Novos Paradigmas de Rede

Segurança rodoviária suportada em computação de proximidade

Rita Gomes[1 [pg50723]], Diogo Pires[1 [pg50334]], and Diogo Matos[1 [pg50340]]

[1] Universidade do Minho, Braga, Portugal
[2] {pg50723,pg50334,pg50340}@alunos.uminho.pt

Abstract. Este relatório descreve o projeto e implementação de um serviço de comunicação veicular que tira proveito da capacidade computacional dos veículos e da infraestrutura de comunicação dos veículos e das *Road Side Units* (RSUs), que visa melhorar a segurança e eficiência no trânsito. O objetivo principal é a implementação de um protocolo de comunicação entre veículos, e com as RSUs. Essa comunicação pode consistir em alertas de segurança de veículos vizinhos, ou com origem na RSU. O projeto foi dividido em três etapas, cada uma com objetivos específicos, abordando desde a implementação básica do serviço até à expansão da área de cobertura e o uso de estratégias de encaminhamento tolerantes a atrasos.

Keywords: comunicação veicular (VANET's), RSUs, protocolo, segurança rodoviária

1 Introdução

O aumento exponencial no número de veículos nas estradas, aliado à crescente necessidade de segurança e eficiência no trânsito, demanda soluções tecnológicas avançadas que possam melhorar a comunicação e a perceção entre os veículos e a infraestrutura rodoviária. Neste contexto, as redes veiculares, que permitem a comunicação espontânea entre veículos e com a infraestrutura, têm se mostrado como uma abordagem promissora para aumentar a segurança e a eficiência no trânsito.

O objetivo deste projeto é desenvolver um protótipo funcional de um serviço de segurança rodoviária baseado em redes veiculares e computação de proximidade. Através da troca de informações entre os veículos e unidades de comunicação fixas instaladas ao longo das vias, a solução proposta busca melhorar a perceção dos veículos em relação ao ambiente ao seu redor, permitindo a identificação de situações de risco e ações preventivas que contribuam para a redução de acidentes e fatalidades no trânsito.

Este trabalho está dividido em três etapas (sendo a última opcional). O objetivo da primeira etapa era desenvolver e testar uma versão simplificada do serviço. Os veículos coletam dados sobre si mesmos e os compartilham com outros veículos e RSUs por *multicast* a um salto de distância. O servidor local da RSU recebe os dados e os processa, gerando informações úteis, como densidade de tráfego e velocidade recomendada. Na segunda etapa, o objetivo era alargar a área de ação da RSU para mais de um salto de distância, usando uma abordagem tolerante a atrasos (DTN) para lidar com perdas temporárias de conectividade. Os veículos armazenam temporariamente as mensagens a enviar, transportando-as algum tempo enquanto esperam por nova conexão, num paradigma de *store-carry-and-forward*. Foi implementada uma estratégia de encaminhamento adequada, baseada na posição e no movimento dos veículos.

O presente relatório descreve em detalhe o desenvolvimento do protótipo, abordando aspetos como a arquitetura global do serviço, comunicações entre as diversas entidades envolvidas, implementação de algoritmos de encaminhamento tolerantes a atrasos e a criação de uma plataforma de emulação para teste e validação da solução proposta.

2 Overview

O objetivo do projeto é melhorar a segurança e a eficiência no trânsito, permitindo uma comunicação eficaz entre veículos e a infraestrutura rodoviária.

O projeto aborda o desafio de lidar com o aumento do número de veículos nas estradas e a necessidade de soluções tecnológicas avançadas para melhorar a segurança rodoviária. Para isso, foi desenvolvido um protótipo funcional que utiliza redes veiculares e computação de proximidade para melhorar a percepção dos veículos em relação ao ambiente ao seu redor.

O protótipo é composto por diferentes componentes, cada um desempenhando um papel específico no sistema. A "Torre" é responsável pela comunicação entre os diferentes elementos do sistema, como veículos e o servidor. A "Torre" envia anúncios e encaminha mensagens relevantes para o servidor. O "Servidor" lida com a comunicação entre os carros e a nuvem, atualizando a lista de carros no alcance e encaminhando informações importantes para a nuvem. A "Cloud" processa as mensagens recebidas e mantém um mapa para rastrear eventos associados a cada torre.

Os carros desempenham um papel crucial no sistema, comunicando-se com as torres próximas e outros carros. Eles coletam informações sobre seu estado atual, como posição, velocidade e direção, e compartilham essas informações com outros carros e torres. Os carros também são capazes de relatar acidentes e enviar notificações de emergência para terceiros, como travagens. Também é possível um veículo de emergência descrever o seu caminho para que carros que se encontrem no percurso na altura da passagem sejam notificados.

O protótipo utiliza um formato de mensagens protocolares (PDU) para facilitar a comunicação entre os diferentes componentes. As mensagens são estruturadas em camadas, como a camada de base, a camada de encaminhamento e a camada de aplicação. Cada camada desempenha um papel específico na transmissão e processamento das mensagens.

Além disso, o projeto incorpora recursos de encaminhamento tolerante a atrasos (DTN) para lidar com a conectividade intermitente e atrasos nas comunicações. O encaminhamento DTN permite que as mensagens sejam armazenadas e encaminhadas quando as conexões estiverem disponíveis, garantindo a entrega das informações mesmo em situações de conectividade limitada.

Para testar se as funcionalidades estavam corretamente implementadas, realizámos dois testes que permitem testar a maior parte das caraterísticas descritas anteriormente.

3 Especificação do serviço e dos protocolos de suporte

De seguida descrevemos quais os tipos de serviços disponibilizados no nosso programa, e tivemos como inspiração o seguinte artigo [4].

3.1 Formato das mensagens protocolares (PDU)

Base Layer Este pequena camada, composta por apenas um *byte*, serve para indicar se o pacote inclui uma camada de *forwarding* ou se vai diretamente encapsulado em IPv6.

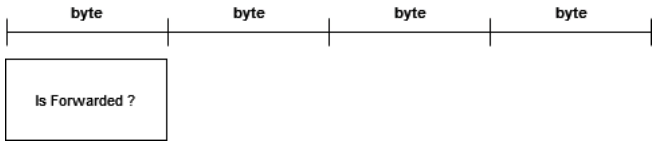

Fig. 1. AWFullPLayer

Forwarding Layer Como o nome indica, esta camada está responsável por incluir informações relevantes para o *forwarding* geográfico dos pacotes, discutido em mais detalhe na secção 3.2.

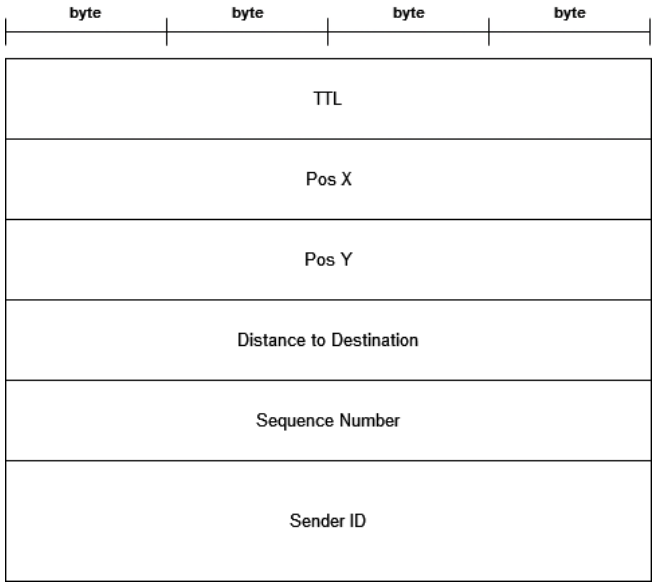

Fig. 2. GeoForwardingLayer

Application Layer Esta camada aborda todos os tipos de mensagens que poderão ser enviadas entre veículos, e entre nodos da infraestrutura. Tem portanto um pequeno cabeçalho a indicar o tipo do conteúdo que se segue.

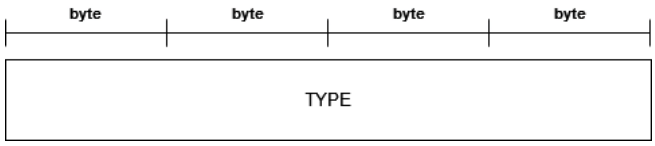

Fig. 3. ApplicationLayer

- **Tower Announce** [14] - Enviados periodicamente pelas torres, servem para avisar os veículos da sua presença, enviando em conjunto a sua posição e informações relevantes sobre a sua zona de influência (i.e. velocidade máxima).
- **Car Hello** [15] - Enviados periodicamente pelos veículos, e contêm informações sobre o seu estado atual, como posição, velocidade, e direção. Decidimos enviar estas informações para que no futuro os outros carros possam construir um mapa do ambiente à sua volta a partir destas mensagens. Dessa maneira, são semelhantes às mensagens CAM definidas pelo ETSI [1].
- **Car In Range** [16] - São enviados pela torre para o servidor, despoletado pela receção da mesma de um $CarHello$. São utilizadas para manter uma lista de veículos dentro da sua zona de influência.
- **Server Info** [17] - Quando o servidor recebe suficientes mensagens $CarInRange$, é formado um *batch* de IDs de veículos na zona, e é enviado para a *cloud*.
- **Car Break** [18] - O seu envio é despoletado no caso de uma paragem brusca por parte de um veículo, servindo para anunciar aos veículos no seu redor para terem atenção e reagirem adequadamente. Tanto esta mensagem como a seguinte usam conceitos das mensagens DENM [2].
- **Car Accident** [19] - O seu envio é despoletado no caso de um acidente de um veículo ou entre veículos, servindo para anunciar aos outros no seu redor para terem atenção e reagirem adequadamente.
- **Ambulance Path** [20] - É utilizado por uma ambulância para estabelecer uma rota por onde vai passar, e é dirigida para a *cloud*. Esta mensagem contém todos os pontos por onde vai passar e quando, para posteriormente a *cloud* notificar os servidores com a mensagem *Ambulance Path to Each Server*.
- **Ambulance Path to Each Server** [21] - Esta mensagem é enviada da *cloud* para os servidores, e para os carros (através da RSU) quando está previsto a ambulância chegar. Desta forma, os outros veículos devem guiar com mais precaução. Pode ser necessário ocorrer *forwarding*, caso o local de passagem

da ambulância seja afastado da RSU, e posteriormente seria possível reservar a passagem na estrada para a ambulância ou outros veículos de emergência.

De seguida, apresentamos algumas informações sobre os vários tipos de mensagens:

Nome da mensage	TTL	Destino	Tamanho (bytes)	Tipo de encaminhamento
CAR_HELLO	1	Não	32	Broadcast topológico
CAR_BREAK	2	Não	24	Broadcast topológico
CAR_ACCIDENT	3	Sim	32	Geo-Unicast
CLOUD_AMBULANCE_PATH	3	Sim (RSU)	4 + Nº paragens x 12	Geo-Unicast
AMBULANCE_PATH	4	Sim (servidor/carros)	16	Geo-Broadcast
TOWER_ANNOUNCE	1	Não	20	Broadcast topológico
CAR_IN_RANGE	Para servidor		16	—
SERVER_INFO	Para cloud		12 + Tam. batch x 8	—

Table 1. Informações genéricas sobre os vários tipos de mensagens

3.2 Forwarding e DTN

De seguida iremos analisar como as várias mensagens circulam na rede, procurando implementar *forwarding* geográfico e usando conceitos de *Delay Tolerant Networks*. Como o forwarding geográfico só funciona entre os carros, pois o resto da rede é fixa, e mais eficiente utilizar os endereços IP's, começamos por explicar: motivos para a organização do *Forwarding Layer*; como os carros implementam *forwarding*; e de seguida a implementação dos serviços implementados.

Especificação Forwarding Layer 3.1 Como descrito anteriormente, o *Forwarding Layer* tem 6 campos, explicados a seguir:

O **TTL** é decrementado sempre que a mensagem é retransmitida por um nodo, e o seu valor inicial varia de mensagem para mensagem, como apresentado na tabela anterior. Com este campo, impedimos que uma mensagem circule por demasiado tempo na rede, prevenindo ciclos infinitos.

As **duas coordenadas de posição** são o destino geográfico da mensagem, e a distância para o destino é calculada a partir do último nodo que envia a mensagem. Decidimos guardar na mensagem a **distância** de quem envia, e não as coordenadas de quem a envia, por duas razões: evitar que todos os recetores calculem a distância, para saber se devem reencaminhar, e para diminuir o tamanho das mensagens.

Por fim, enviámos o **ID** (identificador) de quem envia a mensagem, e o **número de sequência da mensagem**, para verificar se a mensagem é repetida ou não. Cada entidade que envia mensagens tem um contador próprio, correspondente ao número de sequência atual, e cada mensagem enviada possui um número de sequência próprio. Quando uma mensagem é enviada por essa entidade, o seu contador é incrementado. Assim, a partir destes dois fatores, é possível identificar se uma mensagem é duplicada ou não. Estes dois campos não variam na mensagem reencaminhada, mas os restantes são sempre alterados a cada reenvio.

Implementação de *forwarding* nos carros Como temos vários tipos de serviços, com necessidades diferentes, temos diferentes tipos de encaminhamentos conforme foi descrito na tabela anterior **??**, sendo que para o encaminhamento geográfico utilizámos CBF [3]. Decidimos utilizar este método de encaminhamento por diminuir o número de mensagens trocadas, ter um delay reduzido, e ter uma probabilidade de as mensagens chegarem ao destino satisfatória. Apresenta como desvantagem algum custo computacional por mensagem, mas considerámos escalável.

Para implementarmos esta parte, utilizámos duas estruturas de dados importantes, que cada veículo possui:

- **messagesAlreadyReceived** - Consiste num *set* com todas as mensagens que já foram reencaminhadas, e que por isso não devemos voltar a enviar. Esse *Set* apenas contém o resultado de uma função hash com o número de sequência da mensagem, e o identificador de quem a enviou;
- **sendMessagesClasses** - Trata-se de um dicionário onde associámos a parte de Forwarding da mensagem com uma classe (com uma *thread* própria) que envia periodicamente a mensagem. Aqui estão as mensagens que ainda não foram confirmadas, e que por isso devem ser reenviadas. Podemos considerar esta parte como *Carry and Forward*, uma componente da *Delay Tolerant Network*. Decidimos estabelecer um máximo de tentativas, que caso seja atingido, a mensagem deixa de ser enviada. A principal razão para esta decisão é a mensagem só ter importância durante algum tempo.

O encaminhamento nos carros é descrito pelo esquema seguinte, utilizando :

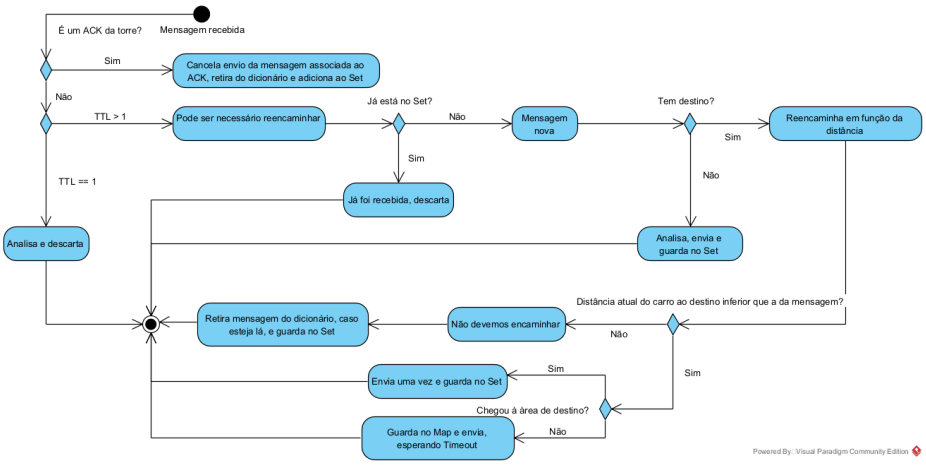

Fig. 4. Funcionamento do encaminhamento das mensagens.

Algumas observações sobre o diagrama:

- A mensagem de ACK da torre é uma mensagem cujo valor de distância é 0, e assim é confirmado que a mensagem chegou ao destino. Caso esta mensagem de ACK não existisse, os carros estariam a mandar a mensagem ao pé da torre até o máximo de tentativas ser atingido;
- As mensagens adicionadas ao dicionário estão numa *thread* que as reenvia algumas vezes, após o *delay* inicial proporcional à distância. Esta *thread* só para de enviar a sua mensagem em duas situações: recebe a mensagem de alguém melhor posicionado (e cancela o envio); ou número de tentativas máximo atingido;
- Quando a mensagem recebida tem destino, e o carro se encontra no destino, ela é armazenada como tendo chegada ao destino;
- A área geográfica considerada como destino é um círculo com um raio definido no ficheiro de constantes do protocolo. O encaminhamento seguido nesta situação é *Simple GeoBroadcast*;
- O *delay* antes de enviar a mensagem quando tem destino serve para reduzir o número de mensagens trocadas.

Serviços implementados Para concluir a especificação do *Forwarding*, decidimos apresentar os vários serviços que o nosso sistema possui:

CAR_HELLO Enviada por um carro para os seus vizinhos, e RSU's. Tanto esta, como as mensagens seguintes, são contabilizadas nos carros, e podem ser verificadas por uma das opções do terminal;

CAR_BREAK Enviada a dois saltos de distância, e pode ser enviada por opção do utilizador no terminal do carro. Posteriormente podia ser disponibilizada numa API, para que o sistema do carro evocasse este método, e os carros vizinhos fossem notificados;

CAR_ACCIDENT Mensagem muito similar à anterior, mas possui um destino geográfico;

TOWER_ANNOUNCE Enviada a um salto de distância, e não possui qualquer tipo de *forwarding*;

Rotas das ambulâncias Como este serviço é mais complexo, decidimos apresentar um diagrama de sequência que apresenta as trocas de mensagens entre os vários intervenientes, descrevendo a seguir alguns pormenores na comunicação:

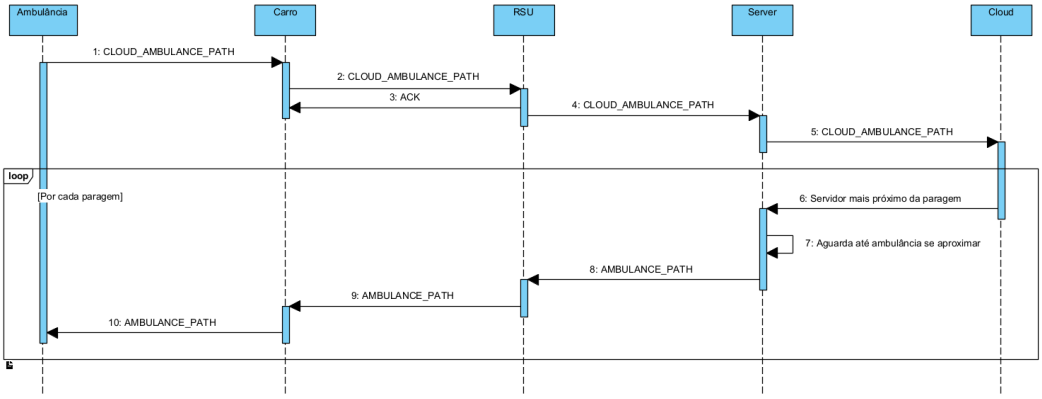

Fig. 5. Diagrama de sequência com comunicações entre ambulância e restantes elementos.

- A comunicação 1 acontece mal a ambulância se inicia, e assumimos que existe uma RSU, ou forma de chegar a ela, mal a execução comece. Fizemos esta assunção porque uma ambulância costuma partir de um hospital, e ele deve ter uma maneira de comunicar com o resto da rede;
- A comunicação 1 pode ser com a RSU diretamente, caso esteja a um salto de distância;
- A comunicação 2 só irá parar quando o carro receber uma confirmação (por causa do funcionamento do protocolo), ou pelo número máximo de tentativas ter acontecido;
- Decidimos que a *cloud* apenas separa o percurso em várias mensagens, e que o servidor é responsável por esperar até ser oportuno mandar mensagem. Desta forma, não sobrecarregamos a *cloud* com muitas responsabilidades, e temos um sistema mais descentralizado;
- É a *cloud* que faz a redistribuição dos caminhos porque possui conhecimento das posições de todas as RSU's.
- As entidades que estão no *loop* não são necessariamente as mesmas de fora, são escolhidas em função das posições da rota da ambulância.

4 Implementação

Diagrama de classes e descrição do que aparece?

4.1 Execução do programa

Para executar o nosso programa, é preciso começar por iniciar a Cloud, de seguida os servidores, e por fim as RSU's. Decidimos desta forma para que o número de argumentos e configurações *default* seja diminuído, e a troca de mensagens também. Assim, a Cloud consegue aprender as torres e RSU's de uma só vez, quando estas se inicializam.

4.2 Detalhes

Torre: Responsável pela comunicação entre a VANET e a parte fixa da rede, nomeadamente entre os carros e os servidores de edge. Assim. envia anúncios para a VANET, recebe mensagens e encaminha mensagens específicas para o servidor. Também envia mensagens do servidor para a rede dos carros.

A torre armazena o conjunto de mensagens já recebidas, *messagesAlreadyReceived*, para que não encaminhe mensagens repetidas para o servidor.

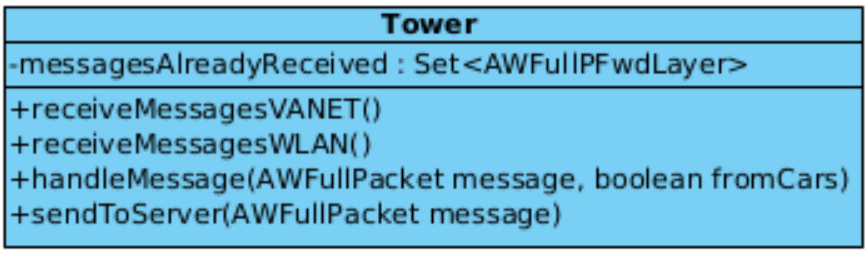

Fig. 6. Diagrama de classes Tower.

Servidor: Manipula a comunicação com os carros, através da RSU, e a Cloud no sistema. Algumas das suas comunicações são: a atualização da sua lista de carros no alcance, e indicação à Cloud; e encaminhamento de mensagens relevantes para a nuvem, como acidentes de carros. O servidor envia lotes de informações (*Batch's*) para a nuvem num intervalo fixo ou quando o tamanho do lote atinge um máximo especificado.

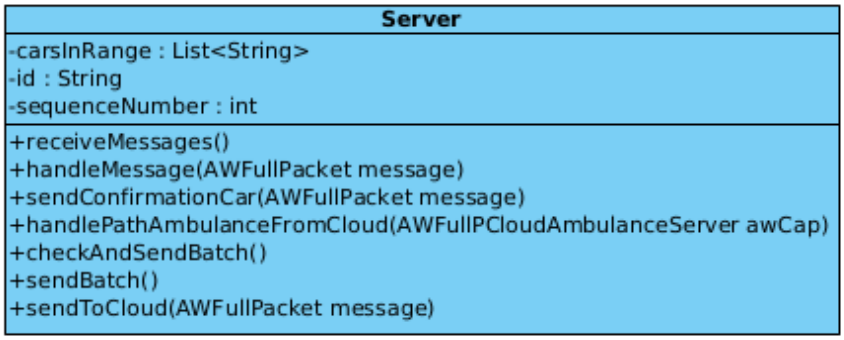

Fig. 7. Diagrama de classes Server.

Cloud: Escuta mensagens de servidores, processa as mensagens com base no seu tipo, regista os eventos e mantém um mapa para acompanhar os eventos associados a cada torre. Contém os seguintes componentes:

- Logger para registrar os eventos.
- DatagramSocket para comunicar com o resto da topologia.
- Um mapa (towerEventMap) para manter o relacionamento entre os IDs das torres e as listas de eventos.
- Um método (receiveMessages) que escuta as mensagens e as manipula usando o método handleMessage.
- O método *handleMessage* que processa mensagens recebidas com base no seu tipo (SERVER_INFO ou CAR_ACCIDENT) e adiciona eventos relevantes ao towerEventMap.
- Como as mensagens recebidas pela ambulância sobre o seu percurso indicam o tempo estimado para passarem no local (tempo relativo ao tempo de envio), e não um tempo preciso, decidimos calcular nesta entidade a hora estimada de passagem. De seguida, enviamos nas mensagens para os servidores as horas previstas. Desta forma, diminuímos o erro na previsão da hora de passagem, porque caso contrário poderia passar bastante tempo até a mensagem ser recebida, e o tempo estimado já ter passado. Decidimos implementar esta funcionalidade na Cloud para reduzir a complexidade nos veículos.

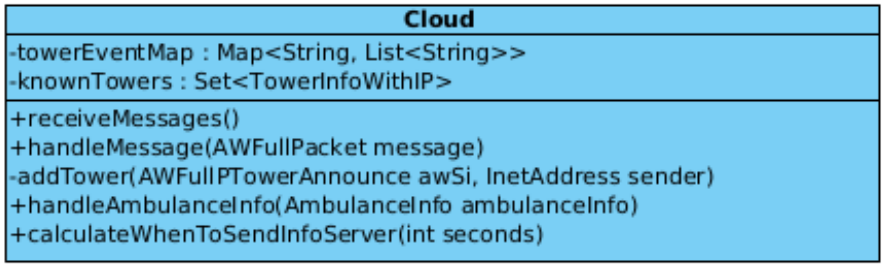

Fig. 8. Diagrama de classes Cloud.

Car: Tem como principais funcionalidades a comunicação com torres próximas e outros carros, tratamento de acidentes de carro e notificação de terceiros sobre acidentes. Usa o package AWFullP para manipulação e comunicação de mensagens, mas vamos ver isso com mais detalhe à frente. De seguida, apresentamos uma visão geral das classes:

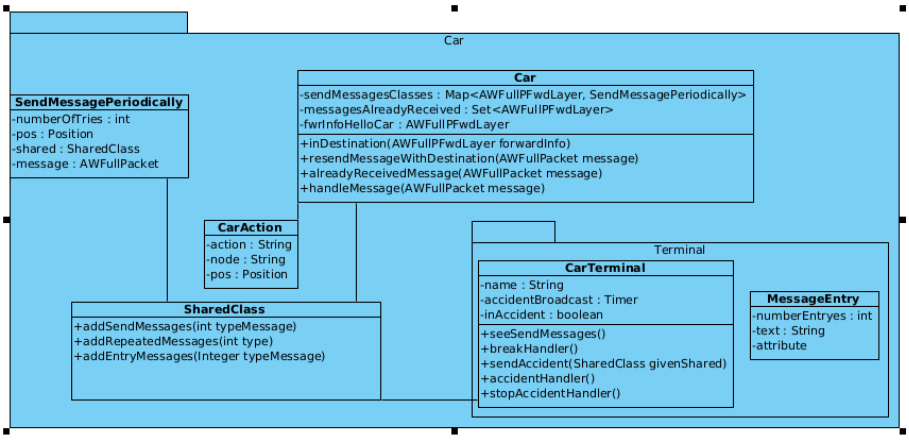

Fig. 9. Diagrama de classes Car.

- Car - A classe principal responsável por lidar com as funcionalidades do carro. Ele inicializa as informações, comunicações e conexões do carro. Ele também escuta as mensagens recebidas e processa-as de acordo.
- SendMessagePeriodically - Uma classe TimerTask que envia periodicamente uma mensagem usando um DatagramSocket, com um número máximo de tentativas.

- SharedClass - Uma classe que contém recursos compartilhados, como *socket*, mensagens recebidas, número de sequência atual e informações das várias torres.
- CarTerminal - Uma classe que fornece um terminal interativo para o utilizador acessar as funções do carro. Os usuários podem acionar eventos como travagem, relatar um acidente e solucionar um acidente. Para facilitar os testes, decidimos desativar a opção de manter o estado de acidente no carro. Anteriormente, quando o estado de acidente era acionado, só podia ser parado pelo terminal, enviando mensagens continuamente.
- MessageEntry - Uma classe que representa uma entrada de mensagens recebidas. Essa classe possui o conteúdo da mensagem ('text'), e 'numberEntryes', que mantém o número de vezes que a mensagem foi recebida.
- CarAction - Armazena uma ação (*action*) que o carro terá no percurso. Esta classe permite fazer testes automáticos, obrigando o carro a executar uma ação quando passa numa dada posição (*pos*)

O package Common contém classes que representam diferentes componentes no sistema para rastrear informações do carro e comunicação entre nós numa rede.

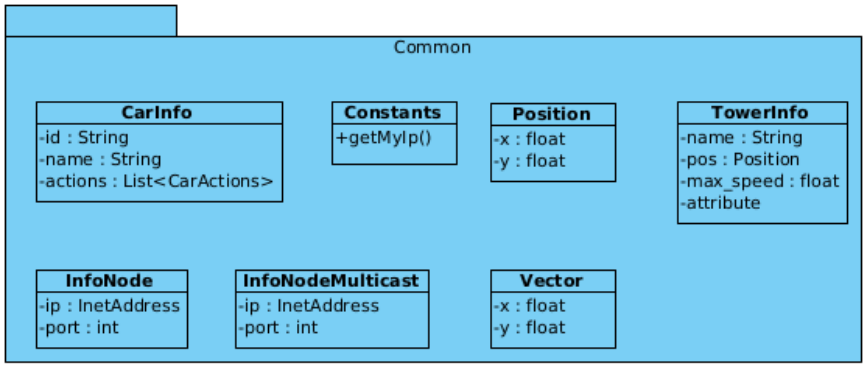

Fig. 10. Diagrama do package Common.

- CarInfo: Esta classe representa as informações de um carro. Inclui o ID do carro, posições (as três posições mais recentes), velocidades (com base nas duas posições mais recentes) e aceleração. Possui métodos de atualização de posição atual, calculados a partir do ficheiro de configuração do core.
- Constants: esta classe contém várias constantes usadas na topologia, como taxa de atualização, números de porta para comunicações entre entidades, endereço de grupo multicast e o IP da nuvem. Também implementámos aqui um método para descobrir o endereço IP do sistema atual.

- InfoNode: Esta classe representa as informações de conexão de um nó. Ele armazena o endereço IP e o número da porta e fornece uma maneira de criar uma instância usando um DatagramSocket.
- InfoNodeMulticast: Esta classe é uma extensão do InfoNode para comunicação multicast. Ele armazena o socket multicast, o endereço IP e o número da porta.
- Position: Esta classe representa a posição de um objeto num plano 2D.
- TowerInfo: Esta classe representa as informações de uma torre. Inclui o nome da torre, posição e velocidade máxima. Possui métodos para obter o nome, posição e velocidade máxima.
- Vector: Esta classe representa um vetor 2D com componentes x e y, utilizado para cálculo das velocidades e acelarações dos veículos. A classe possui métodos para calcular o comprimento do vetor, normalizar um vetor, somar dois vetores, subtrair dois vetores e gerar uma representação em string do vetor.

De seguida vamos fornecer uma visão gerar do package AWFullP.

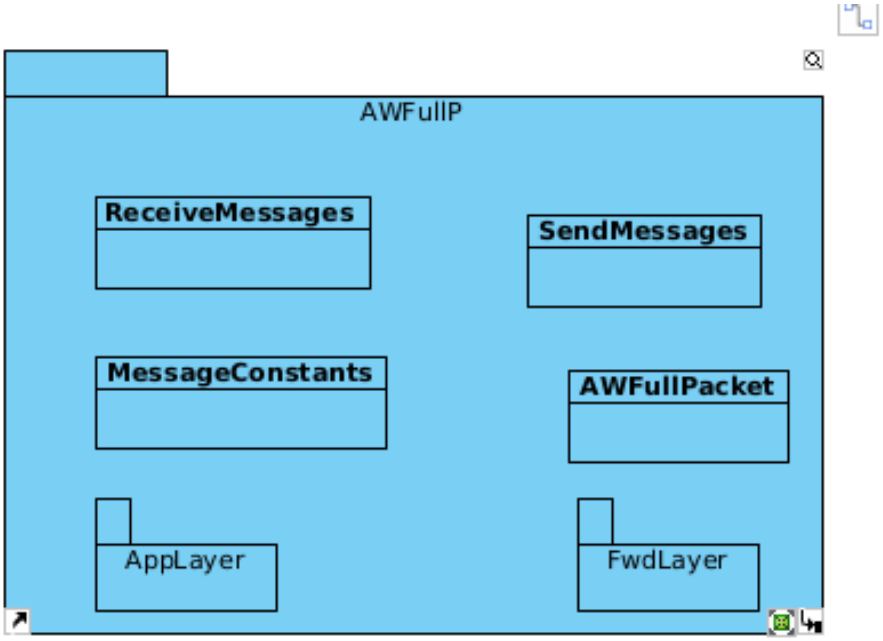

Fig. 11. Diagrama de classes do package AWFullP.

O pacote AWFullP.AppLayer é constituído pelas classes que são usadas para codificar e descodificar mensagens relacionadas a eventos e comunicação do carro, ao nível aplicacional.

A classe AWFullPacket é responsável por lidar com pacotes de dados que são transmitidos no sistema. Ela possui atributos como isForwarded, forwardInfo e appLayer, que armazenam informações sobre o encaminhamento do pacote, informações de encaminhamento e a camada de aplicação, respectivamente.

Vários construtores são fornecidos para criar instâncias de AWFullPacket a partir de diferentes tipos de entrada, como byte[], DatagramPacket e objetos AWFullPAppLayer. Além disso, a classe fornece métodos para converter pacotes em arrays de bytes, verificar se um pacote possui informações de posição de destino e gerar uma representação textual dos pacotes. O método de reencaminhamento depende destes parâmetros, como foi descrito anteriormente.

A classe ReceiveMessages fornece métodos para:

- receiveData(): Receba um DatagramPacket usando um DatagramSocket e converter os dados recebidos em um objeto AWFullPacket.
- parseMessageCar(): Recebe e analisa um pacote, verificando se a mensagem recebida é enviada por ele próprio. Se a mensagem tiver origem no próprio, o método lançará uma exceção SelfCarMessage.
- MaybeForwardMessage(): pode encaminhar uma mensagem recebida com base em certas condições (por exemplo, distância e mensagens duplicadas).

A classe SendMessages fornece métodos para enviar os vários tipos de mensagens descritos anteriormente.

A classe MessageConstants armazena constantes e informações relacionadas aos diferentes tipos de mensagens que são trocadas no sistema. As constantes incluem tamanhos de cabeçalho, tipos de mensagens, tamanhos de mensagens e outros valores relacionados a cada tipo de mensagem, descritos no capítulo relativo ao protocolo.

Estas classes trabalham em conjunto para facilitar a troca de informações entre os componentes do sistema e a manipulação de pacotes de dados de forma eficiente e organizada.

5 Testes e resultados

Para testar as funcionalidades implementadas, decidimos criar dois testes mais gerais, descritos a seguir. Nos dois testes, o percurso dos carros não é o mais realista, mas só desta forma conseguimos provar que o programa funciona corretamente, sendo mais fácil entender o seu funcionamento.

5.1 Notificações de percurso da ambulância, com *handover* de RSU

De seguida apresentamos o esquema do teste, com a representação da troca das mensagens:

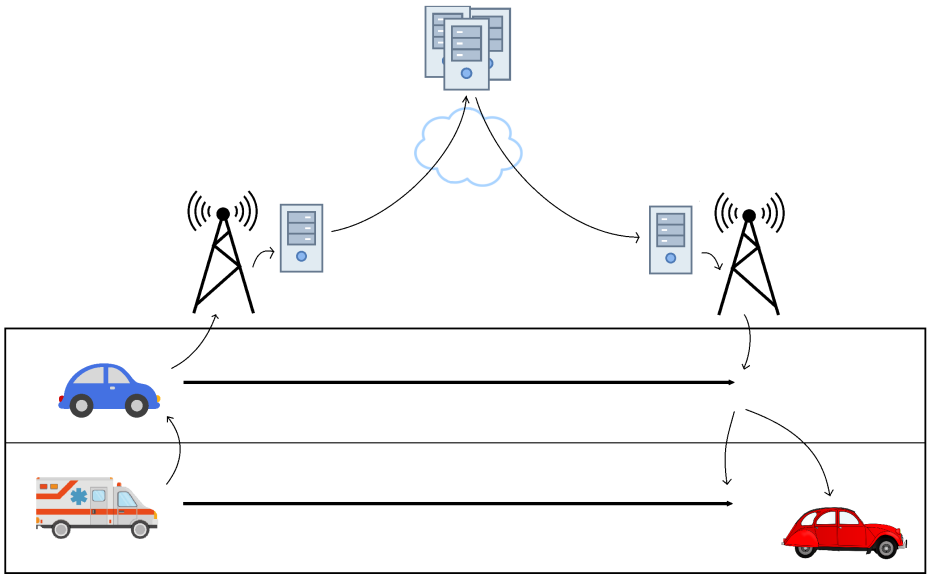

Fig. 12. Teste do percurso das ambulâncias.

Com este teste, pretendemos verificar se a ambulância consegue comunicar o seu caminho para o resto da rede. Neste teste, verificamos os seguintes pontos:

1. Se a ambulância consegue enviar o seu percurso à *cloud* a mais que um salto de distância;
2. Se a *cloud* recebe corretamente a mensagem, e consegue reencaminhar para os servidores mais próximos as mensagens;
3. Se um servidor aguarda o tempo correto antes de enviar a mensagem;
4. Se a mensagem chega à posição de destino, mesmo que demore mais que um salto de distância (carro vermelho do teste)

5.2 Notificação de acidente *multi-hop*, com *store-carry-and-forward*

De seguida apresentamos o esquema do teste do acidente:

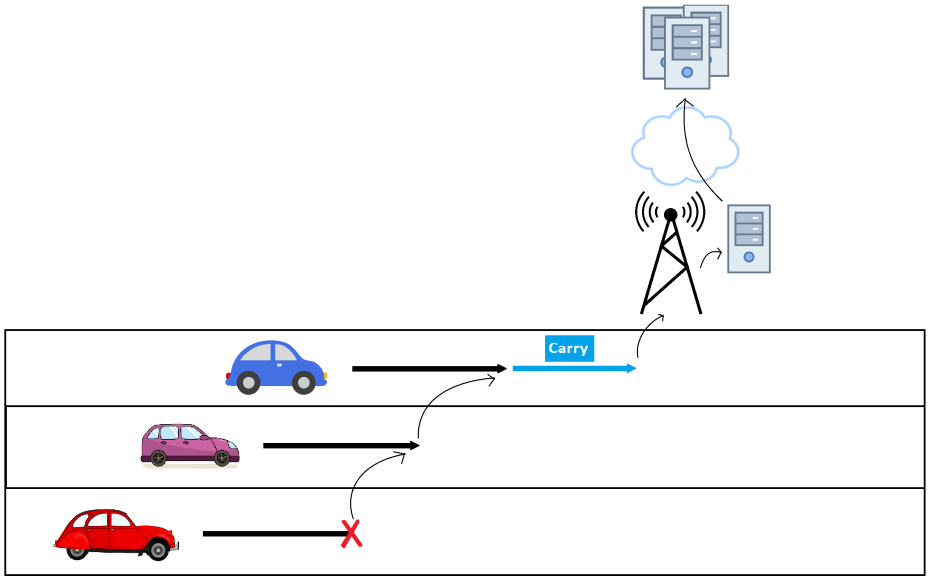

Fig. 13. Teste dos eventos do acidente.

Como podemos ver, o acidente acontece numa zona que não tem cobertura por uma RSU, logo tem de haver carry durante um perídodo curto de tempo. De seguida, apresentamos as caraterísticas do nosso programa que queríamos testar:

1. Se a mensagem de acidente é entregue a mais que um salto;
2. Quando um carro se apercebe que outro está numa posição melhor que ele (carro na fila do meio está numa pior posição que o de cima), cancela o envio da mensagem;
3. Se existe um carry da mensagem, enquanto não recebe confirmação de uma RSU;
4. Se RSU confirma mensagem, e envia para a cloud (através do servidor);
5. Se carro deixa de enviar mensagens após confirmação da RSU;

Os dois testes que realizámos funcionaram, o que nos permitiu, concluir que o sistema implementado funciona de forma correta.

6 Conclusões e trabalho futuro

O projeto e a implementação do serviço de comunicação veicular demonstraram a viabilidade de aproveitar a capacidade computacional dos veículos e da infraestrutura de comunicação das RSUs para fornecer informações úteis em tempo real.

Por fim, gostaríamos de ter implementado as seguintes funcionalidades:

- Apagar conteúdos mais antigos nos veículos, porque eles acumulam o histórico de mensagens, e para testes curtos funciona, mas a longo prazo é insustentável;
- Como foi apenas para *proof-of-concept*, as mensagens de estado dos veículos não estão a ser utilizadas (e por essa razão também não têm muita informação), mas, no futuro, podiam ser usadas para construir um "mapa" da rede, e auxiliar numa estratégia de *forwarding* ainda mais avançada;
- Melhor a componente de *Delay Tolerant Network* do nosso programa, para que utilizasse as informações das mensagens CAM e reencaminhasse conteúdos de forma mais eficiente, minimizando o número de troca de mensagens, e maximizando a probabilidade de uma mensagem chegar ao destino;
- Os *batches* que a *cloud* recebe também estão simplesmente a ser armazenados, mas seria muito interessante processar esses dados e gerar estatísticas relevantes.

References

[1] ETSI. *ETSI EN 302 637-2*. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.04.01_60/en_30263702v010401p.pdf (visited on 05/20/2023).

[2] ETSI. *Specifications of Decentralized Environmental Notification Basic Service*. URL: https://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.02.01_30/en_30263703v010201v.pdf (visited on 05/21/2023).

[3] Mauve M. Hartenstein H Effelsberg W. "Contention-based forwarding for street scenarios". In: *1st International Workshop in Intelligent Transportation* (2004).

[4] et al Al-Sultan S. *A comprehensive survey on vehicular Ad Hoc network*. URL: http://dx.doi.org/10.1016/j.jnca.2013.02.036i (visited on 05/21/2023).

7 Anexos

Fig. 14. TowerAnnounce 3.1

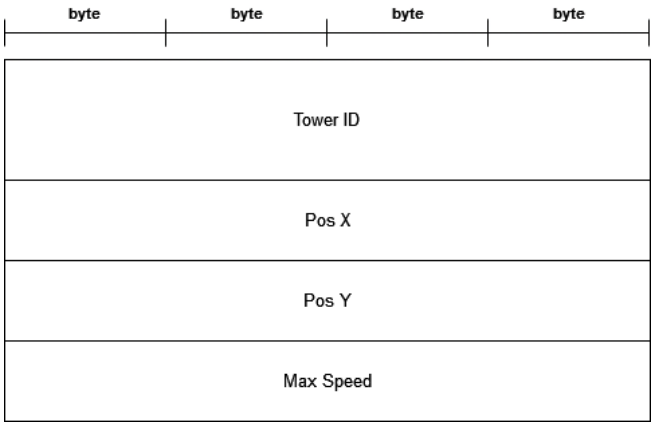

Fig. 15. CarHello 3.1

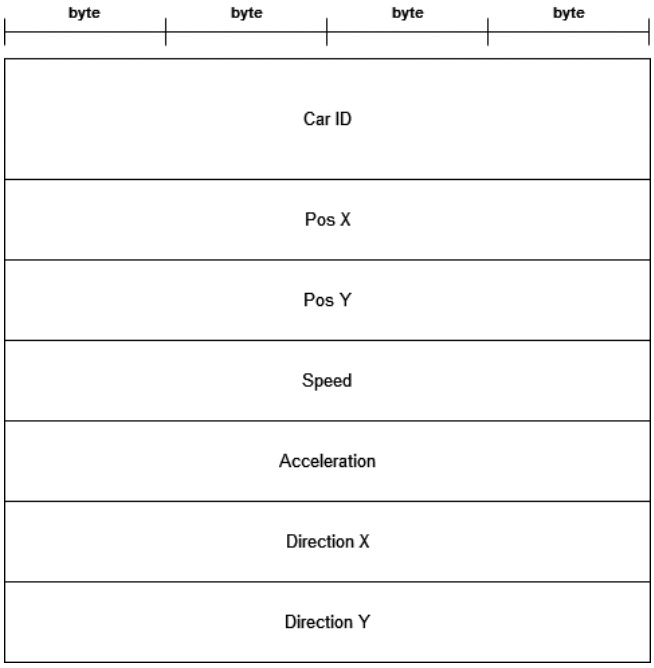

Fig. 16. CarinRange 3.1

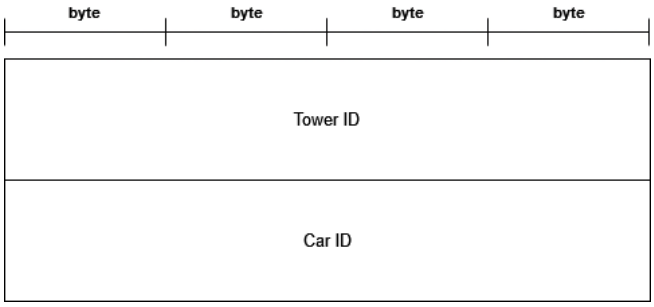

Fig. 17. ServerInfo 3.1

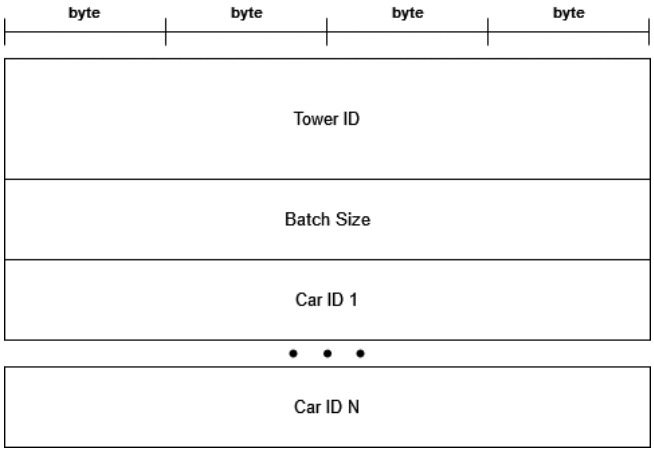

Fig. 18. CarBreak 3.1

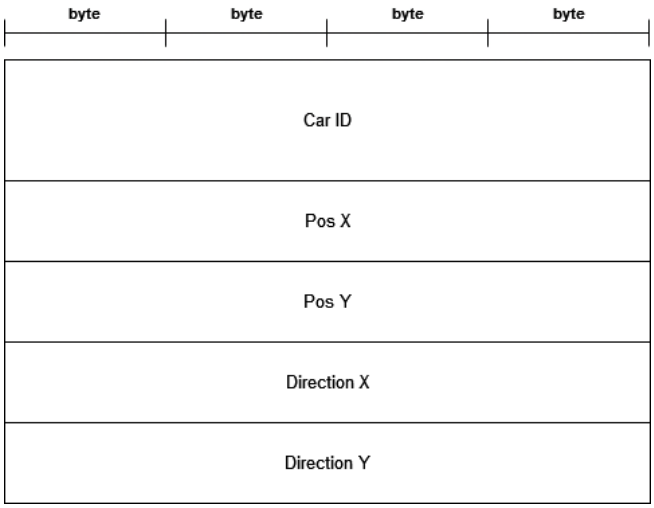

Fig. 19. CarAccident 3.1

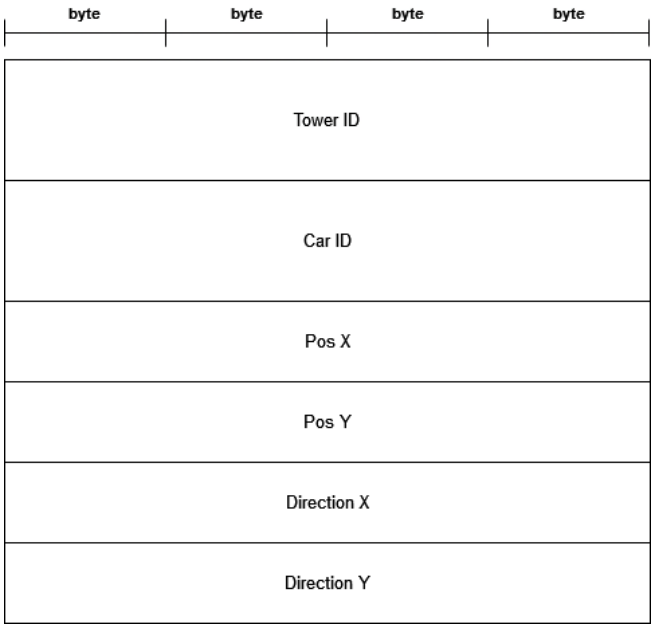

Fig. 20. AmbulancePath 3.1

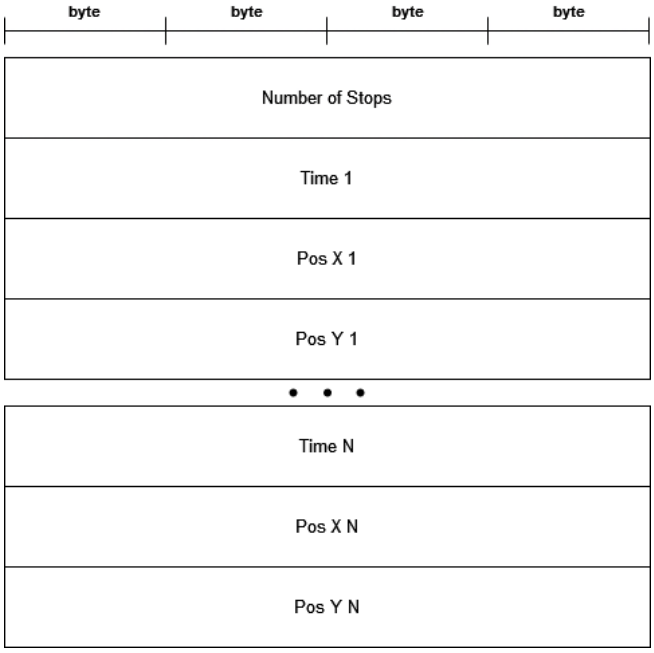

Fig. 21. AmbulancePathtoEachServer 3.1