Capítulo 2

Simulação de Redes Veiculares

Roniel Soares, Sebastião Galeno e André Soares

Resumo

As Redes Veiculares, conhecidas como VANETs (Vehicular Ad Hoc Networks), são redes que permitem a troca de informações entre veículos (comunicação V2V - Vehicle-to-Vehicle) e entre veículos e infra-estruturas localizadas as margens das vias (comunicação V2I - Vehicle-to-Infrastructure). As VANETs possuem caracteristicas particularidades que são desafios na comunicacao entre veículos. Algumas dessas características são: (i) a alta mobilidade dos nos, (ii) mudanças constantes na topologia da rede, (iii) a existência de redes densas que ocasionam altas taxas de perdas de pacotes por congestionamento na rede e (iv) a existência de redes esparsas que dificultam a disseminação de informações [1]. As aplicacoes de Redes Veiculares podem ser divididas em três classes: segurança no transito, entretenimento e assistência ao motorista. O principal desafio para as aplicações de segurança é divulgar rapidamente as informações para que o condutor tenha tempo para reagir. Nas aplicações de entretenimento destacam-se os sistemas de compartilhamento de conteúdo (músicas, filmes) e jogos. Já as aplicações de assistência ao motorista são aquelas que auxiliam o condutor, como por exemplo através de informações sobre as condições de trânsito [2]. O objetivo deste minicurso é apresentar o ambiente de simulação utilizado no Laboratório de Redes e Sistemas Distribuidos da UFPI (DisNeL - UFPI) para avaliação de desempenho de algoritmos e protocolos em redes veiculares. O referido ambiente de simulação é composto pela ferramenta de simulação de rede OMNET++ 4.5 [3] em conjunto com o simulador de tráfego e mobilidade urbana SUMO 0.21.0 [4]. O framework Veins 3.0 que é desenvolvido para redes veiculares e possui suporte para o padrão 802.11p será utilizado para integrar estes dois simuladores [5].

2.1. Introdução

Problemas relacionados ao trânsito em grandes cidades causam impactos negativos em diversas áreas, como na economia, no meio ambiente, na saúde, dentre outras. Um exemplo são os prejuízos causados pelos congestionamentos. Segundo a Fundação Getúlio Vargas, os congestionamentos causaram prejuízos de cerca de R$ 40 bilhões na cidade

de São Paulo, somente no ano de 2012 [6]. Os Sistemas Inteligentes de Transportes, tais como as Redes Veiculares, são vistos hoje como uma das principais formas para solucionar os diversos problemas relacionados ao trânsito [7].

As Redes Veiculares são formadas por veículos automotores e infraestruturas localizadas nas margens das vias. Tais redes permitem que veículos troquem informações entre sí ou com as infraestruturas, permitindo o desenvolvimento de uma ampla variedade de aplicações que visam aumentar a segurança e o conforto no trânsito. As características específicas do ambiente veicular trazem uma série de desafios para a implantação desta tecnologia. Neste cenário, é importante implementar e avaliar novos protocolos e aplicações específicos para este tipo de rede.

O objetivo deste capítulo é apresentar o ambiente de simulação utilizado no DisNeL - UFPI para avaliação de desempenho de algoritmos e protocolos em redes veiculares. O referido ambiente de simulação é composto pela ferramenta de simulação de rede *OMNeT++ 4.5* em conjunto com o simulador de tráfego e mobilidade urbana *SUMO 0.21.0*. O *framework Veins 3.0*, que é desenvolvido para redes veiculares e possui suporte para o padrão 802.11p, é utilizado para integrar estes dois simuladores.

O restante deste capítulo está organizado da seguinte forma. A Seção 2.2 apresenta uma introdução às redes veiculares. Na Seção **??** são discutidos os principais métodos de avaliação de desempenho. Em seguida, o ambiente de simulação utilizado no DisNeL é apresentado nas seguintes seções: 1) 2.3 introdução às ferramentas, 2) tutorial de instalação, 3) execução do primeiro experimento e 4) implementação de um novo protocolo. Por fim, na Seção 2.4 são apresentadas as considerações finais.

2.2. Redes veiculares

As Redes Veiculares, conhecidas como VANETs (*Vehicular Ad Hoc Networks*), são redes que permitem a troca de informações entre veículos (comunicação V2V - *Vehicle-to-Vehicle*) e entre veículos e infra-estruturas localizadas às margens das vias (comunicação V2I - *Vehicle-to-Infrastructure*).

A arquitetura das redes veiculares pode ser caracterizada de acordo com o tipo de comunicação existente. A Figura 2.1 apresenta as três principais arquiteturas que são: 1) ad hoc puro, 2) infraestruturada e 3) híbrida [2]. No modo ad hoc puro, a comunicação ocorre apenas de veículo para veículo (comunicação V2V). A principal vantagem deste modo é o baixo custo, pois ele não necessita da instalação de infraestruturas externas. Porém, a sua principal desvantagem está na conectividade da rede que depende da densidade e da mobilidade dos veículos. Já o modo infraestruturado é caracterizado pela comunicação entre veículo e infraestruturas localizadas nas margens das vias. Tais infraestruturas, chamadas de RSUs (Road Side Units), são nós estáticos que funcionam como pontos de acesso, ou seja, podem viabilizar a conexão com a internet. A principal vantagem deste modo está na alta conectividade da rede, tendo em vista que estes nós estáticos poderão ser posicionados de modo que a rede permaneça conectada. Já a principal desvantagem está no alto custo para implantação e manutenção das RSUs. O modo híbrido, caracterizado pela existência dos dois tipos de comunicações (comunicação V2X) tenta obter o melhor das duas arquiteturas. Neste caso, são posicionados uma quantidade mínima de RSUs para aumentar a conectividade da rede sem elevar demasiadamente o custo da rede.

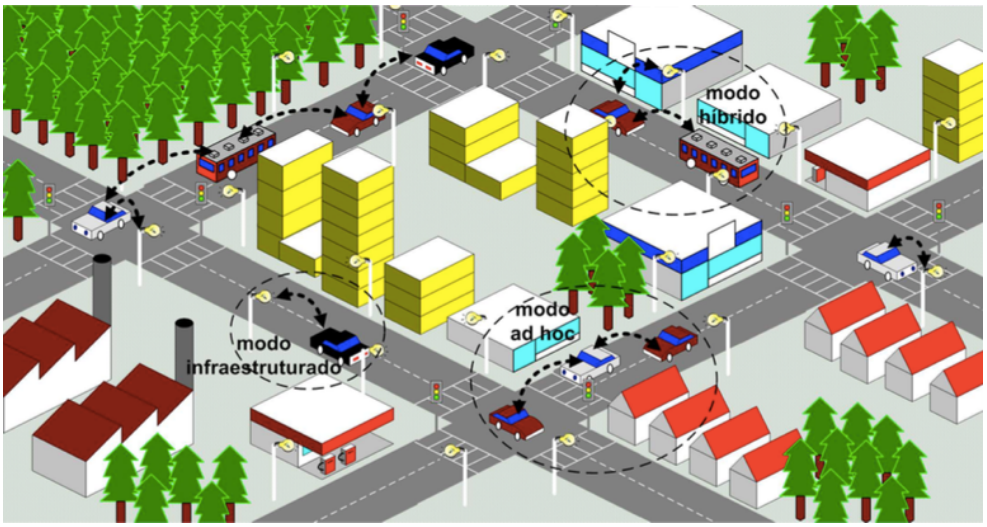

Figura 2.1: Arquiteturas em Redes Veiculares. Figura retirada de [2].

Algumas características específicas do ambiente veicular trazem dificuldades para a comunicação sem fio e representam desafios para a adoção das Redes Veiculares em larga escala. Algumas dessas características são: (i) a alta mobilidade dos nós, (ii) mudanças constantes na topologia da rede, (iii) a existência de redes densas que ocasionam altas taxas de perdas de pacotes por congestionamento na rede e (iv) a existência de redes esparsas que dificultam a disseminação de informações. Estas características fazem com que protocolos existentes para outros tipos de redes não sejam adequados para as redes veiculares [2].

2.2.1. Padrões em redes Veiculares

A primeira tentativa de padronização da comunicação em ambiente veicular ocorreu no final da década de 90. A principal nomemclatura utilizada na época para referenciar às redes veiculares era *Dedicated Short Range Communications* (DSRC). Em 1999 a Comissão Federal de Comunicações (FCC) dos Estados Unidos alocou 75 Mhz do espectro de frequências, na faixa de 5.9 Ghz para ser utilizada especificamente em Sistemas Inteligentes de Transporte (ITS). Apenas em 2008, o Instituto Europeu de Normas de Telecomunicações alocou 30 Mhz do espectro de frequências na mesma faixa para ITS.

Desde o ano de 2004 um grupo de trabalho do IEEE vem desenvolvendo a família de padrões WAVE (*Wireless Access in the Vehicular Environment*). O objetivo deste grupo de trabalho é: permitir o desenvolvimento de dispositivos WAVE que podem se comunicar com baixa latência e baixo overhead em apoio ao transporte seguro, eficiente e sustentável e que podem aumentar o conforto e conveniência do usuário [8]. A nomemclatura WAVE é a mais utilizada atualmente para se referenciar à comunicação veicular. A Figura 2.2 apresenta a arquitetura WAVE e referencia alguns dos documentos publicados pelo grupo de trabalho para padronização da arquitetura, tais como:

- ***IEEE Std 802.11-2012, IEEE Standard for Information technology - Telecommunications and information exchange between systems-Local and metropolitan***

area networks-Specific requirements - Part 11: Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) specifications

Especifica uma subcamada de Controle de Acesso ao Meio (MAC) e várias camadas físicas (PHYs). O IEEE Std 802.11p-2010 está incorporado ao IEEE Std 802.11-2012 e especifica um conjunto de extensões para o IEEE Std 802.11-2012 operar fora de um contexto de um conjunto de serviços basicos (BSS).

- ***IEEE Std 1609.2-2013, IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Security Services for Applications and Management Messages***

  Especifica serviços de segurança para aplicações e para mensagens de gerenciamento.

- ***IEEE Std 1609.3-2010, IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Networking Services***

  Especifica serviços de rede requeridos para a operação de um sistema WAVE e inclui as seguintes características: 1) protocolo *WAVE Short Message*, *WAVE Service Advertisements* e 3) agendamento de canais.

- ***IEEE Std 1609.4-2010, IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Multi-Channel Operation***

  Especifica extensões para o protocolo da camada MAC IEEE 802.11 tais como: 1) temporização e comutação de canal e 2) uso do IEEE 802.11 fora do context de um BSS.

- ***IEEE Std 1609.11-2010, IEEE Standard for Wireless Access in Vehicular Environments (WAVE) - Over-the-Air Electronic Payment Data Exchange Protocol for Intelligent Transportation Systems (ITS)***

  Primeira protocolo da camada de aplicação do padrão IEEE 1609. Especifica um protocolo de pagamentos para, por exemplo, cobranças eletrônicas.

2.2.2. Aplicações de redes veiculares

As aplicações de Redes Veiculares podem ser divididas em três classes: segurança no trânsito, entretenimento e assistência ao motorista. O principal desafio para as aplicações de segurança é divulgar rapidamente as informações para que o condutor tenha tempo para reagir. Nas aplicações de entretenimento destacam-se os sistemas de compartilhamento de conteúdo (músicas, filmes) e jogos. Já as aplicações de assistência ao motorista são aquelas que auxiliam o condutor, como por exemplo através de informações sobre as condições de trânsito [2].

Alguns exemplos de aplicações são: alerta sobre violação de sinal de trânsito, assistência em cruzamentos, alerta sobre aproximação de ambulância, serviços de SOS, extensão das sinalizações de trânsito, alerta de ultrapassagem, alerta sobre colisões, etc.

Outra categoria que vêm chamando bastante atenção são as aplicações relacionadas ao monitoramento e controle de tráfego [1] [9] [10] [7] [11] [12] [13] [14] [15] [16] [17]. Estas aplicações possuem como objetivo transmitir informações de condições

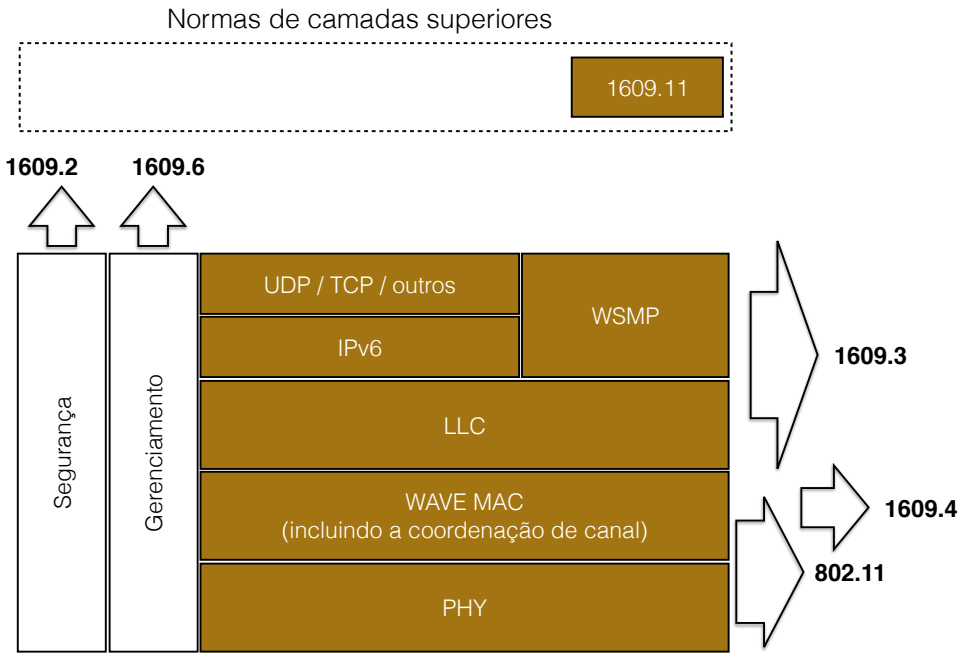

Figura 2.2: Arquitetura WAVE. Figura adaptada de [8].

de tráfego para unidades gestoras de trânsito ou informar diretamente veículos sobre as melhores rotas a serem escolhidas visando a redução dos congestionamentos e consequentemente a redução nos tempos de viagem.

2.3. Veins

2.3.1. Introdução

Veins é um *framework* de código aberto para a execução de simulações de redes veiculares. Baseia-se em dois simuladores bem estabelecidos: OMNeT++, um simulador de redes baseado em eventos discretos, e SUMO, um simulador de tráfego rodoviário. O Veins extende estes simuladores para oferecer um conjunto abrangente de modelos para simulação de comunicações veiculares (*Inter-vehicular communication* - IVC) [18].

Os simuladores OMNeT++ e Sumo são conectados através de um *socket* TCP para que a execução ocorra em paralelo. A comunicação é realizada seguindo um protocolo padronização chamado *Traffic Control Interface* (TraCI). Isto permite simulação bidirecionamente acoplada de tráfego rodoviário e de rede. Os movimentos dos veiculos no simulador de tráfego SUMO são refletidos em movimentos dos nós em uma simulação de rede no OMNeT++. Os nós podem interagir com a simulação de tráfego, como por exemplo, para simular a influência da comunicação veicular no tráfego rodoviário [19].

Além dos módulos para modelar e influenciar o tráfego rodoviário, o Veins oferece um conjunto abrangente de modelos IVC que podem servir como um *framework* para a simulação de aplicações [19]. A Figura 2.3 apresenta a estrutura modular do Veins.

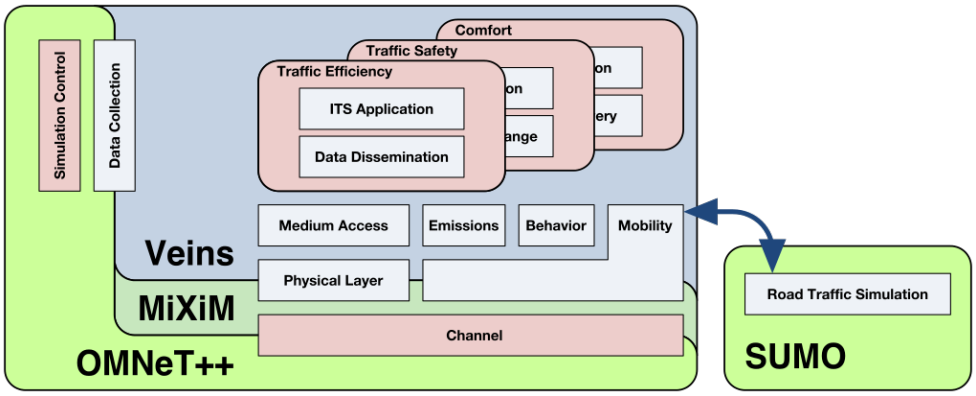

Figura 2.3: Estrutura modular do Veins. Figura retirada de [19].

O Veins inclui um modelo de propagação de informações chamado *Two-Ray Interference* que consegue captar os efeitos de reflexão do solo. Informações detalhadas sobre o modelo podem ser encontradas em [20] e [21].

Com o objetivo de capturar os efeitos de sombreamento de sinais (por exemplo, edifícios bloqueando a propagação de sinal), um Modelo de Sombreamento de Obstáculos é incluído no Veins. A calibração e a validação do modelo foi realizada comparando os resultados de simulações com medidas do mundo real, como ilustrado na Figura 2.4 [19]. Mais informações sobre o Modelo de Sombreamento de Obstáculos podem ser encontradas em [22] e [23].

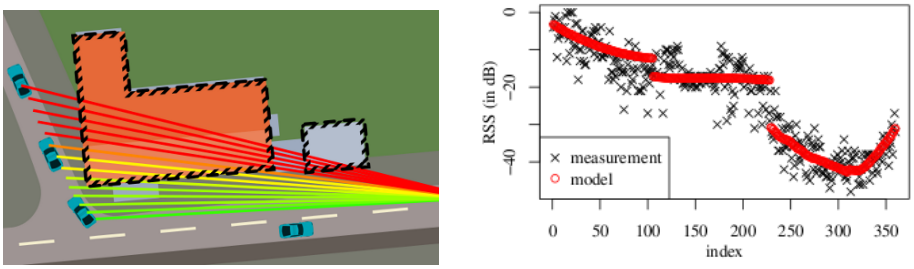

Figura 2.4: Validação do Modelo de Sombreamento de Obstáculos. Figura retirada de [19]

Uma extensão para o Veins, chamada de Veins LTE pode ser utilizada para a simulação de redes veiculares heterogêneas que utilizam, por exemplo, IEEE 802.11p, Wi-Fi e LTE. Mais informações podem ser encontradas em [24] e no *website* `http://floxyz.at/veins-lte/`.

Outra extensão, chamada Plexe, permite a simulação de sistemas automatizados de carros movimentando-se em pelotão. O Plexe está disponível no *website* `http://plexe.car2x.org`. Mais informações podem ser encontradas em [25].

2.3.2. Instalação

Esta seção tem por objetivo orientar e preparar o ambiente necessário para executar simulações de redes veiculares utilizando o Veins, OMNeT++ e SUMO. Como base de orientação para estes passos técnicos, o minicurso baseia-se no tutorial de instalação do site oficial do simulador Veins (`http://veins.car2x.org/tutorial/`).

O simulador Veins pode ser executado nos sistemas operacionais Linux, Mac OS X, e Windows. Por causa de muitos recursos de depuração que ele oferece, Veins é melhor construído e executado em Linux. Para construção no Linux, alguns pacotes precisam ser instalados. No Ubuntu (uma das distribuições Linux), poderá ser usado o comando abaixo para que os pacotes sejam devidamente intalá-los:

```
sudo apt-get install build-essential gcc g++ bison flex
perl tcl-dev tk-dev blt libxml2-dev zlib1g-dev default-jre
doxygen graphviz libwebkitgtk-1.0-0 openmpi-bin
libopenmpi-dev libpcap-dev autoconf automake libtool
libproj0 libgdal1-dev libfox-1.6-dev libgdal-dev
libxerces-c-dev
```

No sistema operacional Mac OS X, pacotes equivalentes deverão ser instalados via Macports através do comando abaixo:

```
sudo port install bison zlib tk blt libxml2 libtool xercesc
proj gdal fox
```

O guia de intalação do OMNeT++, que pode ser encontrado em `http://omnetpp.org/doc/omnetpp/InstallGuide.pdf`, tem dicas úteis sobre a pré e pós configuração mais específicas para cada sistema operacional.

Nas subseções seguintes será mostrado os passos necessários para preparar o ambiente que executará as simulações de redes veiculares, utilizando a IDE OMNeT++ 4, o simulador de tráfego SUMO e o framework Veins. Para a preparação de ambiente neste minicurso foi utilizado o sistema operacional Windows 7. O caminho *C:\Users\user* foi definido como diretório inicial, e todos os softwares necessários foram instalados em *C:\Users\user\src* para este exemplo. As etapas são semelhantes quando for construir e executar as simulações sobre os sistemas operacionais Linux ou Mac OS X, com exceção da janela de linha de comando do OMNeT ++ (MinGW) e dos caminhos de arquivos utilizados.

2.3.2.1. Instalar e Construir o SUMO

O download do simulador SUMO 0.21.0 deve ser feito no endereço `http://www.sumo.dlr.de/wiki/Downloads` e a pasta deverá ser descompactada em um diretório, por exemplo, *C:\Users\user\src\sumo-0.21.0*. Dentro dessa pasta que foi descompactada será possível encontrar o executável sumo.exe em *C:\Users\user\src\sumo-0.21.0\bin\sumo.exe*. Este executável será responsável por executar o simulador SUMO.

Note que versões mais recentes do SUMO binaries requerem a instalação do Microsoft Visual C++ 2010 Redistributable Package (x86), o qual o download pode ser feito através do link `http://www.microsoft.com/en-us/download/details.aspx?id=5555t`. Para uma construção manual do SUMO, instruções detalhadas para diferentes plataformas podem ser encontradas no site do SUMO em `http://www.sumo.dlr.de/wiki/Installing`.

2.3.2.2. Instalar e Construir o OMNeT++

O download do OMNet++ 4.4 para Windows deve ser feito no endereço `http://www.omnetpp.org/omnetpp` e a pasta deverá ser descompactada em um diretório, por exemplo, *C:\Users\user\src\omnetpp-4.4*. Se a descompactação for feita para um diretório diferente, certifique-se que este diretório não contenha espaços. Existe um script que pode ser encontrado no diretório *C:\Users\user\src\omnetpp-4.4\mingwenv.cmd*. Através deste script é possível rodar a janela de linha de comando MinGW, a qual é muito semelhante ao ambiente Linux. Para construir OMNet++ 4 digite o comando ./configure no MinGW, e em seguida digite o comando make para iniciar o processo de construção. Caso não ocorra erro algum, então */c/Users/user/src/omnetpp-4.4/bin/omnetpp* foi construído com sucesso. Finalmente, para executar a IDE OMNeT++ 4 deve ser executado o comando omnetpp. Ao abrir o OMNeT++ é necessário configurar um caminho para o workspace padrão. Neste minicurso foi escolhido o diretório C:\Users\user\src\omnetpp-4.4\samples como workspace.

2.3.2.3. Instalar e Construir o Veins

O download do Veins 3.0 deve ser feito no endereço `http://veins.car2x.org/download/` e a pasta deverá ser descompactada em um diretório, por exemplo, no diretório *C:\Users\user\src\veins-3.0*. Depois disto, o projeto deve ser importado para o workspace do OMNet++ IDE clicando em *File* > *Import* > *General: Existing Projects into Workspace* e então seleciona-se o diretório em que o Veins foi descompactado. Para construir o projeto recém importado deve-se clicar em *Project* > *Build All* no OMNet++ 4 IDE. Depois que o projeto estiver construído, então ele estará pronto para ser executado, mas para facilitar a depuração, o próximo passo será garantir que o SUMO está funcionando como deveria.

2.3.2.4. Verificar o SUMO

Na janela de linha de comando do OMNeT++ (MinGW), deve ser possível executar o SUMO para simular um cenário exemplo. Isso é possível através da alteração do diretório atual para */c/Users/user/src/veins-3.0/examples/veins/* com o seguinte comando:

```
cd ../veins-3.0/examples/veins
```

Em seguida executando o seguinte comando para iniciar o SUMO:

```
/c/Users/user/src/sumo-0.21.0/bin/sumo.exe -c
erlangen.sumo.cfg
```

Feito isso, uma linha com a mensagem *Loading configuration... done* aparece e em seguida um tempo de simulação é executado de 0 a 1000. Para uma visualização gráfica do cenário de exemplo, é possível também executá-lo usando sumo-gui.exe que é um modo alternativo do SUMO com interface gráfica, mas isso não é necessário para que o módulo Veins possa trabalhar.

2.3.2.5. Executar exemplo no Veins

Para evitar que seja necessário executar o SUMO previamente à toda simulação feita no OMNeT++, o módulo do Framework Veins traz junto aos seus arquivos um pequeno script para fazer isso automaticamente. Para realizar isto basta executar o comando no MinGW que foi mencionado anteriormente:

```
/c/Users/user/src/veins-3.0/sumo-launchd.py -vv -c
/c/Users/user/src/sumo-0.21.0/bin/sumo.exe
```

Este script abrirá conexões TCP entre o OMNeT++ e o SUMO, começando uma nova simulação no SUMO para cada conexão aberta de simulação no OMNeT++. O script estará executando na porta 9999 e esperando pelo o início de uma simulação. A janela do MinGW deverá ser mantida aberta enquanto executa o script. Para simular o cenário de exemplo do Veins na IDE OMNeT++ 4 basta clicar com o botão direito do mouse em *omnet.ini*, localizado no workspace em *veins-3.0/examples/veins/omnetpp.ini*, e escolher as opções *Run As* > *OMNeT++ simulation*. Depois de ser executado uma primeira vez assim, nas seguintes será possível re-executar esta configuração clicando no botão verde Run no OMNeT++. Se tudo funcionou como planejado, um cenário de simulação usando OMNeT++ estará trabalhando e o SUMO executando em paralelo para simular um fluxo de veículos que será posteriormente interrompido por um acidente.

Isto conclui a seção de Instalação e preparação de ambiente. Informações mais detalhadas sobre como usar o Veins estão disponíveis na documentação através do link `http://veins.car2x.org/documentation/`.

2.3.3. Primeiro experimento

2.3.3.1. Visão Geral dos Arquivos do SUMO

Esta seção tem como objetivo explicar sobre os principais arquivos de simulação do SUMO que são utilizados na criação de um projeto do simulador Veins. Para realizar uma simulação no Veins é necessário configurar os arquivos com as informações de tráfego. Essa configuração é realizada através da criação de arquivos dos seguintes formatos: *.nod.xml*, *.edg.xml*, *.net.xml*, *.rou.xml* e *.sumo.cfg*.

O arquivo de extensão .nod.xml tem as informações das coordenadas dos nós de interligação das vias. Através do código da Figura 2.5a pode-se observar um exemplo

simples deste tipo de arquivo. Neste arquivo (Exemplo.nod.xml) foram definidos 3 nós, onde cada nó possui um identificador e uma localização com coordenadas x e y que descrevem a distância da origem em metros.

```xml
<nodes>
    <node id="1" x="-250.0" y="0.0" />
    <node id="2" x="+250.0" y="0.0" />
    <node id="3" x="+251.0" y="0.0" />
</nodes>
```

(a) *Exemplo.nod.xml*

```xml
<edges>
    <edge from="1" id="1to2" to="2" />
    <edge from="2" id="out" to="3" />
</edges>
```

(b) *Exemplo.edg.xml*

```xml
<configuration>
    <input>
        <net-file value="hello.net.xml"/>
        <route-files value="hello.rou.xml"/>
    </input>
    <time>
        <begin value="0"/>
        <end value="10000"/>
    </time>
</configuration>
```

(c) *Exemplo.sumo.cfg*

```xml
<routes>
    <vType accel="1.0" decel="5.0" id="Car" length="2.0" maxSpeed="100.0" />
    <route id="route0" edges="1to2 out"/>
    <vehicle depart="1" id="veh0" route="route0" type="Car" />
</routes>
```

(d) *Exemplo.rou.xml*

Figura 2.5: Códigos dos arquivos **.xml*

O arquivo de extensão *.edg.xml* tem as informações das vias, sendo este arquivo responsável pela interligação entre os nós que foram configurados no arquivo *.nod.xml*. Através do código da Figura 2.5b pode-se observar um exemplo simples deste tipo de arquivo. Neste arquivo, (*Exemplo.edg.xml*) foram definidas duas interligações, onde cada um possui um identificador *id*, um nó de origem *from* e um nó de destino *to*.

O arquivo de extensão *.net.xml* é responsável pelas informações do mapa onde será realizado as simulações. Portanto, este arquivo reúne as informações de configuração contidas nos arquivos *Exemplo.nod.xml* e *Exemplo.edg.xml*. A partir desses últimos arquivos mencionados é possível criar o arquivo *Exemplo.net.xml* através da ferramenta ***netconvert*** que pode ser encontrada na pasta *bin*, no diretório do SUMO. O seguinte comando no prompt é utilizado para criar o arquivo *Exemplo.net.xml*:

```
netconvert -node-files=Exemplo.nod.xml
-edge-files=Exemplo.edg.xml -output-file=Exemplo.net.xml
```

O arquivo de extensão *.net.xml* também pode ser gerado a partir de um mapa real. Este mapa pode ser importado do site `www.openstreetmap.org`, então basta salvalo no computador como um arquivo de extensão *.osm*. Depois disso, para criar o arquivo *Exemplo.net.xml* é necessário executar o seguinte comando no prompt:

```
netconvert -osm-files Mapa.osm -output-file Exemplo.net.xml
```

O arquivo de extensão *.rou.xml* reúne as informações dos dados básicos dos veículos (comprimento, aceleração, velocidade máxima, etc.), as rotas a serem seguidas pelos veículos e o lançamento nas vias. Através do código da Figura 2.5d pode-se observar um exemplo simples deste tipo de arquivo. Neste arquivo (*Exemplo.rou.xml*) está sendo definido um tipo de veículo com um identificador igual a *Car*, aceleração igual a 1 m/s^2, desaceleração igual 5 m/s^2, comprimento igual 2 metros e máxima velocidade de 100 m/s. Uma rota está configurada com identificador igual a *route0* e com percurso iniciando na via *1to2* e seguindo para via *out*. Ao final é definido que o veículo de identificador *veh0* do tipo *Car* fará o percurso da rota *route0* partindo após 1 segundo de simulação.

O arquivo de extensão *.sumo.cfg* contém a configuração da simulação composta pelos arquivos anteriores. Através do código da Figura 2.5c pode-se observar um exemplo simples deste tipo de arquivo. Neste arquivo (*Exemplo.sumo.cfg*) tem-se como entrada os arquivos *Exemplo.net.xml* e *Exemplo.rou.xml* para criação do mapa e das rotas, respectivamente. O tempo de simulação está configurado para 10.000 segundos de duração.

Após a criação de todos esses arquivos, e se todos estiverem corretamente configurados, será possível checar a simulação do tráfego no sumo através do seguinte comando no prompt:

```
sumo -c hello.sumocfg
```

Ou então através deste outro comando para utilizar a interface gráfica do SUMO:

```
sumo-gui -c hello.sumocfg
```

Vale ressaltar que existem muitos outros possíveis parâmetros de configuração para os arquivos que foram mencionados e que também existem outros tipos de arquivos. Um exemplo desses arquivos são os de extensão *.poly.xml* que são responsáveis pela coloração de prédios e construções nos mapas *.osm* importados e gerados para o simulador SUMO. Estes arquivos não são obrigatórios para executar as simulações, portanto este minicurso restringiu-se à configuração dos principais arquivos. Para mais informações sobre os diferentes tipos de arquivos e suas respectivas configurações é imprescindível a leitura da documentação do simulador SUMO que pode ser acessada na pasta *docs/userdoc* localizada no diretório do SUMO.

2.3.3.2. Cenário de simulação

Para um primeiro experimento com redes veiculares foi utilizado o exemplo executado no final do processo de instalação, presente na Seção 2.3.2.5. Este exemplo possui um cenário que simula um fluxo de veículos que será posteriormente interrompido por um acidente. Após o acidente será disseminado uma mensagem na rede por inundação (flooding). Com isso, os outros veículos serão alertados que ocorreu uma obstrução daquela via e então os próximos veículos evitaram aquele trecho.

Este cenário é interessante e importante para um primeiro experimento em redes veiculares porque apesar de ser um cenário simples ele reúne três pontos principais: detecção, sinalização e recepção de eventos. A detecção de eventos é realizada através de sensores do veículo, sendo estes responsáveis por detectar, por exemplo, os acidentes. Logo após a detecção de um evento, o veículo transmite mensagens para os demais veículos, assim sinalizando este evento. Por fim, as mensagens que sinalizam determinado evento são recebidas pelos veículos mais próximos, e assim estes poderão escolher rotas alternativas para continuar seu trajeto e será possível evitar que aumente o congestionamento daquela via.

2.3.3.3. Configuração do Projeto

O projeto que foi importado para o workspace do OMNeT++ 4.4 possui algumas pastas e arquivos. Essas pastas contêm arquivos de documentação do simulador Veins, arquivos de funcionamento do próprio simulador Veins e possui também uma pasta com um exemplo. Este exemplo será explicado neste minicurso para facilitar um primeiro contato com experimentos em redes veiculares, utilizando o simulador Veins, o simulador de tráfego SUMO e a IDE OMNeT++.

O funcionamento dos arquivos do exemplo será explicado nesta seção. Estes arquivos podem ser acessados na árvore do projeto no workspace em *veins/examples/veins*. Dentro desta pasta pode-se localizar os arquivos *erlangen.net.xml*, *erlangen.rou.xml*, *erlangen.poly.xml* e *erlangen.sumo.cfg* que são os tipos de arquivos de configuração de tráfego do SUMO.

Neste exemplo, o arquivo *erlagen.net.xml* foi obtido através do mapa da cidade Erlagen, localizada na Alemanha. A Figura 2.6a mostra o mapa desta cidade. Este mapa foi extraído a partir do site `www.openstreetmap.org`, posteriormente preparado e importado para o SUMO. O arquivo *erlangen.poly.xml* foi o responsável pela coloração dos prédios do novo mapa gerado. Este mapa pode ser observado na Figura 2.6b. Os arquivos de extensão *.poly.xml* não são necessários, mas sem eles o mapa gerado pelo arquivo *.net.xml* apresentará apenas as vias.

(a) *Mapa no OpenStreetMap* (b) *Mapa no SUMO*

Figura 2.6: Mapas da cidade Erlagen, Alemanha.

No arquivo *erlagen.rou.xml* está definido um fluxo periódico de 195 veículos que

saem de um ponto localizado na direita do mapa com direção ao lado esquerdo do mapa. Estes veículos foram configurados com aceleração de 2,6m/s^2, desaceleração de 4,5 m/s^2, comprimento de 2,5 metros, velocidade máxima de 14 m/s e cor amarela. Por fim, o arquivo *erlagen.sumo.cfg* foi configurado com os arquivos de entrada: *erlangen.net.xml*, *erlangen.rou.xml*, *erlangen.poly.xml* e o tempo de simulação foi definido com uma duração de 1000 segundos.

O arquivo *omnetpp.ini* é responsável pelos parâmetros da simulação como, por exemplo, o tempo de simulação, configurações da RSU, parâmetros específicos do padrão 802.11p, entre outros. Neste arquivo, também é possível definir dados sobre um acidente. No arquivo *omnetpp.ini* do exemplo desse projeto, um acidente foi configurado para que acontecesse com o primeiro carro da simulação no tempo 75 seg. e que tivesse uma duração de 30 segundos. Estas configurações podem ser observadas nas linhas seguintes deste arquivo:

```
*.node[*0].veinsmobility.accidentCount = 1
*.node[*0].veinsmobility.accidentStart = 75s
*.node[*0].veinsmobility.accidentDuration = 30s
```

Além dos arquivos de configuração do SUMO e do Veins, existe uma pasta de resultados com nome *"results"*. Dentro desta pasta estão os arquivos *nodebug-0.sca*, *nodebug-0.vec* e *nodebug-0.vci* que são gerados a cada simulação. Estes arquivos registram alguns dados da simulação como, por exemplo, a velocidade máxima atingida por um determinado veículo. A partir dos dados registrados nestes arquivos é possível criar um arquivo de análise que possui extensão .anf, e através deste arquivo é possível gerar e analisar gráficos com os resultados da simulação. A Figura 2.7 exemplifica o gráfico de velocidade de um carro da simulação durante seu percurso.

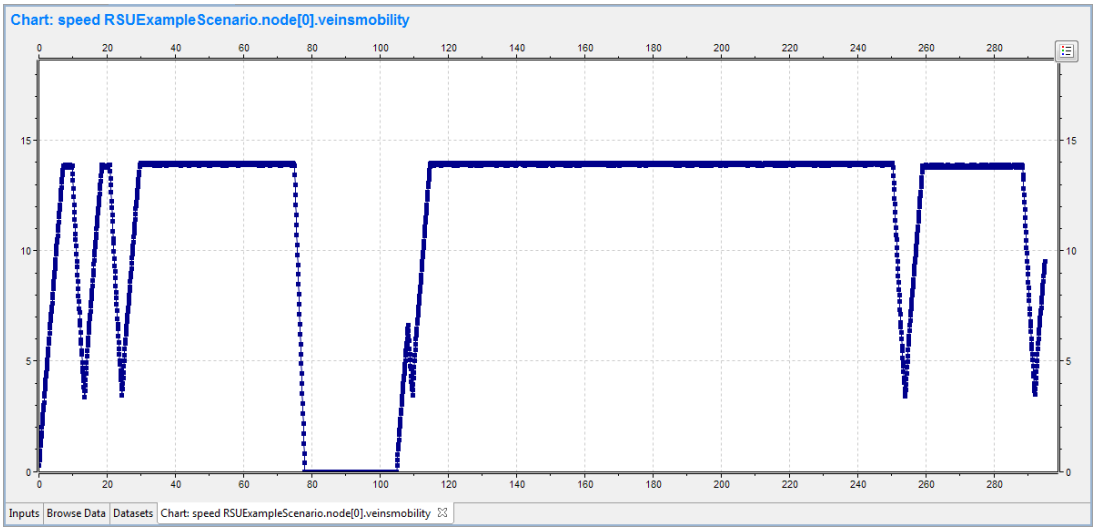

Figura 2.7: Gráfico da velocidade do primeiro carro do fluxo

Para gerar um arquivo *.anf* é necessário dar dois cliques no arquivo *.sca* ou no *.vec*, que se encontram no workspace do OMNeT++, e então será possível salvar este

novo arquivo na mesma pasta que os outros dois se encontram. Após ter feito isso basta abrir o novo arquivo, selecionar os filtros desejados e então com um clique com o botão direito sobre um ou mais veículos selecionados é possível gerar um gráfico através da opção *"plot"*.

2.3.4. Implementação de um Novo Protocolo

Nesta seção será apresentado um tutorial passo-a-passo de como implementar um protocolo simples para redes veiculares utilizando as ferramentas apresentadas. O protocolo em questão tem como foco atacar o problema do monitoramento e da engenharia de tráfego. O principal objetivo é reduzir os congestionamentos de veiculos utilizando apenas a arquitetura V2V, ou seja, não é necessária a presença de RSUs para o seu funcionamento. Com o protocolo proposto, cada veículo irá ter uma visão geral sobre quais as médias de velocidades dos veículos em todas as vias do cenário e assim poderá escolher a melhor rota para alcançar o seu destino. Nesta seção, inicialmente será apresentada a descrição do protocolo e em seguida será apresentado um tutorial de como implementar o protocolo utilizando as ferramentas de simulação apresentadas.

O protocolo proposto funciona da seguinte maneira. Os veiculos transmitem sua velocidade corrente V_c para seus vizinhos através dos *beacons* com uma frequência de 1Ghz. Cada veículo mantém uma estrutura de dados com todos os valores de velocidades recebidos de seus vizinhos (veículos que estão na mesma via e no seu raio de comunicação) nos últimos 5 segundos bem como as suas últimas 5 medidas de velocidade corrente (cada medida é realizada a cada 1 segundo). A cada 5 segundos o veículo calcula a média V_m de todas as velocidades salvas nesta estrutura de dados. Se V_m for menor ou igual a um limiar V_l , o veículo irá verificar se deve ou não transmitir em *broadcast multi-hop* uma mensagem do tipo ***CongestionMessage*** que possui o valor de V_m e o identificador $Road_{id}$ da via atual. Esta mensagem só será transmitida se o veículo não tiver recebido ou transmitido outra mensagem do tipo ***CongestionMessage*** com o mesmo $Road_{id}$ da sua via atual nos últimos 5 segundos.

Cada veículo mantém uma tabela chamada ***CongestionTable*** com a média de todos os valores de V_m recebidos através de mensagens do tipo ***CongestionMessage*** nos últimos 20 segundos para cada $Road_{id}$. Esta média salva na ***CongestionTable*** para cada $Road_{id}$ é chamado de V_{avg}. Caso o veículo não tenha recebido qualquer ***CongestionMessage*** nos últimos 20 segundos para determinada via, o valor de V_{avg} desta via na ***CongestionTable*** será igual à velocidade máxima V_{maxID} permitida na via.

Sempre que um veículo recebe uma nova ***CongestionMessage*** e assim atualiza a sua ***CongestionTable***, sua rota será recalculada da seguinte forma. Será atribuido um peso para cada via através da Equação 1, onde P é o peso atribuído para a via, L_{id} é comprimento em metros da via e V_{avg} é o valor de velocidade média salvo na ***CongestionTable*** para a via. Se V_{avg} for igual a 0, P será igual 1000000.

$$P = \frac{L_{id}}{V_{avg}} \tag{1}$$

Calculados os pesos de todas as vias, o veículo poderá recalcular sua rota através do algoritmo de menor caminho de Dijkstra.

Para implementar o protocolo, inicialmente é necessário definir o cenário da simulação. O mapa que será criado no SUMO será um Manhattan Grid (5x5) sob uma região de 1 Km^2. A Figura 2.8 apresenta o mapa do cenário de avaliação. Cada segmento de via possui extensão de 250 metros, duas faixas em cada direção e limite de velocidade de 50 km/h. Esse tipo de mapa é comumente utilizado em redes veiculares para avaliar protocolos em ambientes urbanos. Vale ressaltar que a versão do *Veins* utilizada nesta seção foi a 2.1.

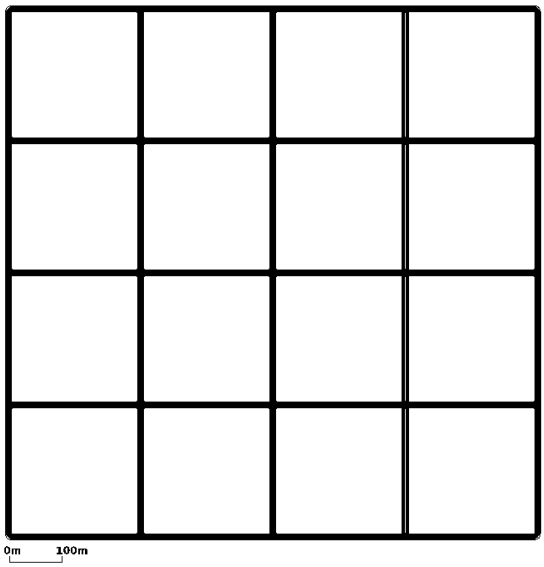

Figura 2.8: Cenário da simulação.

O SUMO possui uma série de ferramentas para criação de mapas. Será utilizado o ***netgenerate*** que pode ser encontrado dentro da pasta *bin*, no diretório do SUMO. O seguinte comando é utilizado para criar o mapa com os parâmetros especificados:

```
./netgenerate -grid -grid.number=5 -grid.length=250
-default.lanenumber=2 -output-file=MyNet.net.xml
```

Com este comando será criado um arquivo chamado *MyNet.net.xml* na pasta onde encontra-se o arquivo *netgenerate*. Este arquivo pode ser lido pelo SUMO e representa o mapa de simulação. Mais informações sobre o comando *netgenerate* podem ser encontradas no *website* `http://sumo.dlr.de/wiki/NETGENERATE`.

Serão geradas 3000 pares aleatórias de origem e destino com a distância mínima de 1 km. Para cada par uma rota será criada composta pela sequência de vias que representam o menor caminho que conectam esse par. Para criar os pares de origem e destino utilizamos o *script* escrito na linguagem python chamado ***randomTrips.py***. Este *script* pode ser encontrado na pasta */Tools/trip*, dentro do diretório do SUMO. O comando a seguir é utilizado para a criação dos pares origem e destino como especificado:

```
./randomTrips.py -net-file=MyNet.net.xml -min-distance=1000
-begin=0 -end=3000 -route-file=MyRoutes-tmp.rou.xml
```

Com isso serão criados os arquivos *trips.trip.xml*, *MyRoutes-tmp.rou.alt.xml* e *MyRoutes-tmp.rou.xml* na mesma pasta onde encontra-se o escript *randomTrips.py*. Será utilizado apenas o arquivo *MyRoutes-tmp.rou.xml*.

Será necessário editar o arquivo *MyRoutes-tmp.rou.xml* para adequá-lo para a simulação. O código na Figura 2.9 escrito na linguagem C pode ser utilizado com esta finalidade. Este código também é responsável por inserir no arquivo de rotas o tipo de carro que será utilizado. Assim será configurada a velocidade máxima de cada veículo como 14 metros por segundo, o que representa aproximadamente 50km/h. Basta compilar o código e executá-lo na mesma pasta onde encontra-se o arquivo *MyRoutes-tmp.rou.xml*. Com isso será criado um arquivo chamado ***MyRoutes.rou.xml***. Este é o arquivo final que possui as 3000 rotas geradas aleatoriamente.

```
#include <sstream>
#include <string>
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    freopen("MyRoutes-tmp.rou.xml","r",stdin);  //Arquivo de entrada gerado
                                                //com o script randomTrips.py
    string line;
    ofstream output;
    output.open("MyRoutes.rou.xml");    //Arquivo que será criado com todas as
                                        //rotas e a definição do tipo de veículo
    output << "<routes>\n\t<vType id=\"Car\" maxSpeed=\"14.0\"/>\n";    //Escrita da definição do tipo
                                                                       //de veículo no arquivo de saída
    int count=0;
    while (getline(cin,line))
    {
        if (line.compare(0,15,"        <route ") == 0)  //Edita cada linha do arquivo de entrada que
        {                                               //representam rotas
            string to = "\t<route id=\"";
            to += "route"+to_string(count++);
            to += "\"";
            to += line.substr(14) + "\n";
            output << to; //Escrita da rota no arquivo de saída
        }
    }

    output << "</routes>"; //Finalização do arquivo de saída

    return 0;
}
```

Figura 2.9: Código utilizado para editar o arquivo de rotas.

Será necessário a criação de mais dois arquivos relacionados ao cenário. Crie dois arquivos de texto com os nomes ***MySimulation.launchd.xml*** e ***MySimulation.sumo.cfg***. O arquivo *MySimulation.launchd.xml* é utilizado pelo *Veins* para detectar os arquivos que definem o cenário no SUMO. Já o arquivo *MySimulation.sumo.cfg* é um dos arquivos necessários para simulação de tráfego no SUMO. Edite os arquivos *MySimulation.launchd.xml* e *MySimulation.sumo.cfg* com os códigos apresentados nas Figuras 2.10a e 2.10b, respectivamente. O tempo de simulação é definido no arquivo *MySimulation.sumo.cfg*, neste caso, 600 segundos.

Agora copie os 4 arquivos criados (*MyNet.net.xml*, *MyRoutes.rou.xml*, *MySimulation.launchd.xml* e *MySimulation.sumo.cfg*) para a pasta */examples/veins* que está localizada no diretório do projeto veins, que por sua vez foi adicionado ao OMNeT++ (mais informações sobre como adicionar o veins ao OMNeT++ podem ser encontradas na seção 2.3.2).

Para finalizar esta etapa, é preciso indicar ao Veins quais são os arquivos de con-

```
<?xml version="1.0"?>
<!-- debug config -->
<launch>
     <copy file="MyNet.net.xml" />
     <copy file="MyRoutes.rou.xml" />
     <copy file="MySimulation.sumo.cfg" type="config" />
</launch>
```

(a) *MySimulation.launchd.xml*

```
<configuration>
    <input>
        <net-file value="MyNet.net.xml"/>
     <route-files value="MyRoutes.rou.xml"/>
    </input>
    <time>
        <begin value="0"/>
        <end value="600" />
     <step-length value="0.1"/>
    </time>
</configuration>
```

(b) *MySimulation.sumo.cfg*

Figura 2.10: Texto para ser inserido nos arquivos *MySimulation.launchd.xml* e *MySimulation.sumo.cfg*

figuração que o SUMO irá utilizar. Isso é feito editando o arquivo ***omnetpp.ini*** que está localizado na pasta */examples/veins*. As seguintes modificações serão feitas no arquivo *omnetpp.ini*:

1. Modifique o tempo de simulação.

   Substituir `sim-time-limit = 6000s` por `sim-time-limit = 600s`

2. Modifique o tamanho do mapa. A adição de 100 metros no tamanho do mapa é feita para evitar erros durante a simulação.

   Substituir `*.playgroundSizeX = 2500m`

   por `*.playgroundSizeX = 1100m`

   Substituir `*.playgroundSizeY = 2500m`

   por `*.playgroundSizeY = 1100m`

3. Informe os arquivos do SUMO

   Substituir

   `*.manager.launchConfig = xmldoc("erlangen.launchd.xml")`

   por

   `*.manager.launchConfig = xmldoc("MySimulation.launchd.xml")`

4. Remova a RSU que vem no exemplo do Veins

   Remover a linha `*.rsu[0].mobility.x = 2000`

   Remover a linha `*.rsu[0].mobility.y = 2000`

   Remover a linha `*.rsu[0].mobility.z = 3`

5. Modifique a potência do rádio dos veículos para 1.3mW. Isso representa um raio de comunicação de aproximadamente 250 metros.

   Substituir `*.**.nic.mac1609_4.txPower = 20mW`

   por `*.**.nic.mac1609_4.txPower = 1.3mW`

6. Ative os *beacons* dos veículos

   Substituir `*.node[*].appl.sendBeacons = false`

   por `*.node[*].appl.sendBeacons = true`

7. Remova o acidente que vem no exemplo do Veins

   Substituir `*.node[*0].veinsmobility.accidentCount = 1`

   por `*.node[*0].veinsmobility.accidentCount = 0`

Antes de começar a codificação do novo protocolo, ainda é necessário remover trechos de códigos relacionados ao exemplo que vem na instalação do *Veins*, já que os mesmos arquivos do exemplo serão utilizados para a implementação do novo protocolo.

1. Remover o módulo responsável pela RSU no arquivo *RSUExampleScenario.ned*, que pode ser encontrado na pasta *examples/veins*.

   Remover as linhas que possuem o código

   ```
   rsu[1]:  RSU {
    @display("p=150,140;b=10,10,oval");
   }
   ```

2. Editar os arquivos *TraCIDemo11p.h* e *TraCIDemo11p.h*. Estes são os arquivos da camada de aplicação. A maior parte do protocolo será implementada nele.

   Removar todas as linhas de código que mencionam a variável `sentMessage` nos dois arquivos.

   Edite o método *void TraCIDemo11p::handlePositionUpdate(cObject* obj)*. Mantenha apenas a seguinte linha de código e apague as demais:

   ```
   BaseWaveApplLayer::handlePositionUpdate(obj);
   ```

   No método *void TraCIDemo11p::onData(WaveShortMessage* wsm)*, apague a seguinte linha de código do método:

   ```
   if (traci->getRoadId()[0] != ':')
   traci->commandChangeRoute(wsm->getWsmData(), 9999);
   ```

3. No método *void TraCIDemo11p::handlePositionUpdate*, apague todas as linhas e mantenha apenas a linha a seguir.

   ```
   BaseWaveApplLayer::handlePositionUpdate(obj);
   ```

Para finalizar, será informado para o *Ve*ins a quantidade de veiculos que deve ser inserido na simulação. O *Veins* ficará responsável por manter esta quantidade fixa, inserindo veículos sempre que necessário selecionando uma das rotas aleatórias que foram criadas anteriormente. Neste exemplo a quantidade de veículos será fixada em 200. Isso é feito adicionando o parâmetro *manager.numVehicles* do *TraCIScenarioManager*. Para isso basta adicionar a seguinte linha de código no arquivo *omnetpp.ini*:

```
*.manager.numVehicles = 200
```

É indicado que esta linha seja adicionado junto às linhas que representam os parâmetros do *TraCIScenarioManager*. Feito isso, a simulação está pronta para ser executada, mas sem o protocolo proposto para redução de congestionamentos. Neste momento poderá ser realizada uma simulação. Para isso basta compilar o projeto e executá-lo. Recomenda-se que seja desligado os beacons dos veículos nesta simulação, já que eles não irão ser utilizados neste momento e isso irá diminuir bastante a duração da execução da simulação. Para isso basta substituir a linha `*.node[*].appl.sendBeacons = true` por `*.node[*].appl.sendBeacons = false` no arquivo *omnetpp.ini*. Não esqueça de executar também o script *sumo-launchd.py*, como apresentado na seção 2.3.2. Ao final da simulação, será criada uma pasta chamada *results* com os resultados da simulação, dentro da pasta */examples/veins*. Copie esta pasta com todos os arquivos presentes nela para um outro local. Assim será possível comparar os resultados da simulação sem o protocolo para redução de congestionamentos com os resultados gerados após a implementação do protocolo. Também é indicado que a simulação seja executada no modo *Express* para que o tempo de simulação seja menor.

A primeira etapa para a implementação do protocolo em sí é fazer com que os veículos transmitam o identificador da sua via atual e a sua velocidade corrente através de seus beacons. Para isso, siga os seguintes passos:

1. Declare as variáveis a seguir no arquivo *WaveShortMessage.msg* que fica localizado na pasta */src/modules/messages/*. Este arquivo representa os pacotes que são transmitidos, incluindo os *beacons*.

   ```
   string roadId = "";
   double senderSpeed = 0.0;
   ```

2. Insira a velocidade e via atual do veículo no *beacon*. Para isso faça as seguintes modificações no arquivo *BaseWaveApplLayer.cc*, que está localizado na pasta */src/modules/application/ieee80211p*:

   Adicione as seguintes linhas no início do arquivo:

   ```
   #include "modules/mobility/traci/TraCIMobility.h"
   using Veins::TraCIMobilityAccess;
   ```

   No método *BaseWaveApplLayer::prepareWSM*, dentro do bloco de *if (name == "beacon")*, adicione as seguintes linhas:

   ```
   wsm->setRoadId(TraCIMobilityAccess().get(getParentModule())
   ->getRoadId().c_str());
   wsm->setSenderSpeed(TraCIMobilityAccess().
   get(getParentModule())->getSpeed());
   ```

Com isso os todos os veículos irão transmitir o identificador da sua via atual *road*Id e a sua velocidade atual *senderSpeed* através dos seus *beacons*.

O próximo passo é declarar as variáveis que irão representar a estrutura de dados que guarda as velocidades recebidas dos vizinhos e a *CongestionTable*. Estas variáveis foram definidas na descrição do protocolo, no início desta seção. A maior parte do protocolo será implementado na camada de aplicação, representada pelos arquivos ***TraCIDemo11p.h*** e ***TraCIDemo11p.cc***. Faça as seguintes alterações no arquivo *TraCIDemo11p.h*:

1. Adicione a seguinte linha após a linha *using Veins::AnnotationManager;*:

   ```
   using namespace std;
   ```

2. Declare a estrutura de dados com as velocidades recebidas dos vizinhos adicionando a seguinte variável como *protected* à classe *TraCIDemo11p*:

   ```
   vector<pair<simtime_t, double> > speedsList;
   ```

3. Declare a *CongestionTable* adicionando a seguinte variável como *protected* à classe *TraCIDemo11p* (mapa onde a chave é o identificador da via e o valor é um vetor com os valores das velocidades recebidas para aquela via e o tempo em que esta velocidade foi recebida):

   ```
   map<string, vector<pair<simtime_t, double> > >
   congestionTable;
   ```

4. Adicione as seguintes variáveis, também como *protected*. ***lastSent*** representa o tempo a última mensagem do tipo *CongestionMessage* foi recebida ou transmitida. ***updateTablesEvt*** é utilizada para a cada 5 segundos atualizar a *CongestionTable* e ***insertCurrentSpeedEvt*** é utilizada para a cada 1 segundo adicionar a velocidade atual do veículo à *speedsList*. ***sentMessages*** identifica todas as *congestionMessage* já recebidas pelo veículo e é utilizada pelo algoritmo de *broadcast multi-hop*. No algoritmo, sempre que um veículo receber uma *congestionMessage*, será verificado se esta é a primeira vez que ele recebe esta mensagem através do seu identificador. Caso seja a primeira vez, o veículo irá retransmitir a mensagem e adicionar o seu identificador à variável *sentMessages*. Por fim, ***messageId*** é uma variável de classe (comum a todos os veículos inserios) que é utilizada para criar os identificadores únicos de cada *congestionMessage*.

   ```
   simtime_t lastSent;
   cMessage* updateTablesEvt;
   cMessage* insertCurrentSpeedEvt;
   std::map<int,bool> sentMessages;
   static int messageId;
   ```

5. Adicione os seguintes métodos como protected. ***handleSelfMsg*** é o método que será disparado a cada 5 segundos e a cada 1 segundo para atualizar as estruturas de dados e verificar se o veículo deve enviar uma *congestionMessage*. Os demais métodos possuem os nomes auto-explicativos.

   ```
   virtual void handleSelfMsg(cMessage* msg);
   ```

```
void updateSpeedList();
void updateCongestionTable();
void verifyAndSendCongestionMessage();
```

6. Para finalizar, adicione a seguinte linha no final do arquivo, antes da linha *#endif* para inicializar a variável *messageId*.

```
int TraCIDemo11p::messageId = 0;
```

Para finalizar os arquivos da camada de aplicação, faça as seguintes alterações no arquivo *TraCIDemo11p.cc*:

1. Inicialize as variáveis adicionando as seguintes linhas dentro do bloco *if (stage == 0)*, no método *void TraCIDemo11p::initialize(int stage)*.

```
lastSent = simTime();
updateTablesEvt = new cMessage("updateTables");
scheduleAt(simTime() + 5, updateTablesEvt);
insertCurrentSpeedEvt = new cMessage("insertCurrentSpeed");
scheduleAt(simTime() + 1, insertCurrentSpeedEvt);;
```

2. Implemente os métodos *handleSelfMsg, updateSpeedList, updateCongestionTable e verifyAndSendCongestionMessage*. Os códigos podem ser encontrados nas Figuras 2.11a, 2.11b, 2.11c e 2.11d, respectivamente.

3. Edite o método *onBeacon*. Este método é invocado sempre que o veículo recebe um *beacon*. Apague todas as linhas presentes no método e adicione as linhas a seguir.

```
if (traci->getRoadId().compare(wsm->getRoadId()) == 0)
 speedsList.push_back(make_pair(wsm->getArrivalTime(),
 wsm->getSenderSpeed()));
```

4. Para finalizar, edite o método *onData* para que ele fique conforme a Figura 2.12. Este método é invocado sempre que o veículo recebe um pacote de dados (no protocolo proposto, ele será invocado apenas quando o veículo receber uma *congestionMe*ssage).

Para finalizar, adicione o seguinte método no arquivo *TraCIMobility.h*:

```
double commandGetLaneLength(std::string laneId){ return
getCommandInterface()->getLaneLength(laneId); }
```

Este método possui a implementação de uma interface para interação com o traci (envio e recebido de métodos do SUMO).

Finalizada a implementação do protocolo, o projeto pode ser compilado e executado. É importante não esquecer de ativar os *beacons* no arquivo *omnetpp.ini* para que o protocolo funcione, bem como realizar as simulações no modo *express*. Ao termino da

```
void TraCIDemo11p::handleSelfMsg(cMessage* msg) {
    if (msg == updateTablesEvt) {
        updateSpeedList();
        updateCongestionTable();
        verifyAndSendCongestionMessage();
        scheduleAt(simTime() + 5.0, updateTablesEvt);
    } else if (msg == updateTablesEvt) {
        speedsList.push_back(make_pair(simTime(), traci->getSpeed()));
        congestionTable[traci->getRoadId()].push_back(make_pair(simTime(), traci->getSpeed()));
        scheduleAt(simTime() + 1.0, insertCurrentSpeedEvt);
    } else {
        BaseWaveApplLayer::handleSelfMsg(msg);
    }
}
```

(a) Implementação do método *handleSelfMsg*

```
void TraCIDemo11p::updateSpeedList() {
    vector<pair<simtime_t, double> > newSpeedsList;

    for(int i = 0; i < speedsList.size(); ++i)
        if (simTime() - speedsList[i].first < 5.0)
            newSpeedsList.push_back(speedsList[i]);

    speedsList.clear();
    speedsList.insert(speedsList.begin(), newSpeedsList.begin(), newSpeedsList.end());
}
```

(b) Implementação do método *updateSpeedList*

```
void TraCIDemo11p::updateCongestionTable() {
    map<string, vector<pair<simtime_t, double> > > newMap;
    for(map<string, vector<pair<simtime_t, double> > >::iterator it = congestionTable.begin(); it != congestionTable.end(); ++it) {
        vector<pair<simtime_t, double> > newList;
        for(int i = 0; i < it->second.size(); ++i)
            if (simTime() - it->second[i].first < 20.0)
                newList.push_back(it->second[i]);
        newMap[it->first] = newList;
    }
    congestionTable.clear();
    congestionTable.insert(newMap.begin(),newMap.end());
}
```

(c) Implementação do método *updateCongestionTable*

```
void TraCIDemo11p::verifyAndSendCongestionMessage() {
    if (simTime() - lastSent < 5.0)
        return;
    if (speedsList.size() == 0)
        return;
    double sum = 0;
    for(int i = 0; i < speedsList.size(); ++i)
        sum += speedsList[i].second;
    double avg = sum / speedsList.size();
    if (avg < 5.0) {
        string toSend = "";
        toSend += traci->getRoadId() + "|" + to_string(avg) + "|" + to_string(++TraCIDemo11p::messageId);
        lastSent = simTime();
        sendMessage(toSend);
    }

    double length = traci->commandGetLaneLength(traci->commandGetLaneId());
    if (length > 0)
        traci->commandChangeRoute(traci->getRoadId(),avg / length);
}
```

(d) Implementação do método *verifyAndSendCongestionMessage*

Figura 2.11: Implementações dos métodos no arquivo *TraCIDemo11p.cc*

```
void TraCIDemo11p::onData(WaveShortMessage* wsm) {
    findHost()->getDisplayString().updateWith("r=16,green");
    annotations->scheduleErase(1, annotations->drawLine(wsm->getSenderPos(), traci->getPositionAt(simTime()), "blue"));
    string data(wsm->getWsmData());
    string road, speed,messageIdentifier;
    int i = 0;
    while(i < data.size() && data[i] != '|') {
        road.push_back(data[i]);
        ++i;
    }
    ++i;
    while(i < data.size()) {
        speed.push_back(data[i]);
        ++i;
    }
    ++i;
    int msgId = atoi(messageIdentifier.c_str());
    if (sentMessages.count(msgId) == 0) {
        sentMessages[msgId] = true;
        sendMessage(wsm->getWsmData());
    }
    while(i < data.size()) {
        messageIdentifier.push_back(data[i]);
        ++i;
    }
    congestionTable[road].push_back(make_pair(simTime(),stod(speed,0)));
    if (road.compare(traci->getRoadId()) == 0)
        lastSent = simTime();
    vector<pair<simtime_t, double> > speeds = congestionTable[road];
    double sum = 0;
    for(int i = 0; i < speeds.size(); ++i)
        sum += speeds[i].second;
    double avg = sum / speeds.size();
    double length = traci->commandGetLaneLength(traci->commandGetLaneId());
    if (length > 0)
        traci->commandChangeRoute(road,avg / length);
}
```

Figura 2.12: Implementação do método *onData*

simulação, a pasta *results* será novamente criada com os resultados da simulação. Será possível então comparar os resultados desta simulação com os da simulação que não fez o uso do protocolo e, por exemplo, verificar se o tempo médio de viagem dos veículos realmente diminuiu com o uso do protocolo.

2.4. Considerações Finais

As redes veiculares representam um avanço tecnológico que pode reduzir os acidentes no trânsito e desperdícios de tempo graças às aplicações que rodam sobre os veículos e as infraestruturas localizadas à margem das vias, sendo gerenciadas de forma eficaz e em tempo real. Como foi mencionado em seções anteriores existem diversas aplicações de diferentes categorias e utilidades, portanto é importante o estudo desse novo cenário presente em redes de computadores.

Este minicurso abordou o âmbito de redes veiculares e um de seus ambientes de simulação, utilizando a IDE OMNeT++ e os simuladores Veins e SUMO. Inicialmente, este minicurso comentou sobre os padrões em redes veiculares e as suas aplicações, objetivando uma introdução aos principais conceitos e o uso de redes veiculares na atualidade. Nas seções posteriores foi demonstrado como preparar um ambiente de simulação para redes veiculares e em seguida os exemplos utilizados foram explicados detalhadamente.

Referências

[1] R. S. de Sousa and A. C. B. Soares, “Estimativa e sinalização de congestionamentos de tr’afego atrav’es de redes veiculares v2v,” in *XXXIII Simpósio Brasileiro de Redes*

de Computadores e Sistemas Distribuídos, Vitória, 2015.

[2] R. dos S. Alves, I. do V. Campbell, and R. de S. Couto, *Redes Veiculares: Princípios, Aplicações e Desafios*. Sociedade Brasileira de Computação, Maio 2009, ch. 5, pp. 199–254.

[3] A. Varga and R. Hornig, "An overview of the omnet++ simulation environment," in *Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, ser. Simutools '08. ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2008, pp. 60:1–60:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1416222.1416290

[4] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MObility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[5] C. Sommer, R. German, and F. Dressler, "Bidirectionally Coupled Network and Road Traffic Simulation for Improved IVC Analysis," *IEEE Transactions on Mobile Computing*, vol. 10, no. 1, pp. 3–15, January 2011.

[6] M. Cintra, "A crise do trânsito em são paulo e seus custos," *GV-executivo*, Julho 2013.

[7] R. Aissaoui, H. Menouar, A. Dhraief, F. Filali, A. Belghith, and A. Abu-Dayya, "Advanced real-time traffic monitoring system based on v2x communications," in *Communications (ICC), 2014 IEEE International Conference on*, June 2014, pp. 2713–2718.

[8] "Ieee guide for wireless access in vehicular environments (wave) - architecture," *IEEE Std 1609.0-2013*, pp. 1–78, March 2014.

[9] J. Geraldo, M. Campista, and L. Costa, "A decentralized traffic monitoring system based on vehicle-to-infrastructure communications," in *Wireless Days (WD), 2013 IFIP*, Nov 2013, pp. 1–6.

[10] M. Gramaglia, M. Calderon, and C. Bernardos, "Abeona monitored traffic: Vanet-assisted cooperative traffic congestion forecasting," *Vehicular Technology Magazine, IEEE*, vol. 9, no. 2, pp. 50–57, June 2014.

[11] M. Milojevic and V. Rakocevic, "Distributed road traffic congestion quantification using cooperative vanets," in *Ad Hoc Networking Workshop (MED-HOC-NET), 2014 13th Annual Mediterranean*, June 2014, pp. 203–210.

[12] M. Younes and A. Boukerche, "Efficient traffic congestion detection protocol for next generation vanets," in *Communications (ICC), 2013 IEEE International Conference on*, June 2013, pp. 3764–3768.

[13] J. Xu, W. Sun, N. Shibata, and M. Ito, "Greenswirl: Combining traffic signal control and route guidance for reducing traffic congestion," in *Vehicular Networking Conference (VNC), 2014 IEEE*, Dec 2014, pp. 175–182.

[14] G. B. Araújo, A. I. J. Tostes, F. de L. P. Duarte-Figueiredo, and A. A. F. Loureiro, "Um protocolo de identificação e minimização de congestionamentos de tráfego para redes veiculares," in *XXXII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, Florianópolis, 2014, pp. 207 – 220.

[15] P. Bellavista, L. Foschini, and E. Zamagni, "V2x protocols for low-penetration-rate and cooperative traffic estimations," in *Vehicular Technology Conference (VTC Fall), 2014 IEEE 80th*, Sept 2014, pp. 1–6.

[16] S. Gupte and M. Younis, "Vehicular networking for intelligent and autonomous traffic management," in *Communications (ICC), 2012 IEEE International Conference on*, June 2012, pp. 5306–5310.

[17] L. Wischoff, A. Ebner, H. Rohling, and R. Halfmann, "Sotis - a self-organizing traffic information system," in *Vehicular Technology Conference, 2003. VTC 2003-Spring. The 57th IEEE Semiannual*, vol. 1, 2003, pp. 2442 – 2446.

[18] Veins framework. [Online]. Available: http://veins.car2x.org

[19] Veins framework. [Online]. Available: http://veins.car2x.org/documentation/

[20] C. Sommer, S. Joerer, and F. Dressler, "On the Applicability of Two-Ray Path Loss Models for Vehicular Network Simulation," in *4th IEEE Vehicular Networking Conference (VNC 2012)*. Seoul, Korea: IEEE, November 2012, pp. 64–69.

[21] C. Sommer and F. Dressler, "Using the Right Two-Ray Model? A Measurement based Evaluation of PHY Models in VANETs," in *17th ACM International Conference on Mobile Computing and Networking (MobiCom 2011), Poster Session*. Las Vegas, NV: ACM, September 2011.

[22] C. Sommer, D. Eckhoff, R. German, and F. Dressler, "A Computationally Inexpensive Empirical Model of IEEE 802.11p Radio Shadowing in Urban Environments," in *8th IEEE/IFIP Conference on Wireless On demand Network Systems and Services (WONS 2011)*. Bardonecchia, Italy: IEEE, January 2011, pp. 84–90.

[23] C. Sommer, D. Eckhoff, and F. Dressler, "IVC in Cities: Signal Attenuation by Buildings and How Parked Cars Can Improve the Situation," *IEEE Transactions on Mobile Computing*, vol. 13, no. 8, pp. 1733–1745, August 2014.

[24] F. Hagenauer, F. Dressler, and C. Sommer, "A Simulator for Heterogeneous Vehicular Networks," in *6th IEEE Vehicular Networking Conference (VNC 2014), Poster Session*. Paderborn, Germany: IEEE, December 2014, pp. 185–186.

[25] M. Segata, S. Joerer, B. Bloessl, C. Sommer, F. Dressler, and R. Lo Cigno, "PLEXE: A Platooning Extension for Veins," in *6th IEEE Vehicular Networking Conference (VNC 2014)*. Paderborn, Germany: IEEE, December 2014, pp. 53–60.