Universidade do Minho
Escola de Engenharia

UNIVERSIDADE DO MINHO

Implementação de um Serviço de difusão de informação de trânsito numa Rede Veicular (VANET de Dados Nomeados)

Autores:

Bruno Rodrigues: pg41066

Carlos Alves pg41840

Paulo Bento a81139

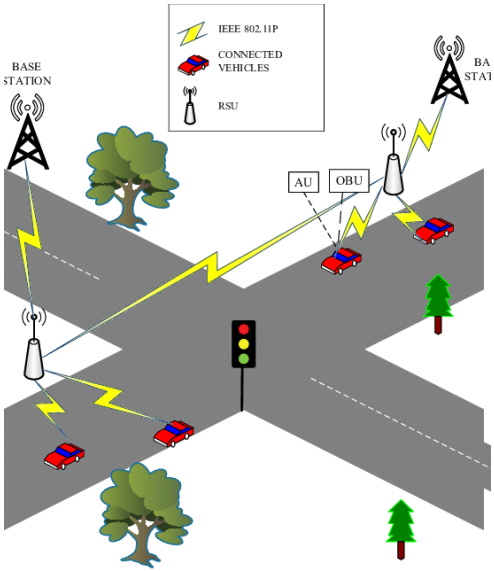
ÍNDICE

INTRODUÇÃO

Este trabalho prático sugerido na Unidade Curricular – Arquitetura Emergentes de Redes tem como objetivo implementar um serviço de difusão de informação de transito numa rede veicular, isto é Vanet de dados Nomeados. Esta implantação está dividida em três fases, sendo elas:

- Fase 1 – Consiste na implementação de um protótipo de uma rede de formação espontânea, Adhoc, em que os nós são veículos automóveis equipados com OBUs(On Board Units) com capacidade Wifi. Nesta fase serão implementados protocolos de encaminhamento, protocolo HELLO.
- Fase 2 – Consiste na implementação do serviço de difusão de informação de transito. Para esta fase é utilizado a implementação da Fase 1, a VANET.
- Fase 3 –

Neste relatório consta uma contextualização da arquitetura em questão, dos protocolos, escolhas tomadas nas implementações das diferentes fases. E por fim, espera-se que o projeto demonstre as vantagens que se verificam entre ambas as abordagens, criação de VANET de Dados Nomeados para disponibilizar um Serviço de difusão de informações de transito.



Exemplo de uma arquitetura VANET

FASE 1 – ESTABELECIMENTO DE UM VANET

Como já referido, esta fase consiste essencialmente na implementação de um protótipo de uma rede de formação espontânea (Adhoc), onde os nós serão veículos equipados com On Board Units com capacidade Wifi e cada um dos nós deverá ser capaz de descobrir rotas para os outros nós(veículos), isto com ajuda dos nós vizinhos. Para isto foi então implementado um protocolo de encaminhamento, neste caso o Protocolo HELLO que abordaremos de seguida.

Protocolo HELLO

Segundo a descrição que está presente no enunciado, o protocolo HELLO é então o responsável por descobrir as tais rotas entre os veículos(nós) – a dois saltos de distancia no máximo, visto que a topologia consistirá numa rede veicular, onde os nós-veículos estarão em constante movimento, logo é necessário que este protocolo esteja sempre em execução de modo a calcular as rotas em diferentes períodos de tempos.

Na implementação deste protocolo tentamos seguir a estratégia sugerida pelos docentes. Antes de desenvolver propriamente o protocolo e as respetivas operações, começamos por declarar o tipo de eventos que poderão ocorrer, as estruturas dos dados a serem retirados ou que cada nó possuirá.

Nos possíveis Eventos temos:

- Hello Init – O evento que inicia o processo de descoberta num raio de 2;
- HELLO_RESPONSE – A resposta do Hello_Init, apenas para propósitos distintos;
- ROUTE_REQUEST – A solicitação de uma inundação para encontrar um host específico;
- ROUTE_REPLY – Resposta de uma Route_Request bem-sucedida;
- GET_INFO – Obtém informações pelo Cliente;
- PUT_INFO – Colocação de informações pelo Cliente;
- REPLY_INFO – Resposta de uma obtenção de informações pelo Cliente;
- ROUTE_SUCESS – Evento interno para informar que a solicitação de rota que "solicitou" foi bem-sucedida;
- ROUTE_FAILED – Evento interno para informar que a solicitação de rota que solicitou falhou.

Na **class Dispatcher**, esta tem como objetivo encarregar-se da distribuição dos pacotes, comunica com o exterior por UDP, ou melhor, fica de imediato á escuta na porta 9999 em UDP. Trata dos eventos, é aqui que ficam em espera os eventos a serem processados ou

redirecionados para determinado lugar onde devem ser processados, por exemplo um novo Hello_Init deve enviar a mensagem broacast para todos, um HELLO_Response deve ser redirecionado para a classe Routing_Manager. Ainda no Dispatcher, este distribui conforme o tipo de pacotes, por exemplo se veio do Net Layes ou do routing.

De forma sucinta, aqui é feita toda a gestão dos eventos (Hello, Route), gestão das tabelas de routing e ainda trata da comunicação com o exterior por UDP.

Routing

Nesta fase fez o protocolo Hello, para isso foi necessário desenvolver configuração do protocolo, onde designamos diversos fatores como:

- O tempo que precisa ocorrer para ser considerado “morto”.
- O tempo entre Hello_init ser adicionado á queue de eventos.
- Número de atualizações na tabela de routing necessárias para verificar se há uma entrada de lixo(inoperante) que deva ser excluída.
- Número de vezes que é preciso ser marcado como morto para ser realmente excluído. (Isto serve para os hosts mais próximos também possam ser classificados como “morto”);
- O tamanho máximo da tabela enviada num pacote HELLO.

Configurações Route Request e Reply

- Número de saltos antes que este pacote seja descartado;
- Tempo de espera antes que a solicitação de rota seja descartada;
- Tempo em que o nó está indisponível apos uma Route_Request falhado.

Foram desenvolvidas algumas funções/mecanismos que tratam o “lixo”, isto é, coletam os nós “mortos”, limpam o lixo de ips marcando-os como “mortos”. Um sistema capaz de descobrir quando o route Request falhou, isto é, através do tempo e de um thread que monitoriza quais já ultrapassaram o tempo por nós declarado (nas configurações) e retira-os da espera. Basicamente, o mecanismo de limpar o lixo consiste num threshold que ativa a recolha de lixo marcando como “mortos” as entradas da routing table. Este threshold é associado com o número de atualizações a tabela.

Essencialmente, o protocolo Hello acorda uma threasd de um determinado tempo a outro, para criar e passar para o Dispatcher os eventos do Hello para este fazer Multicast como é referenciado no enunciado.

Ainda o protocolo route Request que tem como função solicitar a “descoberta” da rota do pacote.

Como já referido, ainda temos um parâmetro de configuração para o tempo entre inits, o raio máximo do route request ttl, quantos updates a tabela para recolher lixo, entre outros. Para uma total Descrição dos parâmetros utilizados é aconselhado que se observe o script Routing_Manager na integra.

Os pacotes do Routing_Manager contam com um sequence number, o hostname, ip e vizinhos no caso do Hello ou seja hosts que estejam a raio 1.

No caso do Route Request temos os parâmetros já mencionados acima, desenvolvemos métodos criam o Route Request recebendo o destino, ttl e tabela. Conforme vai realizando o caminho vai adicionando á tabela as diversas informações (raio, nó e outros), também conforme o caminho que já realizou o Route Request até ao nó onde se encontra.

De volta ao Dispatcher, este quando divide os pacotes, por vezes realiza algumas operações em conjunto como por exemplo tentar procurar o ip do próximo salto e iniciar um route Request caso não consiga determinar qual o nó de destino. Além so adiciona um Route Request caso não esteja outro Route Request com o mesmo ip de destino ou já se conheça o ip de destino, foram estas algumas das operações que acabamos por desenvolver nesta implementação.

Fundamentalmente nesta fase foi implementado o protocolo de HELLO, onde em cada veículo(nó) da nossa topologia é executado uma instância do programa. Num primeiro instante, nenhum veículo se conhece exceto ele próprio. Mesmo assim este nó faz uso do UDP na porta 9999 e fica à escuta, assim poderá se `MULTICAST_IP = "ff02::abcd:1"` comunicar com os restantes veículos, pelo menos aqueles que se encontram na vizinhança. Primeiramente, implementamos o mecanismo de modo que conseguíssemos enviar mensagem Hello para os nós, tais mensagens são enviadas por multicast para o endereço IPv6 do grupo

Portanto, supondo que os veículos ou eles recebessem a mensagem e enviassem para outros vizinhos, no fim foi possível determinar quem era vizinho de quem, qual o melhor caminho, e claro, como os nós estão sempre em movimento, alem disso este processo precisa de ser continuo.

Os pacotes Hello são enviados periodicamente, quando um nó recebe um pacote Hello de um vizinho, responde com outro pacote Hello que contém o identificador de cada nó que conhece. Quando um nó recebe um pacote Hello que contém o seu identificador num dos campos de identificação de vizinhos assume que a vizinhança está. A relação de vizinhança é quebrada com um vizinho caso este deixe de receber pacotes Hello desse vizinho ao fim de um dado intervalo.

Depois de este processo e com auxílio dos métodos e mecanismos presentes na nossa implantação, conseguimos obter uma tabela de encaminhamento deste género.

E é através desta tabela que o processo de entregas pode ser feito sem muita dificuldade, pois já se conhece os vizinhos.

Em relação ao nós fora da vizinhança, estes são tratados pelo protocolo Route_Request que já foi explicado, mas basicamente um mecanismo de flooding, isto quer dizer, enviar mensagem para todos (Para redes muito grandes talvez não seja a melhor escolha). O processo só é atingido caso este consiga determinar uma rota valida, respondendo com um Route_Reply, exatamente como os algoritmos backtrack, este é faz o caminho inverso aquele que percorreu, e claro, as tabelas neste momento são atualizadas com os novos dados do percurso. Neste protocolo é então definido as tais configurações já mencionadas.

FASE 2 – IMPLEMENTAÇÃO DE UM SERVIÇO DE DIFUSÃO DE INFORMAÇÕES DE TRÂNSITO

Nesta fase era pedido para implementar um serviço de difusão de informações de trânsito, isto é, a implementação de um protocolo que permite ao cliente e ao servidor trocar mensagens entre si.

Foi então implementado o Net Layer, que cria os pedidos de get e put para o cliente e servidor. A implementação é igual tanto para o servidor como cliente, sendo o que os define é uma flag (um bool) que nos indica se se trata do cliente ou servidor.

Esta classe vai também permitir a comunicação entre aplicações através do telnet, onde podemos, por exemplo, introduzir o comando “Telnet -6 ip6-localhost 9999”. A partir daqui já podemos fazer um get, ou um put de informações de uma certa zona, para um certo node.

Vamos imaginar que queremos obter informações do CoreNode8 (isto supondo que o CoreNode7 é um servidor). Podemos fazer então “GET UM CoreNode8”, para obter informações

relativas à zona. Também podemos fazer um “put” de informação, onde as informações que queremos enviar são colocadas no segundo argumento do comando.

O comando pode ser então “PUT CoreNode8 UM Acidente”. Isto vai informar o CoreNode8 que houve um acidente na UM.

Estes pedidos são então processados por uma thread criada onde corre a função processing(), onde o processamento acontece de acordo com a sua origem e a sua intenção,

Nesta classe corre também uma socket tcp, gerida pela função listener_TCP(), que nos vai permitir conectar com o telnet.

A classe Dispacher, quando faz um route request, e recebe um route reply, vai buscar os pedidos get e put que estavam à espera de um endereço, e vai enviá-los automaticamente.

Caso exista um route failed, estes pedidos são então retirados da lista de espera do Dispatcher.

Todas a classes têm uma thread processing, que processa os eventos enviados pela classe com que se está a comunicar, por exemplo, os pacotes recebidos da classe net layer e pelo routing manager para a classe Dispatcher, da classe Dispatcher para o routing manager, etc.

FASE 3 – ESTABELECIMENTO DE UMA NDN

A fase 3 do trabalho consiste numa implementação alternativa da anterior usando o paradigma NDN. Assim, implementar-se-á um serviço equivalente ao das fases anteriores. Invés, de IPs as informações são fornecidas através de Nomes. Podendo a informação ser produzida pelos utilizadores, e depois inquirida por outros.

Nesta fase apresenta-se umas classes semelhantes as anteriores, Routing_Manager, Dispatcher, Client. O Routing_Manager trata do processamento de pacotes, de atualizar as estruturas de dados, e, neste caso, de fazer a ligação com a parte direta ao utilizador. Essa parte (Client) só fornece, como na fase anterior, uma conexão TCP para o cliente conectar-se através de Telnet, por exemplo. Como também, as funções de parse para os dados fornecidos pelo utilizador.

O Dispatcher encarrega-se só de encaminhar os pacotes para o Routing_Manger ou fazer MultiCast dos mesmos por meio de um Thread dedicada a esse propósito. Em conjunto, com outra que serve como "conexão" UDP para realizar o MultiCast. Assim, conclui-se que a arquitetura é por Camadas. Sendo o Client a camada superior, o Routing_Manager do meio e a classe Dispatcher a inferior.

As estruturas de dados do Client incluem:

- Content_Store -> Onde guarda-se a informação produzida ou obtido por redirecionamento dos pacotes, mantendo-os em cache
- PIT -> Os pedidos de informação que estão a espera dos dados, sejam do utilizador do Host ou de outro.
- FIB -> Esta estrutura de dados apesar de, estar presente numa arquitetura normal de NDN não é tida em conta nesta, visto que sem algoritmo de routing e sendo o contexto uma VANET tornou-se complicado averiguar como.

Ainda existe duas threads que servem para limpar os conteúdos da PIT e do Content_Store. Os parâmetros que determinam o tempo de vida nesses campos são passados como parâmetros a classe Client. Os conteúdos que são limpos são todos os que passam esses parâmetros e no caso da Content_Store os que não pertencem ao Host e que ultrapassam o tempo limite.

Os pacotes desta fase só tem três categorias possíveis:

- Interest - Neste pacote é incluído o nome do conteúdo, caso queira-se que ele seja recente(Fresh), o nonce para saber se os pedidos são novos e evitar ciclos, lifetime que é o tempo que ele pode ficar na PIT e finalmente o Hop_Limit que é o numero máximo de saltos do pedido.
- Store - Este pedido só é feito dentro do próprio Host, guarda o conteúdo, o número de sequência, o nome associado, o tempo em que foi criado e, também o freshness_period que serve para marcar como novo, no caso da fonte original é marcado com um número negativo.
- Response - Exatamente, o mesmo formato que os pedidos do Store com a ligeira diferença que o Freshness_period é atualizado para um tempo definido pela própria aplicação.

CONCLUSÃO

Neste trabalho era pedido para implementar um Serviço de difusão de informação de trânsito numa Rede Veicular, sendo que consistia em três partes. Na primeira fase foi necessário implementar de uma VANET, onde foi implementado o Protocolo Hello e Route_Request. A segunda fase consistiu na implementação do serviço de difusão de informações de trânsito, onde o objetivo seria permitir a comunicação (troca de mensagens) entre servidor e cliente. A terceira fase consistia numa alternativa às redes IP, a implementação de uma rede NDN. Todas estas fases seriam então emuladas pela plataforma CORE, que nos iria providenciar uma simulação das redes veiculares.

Acreditamos que o trabalho realizado e funcional ficou num nível satisfatório, onde as Fase 1 e Fase 2 foram implementadas completamente, cobrindo exatamente os tópicos pedidos pelo enunciado. A única dificuldade encontrada na realização do trabalho passou-se na Fase 3, na implementação da rede NDN, onde ficou por completar a funcionalidade de pedidos Fresh, isto é, pedidos recentes não ficaram funcionais, mas que o resto das funcionalidades implementadas se encontram funcionais.

É de mencionar que a realização deste trabalho permitiu um aprofundamento e um enorme complemento para o que foi apreendido em sala de aula, e que foi uma mais valia para o desenvolvimento das nossas capacidades e conhecimentos na área de redes.