

Trabalho Prático - Grupo 4

João Abreu, *PG55895*, Miguel Jacinto, *PG57590*, e Ricardo Pereira, *PG56001*

I. INTRODUÇÃO

A segurança rodoviária constitui atualmente um desafio de extrema importância na nossa sociedade contemporânea, exigindo a aplicação de soluções tecnológicas eficazes que permitam uma gestão eficiente e segura das redes veiculares. Neste âmbito, a comunicação V2X assume um papel essencial, oferecendo mecanismos para uma perceção abrangente e atempada dos contextos rodoviários complexos, contribuindo para a prevenção de acidentes e otimização do tráfego.

Este projeto teve como objetivo central o desenvolvimento e implementação de uma ferramenta robusta para comunicação *Multihop* e controlo eficaz da rede veicular, utilizando computação de proximidade baseada em *Fog Computing*. A solução concebida materializou-se num protótipo funcional, suportado pela plataforma de simulação Eclipse MO-SAIC. O trabalho concretizou-se através do desenvolvimento das aplicações necessárias para veículos (*OBV*), unidades fixas nas vias rodoviárias (*RSU*) e nós computacionais *Fog*.

Neste relatório será apresentada uma descrição detalhada do desenvolvimento e implementação de cada uma das aplicações e respetivos protocolos de comunicação, bem como a análise crítica dos resultados obtidos nas simulações realizadas. Serão também discutidas possíveis linhas de investigação e melhorias futuras que poderão aperfeiçoar ainda mais a eficácia e robustez da solução proposta.

II. CENÁRIO

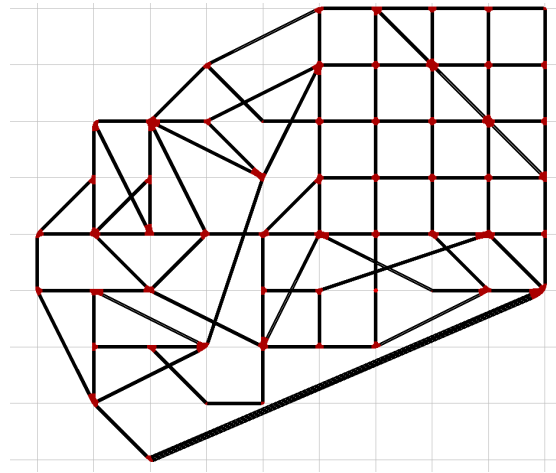


Fig. 1. Cenário utilizado nas nossas simulações.

Este cenário foi desenvolvido com o objetivo de reproduzir uma topologia rodoviária bastante diversificada. No canto superior direito do cenário, simula-se uma grelha de Manhattan interseccionada por uma via na diagonal. No restante cenário, observa-se um conjunto de vias menos organizadas,

típico de algumas zonas urbanas europeias. Finalmente, na zona inferior, incluiu-se uma longa reta com múltiplas vias paralelas, simulando o funcionamento de uma autoestrada a ligar duas extremidades do cenário.

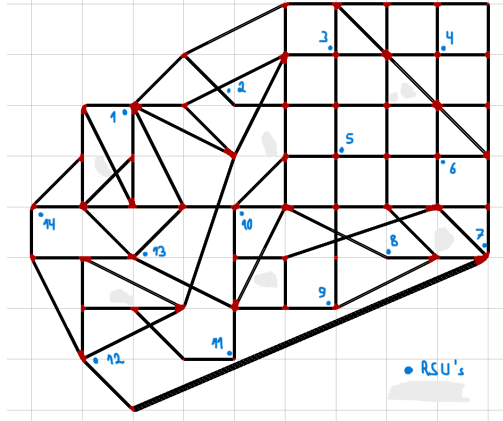


Fig. 2. Localização dos RSUs.

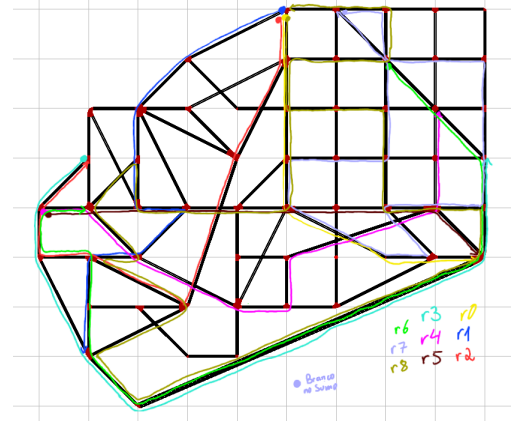


Fig. 3. Rotas definidas. Nem todas estão usadas

Colocamos os RSUs estrategicamente pelo cenário para garantir que cobríamos o máximo de casos possíveis. Existem cerca de 14 RSUs cada um deles distanciado o suficiente (200 metros) para garantir que os veículos são forçados a comunicar entre eles para entregar ou receber mensagens. É importante também referir que o sistema de numeração das instâncias dos RSUs do Eclipse MOSAIC começa sempre a 0 a nossa numeração está exatamente uma unidade acima. Existem várias rotas distintas que partem de vários sítios diferentes do mapa. A nossa ideia é inicialmente ter processamento de CAMS enviadas diretamente entre grupos isolados. À medida que a simulação avança, os veículos começam a cruzar rotas e trocar mensagens entre eles por multihop. Poderíamos ter adicionado mais carros, mais rotas para o cenário ser ainda mais realista mas achamos que cobrimos os casos essenciais para demonstração das funcionalidades do nosso projeto.

III. MENSAGENS

Implementamos quatro tipos de mensagens V2X, cada uma com um formato e papel distinto. De seguida referimos as várias primitivas de comunicação utilizadas e o *Protocol Data Unit (PDU)* definido para cada tipo de mensagem.

A. Primitivas de Comunicação

Através da primitiva `sendV2xMessage()`, é possível transmitir qualquer PDU que estenda a classe `V2xMessage`, pelo método de encaminhamento definido no `MessageRouting`.

No nosso sistema, a comunicação é realizada através de dois tipos de rede: uma Rede *AdHoc*, acedida através do `getAdHocModule()`, e uma Rede Celular, representada pelo `getCellModule()`. Estas redes simulam, de forma abstrata, a infraestrutura de comunicação entre os veículos, os RSUs e o Fog.

Embora o MOSAIC não forneça uma tradução direta para uma ligação cablada entre os RSUs e o Fog, essa ligação é simulada por meio do `getCellModule()`.

Os veículos utilizam exclusivamente a rede *AdHoc* para comunicarem entre si, recorrendo a transmissões em *broadcast* para todos os vizinhos dentro do alcance de rádio. O *Fog* utiliza apenas a rede celular, estabelecendo ligações diretas com os *RSUs*. Já os *RSUs* desempenham um papel intermédio fundamental no sistema: são os únicos nós com acesso a ambas as redes e funcionam como *relays*, permitindo que os veículos comuniquem indiretamente com o *Fog*.

Para a comunicação por *AdHoc*, definimos a intensidade do sinal como 50dB e um alcance de 140 metros para tentar simular o que seria num cenário ideal da rede, no entanto em situações reais de centros urbanos, bastante densas e com muitos obstáculos o sinal estaria enfraquecido e teria um alcance muito menor e imprevisível.

A receção de mensagens é tratada pela seguinte primitiva:

```
onMessageReceived(ReceivedV2xMessage message)
```

Todas as mensagens recebidas pelo veículo são processadas neste método, permitindo à aplicação reagir de acordo com o tipo e conteúdo da mensagem. Para lidar com eventos periódicos utiliza-se a seguinte primitiva:

```
getOs().getEventManager().addEvent(timestamp, this);
```

Este método agenda a execução futura da aplicação num determinado instante de tempo, sendo fundamental para controlar a emissão regular de mensagens, como as *CamMessage*.

B. *CamMessage*

Como o nome indica, serve o propósito de uma CAM, contendo informações relevantes sobre o veículo que a gera. Apenas é gerada

- **vehId** (String): Identificador único do emissor.
- **timestamp** (long): Instante de emissão, utilizado para filtrar mensagens desatualizadas.
- **hopsToLive** (int): Número de saltos restantes. Útil para controlar o flood da mensagem.
- **position** (GeoPoint): Coordenadas geoespaciais do emissor, necessárias para atualizar o LDM.
- **neighbors** (Set<String>): Lista de identificadores dos veículos próximos ao emissor (vizinhos).

C. *EventMessage*

Implementada como uma classe abstrata, define um evento genérico na rede, simulando o propósito de uma DENM.

- **timestamp** (long): Instante de emissão.
- **expiryTimestamp** (long): Instante após o qual a mensagem não deve ser retransmitida ou processada.
- **target** (String): Identificador do veículo a que é destinada.
- **forwardingTrail** (List<String>): Sequência de nós já percorridos, pela ordem de passagem, incluindo o RSU. Cada nó adiciona o salto seguinte antes de enviar a mensagem.

D. *AccidentEvent*

Definida como uma classe concreta que estende a anterior, herdando assim o seu PDU, com a especialização própria do tipo de evento que define.

- **severity** (int): Determina a gravidade com que o recetor deve tratar o evento.

E. *EventACK*

Serve de confirmação de receção de um evento por parte do veículo ao qual era destinado. Segue o caminho inverso da mensagem de evento para otimização do encaminhamento sem sobrecarga na rede.

- **timestamp** (long): Instante de emissão do ACK.
- **expiryTimestamp** (long): Validade do ACK.
- **checklist** (List<String>): Lista de IDs dos nós restantes no caminho inverso; cada nó remove o seu ID antes de reenviar ao próximo.

IV. ENCAMINHAMENTO

No nosso sistema, as mensagens circulam em duas direções:

- **Downstream** (Fog \rightarrow Veículo): Envia ordens ou alertas do Fog para um veículo específico.
- **Upstream** (Veículo \rightarrow Fog): Transmite confirmações (ACK) do veículo para o fog.

As mensagens com dados de posição (CAM) são enviadas livremente pela rede, navegando principalmente entre veículos, e chegam eventualmente a um RSU que as encaminha para o Fog.

A. *Encaminhamento Downstream (Fog \rightarrow Veículo), quando é necessário multihop para o encaminhamento*

Pré-Condição necessária: O veículo está a mais de 140m do RSU.

Fluxo:

- Fog cria **AccidentEvent** para “V_y”, com **forwardingTrail** = [“R_j”].
- Envia a mensagem ao RSU “R_j”.
- “R_j” chama **selectNextHop** e escolhe “V₁”. Lista: [“R_j”, “V₁”].
- “R_j” transmite em broadcast para “V₁”.
- “V₁” repete o processo: escolhe “V₂”, lista = [“R_j”, “V₁”, “V₂”].
- O processo continua até um veículo alcançar “V_y”.
- “V_y” processa o evento.

Resultado: A mensagem percorre vários saltos até alcançar o destino.

B. Encaminhamento Upstream (Veículo \rightarrow Fog) de mensagens ACK

Fluxo:

- Veículo “V1” copia a ForwardingTrail do evento que recebeu e processou.
- Veículo “V1”, remove o seu ID da lista e gera um ACK com o mesmo ID do evento.
- “V1” envia o ACK para o próximo nó na lista.
- Outros nós repetem o processo enquanto a lista tiver nós.
- Quando o RSU recebe o ACK, remove o seu ID da lista e envia ao Fog.
- Fog confirma a conclusão do evento.

Resultado: O ACK é transmitido de veículo em veículo até alcançar o RSU, pelo mesmo caminho que o evento seguiu.

C. Encaminhamento de mensagens CAM

Fluxo:

- Veículo “V1” emite CAM com HTL = 5 (Hops To Live).
- Veículo “V2”, a menos de 140m de “V1”, recebe essa CAM.
- “V2” verifica que o HTL > 0 , atualiza o seu mapa e retransmite com HTL = 4.
- Outros veículos repetem o processo enquanto HTL > 0 .
- Quando um deles está a < 120 m de um RSU “Rj”, este também recebe a CAM.
- “Rj” atualiza o seu mapa e envia ao Fog com HTL = 0.

Resultado: A CAM é transmitida de veículo em veículo até alcançar um RSU.

D. Comparação Resumida

Paradigma	Encaminhamento Direto	Encaminhamento Multihop
Upstream (Veículo \rightarrow Fog)	Veículo a ≤ 140 m de um RSU \rightarrow RSU \rightarrow Fog	A CAM é encaminhada de veículo a veículo até que eventualmente chegue a um RSU
Downstream (Fog \rightarrow Veículo)	Fog envia via RSU \rightarrow RSU transmite para o veículo (Ad Hoc)	Fog envia via RSU \rightarrow Mensagem passa de veículo a veículo até ao destino final
Saltos no Caminho	2 Saltos: (Veículo \rightarrow RSU, RSU \rightarrow Fog) ou (Fog \rightarrow RSU, RSU \rightarrow Veículo)	3 ou mais saltos com múltiplos veículos até alcançar um RSU ou um veículo destino
Critério de Encaminhamento	Apenas distância (≤ 140 metros)	Seleção baseada em proximidade, conhecimento de dois saltos, tempo de registo e uso anterior

Paradigma	Encaminhamento Direto	Encaminhamento Multihop
TTL/Validade	Eventos usam <code>expiryTimestamp</code>	CAM usa HTL (decrementa a cada salto); eventos usam <code>expiryTimestamp</code>
Confirmação de Entrega	Veículo envia ACK direto para um RSU, que envia ao Fog	ACK percorre a <code>forwardingTrail</code> inversamente até ao RSU, que envia ao Fog
Overhead para o RSU	Baixa: RSU apenas recebe e envia mensagens diretas	Elevada: RSU participa no encaminhamento e manutenção de rotas multihop

V. ARQUITETURA

A seguir detalha-se, com referências específicas a classes, métodos e constantes do código, como cada parte da nossa solução é implementada. O texto concentra-se nos mecanismos centrais: geração de eventos no Fog, construção e manutenção dos grafos de vizinhança, envio e receção de CAMs, roteamento de eventos (direto e multihop) e confirmação via ACK.

A. Receção de CAMs e Armazenamento de Estado

Classe/Método: `FogApp.onMessageReceived(ReceivedV2xMessage incoming)`

```
if (msg instanceof CamMessage cam &&
    (!seenCams.containsKey(cam.getVehId()) || cam.getId() >
     seenCams.get(cam.getVehId()).getId()))
{
    // Store the last cam message for the vehicle
    seenCams.put(cam.getVehId(), cam);
}
```

Sempre que o Fog recebe um `CamMessage`, extrai o `cam.getVehId()` e armazena o objeto `CamMessage` em `seenCams`, um `Map<String, CamMessage>`. Cada entrada guarda, para um certo veículo, a última mensagem CAM recebida do mesmo.

Periodicamente, em `processEvent(Event event)`, o Fog chama `purgeCams()`:

```
private static final long CAM_TTL = 2 * TIME.SECOND;

private void purgeCams() {
    long now = get0s().getSimulationTime();
    seenCams.entrySet().removeIf(
        e -> now - e.getValue().getTimestamp() > CAM_TTL
    );
}
```

Remove-se todos os registos cuja diferença entre tempo atual (`now`) e `cam.getTimestamp()` exceda 2s, garantindo que o Fog só armazena estados de veículos ativos e que contenham informação fiável da posição atual dos mesmos. Informação antiga tem uma probabilidade acrescida de estar desatualizada e gerar erros no encaminhamento das mensagens.

B. Geração de AccidentEvent

Classe/Método: FogApp.maybeGenerateEvent()

```

private static final long EVENT_TTL = 8 * TIME.SECOND;
private static final double EVENT_PROB = 0.02;

private void maybeGenerateEvent() {
    // Do nothing if there are no vehicles or the random chance fails
    if (seenCams.isEmpty() || random.nextDouble() >= EVENT_PROB) {
        return;
    }

    // Select a random vehicle to be the target of the event
    String target = new ArrayList<>(seenCams.keySet())
        .get(random.nextInt(seenCams.size()));
    CamMessage targetInfo = seenCams.get(target);

    long now = getOs().getSimulationTime();
    int id = eventSeq.getAndIncrement();

    // Select a random severity level
    int severity = random.nextInt(3); // 0: minor, 1: moderate, 2: severe

    // Select the best RSU based on the target vehicle's position
    String rsuId = getClosestRsu(targetInfo.getPosition());
    logInfo(String.format(
        "EVENT_ROUTING : ID: %s | RSU: %s",
        id, rsuId
    ));

    // Generate the event message, based on the type
    EventMessage event = new AccidentEvent(
        newRouting(rsuId),
        id,
        now,
        now + EVENT_TTL,
        target,
        new ArrayList<>(List.of(rsuId)),
        severity
    );

    // Send the event message
    getOs().getCellModule().sendV2xMessage(event);
    openEvents.add(id);
    logInfo(String.format(
        "EVENT GENERATED AND SENT : UNIQUE_ID: %d | TARGET: %s", id, target
    ));
}

private MessageRouting newRouting(String rsuId) {
    return getOs().getCellModule()
        .createMessageRouting()
        .destination(rsuId)
        .topological()
        .build();
}

```

Em cada chamada de processEvent(...), o método maybeGenerateEvent() é invocado. Com probabilidade 2%, e se o mapa de CAMs não estiver vazio, escolhe-se um vehId aleatório para ser o destino de um evento do tipo AccidentEvent, com severity também ela aleatória.

A mensagem é enviada ao RSU mais próximo do destino escolhido. O ID único do evento é armazenado na lista openEvents para o Fog manter um estado de todos os eventos que ainda não

recebeu confirmação. Pode ser útil no futuro para implementação de métodos de recuperação de falhas, caso um ACK demore a chegar que pode indicar uma perda de mensagem.

Para reenviar esta e qualquer mensagem, é necessário criar uma cópia da mesma uma vez que o SNS Ambassadors não permite mensagens com TTL superior a 1.

C. Tratamento de EventACK

Classe/Método: `FogApp.onMessageReceived(...)` e `FogApp.handleAckReceived(EventACK ack)`

```
else if (msg instanceof EventACK ack) {
    handleAckReceived(ack);
}

private void handleAckReceived(EventACK ack) {
    // Check if the ACK followed the correct trail
    if (!ack.getChecklist().isEmpty()) {
        logInfo(String.format(
            "ACK_ERROR : ACK_ID: %d | CHECKLIST_SIZE: %d",
            ack.getId(), ack.getChecklist().size()
        ));
    }

    // Remove the event from the open events list
    openEvents.remove((Integer)ack.getId()); // Integer cast to avoid ambiguity
    and ensure use of remove(Object)
    logInfo(String.format(
        "EVENT_CLOSED : EVENT_ID: %d",
        ack.getId()
    ));
}
```

Ao receber um `EventACK`, o Fog verifica se a `checklist` está vazia. Esta verificação é importante para verificar se a lista foi percorrida exatamente na mesma direção em downstream e upstream. Se a lista não estiver vazia, indica que existiu algum problema intermédio de encaminhamento. Caso contrário, remove-se o id de `openEvents`, encerrando o ciclo daquele evento.

D. Construção da lista de vizinhos a partir de CAMs

Método: `onMessageReceived(...)`

```
// Vers o do RSU

// If the sender is not the server, add it to the neighbors
String senderId = msg.getRouting().getSource().getSourceName();
if (!senderId.equals("server_0")) {
    neighbors.add(senderId);
}

// Vers o do Veiculo

// If the sender is a vehicle, add it to the neighbors
String senderId = msg.getRouting().getSource().getSourceName();
if (senderId.contains("veh")) {
    neighbors.add(senderId);
}
```

Sempre que recebem uma mensagem, se a mensagem vier de um veículo, este é diretamente adicionado à lista de vizinhos pois caso não fosse um vizinho direto seria impossível receber uma mensagem dele.

Método: `onMessageReceived(...)` e `handleCamReceived(CamMessage cam)`


```

private void handleCamReceived(CamMessage cam) {

    String camVehId = cam.getVehId();
    seenCams.put(camVehId, cam);

    double distance = get0s().getPosition().distanceTo(cam.getPosition());
    boolean reachable = distance < TX_RANGE_M;
    if (reachable) {
        neighbors.add(camVehId);
    } else {
        neighbors.remove(camVehId);
    }

    // Forward message if HTL allows
    if (cam.getHopsToLive() > 0) {
        forwardCamMessage(cam);
    }
}

```

A cada CAM é inspecionada a posição do veículo que gerou a mesma e atualizada a lista de vizinhos consoante a informação obtida. Este mecanismo adiciona uma camada de robustez à lista de vizinhos, mantendo-se mais atualizada.

Enquanto cada veículo que receba uma CAM reenvia-a por broadcast para todos os seus vizinhos, os RSUs que recebam uma CAM apenas a propagam para o Fog, deixando a tarefa de informar outros veículos para os mesmos, reduzindo um pouco a sobrecarga da rede.

E. Encaminhamento de Mensagens AccidentEvent (Downstream)

Métodos: `handleEventReceived(EventMessage event)`, `onMessageReceived(V2xMessage msg)`, `selectNextHop(String target)` e `getClosestNeighbor(String target)`

```

private void handleEventReceived(EventMessage event) {
    // If the event is expired, do not process it
    if (event.getExpiryTimestamp() < get0s().getSimulationTime()) {
        logInfo(String.format(
            "EVENT_NOT_FORWARDED : UNIQUE_ID: %s | VEHICLE_TARGET: %s | "
            "EVENT_EXPIRED",
            event.getId(), event.getTarget()));
        return;
    }

    // Select next hop for forwarding
    String target = event.getTarget();
    String nextHop = selectNextHop(target);
    // If no next hop is found, do not forward the event
    if (nextHop == null) {
        logInfo(String.format(
            "EVENT_NOT_FORWARDED : UNIQUE_ID: %s | VEHICLE_TARGET: %s | "
            "NO_NEXT_HOP_FOUND",
            event.getId(), target));
        return;
    }

    // Add the next hop to the forwarding trail
    List<String> forwardingTrail = event.getForwardingTrail();
    forwardingTrail.add(nextHop);

    // Copy the message to forward the event and send it
    EventMessage eventCopy = new AccidentEvent(...);
    get0s().getAdHocModule().sendV2xMessage(eventCopy);
}

```

```

else if (msg instanceof EventMessage event && event.hasNextHop() &&
    event.getNextHop().equals(vehId) && processedEvents.add(event.getId()))
) {
    // Process the Event message if it is for this RSU
    handleEventReceived(event);
}

```

Um evento só é processado caso seja direcionado ao nó que recebeu a mensagem e não tenha ainda sido processado antes, evitando ciclos. Para além disso é ainda verificada a validade temporal do mesmo, sendo descartado se já tiver expirado.

```

private String selectNextHop(String target) {
    // Check if the destination is reachable directly (1 hop)
    if (neighbors.contains(target)) { return target; }

    // If no direct neighbor is found, select the closest neighbor to the
    // destination
    return getClosestNeighbor(target);
}

private String getClosestNeighbor(String target) {
    CamMessage targetCam = seenCams.get(target);
    if (targetCam == null) {
        return null; // No CAM info available for the target
    }

    // Get the current target position
    GeoPoint targetPosition = targetCam.getPosition();
    double minDistance = Double.MAX_VALUE;
    String closestNeighbor = null;

    // Search for the closest neighbor to the target position
    for (String neighbor : neighbors) {
        CamMessage neighborCam = seenCams.get(neighbor);
        if (neighborCam != null) {
            if (neighborCam.getNeighbors().contains(target)) {
                // If the neighbor is directly connected to the target, return it
                // immediately
                return neighbor;
            }
            GeoPoint neighborPosition = neighborCam.getPosition();
            double distance = neighborPosition.distanceTo(targetPosition);
            if (distance <= minDistance) {
                minDistance = distance;
                closestNeighbor = neighbor;
            }
        }
    }
    return closestNeighbor;
}

```

Para o encaminhamento foi adotada uma implementação baseada no algoritmo de Greedy Forwarding. Primeiro o destino é procurado na lista de vizinhos diretos. Caso não seja encontrado, ou seja esteja a mais de 140 metros do nó atual, passamos a procurar pelo vizinho que está mais próximo do destino geograficamente. Para isso é necessário saber a localização teórica do destino, no entanto como as CAMs estão limitadas a 5 saltos isso nem sempre é possível. No caso de não conseguir encontrar um bom candidato a reenviar a mensagem, a mensagem é descartada, deixando um aviso “NO_NEXT_HOP_FOUND” nos logs para análise. Isto evita que mensagens possam ficar eternamente na rede à procura de um destino que esteja inacessível, inundando por completo a mesma.

Durante a pesquisa pelo vizinho mais próximo, são também analisados os vizinhos destes, para o caso de algum vizinho ter já o destino na sua lista de vizinhos, podendo terminar a pesquisa mais cedo, aliviando um pouco a carga imposta na simulação. No futuro, seria recomendado implementar um método de pesquisa que permitisse a remoção completa deste ciclo para tornar a implementação mais escalável sem grandes perdas de desempenho.

VI. TESTES E RESULTADOS

Para testar a nossa rede veicular, realizamos uma série de testes, analisando o seu comportamento com base nos diversos logs gerados.

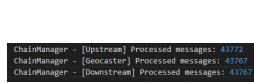


Fig. 4. Mensagens trocadas no cenário.

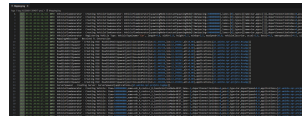


Fig. 5. Criação e posicionamento dos RSUs.

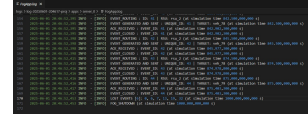


Fig. 6. Receção dos eventos no Fog.

Na Figura 4, é possível observar uma série de mensagens que foram processadas durante uma simulação completa. Este resultado comprova que a simulação está funcional e apresenta um nível razoável de complexidade devido à quantidade elevada de mensagens geradas.

Na Figura 5, apresentam-se os logs referentes à criação das RSUs na simulação. Observa-se que todos os componentes foram inicializados corretamente, confirmando o funcionamento adequado do sistema.

O Fog gera os eventos como exibido na Figura 6, onde são criados os eventos e enviados via RSU. Podemos observar que os veículos destino mudam. Também recebe corretamente os ACKs e mostra os eventos perdidos. Alguns dos eventos perdidos são simplesmente perdido devido à criação de grupos de veículos isolados que apesar de trocarem mensagens entre si, não comunicam com o Fog.

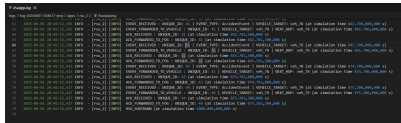


Fig. 7. Receção de eventos e ACKs nos RSUs.

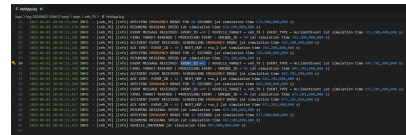


Fig. 8. Eventos e ACKs na aplicação do veículo.

Na Figura 7, é possível observar a receção de mensagens no RSU enviadas pelo Fog. Logo após a receção dessas mensagens, acontece o processo de colocação da mensagens na rede veicular. Neste caso em específico, o Rsu_2 envia os eventos recebidos do Fog, para os veículos 78 e 79. Logo após o envio, é recebido o ACK de volta o que significa que mensagem foi colocada na rede veicular com sucesso.

Já na Figura 8, podemos verificar que os veículo destino (Neste caso o veículo 79) recebe constantemente as notificações de evento enviadas pelo Fog. Além disso, também é possível observar a alteração de comportamento do veículo, reagindo conforme os parâmetros enviados na mensagem.

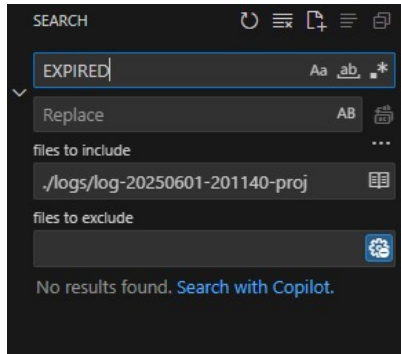


Fig. 9. Eventos expirados antes de serem entregues.

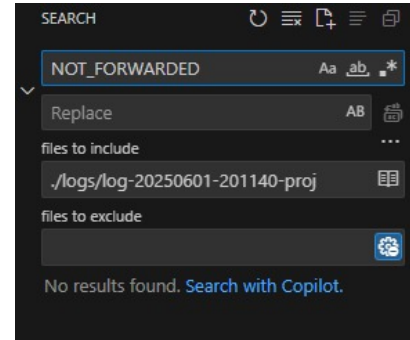


Fig. 10. Eventos que não foram encaminhados.

Nas Figuras 9 e 10, analisamos os eventos que foram marcados como *Expired* e *Not Forwarded*, respectivamente. Verifica-se que praticamente não há mensagens nestas categorias, o que indica que a maioria dos eventos não falha devido a expiração do tempo de vida (*TTL*) nem por falta de vizinhos (*no next hop*).

Isto sugere que os eventos perdidos resultam de tentativas de envio para nós que, embora anteriormente considerados o melhor próximo salto, já se encontram fora do alcance de comunicação no momento da transmissão. Ou seja, o caminho calculado deixou de ser válido devido à mobilidade dos veículos, comprometendo a entrega da mensagem.

VII. CONCLUSÃO E TRABALHO FUTURO

O presente trabalho prático, consistiu na criação de um protótipo de simulação de uma rede veicular inteligente, recorrendo à plataforma Eclipse MOSAIC. O sistema simulado inclui três tipos de nós principais: veículos, RSUs e um *Fog*, permitindo reproduzir de forma realista um cenário de comunicação comunicação V2X.

O principal objetivo foi permitir a comunicação entre veículos através de mensagens CAM, com partilha periódica de informação de mobilidade, e a disseminação de eventos de perigo (como acidentes), originados ou processados no nó *Fog*. Para isso, foram implementadas três aplicações distintas — **VehApp**, **RsuApp** e **FogApp** — com responsabilidades bem definidas e mecanismos de encaminhamento baseados em AdHoc. As RSUs funcionaram como nós intermédios, com acesso a ambas as redes, como os relays, permitindo encaminhar mensagens entre veículos e o *Fog*.

Entre as principais funcionalidades destacam-se:

- **Mensagens CAM Dinâmicas:** Os veículos emitem mensagens CAM com intervalos adaptativos (100 ms–1 s), sempre que detetadas alterações relevantes na posição, direção ou velocidade.
- **Alertas de Acidente (AccidentEvent):** Gerados pelo **FogApp** com 2% de probabilidade, incluem severidade (0 a 2) e são válidos por 8 s. São enviados ao RSU mais próximo do veículo-alvo, com base na sua localização.
- **Reação a Eventos:** Os veículos reagem automaticamente a eventos destinados a si, realizando uma travagem de emergência por 15, 30 ou 60 segundos, consoante a severidade.

- **Encaminhamento Multihop:** As aplicações **VehApp** e **RsuApp** escolhem o próximo salto com base na proximidade geográfica ao destino, propagando eventos e ACKs até ao seu destino ou até atingir o limite de saltos.
- **Gestão de Vizinhos em Tempo Real:** As listas de vizinhos são atualizadas com base nas CAMs recebidas e expurgadas se estiverem desatualizadas (> 2 s).
- **Registo de Logs Detalhado:** Todas as aplicações registam eventos como envios, receções, perdas e expirações, facilitando a validação, testes e análise do desempenho da rede.

Entre algumas funcionalidades que poderiam ser implementadas como trabalho futuro:

- **Casos de uso alargados:** Incluir cenários mais complexos com mais mensagens de acidente e diferentes tipos de risco, aumentando a cobertura funcional do sistema.
- **Mensagens CAM mais completas:** Incluir mais campos com informações nas mensagens CAM.
- **Cálculo de risco e decisão baseada em CAMs:** Substituir a geração aleatória de eventos por uma abordagem baseada na análise das mensagens CAM recebidas, permitindo calcular o risco real com base em dados como velocidade, posição e direção, e tomar decisões mais informadas e contextuais sobre a criação e disseminação de alertas.
- **Integração com Fog Cloud:** Possibilidade de encaminhar certos eventos para a Fog Cloud para análises globais.
- **Encaminhamento inteligente nos veículos:** Desenvolver um método mais avançado de encaminhamento de mensagens nos veículos, como o Adaptive Forwarding (AF), que considere critérios dinâmicos como proximidade ao destino, qualidade do enlace e mobilidade dos nós para melhorar a entrega das mensagens.
- **Algoritmo para a seleção de vizinhos mais eficiente:** Substituir o processo iterativo com **for** não recomendado por uma estrutura baseada em grafos para selecionar os melhores vizinhos.

De forma geral, consideramos que cumprimos os objetivos propostos, desenvolvendo um protótipo funcional que simula eficazmente uma rede veicular inteligente com comunicação V2X.