, hoje em dia é comum encontrar carros com GPS, câmeras, radares, serviços como controle de estabilidade e estacionamento automático.

Apesar do impacto imediato que estas tecnologias trazem aos usuários, elas não promovem a interação entre veículos. O próximo passo nesta evolução seria um sistema capaz de possibilitar a comunicação entre os veículos e oferecer condições para que aplicações com diversos requisitos sejam atendidas. Estas aplicações compõem um Sistema Inteligente de Transporte (*Inteligent Transportation System* - ITS). Em um ITS os veículos seriam capazes de trocar informações o que permitiria aplicações como monitoramento colaborativo do tráfego, prevenção de colisões e auxílio em cruzamentos mal sinalizados.

Estes sistemas de comunicação entre veículos são chamados de Redes Veiculares ou VANETs (*Vehicular Ad hoc Networks*). Estas redes são compostas por carros, ônibus, caminhões e também infraestrutura às margens das rodovias. Apesar de serem redes sem fio as VANETs apresentam vários desafios para sua adoção em larga escala, como comunicação intermitente entre os nós, variação de escala e densidade, dinamicidade do ambiente (KARAGIANNIS et al., 2011) .

Dado a natureza desses sistemas e sua importância, uma vez que, há riscos envolvendo a vida humana no trânsito. A coleta de dados para realização de testes para esses sistemas não é uma tarefa trivial. Podendo ser realizada usando três métodos diferentes: Análise matemática, testes operacionais em campo e simulações.

O método de análise matemática é um estudo analítico do problema, distribuições estatísticas são utilizadas para gerar os modelos necessários para a simulação. Isso tende a simplificar alguns parâmetros da simulação, acarretando assim, em resultados incorretos. Testes operacionais expõem os protocolos desenvolvidos

a ambientes reais, portanto, apresentam os melhores resultados. As desvantagens desse tipo de abordagem são o alto custo em termos de verba e tempo e a dificuldade de realizar testes em larga escala. Simulações conseguem testar novos protocolos em larga escala a um custo baixo. (SILVA et al., 2018).

Geralmente, a qualidade das simulações depende da acurácia dos modelos subjacentes. Nos últimos anos vários modelos foram desenvolvidos representando vários aspectos de comunicação de redes veiculares. Isso aliado a multitude de simuladores disponíveis hoje, dificulta ainda mais a tarefa de avaliação das aplicações de forma comparativa e objetiva (BRUMMER; GERMAN; DJANATLIEV, 2018).

1.1 Objetivos

Esse trabalho tem como objetivo principal realizar uma análise comparativa entre a suíte *Veins* composto dos simuladores OMNet++ e SUMO e o simulador *Airplug*.

Como objetivos específicos temos: portar o algoritmo AND implementado no simulador proprietário *Airplug* para o simulador de código aberto *Veins* fornecendo assim uma fonte de comparação tangível entre os dois simuladores.

1.2 Estrutura do texto

O trabalho encontra-se organizado da seguinte forma: Neste capítulo foi apresentada a introdução. No capítulo 2 são apresentados os conceitos que serviram de base para esse trabalho. O capítulo 4 trata da comparação entre os simuladores *Airplug* e suíte *Veins*. O capítulo 5 apresenta os resultados da adaptação do algoritmo para a suíte *Veins* e compara esses resultados aos obtidos no *Airplug*. O capítulo 6 traz as conclusões, baseadas nas pesquisas feitas durante o trabalho. Por fim, o Apêndice A contém o código fonte do algoritmo desenvolvido.

2 REFERENCIAL TEÓRICO

Neste capítulo são abordados conhecimentos básicos que serviram de base para este trabalho.

2.1 Redes Veiculares

Redes veiculares, também chamadas de VANETs, são redes formadas por veículos automotores e equipamentos fixos às margens das ruas e estradas. Algumas das características de redes veiculares são: alta mobilidade dos nós, conexões intermitentes e dinamicidade da rede. Essas características tornam os algoritmos de redes sem fio padrão pouco efetivos. A principal motivação de pesquisa em redes veiculares é a segurança e comodidade para seus usuários (JAIN; SAXENA, 2017). Se os veículos de uma VANET se comportarem como nós em uma rede e podem interagir entre si. As informações sobre tráfego e estradas podem ser compartilhadas tornando dessa forma, o trânsito mais seguro.

2.1.1 Arquiteturas de Redes Veiculares

A arquitetura de uma rede define como seus nós se organizam. Em Redes Veiculares existem três arquiteturas principais: *ad hoc*, infraestruturada e híbrida.

Na arquitetura *ad hoc*, também chamada de V2V (*vehicle-to-vehicle*), os únicos componentes da rede são os veículos, que trocam informações entre si. Portanto os veículos funcionam como roteadores encaminhando o tráfego através dos saltos. Esse tipo de arquitetura apresenta desafios à conectividade devido à alta mobilidade dos nós e roteamento quando a mensagem precisa alcançar nós distantes.

Para atenuar esses problema a arquitetura infraestruturada, também conhecida como V2I (*vehicle-to-infraestructure*) emprega nós estáticos distribuídos ao longo das ruas ou estradas. Estes nós, conhecidos como RSU (*road side units*) centralizam o tráfego da rede podendo intermediar a comunicação entre os nós

móveis, ou comunicação com redes infraestruturada como a internet.A conectividade neste tipo de rede só é garantida mediante um grande número de RSU, o que pode elevar o custo deste tipo de rede.

A arquitetura híbrida, chamada também de V2X (*vehicle-to-anything*) pode ser vista como uma solução intermediária. Em redes híbridas, os nós se comunicam diretamente (V2V) em áreas onde não há RSUs alcançáveis. Quando estiverem em áreas cobertas por RSUs, os veículos passam a se comunicar através deste dispositivo centralizador (V2I). Essa abordagem permite aumento da conectividade sem aumentar excessivamente o custo de implantação (ALVES et al., 2009).

2.1.2 Padrões de Redes Veiculares

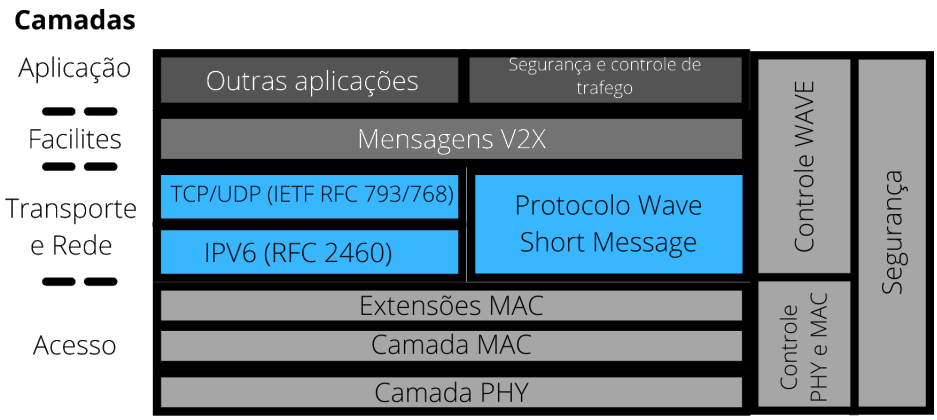
Nos estados unidos as primeiras tentativas de padronização de redes veiculares começaram em 1999 com a alocação de 75MHz do espectro de frequências na faixa 5,9GHz para aplicações DSRC (*Dedicated Short Range Communications*)

Em 2004 o IEEE (*Institute of Electrical and Electronics Engineers*) começou o processo de padronização das redes veiculares. Inicialmente como parte do grupo 802.11, o padrão ficou conhecido como IEEE 802.11p WAVE (*Wireless Access in the Vehicular Environment*). O padrão WAVE é definido em outros seis documentos: IEEE P1609.1, IEEE P1609.2, IEEE P1609.3, IEEE P1609.4, IEEE 802.11 e IEEE 802.11p.O padrão IEEE 802.11p define as camadas físicas e camada MAC com base no padrão IEEE 802.11a, que opera próxima a frequência alocada para as redes veiculares. Os padrões da família IEEE 1609 definem as camadas superiores do protocolo WAVE , incluindo questões de segurança para comunicação DSRC, uso de múltiplos canais e uma alternativa ao protocolo IP (VIVEK et al., 2017)

A família de padrões IEEE 1609 visa fornecer um padrão de interface e comunicação para ser usado no desenvolvimento de aplicativos V2V, V2I ou V2X, este padrão é importante para manter a interoperabilidade entre as aplica-

ções desenvolvidas pelos diversos fabricantes de automóveis ou desenvolvedores independentes. A figura 2.1 apresenta a pilha de protocolos do padrão WAVE.

Figura 2.1 – Pilha do protocolo WAVE

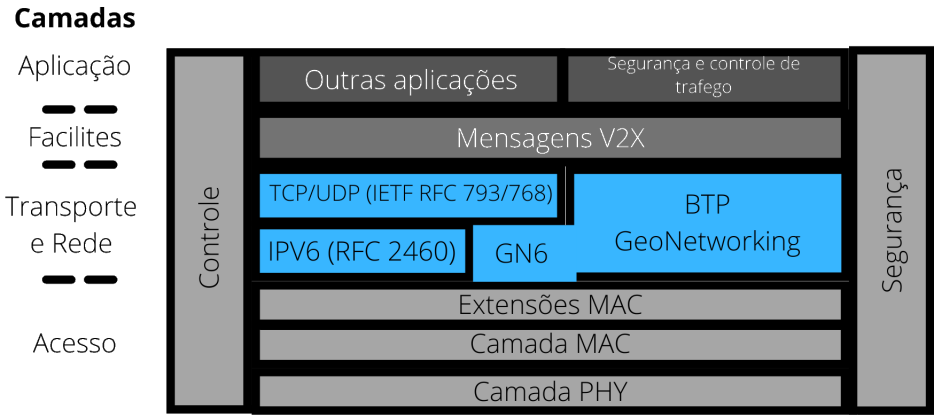


Na Europa os padrões de redes veiculares são responsabilidade do ETSI (*European Telecommunications Standards Institute*). Os primeiros esforços pela padronização datam da década de 90 com as especificações do TMC (*Traffic Message Channel*) e EFC (*Electronic Fee Collection*) desenvolvidos no contexto de RTTI (*Real-Time Traffic Information*). Avanços mais significativos foram feitos recentemente com a conclusão da primeira versão do C-ITS (*Cooperative Intelligent Transportation Systems*) em 2013 (FESTAG, 2015).

O padrão C-ITS mantém a mesma estrutura de camadas horizontais que o padrão WAVE nas camadas de acesso, rede, transporte, mensagens V2X, aplicação e entidades de gerenciamento e segurança. As aplicações C-ITS também não são padronizadas diretamente. São fornecidas os requisitos mínimos funcionais e de performance para três grupos de aplicações: *Road hazard signaling* (RHS - Sina-

lização de perigos na estrada) inclui casos de uso como aproximação emergencial de um veículo, local perigoso e luzes de freio emergencial eletrônicas. *Intersection collision risk warning* (ICRW - Alerta de risco de colisões em cruzamentos) e *Longitudinal collision risk warning* (LCRW - Alerta de risco de colisões longitudinais) referem-se a potenciais colisões em cruzamentos e colisões frontais/traseiras. A figura 2.2 apresenta a pilha de protocolos ITS.

Figura 2.2 – Pilha do protocolo C-ITS



2.2 Simuladores de Redes Veiculares

Avaliar protocolos e aplicações para Sistemas Inteligentes de Transporte é o primeiro passo antes de implantá-los no mundo real. Os componentes e testes de campos para este tipo de sistema tem um custo muito elevado, envolvendo recursos como veículos, motoristas e vias públicas dependendo dos testes a serem realizados. Para diminuir estes custos, simuladores foram desenvolvidos e/ou adaptados para estudos de soluções para Redes Veiculares.

Simulações resultam em avaliações escaláveis e de baixo custo. Porém, para produzir resultados confiáveis, os simuladores devem implementar um modelo mais próximo da realidade quanto possível.

2.2.1 Categorias de simuladores de Redes Veiculares

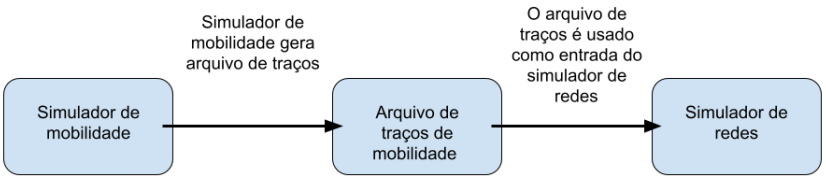
Simuladores de Redes Veiculares geralmente são compostos de: um simulador de redes e um simulador de trânsito. Portanto, o modelo de mobilidade é desenvolvido por engenheiros de tráfego e o modelo de comunicação por cientistas da computação.

Dessa forma, é possível classificar os simuladores de Redes Veiculares em três categorias distintas, de acordo com suas abordagens: Simuladores isolados ou *offline*, simuladores integrados e simuladores federados ou *online*. (ZEMOURI; MEHAR; SENOUCI, 2012)

Na primeira categoria, simuladores *offline*, a interação entre o simulador de rede e de trânsito é limitada ou não existente pois o modelo de mobilidade está preso a um traço de mobilidade estático (ZEMOURI; MEHAR; SENOUCI, 2012). Existem vários *data sets* de arquivos de traço disponíveis publicamente, porém estes geralmente são coletados de veículos específicos, como frotas de ônibus ou táxis. Estes veículos geralmente não representam o comportamento do tráfego como um todo. Além disso o uso destes conjuntos de dados garante apenas que a aplicação funciona bem na mesma classe de veículos do conjunto de dados. Exemplos de simuladores desta categoria são: *BonnMotion*, *MOVE* e *VanetMobiSim* (SILVA et al., 2019). A figura 2.3 demonstra o esquema da categoria dos simuladores *offline*.

Na abordagem integrada os simuladores de rede e trafego estão acoplados de forma nativa, formando um único simulador. Como exemplos de simuladores nesta categoria: *MoVES*, *NCTUns* e *VCOM*. Apesar de estarem integrados de forma nativa, estas soluções sofrem de recursos limitados, tanto do ponto de vista de um simulador de redes e também como de um simulador de trânsito (ZE-

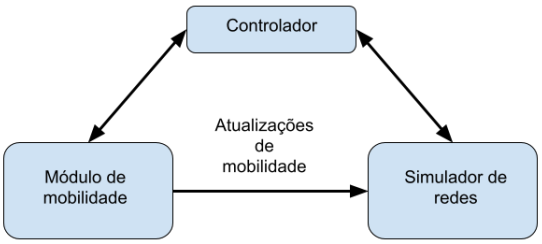
Figura 2.3 – Modelo de simulador offline



MOURI; MEHAR; SENOUCI, 2012). A figura 2.4 remete ao esquema utilizado na categoria de simuladores integrados.

A abordagem *online* foi criada para superar as limitações das abordagens anteriores. Nesta categoria, uma conexão bidirecional é estabelecida pelos simuladores. O simulador de redes controla o simulador de trânsito enviando comandos para modificar o comportamento dos nós. O simulador de trânsito responde com a posição do nó afetado. Esta abordagem é considerada, até o momento, a melhor solução para simulação de Redes Veiculares, devido ao alto nível de realismo (SILVA et al., 2019). Como exemplos dessa categoria temos: *TraNS* (*SUMO* e *ns-2*), *iTETRIS*, *Ovnis*(*SUMO* e *ns-3*) e *Veins* (*SUMO* e *OMNET++*). Apesar de apresentar os resultados mais realistas esta abordagem exige um alto custo computacional e temporal (ZEMOURI; MEHAR; SENOUCI, 2012). A figura 2.5 representa o modelo de comunicação presente nos simuladores online

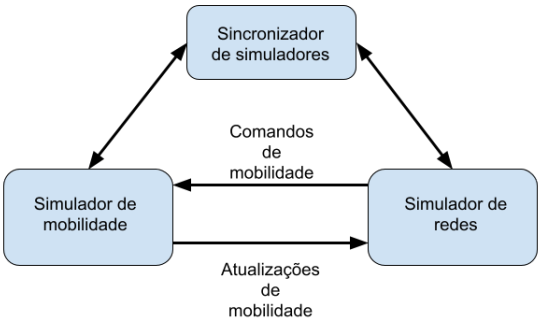
Figura 2.4 – Modelo de simulador integrado



2.2.2 OMNeT++

O *OMNeT++* (*Objective Modular Network Testbed in C++*) é um simulador de eventos discretos de código aberto, sendo usado na modelagem de comunicações de redes, sistemas paralelos e sistemas distribuídos. Disponível desde 1997 para a comunidade acadêmica, OMNeT++ foi projetado para suportar simulações de larga-escala, portanto, foi desenvolvido completamente modular. Essa característica permite que cada módulo desenvolvido possa ser implementado separadamente e depois incorporado formando uma pilha de protocolos como os utilizados no mundo real. O *framework* INET é um dos módulos que contem modelos de simulação de redes móveis, e sem fio (SILVA et al., 2018).

Figura 2.5 – Modelo de simulador online



2.2.3 Sumo

SUMO (*Simulation of Urban Mobility*) é um simulador de trânsito urbano de código aberto, desenvolvido pelo Centro Geoespacial Alemão (German Aerospace Center) tendo sua primeira versão lançada em 2002. SUMO ao longo dos anos cresceu de um simulador de trânsito para uma suíte completa de aplicações para preparação e execução de simulações de trânsito. É um simulador puramente microscópico, ou seja, cada veículo é definido de maneira explicita. Em 2006 SUMO foi estendido para aceitar conexões com aplicação externas através de *sockets* com a API TraCI, permitindo assim integração com outros simuladores. Entre as aplicações disponíveis no SUMO destacam-se: Comunicação veicular como modelos de comunicação V2X; Escolha de rotas e navegação dinâmica; Algoritmo para controle de semáforos; Avaliação de sistemas de vigilância de trafego; Modelos de emissão de poluentes e de ruido (BEHRISCH et al., 2011).

2.2.4 Veins

Veins (*Vehicles in Network Simulation*) é um *framework open source* para Redes Veiculares. Proposto inicialmente em 2008, começou a ganhar visibilidade pela comunidade acadêmica somente em 2011 (SILVA et al., 2019). *Veins* foi desenvolvido estendendo os simuladores *OMNeT++* e *SUMO* com módulos de comunicação dedicados, sendo assim um simulador *online*. *OMNeT++* é um simulador de rede baseado em eventos, portanto, lida com a mobilidade dos nós agendando seus movimentos em intervalos regulares. Isso se encaixa bem com a abordagem do *SUMO*, que também avança a simulação em etapas discretas (SOMMER; GERMAN; DRESSLER, 2010). Portanto, *Veins* oferece a robustez de dois simuladores de renome no meio acadêmico.

2.2.5 Airplug

O *framework Airplug*, desenvolvido e mantido pela *Université de Technologie de Compiègne*, lançado em 2013 como a *The Airplug Software Distribution*. Trata-se de um *framework* proprietário da faculdade, o que requer uma licença para utilizá-lo. Portanto, a análise comparativa realizada, levou em consideração apenas a bibliografia disponível sobre o *framework Airplug*. O foco do *Airplug* é ser um *framework* leve, portável e robusto. Para isso ele depende das funcionalidades oferecidas pelos sistemas operacionais padrão, como, alocação de recursos, escalonamento de processos, gerenciamento em tempo real, entre outros. O *Airplug* conta também com diversos "modos", cada um deles funcionando de forma independente e complementar.

2.3 Algoritmos para Redes Veiculares

Disseminação de informações em VANETs é um problema único. Ao contrário dos dados *unicast* normalmente transmitidos em redes como a internet, as informações de tráfego geral tem uma natureza orientada ao *broadcast*. Em

outras palavras, as informações de transito são de interesse publico e geralmente beneficiam um grupo de usuários. A principal vantagem de um esquema *broadcast* é que um veículo não precisa saber um endereço de destino e uma rota para um destino específico. Isso elimina a complexidade de descoberta de rota, resolução de endereço e gerenciamento de topologia. (PANICHPAPIBOON; PATTARA-ATIKOM, 2012).

Os algoritmos de disseminação de informações podem ser classificados em duas categorias: protocolos *broadcast* e protocolos *geocast* (ALLANI, 2018).

Protocolos de *broadcast* são usados frequentemente em VANETs para compartilhar dados de transito, meteorologia, entretenimento e anúncios. O objetivo é divulgar a informação para todos os veículos, sem exceção, através de mecanismos chamados de inundação (*flooding*). Essa abordagem pode aumentar a acessibilidade, informando todos os veículos interessados. No entanto, isso leva a problemas de congestionamento de canal, conflito e colisões (ALLANI, 2018). Protocolos de *broadcast* ainda podem ser subdivididos em *broadcast* de salto único (*single-hop*) e de múltiplos saltos (*multi-hop*).

Protocolos de *geocast* visam disseminar informações apenas para veículos dentro de uma área geográfica especifica chamada de ZOR (Zona de relevância) (BAKO; WEBER, 2011). Após a ZOR ter sido determinada, técnicas de *broadcast* sofisticadas são usadas para disseminar a mensagem dentro da ZOR. Os veículos que recebem a mensagem fora da área especificada simplesmente ignoram a mensagem (ALLANI, 2018).

Como exemplo de protocolos *broadcast multi-hop* temos o *Weighted p-Persistence* (p-Persistencia com peso, tradução livre), nesse protocolo, quando um veículo recebe um pacote ele primeiro calcula a probabilidade de retransmissão baseado na distancia entre ele e o transmissor. Assim sendo, veículos que estão mais longe do transmissor terão uma probabilidade maior de retransmissão. O calculo da probabilidade não leva em conta a densidade de veículos em consideração,

portanto, o numero de retransmissões ainda pode ser grande se a rede for densa (PANICHPAPIBOON; PATTARA-ATIKOM, 2012).

O protocolo *Dynamic Time-Stable Geocast Protocol* (DSTG) é um protocolo *geocast* com foco em garantir a entrega da mensagem com um custo baixo, isso é feito, atualizando dinamicamente o tempo de espera estável. Esse protocolo inclui duas fases: um período pré-estável e um período estável. A primeira fase consiste em fazer o *broadcast* da mensagem na região especifica. Sempre que um veículo detecta um evento crítico, ele transmite e continua a retransmitir esse evento. Esse processo termina quando o transmissor recebe a mesma mensagem de outro veículo. O veículo mais distante recebe a missão de retransmissão com base em um período de tempo estável. O objetivo desse período estável é manter viva a mensagem dentro da região de *geocast* sempre que ainda for pertinente (ALLANI, 2018) mantê-la.

2.3.1 CAS - *Cooperative Awareness Service*

CAS (*Cooperative Awareness Services*) é um algoritmo descrito no documento: EN 302 637-2, mantido pela ETSI. As mensagens CAM (*Cooperative Awareness Messages*) são mensagens usadas na rede ITS entre estações ITS, chamadas de ITS-S (Intelligent Transport System Station). As mensagens são gerenciadas pelo serviço CA (*Cooperative Awareness*), que é uma entidade da camada *Facilities* do protocolo ITS. O serviço CA é responsável por codificar e decodificar as mensagens CAM, gerenciar a transmissão das mensagens, isto inclui: Ativação do serviço de transmissão de mensagens, controle da frequência das mensagens e criação das mensagens. O gerenciamento da recepção das mensagens, consiste em: Invocar o método de decodificação de mensagens quando uma mensagem é recebida, Prover as informações recebidas pelas mensagens CAM para as outras camadas do protocolo.

Uma estação ITS pode ser qualquer tipo de veículo, pedestres e até equipamentos de infraestrutura da rodovia como placas, semáforos, barreiras ou portões. Uma mensagem CAM contêm informações de status e atributos da estação de origem. O conteúdo varia dependendo do tipo de estação que está enviando a mensagem. Para veículos, as informações de status incluem tempo, posição, estado de movimento, sistemas ativados, entre outros. As informações de atributos por sua vez incluem: Dimensões do veículo, tipo do veículo, função no trafego (se este é um veiculo de auxilio, veiculo pessoal, etc), dentre outros. As mensagens CAM são transmitidas somente para as estações ITS dentro da faixa de comunicação, e em um único salto, um CAM recebido não é repassado a outras estações ITS.

A frequência de geração de mensagens CAM é gerenciada pelo serviço CA, ele define o intervalo de tempo entre duas gerações de mensagens consecutivas.

Dentro destes limites a geração de uma mensagem CAM deve ser acionada dependendo da dinâmica da estação ITS e do nível de congestionamento do canal. Caso a dinâmica da estação ITS resultar em uma diminuição do intervalo de geração, esse intervalo deverá ser mantido por um numero de CAMs consecutivos.

2.3.2 AND - *Adaptative Neighbor Discovery*

AND (*Adaptative Neighbor Discovery*) é um algoritmo orientado a eventos. Desenvolvido pelo Professor Dr. Hermes Pimenta de Moraes Júnior em sua tese de doutorado. Foi desenvolvido com uma estratégia cooperativa em mente. Levando em consideração as normas e padrões definidos pelo ETSI (MORAES, 2018).

A cada recepção de mensagem o nó receptor salva os dados do remetente (número de sequência, estimativa de confiabilidade e vizinhos) calcula a velocidade relativa e a distância entre eles e verifica se há problemas na vizinhança. Os valores de velocidade relativa e distancia são usados para calcular a estimativa de

tempo de vida da vizinhança. Problemas com a vizinhança estão relacionados à supostos vizinhos desconhecidos. Considerando o nó **V** como o nó receptor e o nó U como remente, **V** verifica a lista de vizinhos de **U** (recebida pela mensagem). Se existe um nó dentro da faixa central **R**/2 (**R** representando a distância máxima de comunicação estipulada pelo algoritmo) de **U**, que não é vizinho de **V**, as mensagens estão sendo perdidas (v deveria ter reconhecido esse nó). Esse valor é usado posteriormente para complementar o calculo da estimativa de confiabilidade de rede (MORAES, 2018).

O algoritmo apresenta um comportamento periódico guiado por uma constante chamada `aTimer`. Esse temporizador representa o menor intervalo de tempo permitido no algoritmo (100 milissegundos). A cada fim do temporizador, AND exclui todos os dados relacionados a vizinhos antigos de acordo com seus tempos de vida. Dessa forma o nó mantêm uma vizinhança atualizada. Em seguida, se alguma mensagem foi recebida desde o último fim do temporizador, os parâmetros de rede são atualizados. AND utiliza uma abordagem cooperativa de alto nível para lidar com essa tarefa. Permitindo estimar a taxa de perdas, e, dessa forma, o número de tentativas necessárias para entregar uma mensagem (definido pela variável `p`). Então o valor do temporizador `timeToSend` é diminuído em um `aTimer`, uma mensagem é enviada se o temporizador `timeToSend` chegar a zero. Como ultimo passo, o *inter-messages delay* (IMD) é atualizado e, em caso de alteração, o temporizador `timeToSend` é adaptado de acordo.

Para estimar a confiabilidade de rede, cada nó mantém contadores e números de sequencia para cada mensagem recebida. Esses valores são usados para calcular a quantidade de mensagens recebidas e perdidas, que, por sua vez, permitem o calculo de uma taxa de perda local. A estimativa local é enviada para os nós vizinhos, proporcionando que cada vizinho calcule uma estimativa para a vizinhança geral. A estimativa geral é obtida por meio das somas das estimativas locais de cada nó, dividida pelo numero de vizinhos de cada nó.

O algoritmo AND sempre tenta enviar mensagens com o menor IMD permitido, mantendo este comportamento até que seja detectada uma perda de pacotes. Quando isso acontece a frequência de envio é ajustada de acordo com uma abordagem de Aumento aditivo / diminuição multiplicativa (AIMD). Cada vez que as condições da rede parecem ser melhores, ou seja, a taxa de perda diminui, o IMD também diminui em um fator aditivo. Cada vez que as condições de rede parecem piores, ou seja, se a taxa de perda aumenta, o IMD aumenta em um fator multiplicativo. Por ultimo o valor do IMD é limitado pelas bordas inferior e superior.

Como o Airplug é um simulador proprietário, não podemos utiliza-lo diretamente nessa comparação. No entanto, como já possuímos resultados publicados com o uso do Airplug (MORAES, 2018), implementamos o algoritmo AND no simulador Veins para então comparar os resultados e diferenças entre os simuladores. No próximo capítulo algumas características dos simuladores são apresentadas e, em seguida, é feita uma análise comparativa entre eles.

3 METODOLOGIA

A metodologia utilizada nesse trabalho foi a de pesquisa descritiva e exploratória. Uma vez que, foi necessário um aprofundamento no tópico de redes veiculares e a familiarização com o simulador escolhido (a suíte *Veins*). A coleta de dados foi feita através de pesquisa bibliográfica com abordagem qualitativa com foco nos dois simuladores a serem comparados.

Os procedimentos realizados ao longo da pesquisa foram: Pesquisa bibliográfica com o objetivo de aprofundar o conhecimento sobre os conceitos pertinentes a esse trabalho. Após isso foi realizada uma pesquisa em laboratório, onde foi desenvolvido o algoritmo AND no simulador *Veins*. A última etapa do trabalho consistiu em uma análise comparativa dos dois simuladores, de forma a evidenciar suas diferenças e similaridades.

A pesquisa bibliográfica contemplou tanto os conceitos básicos presentes no referencial teórico, quanto uma pesquisa mais aprofundada sobre os dois simuladores contemplados. Para a pesquisa do referencial teórico foram consultados os documentos referentes ao protocolo WAVE e ao CAS; artigos sobre história de redes veiculares; tipos de redes; o estado da arte referente a simulação e algoritmos existentes. A pesquisa sobre o *Airplug* e *Veins* contemplou os artigos publicados sobre os simuladores.

A pesquisa em laboratório se caracteriza por realizar experiências limitadas em um ambiente fechado e sujeito a manipulações. Essas experiências requerem instrumentos próprios, no caso deste trabalho, foi utilizado o simulador *Veins* e um computador. O objetivo dessa pesquisa foi portar o algoritmo AND que foi desenvolvido no simulador proprietário *Airplug* para o simulador aberto *Veins*, para então analisar o desempenho das duas versões do algoritmo e apontar possíveis diferenças nos simuladores a partir da análise. A pesquisa foi conduzida através de consulta o pseudocódigo do algoritmo AND descrito em (MORAES, 2018) e analisando o código fonte original, levando em consideração as estruturas

de dados utilizadas no código para manter a adaptação mais próxima o possível. Depois de portado o algoritmo, testes foram realizados em um cenário semelhante ao utilizado no *Airplug*. Dados sobre os testes foram coletados e transformados em gráficos para melhor visualização.

Na análise comparativa, os artigos que descrevem o funcionamento dos simuladores e manuais foram estudados. Esses serviram como fonte principal da comparação, enquanto, os resultados dos testes do algoritmo serviram como fonte adicional. Importante ressaltar, que o *Airplug* é um simulador proprietário, portanto, o manual e informações mais detalhadas não estão disponíveis para livre acesso. A análise focou em aspectos estruturais como a linguagem empregada, capacidade de paralelismo, esquema de endereçamento, entre outros. Sendo que, são essas características as que mais influenciam os resultados das simulações.

No próximo capítulo são apresentados os dois simuladores e algumas de suas características.

4 COMPARAÇÂO ENTRE OS SIMULADORES

Nesse capitulo é feita a comparação entre os simuladores. Primeiro é apresentado algumas características dos simuladores e particularidades de seus funcionamentos. Após isso a comparação é feita apresentando a adaptação do algoritmo AND e evidenciando as diferenças.

4.1 Veins

Veins [1] teve sua primeira versão publica em meados de 2006, lançado como uma extensão para o *framework* INET. Por causa de limitações na fidelidade do modelo de canais sem fio da época, para sua versão 1.0, *Veins* foi portado para ser uma extensão do MiXiM (Uma biblioteca, agora descontinuada, de modelos de canais sem fio para o simulador *OMNeT++*). *Veins* foi ampliado ao longo do tempo incorporando modelos novos como, por exemplo, IEEE 802.11p, IEEE 1609.4, e WAVE, que mais tarde foram refatorados até a camada física para a versão 2.0. Conforme mais refatorações e reescritas aconteciam nos modelos de canais, *Veins* 3.0 se tornou um *fork* do MiXiM, porém, ainda foi mantido a compatibilidade com simulações mistas incorporando modelos do *framework* INET (SOMMER et al., 2019).

4.1.1 Arquitetura

A suíte *Veins* não inclui modelos customizados de veículos rodoviários. Em vez disso, ele faz com que as simulações estabeleçam conexão com o SUMO que é executado como um processo separado. Dessa forma, é possível se beneficiar de anos de pesquisa e desenvolvimento por especialistas no domínio de simulação de tráfego rodoviário. O simulador de tráfego rodoviário com o qual Veins foi projetado para interoperar é o *SUMO* (SOMMER et al., 2019).

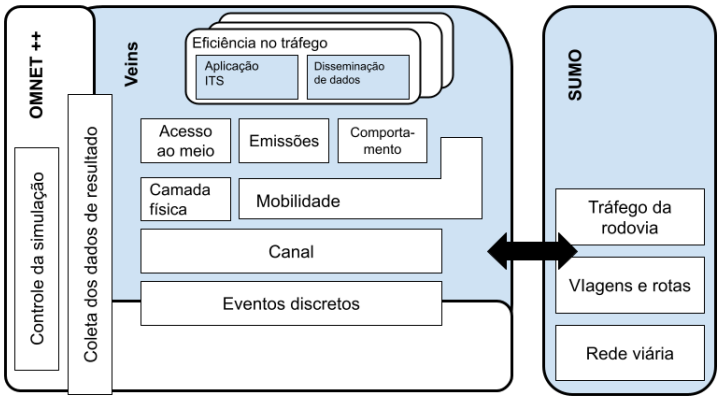
[1] <https://veins.car2x.org/>

SUMO pode simular redes viárias de médio e grande porte de cidades, áreas urbanas e rodovias. Neles, pode-se simular o movimento de veículos como carros e caminhões, motos e bicicletas, de pedestres e trens. SUMO oferece suporte a uma ampla gama de modelos de mobilidade diferentes , um conjunto de controladores diferentes para cada interseção (de vias preferencias, a semáforos acionados por demanda), e uma ampla gama de formatos de entrada para a rede de estradas (como *OpenStreetMap*, *TIGER*, *Geographic Information System*).

A cada intervalo de tempo *OMNeT++* envia todos os comandos do *buffer* para o SUMO e aciona o intervalo de tempo correspondente no simulador de trânsito. Após a conclusão do intervalo de tempo do *SUMO*, esse envia uma série de comandos e posições de todos os veículos instanciados de volta ao *OMNeT++*. Isso permite que o *OMNeT++* reaja ao traço de mobilidade recebido introduzindo novos nós, deletando nós que alcançaram seu destino e movendo nós de acordo com o simulador de trânsito. Depois de processar todos os comandos recebidos e mover todos os nós de acordo com as informações de mobilidade, o *OMNeT++* irá então avançar a simulação até próximo intervalo de tempo, permitindo que os nós reajam as alterações nas condições do ambiente, como, por exemplo, influenciando nas suas rotas ou velocidades (SOMMER; GERMAN; DRESSLER, 2010). A figura 4.1 apresenta uma visão alto nível da arquitetura do *framework Veins*.

Usando um simples protocolo de pergunta / resposta, o trafego rodoviário no SUMO pode ser influenciado pelo *OMNeT++* de várias maneiras. O mais importante, intervalos de tempo são gerados para avançar a simulação no *SUMO*. Além disso, veículos podem ser parados para criar congestionamentos artificiais, podem ser retomados para resolver esses congestionamentos e cada veículo simulado pode ser reencaminhado individualmente em torno de segmentos de estrada arbitrários. Dessa forma, *Veins* reflete a maneira como motoristas que sabem sobre uma obstrução no trânsito tentarão evitá-la (SOMMER; GERMAN; DRESSLER, 2010).

Figura 4.1 – Visão alto nivel da arquitetura Veins



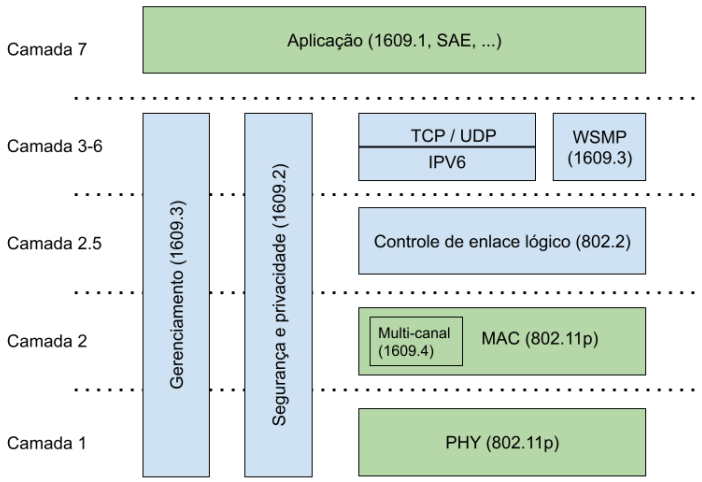
4.1.2 Camadas inferiores

Um dos recursos principais do *Veins* é a modelagem detalhada das camadas inferiores dos protocolos de Comunicação Inter-Veicular. A tecnologia frequentemente usada nesses protocolos é o IEEE WAVE. Embora seja possível implementar e integrar as camadas padrões, *Veins* coloca foco nas camadas inferiores, pois estas, são decisivas para o acesso real aos canais de transmissão de pacotes.

Cada nó, seja este um veiculo, RSU, ou até mesmo um pedestre ou ciclista, faz uso de comunicação sem fio na simulação, consistindo em, no mínimo, uma placa de interface de rede (NIC) 802.11p para se comunicar com os outros dispositivos. As camadas superiores são conectadas diretamente ao NIC, que por sua vez, é um modelo composto consistindo nas camadas MAC e PHY. Isso resulta em uma arquitetura simples *aplicação-mac-phy* para cada nó.

No *OMNeT++* cada módulo pode trocar mensagens com outros se estiverem conectados. Estas mensagens podem ser de diferentes tipos: podendo ser mensagens simples encapsuladas, mensagens WAVE padrão (WSM) ou mensagens WAVE de anuncio (WSA). Dentro de cada nó as mensagens podem ser mensagens com informações definidas pelos usuários que são encaminhadas para as camadas abaixo ou acima, ou mensagens de controle para disparar uma determinada ação na camada receptora, dependendo do tipo, uma função diferente será chamada. A figura 4.2 demonstra as camadas do protocolo WAVE implementadas no *framework Veins*.

Figura 4.2 – Pilha do protocolo WAVE no *framework Veins*



4.1.3 Camada MAC

O comportamento da camada MAC pode ser especificado na forma de uma máquina de estados, porém essa estratégia é muito custosa porque pacotes de camadas superiores podem chegar independente do estado atual da camada e há

vários temporizadores executando em paralelo. Isso levou a equipe do *Veins* a seguir uma abordagem diferente: Apenas um contador principal é utilizado, quando um evento ocorre que afeta a camada, o contador é remarcado e as filas correspondentes são atualizadas. Quando o canal fica ocioso a operação se resume e os contadores adicionais são ajustados. Portanto, *Veins* apresenta um canal único com apenas mensagens de transmissão, uma configuração bastante comum em redes veiculares (SOMMER et al., 2019).

Para transmitir um pacote, a camada MAC espera uma mensagem do tipo WAVE das camadas superiores, com informações sobre qual canal deve ser utilizado e uma prioridade definida pelo usuário, essa prioridade será mapeada para uma fila. O pacote entrará na fila de acordo com o contador e esse será atualizado caso for necessário. Caso o canal estiver ocupado e contador estiver expirado com o pacote na frente da fila, então um procedimento de *backoff* é invocado de acordo com o padrão.

Na recepção de pacotes a camada MAC tem um papel mais simples. Se a camada PHY enviar um pacote, o MAC verificará se o endereço de destino é o endereço de *broadcast* da camada, ou se o endereço corresponde ao seu próprio. Nesse caso, o pacote será desencapsulado e a mensagem sera entregue a camada de aplicativo. Ao lidar com transmissões *unicast*, o pacote recebido pode ser uma confirmação de pacote recebido (ACK). A recepção e retransmissão de ACKs funciona da mesma forma que em redes cabeadas. A suíte*Veins* utiliza várias mensagens de controle e por meio dessas consegue estatísticas como: pacotes recebidos e falhas, podendo ser separadas por causas da falha e por tipo de envio (*broadcast*, *unicast*), e informações sobre o estado interno dos nós (SOMMER et al., 2019).

4.1.4 Camada Física

O beneficio de simulações em nível de pacotes é a capacidade de, na medida do possível, determinar de forma realista para cada pacote se este pode ser recebido com sucesso. Vários fatores afetam a decodificação de um pacote: posição do emissor e receptor, características das antenas, obstáculos bloqueando a linha de visão, interferência entre os outros nós. Embora o CSMA/CA reduza significativamente a chance de dois nós próximos enviarem mensagens ao mesmo tempo, ele não oferece uma solução para o problema do terminal oculto. Todos esses efeitos podem ser capturados no *Veins*.

Os modelos da Camada Física e canais sem fio do *Veins* considera sinais de rádio como objetos n-dimensionais genéricos, por exemplo, níveis de potencia expressos em tempo e frequência, e fornece um conjunto de funções matemáticas para trabalhar com estes objetos. Embora esta seja a forma mais flexível de modelar sinais de radio é também a forma mais custosa computacionalmente. Assim sendo, é utilizada uma abstração mais especifica de sinais de rádio, adaptados para comunicação V2V padrão. Apesar da desvantagem de não ser capaz de modelar sinais de rádio em dimensões diferentes de tempo e espaço, essas adaptações permitem que as simulações sejam executadas em até duas ordens de magnitude mais rápidas (SOMMER et al., 2019).

4.1.5 Modelos de interferência

Os modelos de perda incluem: *Free Space path loss* (FSPL) e o modelo de interferência de dois raios, que considera o sinal refletido na estrada podendo causar o cancelamento e amplificação do sinal recebido. Para representar o efeito de *fading* Veins utiliza o modelo de *Nakagami-m*, que é um método probabilístico que reflete a propagação de múltiplos caminhos em ambientes urbanos. O efeito de obstáculos também é contabilizado usando o modelo de perda. Assumindo que cada obstaculo é um polígono, então a energia de recepção é reduzida com base

nos **n** números de bordas em que o sinal está cruzando e a distancia **m** percorrida dentro dos polígonos. Esses valores são ponderados usando os parâmetros calibrados usando medições reais. Podendo ser alterados de acordo com o material, por exemplo, tijolos, concreto, madeira, entre outros.

Depois que todos os modelos de perda foram aplicados, o pacote é entregue a uma classe externa que determina se os pacotes podem ser decodificados com sucesso. Se a energia recebida estiver abaixo da sensibilidade configurável, o pacote não conseguirá definir o canal como ocupado e com isso a camada MAC não será notificada. Se o pacote estiver acima do limite, a função verifica se o nó já esta transmitindo ou recebendo outro pacote. Em ambos os casos, o pacote não será decodificado.

Veins também conta com um modelo de erro de bit para decodificação dos pacotes. Dependendo do esquema de modulação, uma equação diferente é aplicada para calcular a probabilidade de um bit ser decodificado incorretamente. São calculadas duas taxas de erro, uma para a carga do pacote e uma para seu cabeçalho. Essas taxas são combinadas e aplicadas ao comprimento do pacote para se obter a taxa de erro do pacote. Caso o pacote foi decodificado com sucesso ele é encaminhado para a camada MAC ou, em caso de erro, é enviada uma mensagem de controle.

4.1.6 Padrões de Antenas

Antenas são cruciais para a comunicação sem fio, porque, constituem a interface entre o dispositivo de rádio e o meio de transmissão. No entanto, apesar da infinidade de modelos detalhados para as camadas MAC e PHY, o impacto dos padrões de antena não foi levado em consideração em simulações VANET por um longo tempo, embora, o ganho ou perda de uma antena influencie criticamente o poder de recepção e, consequentemente, a decodificação de uma mensagem enviada.

A potência dos sinais enviados e recebidos depende de vários aspectos, o primeiro sendo, o tipo da antena utilizada. Podendo ser: Antenas isotrópicas, omnidirecionais e direcionais. além do mais, no contexto de redes veiculares, os veículos também influenciam as características de radiação das antenas. Nesse aspecto destaca-se o local de montagem da antena e as propriedades dos materiais das peças ao redor da antena.

Veins utiliza uma superclasse para antenas, com essa estrategia é possível adicionar uma antena para cada nó da simulação, independente de seu tipo. Isso auxilia também na modelagem especifica dos ganhos de antenas, uma vez que, os modelos base do podem ser expandidos. O simulador conta com implementações de antenas em 2D (considerando apenas o plano horizontal) e 3D (considerando também a elevação).

Para o modelo 2D o angulo de incidência do sinal é calculado, como todas as variáveis para esse calculo são conhecidas pela simulação, um simples calculo do produto escalar é executado. Para o modelo 3D além do angulo de incidência do sinal, é necessário também, a posição 3D da antena. Para isso, é feita uma interpolação de quatro valores de ganho mais próximos aos planos comparados e uma soma ponderada sobre esses valores. O uso do padrão 3D só faz sentido em simulações 3D, para isso o cenário da simulação deve ser modelado tridimensionalmente e tanto *OMNeT++* quanto o *SUMO* devem estar configurados para suportar esses dados.

4.2 Airplug

O *framework* Airplug[2] foi desenvolvido pela *Université de Technologie de Compiègne*, lançado em 2013 como a *The Airplug Software Distribution*. O *framework* conta com diversas aplicações, chamadas de modos, esses modos oferecem funcionalidades que podem ser executadas de maneira independente e tam-

[2] <https://airplug.hds.utc.fr>

bém de maneira complementar. É distribuído como uma licença gratuita para fins educativos e comercial. A licença gratuita contem o "esqueleto", que contem apenas exemplos de aplicações. Para acessar todas as funcionalidades e modos, é necessário entrar em contato com a universidade para se adquirir a licença comercial.

4.2.1 Modos

O *Airplug* apresenta seis implementações distintas chamadas de modos. São elas: *Airplug-term*, O modo padrão do *Airplug*, executa em um *desktop* Linux porque depende das funcionalidades do terminal POSIX; *Airplug-emu*, modo de emulação, usado para reproduzir redes dinâmicas; *Airplug-ns*, esse modo adapta parte do *Airplug* para executar em conjunto com o simulador de rede *ns-2*, permitindo assim alimentar ambas as simulações (*Airplug* e *ns-2*) com dados de maneira conectada. *Airplug-live*, esse modo é utilizado para fazer testes de campo, assim, obtendo medições reais; *Airplug-rmt*, modo remoto, para execução em rede, podendo dividir um nó entre vários computadores; *Airplug-notk*, para execução em computadores que não tem ambiente gráfico, ideal para execução em sistemas embarcados. Cada um desses modos são compatíveis e complementares: uma aplicação pode ser usada em qualquer um desses modos. Oferecendo assim diferentes perspectivas da simulação sendo feita. Uma breve descrição dos modos é feita a seguir.

Airplug-term: Implementação do *framework Airplug* para terminais do UNIX. É bem adaptado para prototipagem rápida e propõe vários recursos por meio de suas bibliotecas, apresentando um formato de mensagem extensível de três tipos: O formato *what* onde só o conteúdo das mensagens é enviado, podendo ser usado, como por exemplo, para prototipação de uma aplicação distribuída. Como há uma única aplicação por nó, as mensagens podem conter somente o conteúdo: cada processo iniciado pelo terminal representa um nó. No formato

whatwho o conteúdo e identificadores de envio e recebimento são enviados. Útil quando é necessário desenvolver uma aplicação distribuída que precisa interagir com outra. No formato *whatwhowhere* as mensagens incluem também o campo de *host* e representa o formato completo da mensagem. Utilizado quando uma aplicação distribuída depende do resultado de outra(DUCOURTHIAL, 2013).

Airplug-emu: É um simulador de rede, as camadas superiores são idênticas aos experimentos reais, enquanto, as camadas inferiores (comunicações *wireless*) são reproduzidas artificialmente utilizando recursos do *shell*. A descrição do cenário é feita em um arquivo XML: Aplicações executando em cada nó, movimento de cada nó, entre outras informações. Ao analisar a posição dos nós, *Airplug-emu* modifica as conexões entre os processos. É possível modificar dinamicamente o alcance e a confiabilidade da comunicação, a taxa de perda ou o atraso. Também podendo baixar mapas (*OpenStreetMap*) para os nós móveis ao longo das estradas. Vários padrões de mobilidade podem ser usados, como, por exemplo, arquivos de registro de GPS obtidos durante testes reais ou arquivos de traço *ns-2*. Os movimentos podem ser acelerados e desacelerados dinamicamente e também podem ser emulados nós imoveis. *Airplug-emu* pode também ser estendido para ser usado em vários computadores com o modo *Airplug-rmt*. Permitindo a execução remota das aplicações conectadas através de uma aplicação especifica chamada de *RMT* que retransmite as mensagens entre os computadores (DUCOURTHIAL, 2013).

Airplug-live: Esse modo foi criado para uso real. É composto de um programa central escrito em C, que gerencia a comunicação local e inter-nó. Este programa permite reutilizar as aplicações prototipadas usando o *Airplug-term* e estudados com o *Airplug-emu* sem qualquer modificação. O programa atua como um *middleware* entre os aplicativos e as interfaces de rede, enquanto é executado no modo usuário de um sistema Linux. O programa central se inicia e todos os aplicativos locais como processos independentes. Para cada um deles, a E/S padrão são redirecionados de e para o programa central. Então, cada vez que uma aplica-

ção local escreve na saída padrão, o programa central do *Airplug* recebe os dados reciprocamente. O programa central também examina os links das aplicações locais bem como as interfaces de rede e encaminha as mensagens para destinatário apropriado, seguindo o esquema de endereçamento do *Airplug* (DUCOURTHIAL, 2013).

4.2.2 Arquitetura

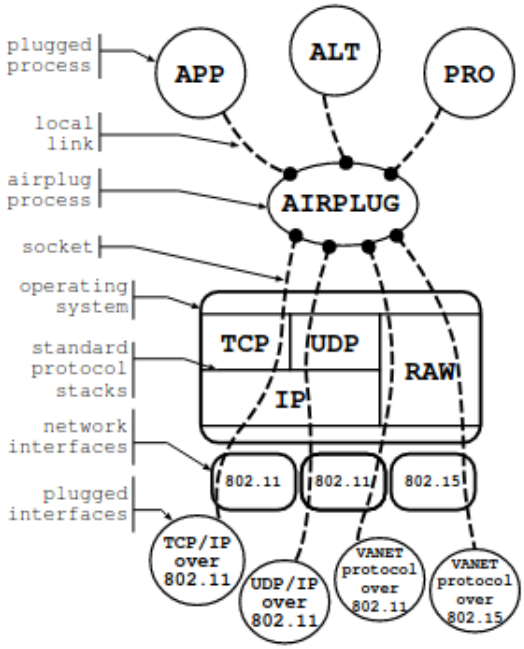
O foco do *framework Airplug* é ser um *framework* leve, portável e robusto. Para isso ele depende das funcionalidades oferecidas pelo sistema operacional, como, alocação de recursos, escalonamento de processos, gerenciamento em tempo real, entre outros. Dessa forma, não há redundância entre o *framework* e o sistema operacional, e, permite tirar proveito de qualquer melhoria feita nessas estruturas.

A arquitetura foi projetada para permanecer aberta para soluções futuras. Consequentemente, ela impõe muito poucas convenções as suas aplicações, que podem ser desenvolvidas em qualquer linguagem de programação. Não precisando incluir operações de rede nem comandos complexos de comunicação interprocesso. Assim sendo, as aplicações não dependem do *framework* e podem ser utilizadas de forma autônoma. Permitindo que as aplicações se beneficiem de qualquer melhoria futura tanto nos paradigmas de programação, ou protocolos de rede ou até, no próprio *framework*. O *framework* é concentrado em um único programa central, chamado de *Airplug*, presente em cada nó móvel, executado como um processo padrão sob o sistema operacional. Não incluindo nenhuma parte do *framework* no *kernel* nem nas aplicações, a independência com o sistema operacional bem como a programação das aplicações é mantida (DUCOURTHIAL, 2007).

Airplug aceita aplicações locais ou distribuídas. Uma aplicação local não interage com aplicações remotas. As aplicações locais são executadas em processos separados com seu próprio espaço de memória. Isso permite delegar o

escalonamento e o gerenciamento para o sistema operacional. O *framework* deve apenas definir as prioridades em relação aos parâmetros e o sistema operacional se encarrega do resto. Todos os processos são iniciados pelo *Airplug*, os processos filhos são criados utilizando a função `fork()` do Linux e são chamados de processos *plugged*, desse modo, o *Airplug* é informado sobre toda a comunicação entre as aplicações, capturando os sinais relacionados enviados pelo sistema operacional para o processo pai. A figura 4.3 apresenta uma visão geral de um processo *Airplug*.

Figura 4.3 – Visão geral de um processo *Airplug*



Fonte: (DUCOURTHIAL, 2007)

4.2.3 Comunicação Inter-processo

A comunicação no *Airplug* é baseada na passagem de mensagens assíncronas. Isso é feito através da E/S padrão. O motivo dessa escolha se da porque todos os processos possuem E/S padrão. Sendo assim, essa funcionalidade é então

suportada por qualquer linguagem de programação e não necessita de nenhum requisito adicional para a programação da aplicação. A E/S padrão é suficiente para realizar as comunicações locais inter-processo e pode ser facilmente estendida para garantir comunicações inter-processo remotas. Esse esquema de comunicação permite o uso isolado das aplicações (sem *Airplug*) e também permite reutilização das aplicações existentes (DUCOURTHIAL, 2007).

Para cada processo *plugged* a entrada e saída padrão são redirecionadas de e para o *Airplug* através de links locais. Há um link para cada processo conectado. Desse modo, cada vez que um processo *plugged* escreve em sua saída padrão, o *Airplug* receberá os dados por meio do link correspondente. Analogamente, cada vez que o *Airplug* escreve em um link, o processo conectado ao link receberá os dados. Portanto, o *Airplug* funciona quase como um barramento conectando todos os processos *plugged*. As interfaces de rede também são conectadas a esse barramento, permitindo que qualquer aplicação nas camadas mais altas de uma instância *Airplug* sejam conectadas a um barramento comum. Esse barramento é simples, e pode então, ser implementado de forma eficiente, evitando qualquer tipo de sincronização de processos indesejados. Fornecendo comunicação rápida, com poucas convenções comuns e sem gerenciamento global tornando o próprio para o ambiente de redes veiculares (DUCOURTHIAL, 2007).

4.2.4 Independência de linguagem

No *Airplug* todos os elementos relacionados a comunicação são implementados no próprio *Airplug* e as aplicações precisam apenas ler e gravar em suas E/S. A única restrição existente é para programas com mais de uma entrada padrão. Esses programas devem ser capazes de ler todas suas entradas, isso pode ser feito, por exemplo, por leitura assíncrona.

Graças a essa flexibilidade quanto ao uso de linguagens o paradigma de programação mais adaptado pode ser escolhido para o desenvolvimento de uma

aplicação *Airplug*. Assim como critérios que dependem do programador, como, capacidade de reutilização do código, compatibilidade com outras plataformas e ambientes. Pequenas adaptações são necessárias no código desenvolvido para formatar a saída no padrão do *Airplug*. Essa solução pode ser implementada como um programa independente inserido entre a aplicação e o *Airplug*. As aplicações podem então ser implantadas usando ferramentas de gerenciamento de pacotes de sistemas operacionais.

4.2.5 Integração de Rede

Uma estrutura em camadas com o *framework* sob o sistema operacional é vantajoso pois evita qualquer redundância. A estrutura ideal para a simulação de redes veiculares ainda não é conhecido. Levando isso em consideração o *Airplug* disponibiliza uma arquitetura versátil em vez de uma estrutura estrita em camadas entre a rede e as aplicações, porque, assim é possível focar nos aspectos relevantes.

Na arquitetura *Airplug*, as interfaces de rede são acessadas por meio do programa *Airplug*, sendo chamadas de interfaces *plugged*. As interfaces *plugged* são gerenciadas da mesma forma que os processos *plugged*, com exceção de que são conectadas ao *Airplug* por meio de *sockets*. Assim sendo, a rede é endereçada pelas aplicações da mesma forma que eles endereçam uma mensagem para outra aplicação, simplesmente escrevendo em sua saída padrão. O programa *Airplug* recebe os dados enviados para a interface desejada através do *socket* relacionado (BUISSET et al., 2010).

4.2.6 Endereçamento e primitivas de comunicação

Como o ambiente de redes veiculares apresenta características altamente instáveis, o *Airplug* implementa um esquema de endereçamento bastante simples. Uma mensagem pode ser enviada a uma aplicação local, utilizando o identificador LCH (*localhost*); Para outros nós próximos pela interface sem fio, com o iden-

tificador AIR, ou, para ambos, com o identificador ALL. Um quarto modo pode ser usado quando um serviço de descoberta permite detectar os nomes dos nós vizinhos. Para isso um nome do vizinho deve ser usado em vez dos identificadores apresentados anteriormente. Se uma instancia *Airplug* receber uma mensagem endereçada a outro nó, a mensagem sera descartada.

No *Airplug* um processo é endereçado por um par, como, por exemplo (APP, HST), onde APP é o nome de aplicação ou os identificadores ALL para todas as aplicações e CLT para a aplicação de controle.E HST pode ser os identificadores LCH, AIR, ALL ou o nome de um nó vizinho (KHALFALLAH; DUCOURTHIAL, 2010).

Com essa forma de endereçamento, é possível enviar mensagens para uma ou varias aplicações tanto local quanto remotamente por meio de uma única primitiva, identificada como SND. No entanto, uma aplicação não pode receber as mensagens pelo método AIR sem antes notificar o programa *Airplug* sobre a assinatura. Isso aumenta a robustez, evitando qualquer problema em cascata no caso de uma aplicação errônea. As assinaturas são feitas notificando o *Airplug* com as ações BEG e END, para inicio e fim respectivamente. Uma aplicação pode se inscrever a outra especifica ou a todas, localmente com LCH ou remotamente com AIR, também é possível assinar qualquer informação de controle, com CTL. Utilizando o identificador ALL, um aplicativo enviará mensagens para todos os aplicativos que assinaram para receber suas mensagens. O envio pode ser feito periodicamente, ou quando ocorre um evento, como alteração de um valor (KHALFALLAH; DUCOURTHIAL, 2010).

4.2.7 Formato das mensagens

Os formatos de mensagens no *Airplug* são compostas pelos campos de: ação, endereço, controle e carga. Esse formato podendo variar devido ao tipo da comunicação, local ou remota. O campo ação pode assumir os valores SND, BNG

ou END. As assinaturas existem somente no contexto local, portanto, o campo ação só é presente em mensagens locais.

O campo endereço contem um endereço para comunicações locais e dois para comunicações remotas. Uma comunicação distante inclui nós e é feita através da rede. A mensagem contem os endereços dos aplicativos de envio e recepção. Ao contrario, uma comunicação local inclui uma instancia *Airplug* e seu processo filho.

O campo de controle é utilizado para enviar dados opcionais com as mensagens como identificador dos nós, posição geográfica, entre outras. Qualquer aplicação, inclusive o próprio *Airplug* pode acessar os dados deste campo, ao contrário do campo carga.

O campo de carga depende da aplicação e contem os dados que são de fato utilizados e trocados entre os processos. As mensagens são enviadas em formato de texto simples. Facilitando, a leitura, a depuração e também permite a utilização da aplicação de forma isolada. Se necessário, a carga pode ser codificada e uma função otimizada permite a transmissão de cargas binárias.

Os campos são separados por um caractere fornecido no inicio da mensagem, esse caractere não pode estar presente nos campos, exceto, no campo de carga, que depende da aplicação. Assim sendo, os campos podem ter tamanhos variáveis, que, por sua vez, são bem adaptados para o sistema de endereçamento baseado em nomes do *Airplug*. Para fins de otimização, um modo permite usar prefixos dos nomes ao invés dos nomes completos.

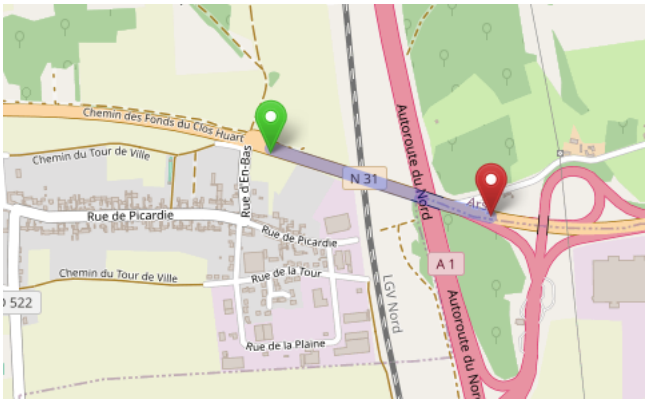
4.3 Comparação prática dos simuladores

Nessa seção é descrita a criação do cenário de simulação e elaboração do código-fonte no simulador *Veins*

4.3.1 Cenário de simulação

O ambiente de simulação utilizado no teste foi o mesmo que o utilizado no simulador *Airplug*. O cenário consiste em um recorte de uma rodovia francesa próximo a cidade de Arsy, consistindo em um trecho de aproximadamente 600 metros. Quatro nós percorrem esse trecho com velocidade de 115 km/h. O nó chamado v0 viaja sozinho em direção aos outros três nós, denominados v1 a v3 respectivamente. O nó v0 passa pelos outros nós trocando mensagens com estes ao longo do percurso. A figura 4.4 apresenta o cenário utilizado na simulação.

Figura 4.4 – Cenário utilizado na simulação no *framework Veins*



O recorte do mapa foi adquirido do site *OpenStreetMaps* e apresenta boa parte do vilarejo devido as proporções arbitrarias de exportação do site. O arquivo exportado tem o formato `.osm`, formato padrão do *OpenStreetMaps*. Foi necessário a conversão desse formato para o formato `.xml` utilizando o comando `netconvert` do SUMO. Esse novo arquivo foi então combinado com as informações das formas geométricas do cenario (que também se encontram no arquivo exportado do *OpenStreetMaps*) para formar o modelo completo do cenário, usando o comando `polyconvert` do *SUMO*.

Com o arquivo de cenário completo a próxima etapa consistiu em criar os fluxos de veículos que fariam parte da simulação. Para isto foi criado um arquivo XML contendo informações das rotas como: Tempo de instanciação na simulação,

número de veículos instanciados, ponto de inicio da rota e ponto de chegada. Após a criação desse arquivo, o comando `duaroute` do SUMO foi utilizado passando como parâmetros o arquivo de fluxo e o arquivo do cenário completo. Gerando assim o arquivo de rotas utilizado pelo *Veins*.

Com os arquivos de rotas, de cenário e de formas geométricas prontos. O último passo é copiar esses arquivos para a pasta do projeto e editar o arquivo de configurações do projeto para que inclua esses arquivos. A Tabela 4.1 apresenta os parâmetros utilizados na simulação do algoritmo AND no *framework Veins*.

Tabela 4.1 – Valores utilizados para simulação no *framework Veins*

Parâmetro	**Valor**
Tempo Máximo de simulação	200 segundos
Área da simulação	5500 m x 5500 m x 50 m
Distância entre os pontos inicial e final	525 m
Número de veículos utilizados	4 veículos
Distância de interferência máxima	2600 m
Potência de transmissão	20 mW
Taxa de bits	6 Mbps
Velocidade dos veículos	115 Km/h

4.3.2 Adaptação do algoritmo AND para o framework Veins

O algoritmo AND foi originalmente desenvolvido para o simulador *Airplug*. Escrito em Tcl/Tk e utilizando o modo *Airplug-emu*.

Para o desenvolvimento do algoritmo no *framework Veins* foram estudados o pseudocódigo do algoritmo, sua implementação em Tcl/Tk, a estrutura e as classes disponíveis no *Veins*. O algoritmo foi desenvolvido utilizando a linguagem C++.

A estrutura principal do algoritmo foi desenvolvida em 3 funções chamadas: `handleSelfMessage`, `sendWSM` e `onWSM`. Essas funções são todas padrões da suíte *Veins*, responsáveis por agendar mensagens, enviar mensagens e receber mensagens respectivamente. A lista de vizinhos de cada nó é a estrutura de dados principal do algoritmo, ela é formada por uma tabela *hash* não-ordenada (No C++ `unordered_map` com o par de identificação sendo um inteiro `int` e uma lista `list` de uma estrutura de dados que representa as informações que são enviadas nas mensagens chamada de `neighborData`.

O método `handleSelfMessage` é a mais importante pois é nesse método que ocorre o "controle"do algoritmo. Ao final desse método é feito um agendamento de acordo com uma constante, esse agendamento nada mais é que um atraso para a invocação desse método, funcionando assim, como uma forma de iteração. Essa função é responsável por manter o contador principal do algoritmo chamado de `timeToSend`, quando este contador for zerado, a função `sendWSM` é acionada. Após isso, um método de atualização da vizinhança é invocada. Caso uma mensagem tenha sido recebida, um método para computar a estimativa de confiabilidade da rede é acionado. Em seguida, o método responsável por calcular o intervalo entre-mensagens (IMD) é executado, os valores de mensagens recebidas e perdidas são atualizados e o agendamento da próxima iteração é feita.

O método `sendWSM` é responsável por enviar mensagens para os nós da rede. É nesse método que a mensagem é construída a partir dos dados do veículo e informações recebidas pelos nós vizinhos. A classe de mensagem no formato *WAVE* já é disponibilizada pela suíte *Veins*, porém, alguns ajustes foram necessários para comportar os dados utilizados pelo algoritmo. O método preenche a mensagem com os campos requisitados e a envia o mais rápido possível.

O método `OnWSM` por sua vez, se encarrega de processar as mensagens recebidas e calcular algumas estatísticas sobre a vizinhança do nó. Como primeiro passo, estruturas temporárias são criadas para armazenar os dados sobre a vizi-

nhança, esses dados são preenchidos com as informações contidas na mensagem recebida. Depois da mensagem ser processada, a tabela de vizinhos é atualizada caso a mensagem for de um novo vizinho; estimativas sobre o tempo de vida da vizinhança é feita para o nó que enviou a mensagem; e, por fim, o número de mensagens perdidas é atualizado, se necessário.

Além dos métodos discutidos anteriormente, vários outros foram desenvolvidos. O método `updateNeighbor` tem como tarefa atualizar a lista de vizinhos. Isso é feito comparando o tempo de vida de cada vizinho com o tempo atual da simulação, caso o tempo de simulação for maior que o tempo de vida do nó comparado, todas as informações do nó vizinho são deletadas da lista. O método `computeReliability` realiza todos os cálculos necessários para determinar a estimativa de confiabilidade da rede. O método `updateIMD` utiliza as informações da estimativa de confiabilidade de rede para calcular o tempo IMD. Caso o leitor queira conhecer mais detalhes do algoritmo e de sua implementação, verifique o Apêndice A

4.4 Comparação teórica dos simuladores

Dadas as informações apresentadas na seções anteriores, foi realizada uma análise comparativa sobre os dois simuladores.

A suíte *Veins* apresenta um ambiente de simulação mais realista e robusto, porque, conta com o acoplamento de dois simuladores bastante utilizados na comunidade acadêmica. Isso permite usufruir do melhor de ambos os simuladores. Conta com uma licença gratuita, disponível para *desktops* Linux que sejam compatíveis com as versões do *SUMO* e *OMNET++*. Aceitando varias opções de arquivos de redes viárias como, *TIGER*, *OpenStreetMaps*, *GIS*, entre outros. Os programas a serem escritos na suíte *Veins* devem obrigatoriamente ser escritos em *C++*, linguagem utilizada no simulador *OMNET++*.

No contexto de simulações, a suíte *Veins* foca na simulação fidedigna das camadas inferiores, MAC e PHY. Modelando de forma precisa os protocolos IEEE 1609.4 e IEEE 802.11p, Sendo possível, observar todo o conteúdo dos pacotes nas diversas camadas. Apresentando uma simulação de tráfego microscópica, ou seja, cada carro é controlado de maneira individual. As mensagens trocadas pelos nós da simulação podem ser representadas por mensagens ASCII em texto puro ou encapsuladas no padrão WSM (*WAVE Short Message*). Essas mensagens estão sujeitas a vários modelos de interferências como: Interferência de dois raios, sombreamento da infraestrutura urbana e também de outros carros presentes na simulação. A suíte *Veins* também leva em consideração o tipo de antena e a posição em que ela se encontra no nó, uma vez que, antenas são de suma importância para a simulação de comunicação sem fio. É possível realizar testes automatizados a partir de módulos específicos do *Veins* como também os já disponíveis no *OMNET++*. O *framework Veins* não apresenta suporte a parametrização rápida dos parâmetros (Quantidade de carros, taxa de perda artificial, etc), porque, para que isso seja feito é necessário alterar os arquivos utilizados pelo simulador SUMO e depois copiar esses arquivos para a pasta do projeto do *Veins*. Também não apresenta muitas opções de portabilidade, devido as dependências com os simuladores que o compõem.

O *framework Airplug* apresenta mais flexibilidade quando comparado a suíte *Veins*. Essa flexibilidade é evidenciada em diversos fatores: É um simulador independente de linguagem, uma vez que, o *Airplug* utiliza entrada e saída padrão para comunicação entre os nós. Pode rodar em qualquer sistema Linux, inclusive em sistemas embarcados. Apresenta possibilidade de parametrização rápida dos elementos da simulação (taxa de perda de mensagens, distancia de comunicação dos nós), geração de *scripts shell* para reprodução dos testes.

Quanto ao ambiente de simulação a flexibilidade do *Airplug* se torna um ônus quanto a fidelidade da simulação. O formato das mensagens disponíveis no *Airplug* são texto simples ou binário, que são endereçadas a um modelo próprio

consistindo em nome do nó e o nome da aplicação. As mensagens estão sujeitas a um modelo de interferência artificial. O modelo de tráfego disponível é baseado em arquivos de traços extraídos de experimentos de campo, não sendo possível gerar trajetos artificiais. Outro fator negativo é a presença de uma licença comercial para ter acesso a todos os modos disponíveis no *Airplug*, sendo necessário entrar em contato com a faculdade caso desejássemos utilizar o *framework* completo. Como o *Airplug* depende dos protocolos implementados pelo sistema operacional, se for desejado usar um protocolo não disponível, este deve ser implementado. Existe também a possibilidade de se ignorar certos protocolos da pilha, o que acarreta em um impacto significativo no realismo da simulação. Desse modo, o *Airplug* demonstra ser um simulador indicado para teste rápidos de novos protocolos, onde o que se deseja testar e avaliar é o desempenho do protocolo em si, ignorando fatores externos como interferência de sinal, potencial dos sinais de antenas utilizadas, entre outros.

Em contraste, a suíte *Veins* oferece robustez em suas simulações, disponibilizando amplas estatísticas sobre as mensagens trocadas entre os nós, sendo assim, a suíte *Veins* apresenta um ambiente mais recomendado para simulação realistas, capaz de prover informações importantes sobre o protocolo que está sendo testado e como este se comportaria no mundo real. Sendo uma alternativa mais viável que testes em ambientes reais. As principais diferenças entre os simuladores são apresentados na Tabela 4.2.

Tabela 4.2 – Tabela comparativa dos simuladores Airplug e Veins

Característica	***Airplug***	***Veins***
Categoria de simulador	Embarcado	Online
Plataforma	Qualquer sistema Linux, inclusive embarcados	*Desktops* Linux
Tipo de licença	"esqueleto" gratuito, licença comercial para uso completo	Gratuita
Linguagem	Independe, Deve ser uma linguagem que aceite E/S padrão	C++
Formato das mensagens	ASCII e binário	Texto puro, pacotes encapsulados (WSM))
Esquema de endereçamento	Nomes únicos, par nome/aplicação	IEEE 1609.4 e IEEE 802.11p
Simulação das pilhas de protocolos	Camadas superiores detalhadas, inferiores simplificadas	Camadas inferiores detalhadas, superiores simplificadas
Modelo de simulação de tráfego	Informações sobre os veículos retirado de experimentos reais	Microscópico (cada nó é controlado de maneira independente)
Modelo de interferência	Artificial	Interferência de dois raios, sombreamento de veículos, sombreamento de objetos
Modelos de antena	Não possui	Antenas isotrópicas, 2D e 3D
Testes automatizados	Geração de *scripts shell* para reprodução dos testes.	Disponível através de módulos dentro do próprio simulador
Granularidade	Um processo *Airplug* por nó	Um par de módulos por nó, um do *OMNET++* e um do *SUMO*
Portabilidade	Pode ser executado em sistemas embarcados em testes de campo	Não possui
Paralelismo	Distribuição de nós pela rede.	Pode ser distribuído em *clusters*.
Formato de arquivo de mapas suportados	*OpenStreetMaps*, arquivos de traços real	*OpenStreetMaps*, *TIGER*, *GIS*, arquivos de traços real

5 RESULTADOS E DISCUSSÃO

Nesse capitulo são descritos os resultados obtidos nesse projeto, como foi realizada a coleta e comparação dos resultados.

5.1 Coleta dos resultados

A coleta de dados foi feita com a criação de dois métodos no código-fonte, porque, os arquivos de *logs* gerados pela suíte *Veins* incluem muitas informações exibidas em um formato difícil de se entender. A coleta é feita criando ou abrindo um arquivo (se esse já existe) para escrita em formato texto e salvando as informações relevantes. O primeiro método criado foi `getCurrentTime`, esse método simplesmente retorna a data atual, pois, é possível passar uma data qualquer para o método de *log*. Dependendo da data passada o método retorna a data formatada. Isso é útil para evitar conflito de nomes no momento de criação do arquivo de *log*, caso sejam executadas varias simulações em um dia. O segundo método, `logger`, é responsável pela criação e escrita do arquivo de *log*. O método `logger` recebe como parâmetro uma *string* representando os dados que devem ser escritos no arquivo de *log*. Isso apresenta uma desvantagem, pois para cada informação que deve ser gravada, é necessário a concatenação dos valores em uma mensagem só.

Foram coletados dados sobre a distância, número de mensagens recebidas, remetente e receptor das mensagens enviadas e informações sobre as bordas superior e inferior juntamente com o tempo médio entre mensagens (IMD). Todos esses dados, levando em consideração o nó v0 como o nó principal. O arquivo de *log* gerado pela simulação foi então dividido manualmente em dois arquivos, um contendo os dados das informações trocadas entre os nós e o outro contendo informações sobre o IMD e as variações observadas nas bordas inferiores e superiores. Esses arquivos de *log* no formato texto passam por um parser escrito em Python com o objetivo de separar as informações relevantes para nosso estudo em um arquivo novo no formato CSV. O arquivo de *log* com as informações sobre as

mensagens trocadas entre os nós passa por mais um parser. Este segundo parser, também escrito em Python, ordena o arquivo CSV por tempo de simulação e por remetente das mensagens. Após essa segunda ordenação o arquivo é separado manualmente com todas as mensagens que foram recebidas pelo nó v0 em um novo arquivo para cada nó.

Os arquivos CSV (*log* de mensagens e *log* de variação do IMD) gerados na etapa anterior servem como entrada para *scripts* feitos na ferramenta Jupyter Lab. Esses *scripts* exibem os dados da simulação em dois gráficos: Um gráfico exibindo as mensagens trocadas pelos nós ao longo da simulação e um gráfico exibindo a variação do valor IMD e as bordas superior e inferior.

5.2 Comparação dos resultados

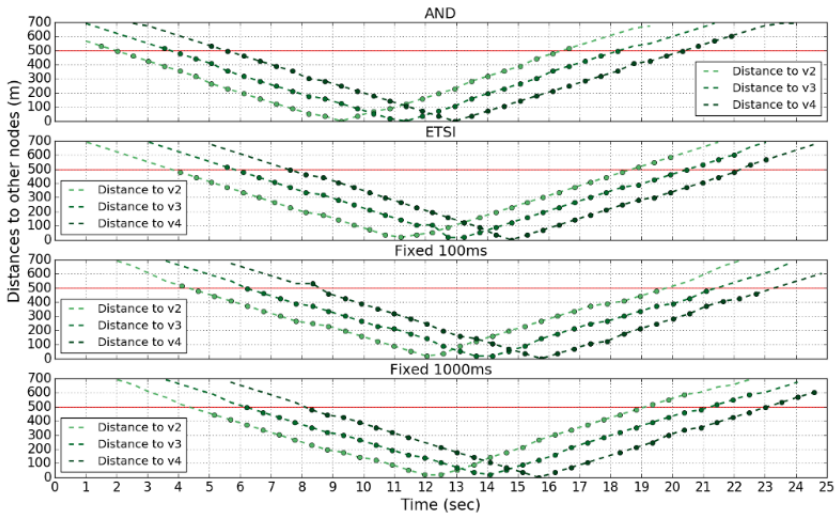
Para comparação dos resultados do algoritmo utilizamos os resultados obtidos em (MORAES, 2018) com a devida autorização do autor. A comparação baseou-se na distância de descobrimento entre os nós, além da variação do IMD.

A distância de descobrimento entre os nós é de extrema importância, porque, em um ambiente real, isso significa a capacidade do protocolo de evitar um acidente. A variação do IMD, por sua vez, é uma métrica interessante pois evidencia a capacidade de adaptação do protocolo.

Nos experimentos realizados em (MORAES, 2018) o algoritmo AND é comparado com outros algoritmos: ETSI CAS, um algoritmo com tempo de envio de mensagens fixo de 100 ms e outro com tempo fixo de 1000 ms. As figuras 5.1 e 5.2 apresentam os resultados no *Airplug*e *Veins* respectivamente.

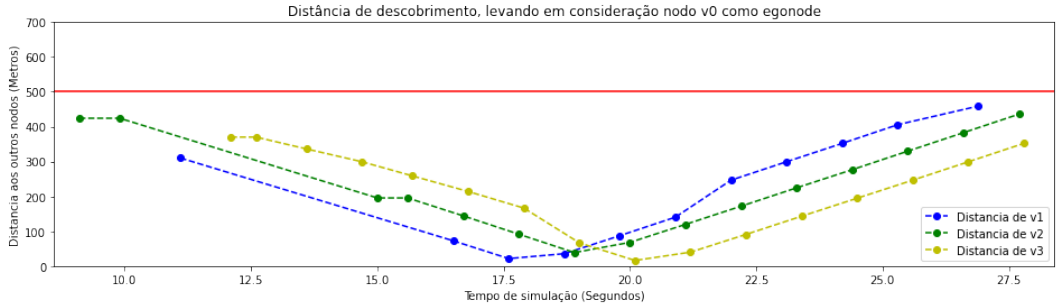
Como podemos analisar nas figuras o resultado obtido no simulador apresenta-se mais homogêneo para todos os nós ao longo da simulação, enquanto, na suíte *Veins* os resultados obtidos são mais discrepantes. Nota-se também, que nos resultados apresentados no *Veins* nem um nó foi descoberto antes da distância máxima

Figura 5.1 – Resultados do algoritmo AND no *Airplug*



Fonte: (MORAES, 2018)

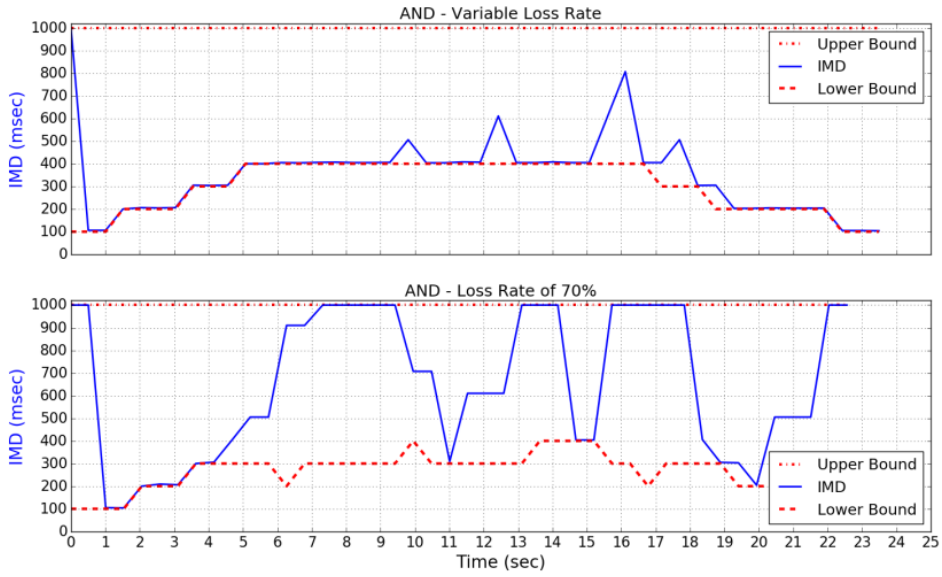
Figura 5.2 – Resultados do algoritmo AND no *Veins*



de comunicação (500m) enquanto no *Airplug* nós foram descobertos antes dessa distância.

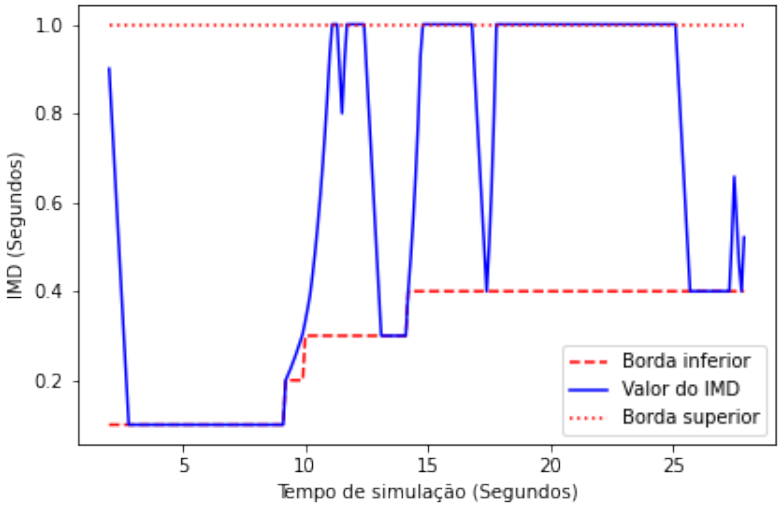
Para a comparação dos resultados da variação dos valores do IMD (MORAES, 2018) apresenta dois gráficos, correspondendo a simulação com uma taxa de perda variável e outra com uma taxa de perda de 70%. Na suíte *Veins* essa opção é possível, porém, requer ajustes em vários parâmetros complexos, o que demanda diversos ajustes. As figuras 5.3 e 5.4 apresentam as variações nos valores assumidos pelo IMD no *Airplug* e *Veins* respectivamente.

Figura 5.3 – Variação do IMD no Airplug



Fonte:(MORAES, 2018)

Figura 5.4 – Variação do IMD no Veins



Nesta comparação podemos observar que a variação do IMD na suíte *Veins* se aproxima mais da simulação com perda de 70%. A simulação no *Veins* apresenta-se mais estável quanto a descoberta de vizinhança, sendo que a borda

inferior, após descobrir um nó novo, não diminui ao longo do tempo restante da simulação , ou seja, o nó descoberto não teve seu tempo de vida expirado.

Como podemos perceber nas figuras acima, os resultados obtidos da adaptação do algoritmo AND para o simulador *Veins* apresentam resultados satisfatórios, uma vez que, não apresentaram valores tão discrepantes comparados com sua implementação original no *Airplug*. A implementação no *Veins* também demonstrou que o algoritmo é capaz de descobrir nós em uma distancia aceitável, mesmo em condições de simulação mais rigorosas.

6 CONCLUSÃO E TRABALHOS FUTUROS

O objetivo desse projeto foi realizar uma análise a fundo do impacto que a modelagem de um simulador tem nos algoritmos desenvolvidos. Como demonstrado nesse projeto os simuladores *Airplug* e *Veins* apresentam bastante diferenças.

O simulador *Airplug* foi desenvolvido para ser leve e portável, não impondo restrição nenhuma sobre a linguagem na qual as aplicações são desenvolvidas, o formato das mensagens que podem ser enviadas, o esquema de endereçamento envolvido e até que protocolos devem ser utilizados para entregar as mensagens. O ponto positivo dessa abordagem é a total liberdade para os pesquisadores que desenvolvem para esse simulador, o que de certa forma, acaba sendo uma desvantagem também, pois a responsabilidade toda fica nas mãos do pesquisador, tendo que definir os campos que serão utilizados pelos protocolos, que camadas serão utilizadas, para qual aplicação as mensagens são destinadas entre outros fatores. Uma das grandes vantagens do *Airplug* está na possibilidade de utilizar a aplicação desenvolvida em seus vários modos, podendo até, embarcá-la de maneira trivial em sistemas ativos para realização de testes de campo.

O simulador *Veins* por outro lado, serve quase como um completo oposto, é extremamente robusto em sua implementação, sendo composto por dois simuladores de bastante renome na comunidade acadêmica. As aplicações desenvolvidas para o *Veins* devem seguir os modelos já definidos, uma vez que, esses modelos são extremamente complexos e implementar um protocolo do inicio é uma tarefa árdua. No entanto, as simulações na suíte *Veins* apresentam resultados muito próximos da realidade, esse sendo um dos principais motivos para a escolha desse simulador. Os diversos modelos de interferência disponíveis agregados aos diversos modelos de mobilidade presente no SUMO permitem ao *Veins* simular cenários complexos de maneira muito detalhada. Porém essa complexidade serve como um ônus em alguns casos, como por exemplo, o tempo necessário para se aprender a usar o simulador, a falta de manuais e tutoriais, também acaba acarretando numa

dificuldade maior ao se aprender. *Veins* também é restringido a somente *desktop* Linux uma vez que depende de versões específicas do SUMO e OMNET++. A falta de opções para parametrizar a simulação de forma rápida é mais outro empecilho, levando em conta a demora para ajustar alguns parâmetros dentro do simulador.

Como trabalhos futuros considera-se a implementação do algoritmo CAS no simulador *Veins* para a análise e comparação.

REFERÊNCIAS

ALLANI, S. **Data Dissemination and Aggregation in Vehicular Adhoc Network**. Tese (Doutorado) — Université Pau et des Pays de l'Adour ; Université de Tunis El Manar, Nov 2018. Disponível em: <https://hal.archives-ouvertes.fr/tel-02298397>.

ALVES, R. dos S. et al. Redes Veiculares: Princípios, Aplicações e Desafios. **ResearchGate**, p. 199–254, May 2009. Disponível em: <https://www.researchgate.net/publication/320595628_Redes_Veiculares_Principios_Aplicacoes_e_Desafios>.

BAKO, B.; WEBER, M. Efficient Information Dissemination in VANETs. In: **Advances in Vehicular Networking Technologies**. Croatia: IntechOpen, 2011. ISBN 978-953-307-241-8.

BEHRISCH, M. et al. Sumo – simulation of urban mobility: An overview. In: . [S.l.: s.n.], 2011. v. 2011. ISBN 978-1-61208-169-4.

BRUMMER, A.; GERMAN, R.; DJANATLIEV, A. On the Necessity of Three-Dimensional Considerations in Vehicular Network Simulation. In: **2018 14th Annual Conference on Wireless On-demand Network Systems and Services (WONS)**. [S.l.]: IEEE, 2018. p. 75–82.

BUISSET, A. et al. Vehicular Networks Emulation. In: **2010 Proceedings of 19th International Conference on Computer Communications and Networks**. [S.l.]: IEEE, 2010. p. 1–7.

DUCOURTHIAL, B. About efficiency in wireless communication frameworks on vehicular networks. In: **WINITS '07: The First International Workshop on Wireless Networking for Intelligent Transportation Systems**. New York, NY, USA: Association for Computing Machinery, 2007. p. 1–9. ISBN 978-1-59593758-2.

DUCOURTHIAL, B. Designing applications in dynamic networks: The Airplug Software Distribution. **ResearchGate**, Sep 2013. Disponível em: <https://www.researchgate.net/publication/281885513_Designing_applications_in_dynamic_networks_The_Airplug_Software_Distribution>.

FESTAG, A. Standards for vehicular communication—from IEEE 802.11p to 5G. **Elektrotech. Inftech.**, Springer Vienna, v. 132, n. 7, p. 409–416, Nov 2015. ISSN 1613-7620.

JAIN, M.; SAXENA, R. Overview of VANET: Requirements and its routing protocols. In: **2017 International Conference on Communication and Signal Processing (ICCSP)**. [S.l.]: IEEE, 2017. p. 1957–1961.

KARAGIANNIS, G. et al. Vehicular Networking: A Survey and Tutorial on Requirements, Architectures, Challenges, Standards and Solutions. **IEEE Commun. Surv. Tutorials**, v. 13, n. 4, p. 584–616, 2011. ISSN 1553-877X. Disponível em: <https://ieeexplore.ieee.org/document/5948952>.

KHALFALLAH, S.; DUCOURTHIAL, B. Bridging the Gap between Simulation and Experimentation in Vehicular Networks. In: **2010 IEEE 72nd Vehicular Technology Conference - Fall**. [S.l.]: IEEE, 2010. p. 1–5.

MORAES, H. Pimenta de. **Adaptive solutions for data sharing in vehicular networks**. Tese (Theses) — Université de Technologie de Compiègne, maio 2018. Disponível em: <https://tel.archives-ouvertes.fr/tel-02052847>.

PANICHPAPIBOON, S.; PATTARA-ATIKOM, W. A Review of Information Dissemination Protocols for Vehicular Ad Hoc Networks. **Communications Surveys & Tutorials, IEEE**, Institute of Electrical and Electronics Engineers, v. 14, n. 99, p. 1–15, Jan 2012. ISSN 1553-877X.

SILVA, M. J. et al. Survey of Vehicular Network Simulators: A Temporal Approach. In: **Enterprise Information Systems**. Cham, Switzerland: Springer, 2019. p. 173–192. ISBN 978-3-030-26168-9.

SILVA, M. J. et al. Temporal evolution of vehicular network simulators: Challenges and perspectives. In: INSTICC. **Proceedings of the 20th International Conference on Enterprise Information Systems - Volume 2: ICEIS,**. [S.l.]: SciTePress, 2018. p. 51–60. ISBN 978-989-758-298-1.

SOMMER, C. et al. Veins – the open source vehicular network simulation framework. In: VIRDIS, A.; KIRSCHE, M. (Ed.). **Recent Advances in Network Simulation**. [S.l.]: Springer, 2019. ISBN 978-3-030-12841-8.

SOMMER, C.; GERMAN, R.; DRESSLER, F. Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis. **IEEE Trans. Mob. Comput.**, IEEE, v. 10, n. 1, p. 3–15, Jul 2010. ISSN 1558-0660.

VIVEK, N. et al. Implementation of IEEE 1609 WAVE/DSRC stack in Linux. **ResearchGate**, p. 1–5, Jul 2017.

ZEMOURI, S.; MEHAR, S.; SENOUCI, S.-M. HINTS: A novel approach for realistic simulations of vehicular communications. **2012 Global Information Infrastructure and Networking Symposium, GIIS 2012**, p. 1–6, Dec 2012.

APÊNDICE A – Código fonte algoritmo AND

O Código fonte do algoritmo AND foi desenvolvido na linguagem C++, adaptações foram feitas considerando quais estruturas de dados seriam as mais aproximadas as estruturas utilizadas no código em Tcl/Tk. O código foi desenvolvido a partir do código exemplo já disponível na maquina virtual Veins, dessa forma, os únicos arquivos alterados foram `TraCIDemo11p.h` , `TraCIDemo11p.cpp` e `TraCI11pMessage.msg`, correspondendo a biblioteca, sua implementação e o arquivo de formato das mensagens, respectivamente.

Listing 1 – TraCIDemo11p.h

```
1  /* Original code by Christoph Sommer <christoph.sommer@uibk.ac.at>
2   * Modifications by Guilherme Danrley Silva Hanauer <gdsh@estudante.ufla.br , gdshanauer@gmail.com>
3   * February - August 2021
4   */
5
6  #pragma once
7
8  #include "veins/modules/application/ieee80211p/DemoBaseApplLayer.h"
9
10 #include <unordered_map>
11 #include <list>
12 #include <vector>
13 #include <string>
14 #include <fstream>
15 #include <sstream>
16
17 namespace veins {
18
19     int R = 500; //Communication Range, fixed value. Alcance de comunicacao
20     double s_max = (double) 130/3.6; // French maximum speed in m/s. velocidade maxima francesa em m/s
21     double q = 0.99; // Assurance of 99% for a message reception.
22     double weight = 0.8;
23     double delta = 2.0; // 2 seconds
24
25     struct neighborData {
26         int seq_message;  // Message sequence number
27         double reliability; // Estimated network reliability at the time
28         double distance;  // Distance between the nodes
29         double speed;  // Relative speed between the nodes
30         double life;  // Relative life expectancy of the neighborhood
31         int numberOfNeighbors; // The total number of neighbors node V has
32     };
33
34 class VEINS_API TraCIDemo11p : public DemoBaseApplLayer {
35
36 public:
37     void initialize(int stage) override;
```

```
38      void finish(void) override;
39
40  protected:
41      simtime_t lastDroveAt;
42      bool sentMessage;
43      int currentSubscribedServiceId;
44      int msgFlag;
45
46      // Timers
47      double aTimer = 0.1; // Lowest timer value allowed (in milliseconds). Valor minimo permitido para o timer
48      double IMD = 1; //Time value between consecutive sent messages
49      double timeToSend = IMD; // Timer to send the next message.
50      double lower_bound = aTimer; // IMD lower bound. Limite inferior para o IMD.
51      double upper_bound = 1.0; // IMD upper bound. Limite superior para o IMD
52      double p = 1.0; //Number of attempts required to ensure a message reception
53      double lowerBoundLimit = 0.1;
54      double upperBoundLimit = 1.0;
55
56      // Arrays indexed by nodes IDs
57      std::unordered_map<int, std::list<neighborData>> tab_neigh;
58      std::list<int> neigh_List;
59
60      simtime_t lastSentTime;
61
62      Coord curPosition;
63      Coord lastSentPosition;
64      double curHeading;
65      double lastSentHeading;
66      double lastSentSpeed;
67
68      double msg_rcvd_smt = 0;
69      double msg_lost_smt = 0;
70
71      int msg_rcvd_old = 0;  // Number of messages received on the previous iteration
72      int msg_lost_old = 0;  // Number of messages lost on the previous iteration
73      int msg_rcvd = 0; // Number of messages received during aTimer
74      int msg_lost = 0;
75
76  protected:
77      void onWSM(BaseFrame1609_4* wsm) override;
78      void handleSelfMsg(cMessage* msg) override;
79      void handlePositionUpdate(cObject* obj) override;
80
81      void updateIMD();
82      void computeReliability();
83      double headingComputation(Coord P1, Coord P2);
84      double distanceComputation(Coord P1, Coord P2);
85      double speedComputation(double heading, double XSpeed, double YSpeed);
86      bool checkDynamicConditions(void);
87      void sendWSM(void);
88      void updateNeighbor(void);
89      inline std::string getCurrentDateTime(std::string time);
90      inline void logger(std::string logMessage);
91
92  };
```

```
93
94 } // namespace veins
```

Listing 2 – TraCIDemo11p.cpp

```
1  /* Original code by Christoph Sommer <christoph.sommer@uibk.ac.at>
2   * Modifications by Guilherme Danrley Silva Hanauer <gdsh@estudante.ufla.br , gdshanauer@gmail.com>
3   * February - August 2021
4   */
5
6  #include "veins/modules/application/traci/TraCIDemo11p.h"
7  #include "veins/modules/application/traci/TraCIDemo11pMessage_m.h"
8  #include <cmath>
9
10 using namespace veins;
11
12 Define_Module(veins::TraCIDemo11p);
13
14 #define Difference(x, y) (x > y) ? (x - y) : (y - x)
15 #define Pi 3.14159265
16
17 void TraCIDemo11p::initialize(int stage)
18 {
19     DemoBaseApplLayer::initialize(stage);
20
21     if (stage == 0) {
22         sentMessage = false;
23         lastDroveAt = simTime();
24         currentSubscribedServiceId = -1;
25         this->lastSentHeading = 0.0;
26     }
27
28     else {
29         // When first creating a vehicle we need to create
30         // a temporary list to insert it on the map
31         std::list<neighborData>  tempList;
32         neighborData element;  // Temporary object needed to store the default data
33
34         // fill our template object with default values
35         element.seq_message = 0;  // Default sequence number is zero (no message was sent)
36         element.reliability = 1;  // Default network reliability is 1
37         element.distance = 0;  // Default distance between nodes is zero
38         element.speed = 0;  // Default speed distance between nodes is zero
39         element.life = upper_bound;  // the default neighbor lifetime is the upper_bound limit
40         element.numberOfNeighbors = 0;  // Default number of neighbors is zero
41         tempList.push_front(element);  // Insert the template object in the list of messages
42
43         // Primeiro carro que e instanciado na simulacao inicia o arquivo de log
44         // Alterar para uma forma mais otimizada e generica
45         if(myId == 9)
46         {
47                 logger("--- Initializing Simulation ---");
48         }
49
```

```
50          tab_neigh.insert(std::make_pair(myId, tempList));
// Insert the list of messages in our map
51          // Schedule the next message to be sent at simTime() + 1 second
52          scheduleAt(simTime().dbl() + 1, new TraCIDemo11pMessage());
53      }
54  }
55
56  void TraCIDemo11p::finish(void) {
57
58      if(myId == 33)
59      {
60          logger("--- End of Simulation ---");
61      }
62  }
63
64  void TraCIDemo11p::onWSM(BaseFrame1609_4* frame)
65  {
66      TraCIDemo11pMessage* wsm = check_and_cast<TraCIDemo11pMessage*>(frame);
67
68      std::list<neighborData>  tempList;  // Temporary list in case of the sender is not a neigbor of node V
69      neighborData tempNeigh;  // Temporary struct so we can store the data from the sender node
70      neighborData lastMessageData;
71      double neigh_spa;  // Variable responsable for calculating the neighborhood space estimation
72      int unknown_neighbors = 0;
73      int neighborListSize, neighborListID;
74      // variable responsable for calculating the heading difference between node V and sender node
75      double headingDiff;
76      std::stringstream stream;
77      std::string streamResult[5];
78
79
80      // In this fuction we receive a message and add them to the list of neighbors of node V
81      msg_rcvd += 1;  // Increase the number of received messages
82      // fill our temporary structure with data received from the message
83      tempNeigh.seq_message = wsm->getSequenceNumber();
// get the sequence number of the message
84      //std::cout << "\n\n The sending node is
" << tempNeigh.seq_message << endl;
85      tempNeigh.reliability = wsm->getReliability();
// get the estimated reliability of the sender node
86      //std::cout << "\n\n The received reliability was:
" << tempNeigh.reliability << endl;
87      // Calculate the estimated distance between the sending node and the receiving node
88      tempNeigh.distance = distanceComputation(mobility->getPositionAt(simTime()), wsm->getPosition());
89      //std::cout << "\n\n The received estimated distance between the nodes is:
" << tempNeigh.distance << endl;
90      // Calculate the estimated speed between the sending node and the receiving node
91      headingDiff = headingComputation(mobility->getPositionAt(simTime()), wsm->getPosition());
92      //std::cout <<  "\nThis is node:  " << myId << " The heading difference is: " << headingDiff << "\n";
93      tempNeigh.speed =  (double) speedComputation(headingDiff, mobility->getSpeed(), wsm->getSpeed());
94      neighborListSize = wsm->getNeighborsArraySize();
95      tab_neigh[myId].front().numberOfNeighbors += 1; // Add to the number of node V total neighbors
96
97      // Here we try to estimate the neighborhood space
98      // If the difference in speed is positive the nodes are approaching each other
```

```
99      if(tempNeigh.speed >= 0)
100     {
101         neigh_spa = tempNeigh.distance + R;
102     }
103     // if not, the nodes are getting away from each other
104     else {
105         neigh_spa = tempNeigh.distance - R;
106         tempNeigh.speed = tempNeigh.speed * -1.0; // Revert the signal for the next calculations
107     }
108
109     tempNeigh.life = neigh_spa / tempNeigh.speed; // Compute the neighborhood lifetime estimation
110
111     if(tempNeigh.life < 0)
112     {
113         tempNeigh.life = tempNeigh.life * -1;
114     }
115
116     tempNeigh.life = (double) (simTime().dbl() + tempNeigh.life);
117     tempList.push_front(tempNeigh);  // Insert the temporary data into the temporary list
118
119     /* Here is a bit of wizardry, the insertionResult is the result of our insertion in the map
120        the insertion will fail if we try to insert another message from a node who is already a neighbor
121        of node V (us). So we use this result (it returns an iterator to the location where the element was
122        inserted and a boolean indicating if the operation was sucessful. We're only interested in the boolean)
123         to check if it fails or not, if it fails we only have to add he new element on the list, if not the new
124         entry in the map is made and we don't have to worry about it
125     */
126
127     auto const insertionResult = tab_neigh.insert(std::make_pair(wsm->getSenderAddress(), tempList));
128     auto const wasAlreadyInSet = !insertionResult.second;
129         // If the element was already in the map, we only add another entry to the neighbors message list
130     if(wasAlreadyInSet)
131     {
132         tab_neigh[myId].front().numberOfNeighbors -= 1; // If the neighbor is not a new one, we
133         // Access the last message received from node U, to check the last message sequence data
134         lastMessageData = tab_neigh[wsm->getSenderAddress()].front();
135         // Here we calculate the number of lost messages, the + 1 in the last messages stored
136         // is because if the difference between the newly received message and last stored is
137         // one there was no message lost,
138         msg_lost = tempNeigh.seq_message - (lastMessageData.seq_message + 1);
139         // Insert the received message in the list of received message of the neighbor node
140         tab_neigh[wsm->getSenderAddress()].push_front(tempNeigh);
141     }
142
143     /* Steps required for filling up the log file
144      * if(myId == 9 || (wsm->getSenderAddress() == 7))
145     {
146         stream << simTime();
147         stream >> streamResult[0];
148         stream.clear();
149         stream << myId;
150         stream >> streamResult[1];
151         stream.clear();
152         stream << wsm->getSenderAddress();
153         stream >> streamResult[2];
```

```
154         stream.clear();
155         stream << std::to_string(tempNeigh.distance);
156         stream >> streamResult[3];
157         stream.clear();
158         stream << std::to_string(neighborListSize);
159         stream >> streamResult[4];
160         stream.clear();
161         //std::cout << "\n" << streamResult[2] << endl;
162         logger(" - SIM TIME: " + streamResult[0] + " - Message Received: | RCV Node: " + streamResult[1] + " || SND Node: " +
163                 streamResult[2] + "|| Node DTC: " + streamResult[3] + "|| Node N NBR: " + streamResult[4]);
164     }*/
165
166     // Here we check if node U has neighbors that node V doesn't have
167     // We are checking directly on node U message
168     for(int i = 0; i < neighborListSize; i++)
169     {
170         neighborListID = wsm->getNeighbors(i);
    // Get every neighbor of node U
171
172         // Check if the neighbor exists in Node V list, if it exist,
173         //it already returns its position in the table
174         auto isNeighbor = tab_neigh.find(neighborListID);
175         // If the node DOESNT exist in node V table AND the sender neighbor is still in range
176         if((isNeighbor == tab_neigh.end()) && (lastMessageData.distance < R/2))
177         {
178             unknown_neighbors++;  // then node U neighbor is indeed an unknown neighbor
179         }
180     }
181     // The number of message lost also compromises the number of unknown neighbors
182     msg_lost = msg_lost + unknown_neighbors;
183 }
184
185 void TraCIDemo11p::handleSelfMsg(cMessage* msg)
186 {
187     if (TraCIDemo11pMessage* wsm = dynamic_cast<TraCIDemo11pMessage*>(msg)) {
188
189     }
190
191     timeToSend = timeToSend - aTimer;  // Subtract from the send message counter
192     // Check if it's time to send a message
193     if(timeToSend <= 0)
194     {
195         this->sendWSM(); // If is, call function to prepare and send a message
196         timeToSend = IMD;
197     }
198     // After sending a message check the list of neighbors
199     //if any should be deleted (because the lifetime has expired)
200     updateNeighbor();  // Call function responsible for deleting any expired node from the neighbors table
201
202     // If there was a message reception, update related values
203     if(this->msg_rcvd > 0)
204     {
205         // Smoothed number of messages received
    during aTimer
206         this->msg_rcvd_smt = (double) (this->msg_rcvd * weight) + (msg_rcvd_old * (1 - weight));
```

```
207         // Smoothed number of lost messages during aTimer
208         this->msg_lost_smt = (double) (this->msg_lost * weight) + (msg_lost_old * (1 - weight));
209         //std::cout << "MSG_RCV_SMT = " << msg_rcvd_smt << endl;
210         //std::cout << "MSG_LOST_SMT = " << msg_lost_smt << endl;
211         // Call the function that handles the Network Reliability estimations
212         computeReliability(); // This fuction calculate the estimatives of network reliability
213     }
214     // After intermediate values are updated, compute the new IMD
215     updateIMD();  // This function calculates the Inter Message Delay and updates lower and upper bound values
216
217     // Check whether is it required or not to update the send IMD
218     // If the Inter Message Delay is bigger than our timer, our timer need to be adjusted
219     if(this->IMD <= this->timeToSend)
220     {
221         this->timeToSend = this->IMD;  // The time to send a new message is now our IMD
222     }
223     // update variables for the next iteration
224     msg_lost_old = this->msg_lost;  // Update the message lost value for the next iteration
225     msg_rcvd_old = this->msg_rcvd;
226     this->msg_lost = 0;  // Reset the number of lost message before initiating the new iteration
227     this->msg_rcvd = 0;  // Reset the number of received message before initiating the new iteration
228     // schedule next message
229     // The next message is scheduled at simulation time + the IMD value
230     scheduleAt((simTime().dbl() + aTimer), new TraCIDemo11pMessage());
231 }
232
233 void TraCIDemo11p::handlePositionUpdate(cObject* obj)
234 {
235     DemoBaseApplLayer::handlePositionUpdate(obj);
236 }
237 /* Function responsible for calculating the Inter Message Delay
238  * Input: Nothing
239  * Output: Nothing
240  */
241 void TraCIDemo11p::updateIMD()
242 {
243     int numb_of_neigh = tab_neigh.size(); // Number of neighbors is equal to table size
244     double tmp_ub; // Temporary upper bound
245     double tmp_IMD; // Temporary IMD
246     std::stringstream stream;
247     std::string logInfo[4];
248     double upperBoundDivisor;
249
250     // The lower bound value is equal to the number of neighbors * the timer interval
251     this->lower_bound = this->aTimer * numb_of_neigh;
252
253     if(this->lower_bound < lowerBoundLimit)
254     {
255         this->lower_bound = lowerBoundLimit;
256     }
257
258     // Check for preventing division by zero
259     if(p <= 0)
260     {
261         p = 1.0;
```

```
262     }
263     // The temporary upper bound value is based on the node
264     // actual speed divided by the number of attempts
265     tmp_ub = (double) ((1 / p) * ((R / (mobility->getSpeed() + s_max)) - delta));
266
267     // If the temporary upper bound is greater than the actual
268     // upper bound we calculate a smoothed value for the upper bound
269     if(tmp_ub > this->upper_bound)
270     {
271         this->upper_bound = (double) (this->upper_bound * weight) + ( tmp_ub * (1 - weight));
272     }
273     else {
274         this->upper_bound = tmp_ub; // If its not the upper bound is just the temporary value
275     }
276
277     if(this->upper_bound > upperBoundLimit)
278     {
279         this->upper_bound = upperBoundLimit;
280     }
281
282     // If we have lost messages, we need to reduce our Inter Message Delay
283     if(msg_lost_smt > 0)
284     {
285         tmp_IMD = (double) IMD * ( 2 - (tab_neigh[myId].front().reliability));
286     }
287     else {
288         tmp_IMD = this->IMD - this->aTimer;
289     }
290     // Our IMD cant be greater than our upper bound value
291     if(tmp_IMD > this->upper_bound)
292     {
293         this->IMD = this->upper_bound;  // So if it is we adjust it to the upper bound
294     }
295     // Our IMD cant be lower than our lower bound value either
296     else if(tmp_IMD < this->lower_bound)
297     {
298         this->IMD = this->lower_bound;  // So if it is we adjust it to the lower bound
299     }
300     // If it isnt any of the previous cases our IMD will just be the temporary IMD value
301     else {
302         this->IMD = tmp_IMD;
303     }
304
305     /* Steps required for filling up the log file
306      * if(myId == 9)
307     {
308         stream << simTime();
309         stream >> logInfo[0];
310         stream.clear();
311         stream << std::to_string(this->IMD);
312         stream >> logInfo[1];
313         stream.clear();
314         stream << std::to_string(this->lower_bound);
315         stream >> logInfo[2];
316         stream.clear();
```

```
317         stream << std::to_string(this->upper_bound);
318         stream >> logInfo[3];
319         stream.clear();
320         logger(" - SIM TIME: " + logInfo[0] + " - IMD Update: | IMD Value: " + logInfo[1] + "|| Lower Bound: " +
321             logInfo[2] + "|| Upper Bound: " + logInfo[3] + "|");
322     }*/
323
324 }
325 /* This function is responsible for computing the reliability of the network
326  * Input: Nothing
327  * Output: Nothing
328  */
329 void TraCIDemo11p::computeReliability()
330 {
331     double old_rel; // temporary variable to store the old reliability
332     double curr_rel = 0;  // Current reliability set to zero, to prevent calculation errors
333     double dividend, divisor; // temporary variables to assist in the computation of the network reliability
334     double dividendP, divisorP;
335     dividend = 0; // Dividend and divisor set to zero, to prevent calculation errors
336     divisor = 0;
337
338     // Get Node V's data, we'll needed to update node V's reliability
339     auto neighborData = tab_neigh[myId].front();
340     // The old reliability is  node V's reliability
341
342     old_rel = neighborData.reliability;
343     // Node V reliability local reliability is the smoothed number of messages
344     // received divided by the total number of messages
345     neighborData.reliability =(double) (this->msg_rcvd_smt / (this->msg_rcvd_smt + this->msg_lost_smt));
346     tab_neigh[myId].front() = neighborData; // Update node V data on the neighbor table
347     // This loop iterates throughout the neighbors table calculating
348     // the global network reliability estimation
349
350     for(auto it = tab_neigh.begin(); it != tab_neigh.end(); it++)
351     {
352         auto loopData = it->second.front();
    // Gets the last sent message from each neighbor
353         // Check done to prevent division by zero
354         if(loopData.numberOfNeighbors <= 0)
355         {
356             loopData.numberOfNeighbors = 1;
357         }
358         // The global reliability estimation is a summation of
359         // each node proportional reliability divided by its inverse
360         dividend += (double) loopData.reliability / loopData.numberOfNeighbors;
361         divisor += (double) 1 / loopData.numberOfNeighbors;
362     }
363
364     curr_rel = (double) (dividend / divisor); // Here we calculate the estimated global reliability
365     // Then we update again node V reliability with the global estimation now
366     neighborData.reliability = (double) (old_rel * weight) + (curr_rel * (1 - weight));
367     // With the final reliability estimation calculated we now calculate the
368     // minimum number of attempts required for a sucessful message delivery
369     // with 99% confiability, this is what the value of P represents
370     p = (double) (log(1.0 - q)) /  (log(1 - neighborData.reliability));
```

```
371
372     // At last, we save our changes to the neighbors table
373     tab_neigh[myId].front() = neighborData;
374
375 }
376 /* This function is responsible for calculating the heading (in degrees) between two positions
377  * Input:
378  *      Coord P1 = 2D coordinates of position 1
379  *      Coord P2 = 2D coordinates of position 2
380  * Output:
381  *      double result = The resulting heading value (in degrees)
382  */
383 double TraCIDemo11p::headingComputation(Coord P1, Coord P2)
384 {
385     // Declaration of auxiliary variables diffence X and diffence Y
386     // and also the result variable which will be the return value
387     double diffX, diffY, result;
388
389     diffX = P1.x - P2.x;  // The difference X is the difference between P1 and P2 Xs values
390     diffY = P1.y - P2.y;  // The difference Y is the difference between P1 and P2 Ys values
391
392     // If any of the differences is zero the heading is the last same as the previously sent
393     if(diffX == 0 || diffY == 0)
394     {
395         result = this->lastSentHeading;
    // There was no difference in headings, so the result
396         return result;  // is the last sent heading
397     }
398
399     else {
400         // If not the difference is given by the arc tangent of cathet X divided by cathet Y
401         result = atan(diffY / diffX); // This result is given in radians, so we have to convert it back to degrees
402         result = (result * 180) / Pi; // Converting the result to degrees
403
404         return result;
405     }
406
407     return -5;  // Error code if something goes wrong (NOT IMPLEMENTED)
408 }
409 /* Function Responsible for calculating the distance between two points
410  * Input:
411  *      Coord P1 = 2D coordinates of position 1
412  *      Coord P2 = 2D coordinates of position 2
413  * Output:
414  *      double result = The distance between the two points (in meters)
415  */
416 double TraCIDemo11p::distanceComputation(Coord P1, Coord P2)
417 {
418     return (double) sqrt(pow(P2.x - P1.x, 2) + pow(P2.y - P1.y, 2) * 1.0);
419 }
420 /* This function is responsible for returning the relative speed between two nodes
421  * Input:
422  *      double heading = The heading difference between the nodes
423  *      double XSpeed = The speed value of the receiving node, node V (our node)
424  *      double YSpeed = The speed value of the sender node, node U
```

```
425  * Output:
426  *       double relativeSpeed = The relative speed value between node V and node U
427  */
428 double TraCIDemo11p::speedComputation(double heading, double XSpeed, double YSpeed)
429 {
430     double relativeSpeed;  // The return variable, the relative speed between node V and node U
431
432     heading = fabs(heading);  // We need the absolute value of our heading
433
434     // If the difference between the heading is greater than 90
435     // degrees the nodes are approaching each other
436     if(heading >= 90.0)
437     {
438         relativeSpeed = XSpeed + YSpeed;
    // if the nodes are approaching we sum their speed
439     }
440     else {
441         relativeSpeed = XSpeed - YSpeed;
    // If not the nodes are in the same directon, subtract their speed
442     }
443
444     return relativeSpeed;
445 }
446 // This function was not used
447 bool TraCIDemo11p::checkDynamicConditions(void)
448 {
449     double direction, speedDiff, distance, headingDiff;
450
451     // calculate the speed difference between nodes
452     speedDiff = Difference(mobility->getSpeed(), this->lastSentSpeed);
453     // If the difference is greater than 0,5 m/s we need to send a message
454     if(speedDiff > 0.5)
455     {
456         return true;
457     }
458     else {
459         // Verifying the distance between the nodes current position and last sent position
460         distance = (this->curPosition - this->lastSentPosition).length();
461         // If the distance is greater than 4 meters we need to send a message
462         if(distance > 4)
463         {
464             return true;
465         }
466         else {
467             // Verifying the last angle between the nodes current and previous position
468             this->curHeading = headingComputation(this->lastSentPosition, this->curPosition);
469             headingDiff = Difference(this->curHeading, this->lastSentHeading);
470             // If the angle is greater than 4 degree we need to send a message
471             if(headingDiff > 4)
472             {
473                 return true;
474             }
475         }
476     }
477
```

```
478     return false;
479 }
480 /* This function is responsible for creating, filling and sending a message
481  * Input: Nothing
482  * Output: Nothing
483  */
484 void TraCIDemo11p::sendWSM(void)
485 {
486     TraCIDemo11pMessage *wsm = new TraCIDemo11pMessage();
487
488     int i = 0;  // i is an iterator for our list of possible neighbors
489
490     auto tempNeighbor = tab_neigh[myId].front();
    // Set a temporary variable to make changes in the structure
491
492     //std::cout << "My message sequence is: " << tempNeighbor.seq_message << endl;
493     this->lastSentHeading = mobility->getHeading().getRad();
494     //std::cout << "\n Heading value is: " <<
    this->lastSentHeading << endl;
495     tempNeighbor.seq_message += 1;  // increase the message sequence number
496
497     // Fill up the message with current speed, heading, position,
498     // reliability, address and sequence number from node V
499     tempNeighbor.speed = mobility->getSpeed();
    // Check the mobility variable for the current speed
500     wsm->setSpeed(mobility->getSpeed());
    // Set the field "Speed" of our message
501     this->lastSentSpeed = mobility->getSpeed();
    // Update the variable with the last speed sent
502     this->curPosition = mobility->getPositionAt(simTime());
    // Update the nodes current position
503     //std::cout << "The Position is: " << this->curPosition << endl;
504     wsm->setHeading(this->curHeading);  // Set the field "heading" of our message
505     this->lastSentPosition = this->curPosition;
    // Update the last position to our current position
506     wsm->setPosition(this->curPosition);
    // Set the field "Position" of our message
507     wsm->setSenderAddress(this->findHost()->getId()); //Set the field "Sender address" of our message
508     wsm->setReliability(tempNeighbor.reliability);
    // Set the field "reliability" of our message
509     wsm->setSequenceNumber(tempNeighbor.seq_message);
    // Set the field "Sequence number" of our message
510     wsm->setNeighborsArraySize(tab_neigh.size() - 1);
    // Set the field "neighbors Array size" of our message
511     // The neighbors array size is 1 unit less than the actual size
512     // Because the first "neighbor" we have is ourselves
513     this->lastSentTime = simTime().dbl();
    // Update the variable
514     // Add the changes back in to the list
515     populateWSM(wsm, -1);  // Create and populate the message node V has to send
516     tab_neigh[myId].front() = tempNeighbor; //THIS WORKS DONT DELETE
517     // This loop is responsible for adding the nodes U in the neighbor list
518     for(auto it = tab_neigh.begin(); it != tab_neigh.end(); it++)
519     {
520         // Checks if the current node is not the node V itself
```

```
521         if(it->first != myId)
522         {
523             //std::cout << "\n Key Value: " << it->first << endl;
524 
525             // If its not them check for the messages sended
526             auto u = it->second.front();
527 
528             //std::cout << "\n Key Value: " << u.seq_message << endl;
529             // If node U distances is less then the range add U to node V list of neighbors
530             // If the distance between node V and node U is less than half the total distance
531             // add node U to the list of neighbors and to the list in our message
532             if(u.distance < (R/2))
533             {
534                 neigh_List.push_back(it->first);
// Add node U to the list of know neighbors
535                 wsm->setNeighbors(i, it->first);
// Add node U to the list on the message we are going to send
536                 i++;
537             }
538         }
539     }
540     // Send the message
541     sendDown(wsm);
542 }
543 /* This is the function responsible for deleting an neighbor from the neighbor list
544  * Input: Nothing
545  * Output: Nothing
546 */
547 void TraCIDemo11p::updateNeighbor(void)
548 {
549     // First we declare a iterator vector for storing pointers to the neighbors we are going to delete
550     std::vector<std::unordered_map<int, std::list<neighborData>>::iterator> neighborsToBeDeleted;
551 
552     /* The reason for this is because when trying to delete an object from the map
553      * during a loop it will crash, because the iterator passed to the erase()
554      * function will be dereferenced, thus, we store all the iterators, and
555      * in a different loop we delete them
556      */
557     // Iterate through the neighbors table with an iterator
558     for(auto it = tab_neigh.begin(); it != tab_neigh.end(); ++it)
559     {
560         if(it->first != myId) // Check if the current node is not node V itself
561         {
562             auto u = it->second.front(); // If its not, check the last message send
563 
564             if((u.life) <= (simTime().dbl()) )
565             {
566                 it->second.clear();  // We erase the list of messages from node U
567                 // And we add node U position on the list to our delete vector
568                 neighborsToBeDeleted.push_back(it);
569             }
570         }
571     }
572     // This second loop iterates through our vector, deleting the neighbors from our table of neighbors
573     for(auto it : neighborsToBeDeleted)
```

```
574     {
575         tab_neigh.erase(it); // Erases the neighbor from the map
576     }
577 }
578
579 inline std::string TraCIDemo11p::getCurrentDateTime(std::string logTime)
580 {
581     time_t now = time(0);
582     struct tm tstruct;
583     char buffer[80];
584
585     tstruct = *localtime(&now);
586
587     if(logTime == "now")
588     {
589         strftime(buffer, sizeof(buffer), "%d-%m-%Y %X", &tstruct);
590     }
591
592     else if(logTime == "date")
593     {
594         strftime(buffer, sizeof(buffer), "%d-%m-%Y", &tstruct);
595     }
596
597     return std::string(buffer);
598 }
599
600 inline void TraCIDemo11p::logger(std::string logMessage)
601 {
602     std::string filePath = "/home/veins/src/veins/examples/veins/log/sim_" + getCurrentDateTime("date") + ".txt";
603     std::string now = getCurrentDateTime("now");
604
605     std::ofstream ofs(filePath.c_str(), std::ios_base::out | std::ios_base::app);
606
607     ofs << now << '\t' << logMessage << '\n';
608     ofs.close();
609 }
```

Listing 3 – TraCIDemo11pMessage.msg

```
1  // Original code by Christoph Sommer <christoph.sommer@uibk.ac.at>
2  // Modifications by Guilherme Danrley Silva Hanauer <gdsh@estudante.ufla.br , gdshanauer@gmail.com>
3  // February - August 2021
4  //
5
6  cplusplus {{
7  #include "veins/base/utils/Coord.h"
8  #include "veins/modules/messages/BaseFrame1609_4_m.h"
9  #include "veins/base/utils/SimpleAddress.h"
10 }}
11
12 namespace veins;
13
14 class BaseFrame1609_4;
15 class noncobject Coord;
```

```
16 class LAddress::L2Type extends void;
17
18 packet TraCIDemo11pMessage extends BaseFrame1609_4 {
19     string demoData;
20     LAddress::L2Type senderAddress = -1;
21     int serial = 0;
22     // dados da mensagem
23     double speed; // Current node speed
24     double heading; // current node heading
25     Coord position; // current node position
26     double reliability; // current nodes reliability
27     int sequenceNumber; // node sequence number
28     int neighbors[];  // node list of neighbors
29
30 }
```