5
Exemplo de Uso do MobiCS2

Neste capítulo será apresentada a implementação de um protocolo de roteamento e de uma simulação no MobiCS2. Porém, antes de abordarmos o protocolo de roteamento, iremos discutir alguns outros protocolos e abstrações adicionais que se fazem necessárias para implementar um protocolo de roteamento e sua simulação.

5.1.
A Interface da SDU de um Protocolo

Como mencionado no capítulo 4, para se desenvolver um novo protocolo no MobiCS2 é necessário estender as classes abstratas `Protocol`, `ServiceDataUnit`, `ProtocolDataUnit` e `Address`.

Porém, para alguns protocolos, além destas extensões, também é necessário que seja definida a interface da sua unidade de dados do serviço (SDU), que representa a funcionalidade mínima que a classe SDU tem que prover para que o protocolo possa processá-la. Isto é necessário pois um protocolo pode receber SDUs de qualquer protocolo "cliente", que podem apresentar diferentes "formatos".

Por exemplo, o protocolo `WiFiMAC` (ver seção 4.4) utiliza a interface `MacSDU`, que define o método `getDestination()`, o qual retorna o `MacAddress` do destino da unidade de dados. Portanto, para que um protocolo "cliente" do `WiFiMAC` possa enviar uma unidade de dados para o `WiFiMAC`, é necessário que a unidade de dados implemente a interface `MacSDU`.

5.2.
Interface de Rede

A abstração da interface de rede define uma interface entre os protocolos de mais alto nível (p. ex., protocolo de roteamento) e os protocolos de mais baixo

PUC-Rio - Certificação Digital Nº 0016060/CA

nível (p. ex., protocolos da camada física/MAC, como o `WiFiPhy` e `WiFiMAC`). A interface de rede oculta detalhes dos protocolos de baixo nível, permitindo que os protocolos de mais alto nível possam interagir com uma variedade de dispositivos de rede diferentes (p.ex., IEEE 802.11 ou Bluetooth) usando as mesmas interfaces.

No MobiCS2 a interface de rede é representada pela classe abstrata `NetIf` (do inglês *Network Interface*), que é derivada da classe `Protocol`. A classe `NetIf` precisa necessariamente ser derivada para uma classe concreta, devendo ser especializada para servir de interface para um determinado dispositivo de rede. Por exemplo, o MobiCS2 disponibiliza a classe `NetIfIEEE802` (derivada de `NetIf`), que é a interface de rede que deve ser utilizada em conjunto com as classes `WiFiPhy` e `WiFiMAC`.

A classe `NetIf` armazena e disponibiliza uma série de dados sobre a interface de rede, como por exemplo, o seu endereço IP (*Internet Protocol*) e o seu endereço físico. O endereço IP é representado pela classe `IpAddress`, derivada da classe `Address`. Já o endereço físico vai depender do dispositivo de rede utilizado. Por exemplo, no caso da classe `WiFiMAC`, o endereço físico é representado pela classe `MacAddress`.

Para facilitar o uso do framework, a classe `NetIfIEEE802` gera automaticamente o seu endereço IP, que será único em toda a rede simulada.

A interface de rede define uma interface para sua SDU, chamada `NetIfSDU`, a qual define dois métodos. O método `getNextHop()` retorna o endereço IP (`IpAddress`) do próximo nó da rede para encaminhamento da unidade de dados. O método `getType()` retorna o tipo do protocolo que gerou a SDU, que serve para que o `NetIf` possa de-multiplexar a unidade de dados recebida no nó de destino e repassá-la para o protocolo correto.

A classe `Node` disponibiliza dois métodos para facilitar o acesso às interfaces de rede de um nó. O método `getNumNetIf()` retorna o número de interfaces de rede que o nó possui. O método `getNetIf()` recebe como parâmetro o número (inteiro `int`) da interface de rede e retorna o objeto da interface de rede (`NetIf`). O número da interface de rede de um nó inicia no número zero. Assim, caso se queira descobrir o endereço IP da primeira interface

PUC-Rio - Certificação Digital Nº 0016060/CA

de rede de um nó chamado "`n1`", pode-se utilizar o comando "`n1.getNetIf(0).getIpAddress()`".

5.3. Conversão de Endereços

Para resolver o problema do mapeamento de endereços IP para endereços físicos, o framework disponibiliza também o protocolo ARP (*Address Resolution Protocol*). Comer (1998) apresenta uma descrição do protocolo ARP, e Comer & Stevens (1999) discutem detalhes de uma implementação deste protocolo.

As seguintes classes e interface fazem parte da implementação do protocolo ARP no MobiCS2:

- Classe `Arp` – derivada da classe `Protocol`. Implementa um cache para mapeamento de endereços IP para endereços físicos, e uma fila para armazenar as unidades de dados que estão aguardando a resposta a uma requisição de endereço.
- Interface `ArpSDU` – define a interface mínima que uma SDU do protocolo ARP deve oferecer.
- Classes `ArpRequest` e `ArpReply` – ambas derivadas da classe `ServiceDataUnit`, e implementam a interface `NetIfSDU`. O `ArpRequest` representa uma requisição de endereço físico do protocolo ARP, e o `ArpReply` representa uma resposta a uma requisição.

A seguir será descrito, de forma resumida, o funcionamento do protocolo `Arp`, com o objetivo de mostrar o relacionamento do `Arp` com os demais protocolos do MobiCS2, exemplificados aqui pelas classes dos protocolos do padrão IEEE 802.11 e pelo protocolo de roteamento, como mostra a Figura 11.

PUC-Rio - Certificação Digital Nº 0016060/CA

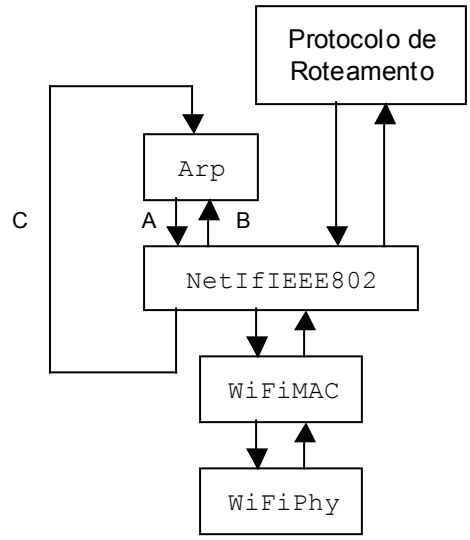

Figura 11 – Relação entre o `Arp` e os demais protocolos

Quando o protocolo de roteamento envia uma unidade de dados para a interface de rede (`NetIfIEEE802`), esta verifica o endereço IP do próximo nó no encaminhamento (`getNextHop()`). Caso este endereço seja um endereço de broadcast, o `NetIfIEEE802` monta a PDU com o endereço físico de broadcast e envia para o `WiFiMAC`. Caso o endereço IP do próximo nó seja um endereço de unicast, o `NetIfIEEE802` monta uma PDU com o campo de endereço físico de destino vazio e a envia para o `Arp` (Figura 11 – seta C) para que este a preencha.

Quando o protocolo `Arp` recebe a sua SDU (`whenSDU()` – Figura 11 – seta C), este verifica no seu cache se possui o endereço físico de destino. Caso possua, então completa o campo de endereço físico do destino e envia a unidade de dados para o `NetIfIEEE802` (Figura 11 – seta A). Caso contrário, o `Arp` coloca a unidade de dados em uma fila de espera, e envia por broadcast uma requisição de endereço físico (`ArpRequest`) ao `NetIfIEEE802` (Figura 11 – seta A).

Quando o protocolo `Arp` recebe um `ArpRequest` (`whenArpRequest()` – Figura 11 – seta B), este verifica se é seu o endereço IP procurado. Caso seja, o `Arp` envia de volta uma resposta à requisição (`ArpReply`) por unicast, informando o seu endereço físico (Figura 11 – seta A). Caso contrário, descarta o `ArpRequest` recebido.

Quando o protocolo `Arp` recebe o `ArpReply` (`whenArpReply()` – Figura 11 – seta B), este atualiza o seu cache, retira da fila de espera todas as

PUC-Rio - Certificação Digital Nº 0016060/CA

unidades de dados relacionadas com esta requisição, completa o campo de endereço físico de destino destas unidades de dados, e as envia para o `NetIfIEEE802` (Figura 11 – seta A).

Como se pode observar, o protocolo `Arp` é ao mesmo tempo cliente e fornecedor do protocolo da interface de rede (`NetIfIEEE802`), pois o protocolo `Arp` utiliza a interface de rede para acesso ao sistema de comunicação (relação de cliente – Figura 11 – setas A e B), e ao mesmo tempo o `Arp` fornece para a interface de rede o serviço de conversão de endereços (relação de fornecedor – Figura 11 – seta C).

Como o protocolo `Arp` sempre utiliza a interface de rede para ter acesso ao sistema de comunicação, o `Arp` é independente do dispositivo de rede utilizado.

A classe `Arp` foi projetada de forma que possibilite que apenas uma instância dela possa ser compartilhada por várias interfaces de rede, ao mesmo tempo, em um mesmo nó.

Para que a classe `Arp` funcione, é necessário que a interface de rede possibilite transmissões em broadcast para a rede física, porque o protocolo `Arp` utiliza o broadcast na rede física para difundir o `ArpRequest`.

5.4. A Classe `WiFiNode`

Para facilitar a criação de simulações, o framework disponibiliza a classe `WiFiNode`, derivada da classe `Node`. O `WiFiNode` é um nó da rede que já instancia automaticamente todos os protocolos de baixo nível necessários para a comunicação de um nó, utilizando o padrão IEEE 802.11, e disponibilizando também a interface de rede e o protocolo ARP.

As classes de protocolos que são instanciadas pelo `WiFiNode` são: `WiFiPhy`, `WiFiMAC`, `NetIfIEEE802` e `Arp`. Esses protocolos já são todos interconectados, formando um grafo de protocolos igual ao mostrado na Figura 11.

Como o protocolo `WiFiPhy` já se conecta (`attach()`) automaticamente à mídia `Air`, e também já aloca um canal (`allocateChannel()`) nesta mídia, então basta instanciar o `WiFiNode` para ter um nó de rede com acesso ao sistema

PUC-Rio - Certificação Digital Nº 0016060/CA

de comunicação, faltando apenas incluir os protocolos de alto nível, que deve começar pelo protocolo de roteamento.

Ao instanciar um protocolo de roteamento, como todo protocolo, este já é incluído automaticamente no nó de rede informado no seu construtor, basta então chamar o método `connectRoutingProtocol()` da classe `WiFiNode`, para que o protocolo de roteamento seja conectado ao restante da pilha de protocolos do `WiFiNode`, como é mostrado na Figura 11.

De forma a padronizar as conexões da interface de rede (classe `NetIf`) com os demais protocolos, foi estabelecido que o protocolo `Arp` é sempre o primeiro protocolo cliente da interface de rede, e o protocolo de roteamento é o segundo cliente. Isto foi definido assim, porque na classe `WiFiNode` o `Arp` é conectado primeiro à interface de rede, durante a execução do construtor da classe `WiFiNode`, e o protocolo de roteamento é conectado num momento posterior pelo método `connectRoutingProtocol()`.

Quanto aos protocolos fornecedores da interface de rede, foi padronizado também que o `Arp` será sempre o primeiro fornecedor, e o segundo fornecedor será o protocolo superior do dispositivo de rede. No caso da implementação do IEEE 802.11 no MobiCS2, este protocolo é o `WiFiMAC`.

A informação desta padronização é necessária para o usuário do framework que queira programar explicitamente as conexões da interface de rede, sem utilizar a classe `WiFiNode`. É útil também para o usuário que queira desenvolver uma nova interface de rede, derivada da classe `NetIf`.

A Figura 12 ilustra as conexões da interface de rede (`NetIf`) com os demais protocolos, destacando a ordem dos protocolos clientes e fornecedores da interface de rede.

PUC-Rio - Certificação Digital Nº 0016060/CA

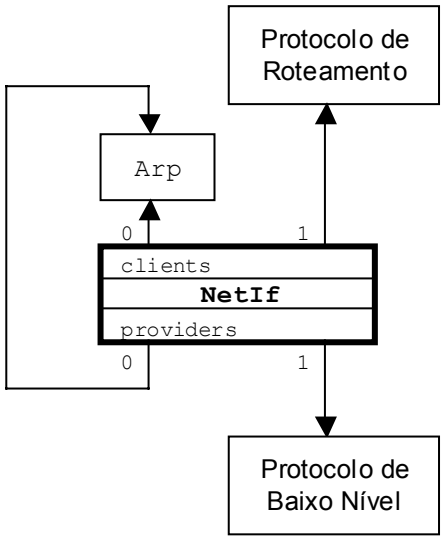

Figura 12 – Conexões da interface de rede (`NetIf`) com os demais protocolos

5.5. Implementação do Protocolo de Roteamento DSR

A partir do estudo que foi feito sobre os diferentes protocolos de roteamento para redes móveis ad hoc (ver Apêndice – Protocolos de Roteamento para MANET), verificou-se que atualmente existem mais de uma dezena de protocolos de roteamento para redes móveis ad hoc. Porém, se observarmos os protocolos de roteamento que estão propostos no IETF (*Internet Engineering Task Force*), no grupo de trabalho (*Working Group*) MANET (*Mobile Ad-hoc Networks*), e compararmos com os protocolos de roteamento implementados pelos dois principais simuladores para redes móveis ad hoc (ns-2 e GloMoSim), veremos que apenas dois protocolos são comuns a todos, são os protocolos de roteamento AODV (ver seção 8.3.1) e DSR (ver seção 8.3.2).

Tanto o AODV como o DSR são protocolos da classe *on-demand* (ver seção 8.3). Segundo Perkins (2001), um dos autores do AODV, o DSR é considerado um dos exemplos mais puros de protocolos *on-demand*.

Por esses motivos, o DSR foi escolhido para ser o primeiro protocolo de roteamento implementado no MobiCS2. O protocolo DSR foi implementado na sua versão básica, exatamente como descrito na seção 8.3.2, sem nenhuma otimização, nem extensões, que são opcionais.

Para implementar o protocolo DSR no MobiCS2, foi seguido o modelo proposto na seção 4.5.2.

PUC-Rio - Certificação Digital Nº 0016060/CA

Inicialmente foi criada uma nova classe, chamada `Dsr` (derivada da classe `Protocol`), para representar o protocolo de roteamento DSR.

Em seguida, foram criadas as classes que representam os diferentes tipos de pacotes do protocolo. Todas estas classes são derivadas ou da classe `ServiceDataUnit` ou da classe `ProtocolDataUnit`. E todas estas classes implementam a interface `NetIfSDU`, para que a interface de rede possa processá-las. A seguir são descritas estas classes:

- Classe `DsrRouteRequest` – derivada da classe `ServiceDataUnit`. Esta classe representa o pacote do tipo *Route Request* do DSR, que apresenta os seguintes atributos:
  - `source` – endereço IP do nó que originou este pacote. Este endereço não pode ser alterado pelos nós intermediários que retransmitem o pacote.
  - `destination` – endereço IP do destino, neste caso é o endereço IP de broadcast.
  - `identification` – número de identificação único da requisição de rota, criado pelo nó de origem.
  - `target` – endereço IP do nó que está sendo procurado pela requisição de rota.
  - `routeRecord` – representa o registro de rota, que é implementado por uma lista encadeada de endereços IP.
- Classe `DsrRouteReply` – derivada da classe `ServiceDataUnit`. Esta classe representa o pacote do tipo *Route Reply* do DSR, que apresenta os seguintes atributos:
  - `source` – endereço IP do nó que originou este pacote.
  - `destination` – endereço IP do destino, copiado do atributo `source` do *Route Request* que gerou este *Route Reply*.
  - `routeRecord` – representa o registro de rota.
- Classe `DsrData` – derivada da classe `ProtocolDataUnit`. Esta classe representa o pacote de dados do DSR, que apresenta os seguintes atributos:

PUC-Rio - Certificação Digital Nº 0016060/CA

- `source` – endereço IP do nó que originou este pacote.
- `destination` – endereço IP do destino.
- `routeRecord` – representa o registro de rota.

- Classe `DsrRouteError` – derivada da classe `ServiceDataUnit`. Esta classe representa o pacote do tipo *Route Error* do DSR, que serve para informar a falha em um enlace, não podendo alcançar o próximo nó do registro de rota, no encaminhamento de um pacote. Esta classe apresenta os seguintes atributos:
  - `source` – endereço IP do nó que originou este pacote (quem detectou a falha no enlace).
  - `destination` – endereço IP do destino. É copiado do endereço IP de origem (`source`) do pacote que não conseguiu ser encaminhado.
  - `unreachable` – endereço IP do nó que não foi alcançado.
  - `routeRecord` – representa o registro de rota.

Como o `Dsr` em nosso caso executa em cima do IEEE 802.11 (`WiFiPhy/WiFiMAC`), o `Dsr` pode assumir que os enlaces são bidirecionais. Isto acontece devido ao funcionamento do IEEE 802.11, que troca quadros RTS (`WiFiRTS`) e CTS (`WiFiCTS`), e implementa entrega com confirmação (`WiFiACK`). Desta forma, quando o `Dsr` gera o pacote `DsrRouteReply`, este pode usar a rota reversa da rota contida no pacote `DsrRouteRequest` para encaminhar o pacote.

Como o IEEE 802.11 implementa entrega com confirmação (`WiFiACK`), o `Dsr` utilizou "*link-layer acknowledgements*" na fase de Manutenção de Rotas (*Route Maintenance*). Como é especificado pelo protocolo DSR, quando se usa "*link-layer acknowledgements*" não é necessário implementar os outros tipos de reconhecimentos da fase de Manutenção de Rotas ("*passive acknowledgements*" e "*network-layer acknowledgements*").

Continuando a implementação do protocolo, e seguindo o modelo proposto da seção 4.5.2, foram criadas duas interfaces Java: `DsrForwarding` e

PUC-Rio - Certificação Digital Nº 0016060/CA

`DsrRouteManager`. A classe `Dsr` implementa estas duas interfaces, que são descritas a seguir:

- Interface `DsrForwarding` – representa o componente *Forwarding* de um protocolo de roteamento, e possui os seguintes métodos:
  - Método `whenDsrData()` – serve para tratar um pacote `DsrData` que está chegando no protocolo.
  - Método `buildPDU()` – representa o componente *BuildPDU* de um protocolo de roteamento, e serve para criar um novo pacote `DsrData`, baseado em uma SDU que recebe como parâmetro.
- Interface `DsrRouteManager` – representa o componente *RouteManager* de um protocolo de roteamento, e possui os seguintes métodos:
  - Método `whenDsrRouteRequest()` – serve para tratar um pacote `DsrRouteRequest` que está chegando no protocolo.
  - Método `whenDsrRouteReply()` – serve para tratar um pacote `DsrRouteReply` que está chegando no protocolo.
  - Método `whenDsrRouteError()` – serve para tratar um pacote `DsrRouteError` que está chegando no protocolo.
  - Método `routeDiscovery()` – inicia uma fase de Descobrimento de Rota (*Route Discovery*). Entre outras coisas, é o responsável por criar e enviar um pacote `DsrRouteRequest`.
  - Método `whenRouteReplyTimeout()` – serve para tratar um evento de tempo, neste caso o estouro do tempo para receber um pacote `DsrRouteReply` na fase de Descobrimento de Rota (*Route Discovery*). Este método está relacionado ao componente *Timer* de um protocolo de roteamento. Faz parte também do protocolo a classe `RouteReplyTimeout`, que representa o evento de tempo.

Note-se que as interfaces `DsrForwarding` e `DsrRouteManager` não precisariam ser definidas, já que seus métodos poderiam ter sido também criados

PUC-Rio - Certificação Digital Nº 0016060/CA

diretamente na classe `Dsr`. No entanto, estas interfaces servem para caracterizar melhor as funções específicas de cada componente do protocolo de roteamento.

Concluindo a implementação do protocolo de roteamento `Dsr`, foram criadas as seguintes três classes:

- Classe `DsrQueue` – representa o componente *Queue* de um protocolo de roteamento, que implementa uma fila para armazenar os pacotes que estão pendentes para encaminhamento, aguardando a resposta (`DsrRouteReply`) de um Descobrimento de Rota (*Route Discovery*). Esta classe é instanciada pela classe `Dsr`.
- Classe `DsrRouteData` – representa o componente *RouteData* de um protocolo de roteamento, que implementa um cache de rotas (*route cache*) para armazenamento das rotas (*source routes*). Esta classe é instanciada pela classe `DsrRouteAccess`.
- Classe `DsrRouteAccess` – representa o componente *RouteAccess* de um protocolo de roteamento, que implementa uma interface para acessar as rotas armazenadas no `DsrRouteData`, sem expor a estrutura do cache. Esta classe é instanciada pela classe `Dsr`.

Todas as classes e interfaces do protocolo de roteamento DSR descritas nesta seção, foram colocadas em um único pacote Java chamado "`dsr`".

O framework disponibiliza ainda a interface `RoutingSDU`, que define a interface da SDU de qualquer protocolo de roteamento, padronizando assim a interface entre estes protocolos e os protocolos de níveis superiores, como p.ex., um protocolo de transporte. Um método desta interface é o `getDestination()`, que retorna o endereço IP (`IpAddress`) do nó de destino.

5.6. Geração de Tráfego para a Simulação

Para testar um protocolo em uma simulação é necessário criar algum elemento que gere tráfego de dados para este protocolo. No caso do MobiCS2, este elemento gerador de tráfego tem que ser necessariamente derivado da classe

PUC-Rio - Certificação Digital Nº 0016060/CA

`Protocol`, pois no framework apenas protocolos podem enviar e receber dados de outros protocolos.

No MobiCS2 não existe uma classe específica para representar um processo "aplicação" que gere e consuma tráfego de dados, pois este na verdade seria muito parecido com a classe `Protocol` (apenas não teria referências para protocolos clientes). Portanto, foi uma decisão de utilizar a classe `Protocol` também para representar a abstração de um processo "aplicação".

Um tipo fonte geradora de tráfego muito utilizado em simulações é o tipo CBR (*Constant Bit Rate*), ou seja, uma fonte que gera tráfego de dados numa taxa de bits constante. Assim, o MobiCS2 disponibiliza a classe abstrata `Cbr`, derivada da classe `Protocol`, que representa a abstração de uma fonte geradora de tráfego CBR.

A classe `Cbr` tem três parâmetros: o endereço do destino do tráfego (`Address destination`); o tamanho (em octetos) do pacote (unidade de dados) a ser gerado (`int packetSize`); e a taxa de envio (`double sendingRate`), que é o número de pacotes enviados por segundo. Estes três parâmetros são informados através do método `setParameters()`, antes de iniciar a geração de tráfego através do método `startCbr()`. O método `stopCbr()` serve para interromper a geração de tráfego.

A classe `Cbr` utiliza o padrão de projeto *Template Method*. Esta classe possui o método *template* `sendPacket()`, que serve para enviar um único pacote para o protocolo fornecedor, utilizando os parâmetros `destination` e `packetSize`. Todo pacote gerado é numerado, iniciando em zero, e a cada envio este número é incrementado. O pacote também possui o nome do nó que o originou. Estas informações do pacote aparecem no *trace* da classe `Cbr`.

A classe `Cbr` também é consumidora de tráfego, através do método `consumePacket()`. A classe `Cbr` pode gerar informação de *trace* no momento do envio e do recebimento de um pacote.

A classe abstrata `CbrPacket`, derivada de `ServiceDataUnit`, representa um pacote da classe `Cbr`.

O método *template* `sendPacket()` chama o método abstrato `createPacket()`, para criar um pacote "vazio", que será preenchido pelo

PUC-Rio - Certificação Digital Nº 0016060/CA

próprio `sendPacket()`. O método abstrato `createPacket()` tem que retornar um objeto de uma classe derivada de `CbrPacket`, e que implemente a interface da SDU do protocolo fornecedor.

Quando a geração de tráfego é iniciada (`startCbr()`), o método `sendPacket()` é chamado automaticamente pela classe `Cbr` para enviar um pacote, e o processo é repetido na taxa definida pelo parâmetro `sendingRate`. Porém, o método `sendPacket()` também pode ser chamado explicitamente pelo usuário do framework num script de simulação, mesmo quando a classe `Cbr` não estiver gerando tráfego de forma automática (`stopCbr()`). Esta funcionalidade pode ser útil numa simulação determinística (`DetermSimulation`).

O MobiCS2 disponibiliza a classe concreta `RoutingCbr`, derivada da classe `Cbr`, especializada para a geração de tráfego CBR para protocolos de roteamento. Esta classe implementa o método `createPacket()`, que cria um objeto da classe `RoutingCbrPacket`, derivada da classe `CbrPacket`, e que implemente a interface `RoutingSDU`. Esta classe implementa também o método `whenRoutingCbrPacket()`, que chama o método `consumePacket()` para consumir o pacote.

PUC-Rio - Certificação Digital Nº 0016060/CA

5.7. Exemplo de uma Simulação

Nesta seção descrevemos uma simulação bem simples no MobiCS2, que no entanto contemplará de forma direta ou indireta praticamente todo o framework. O objetivo desta simulação é apenas exemplificar o uso do MobiCS2. O resultado concreto da simulação não será de relevância no momento, isto é, não vamos aqui avaliar o desempenho de um protocolo, mas apenas mostrar os principais recursos do framework para criar uma simulação de uma rede móvel ad hoc.

Para criar uma simulação é necessário primeiramente especificar o cenário da simulação, cujos principais parâmetros, do nosso exemplo, foram inspirados no trabalho de Broch et al (1998).

O cenário do nosso exemplo é uma rede móvel ad hoc constituída de 50 nós, que se movem em uma região retangular plana (1500m x 300m), durante 300 segundos de tempo simulado.

Em relação à pilha de protocolos, todos os nós executam o protocolo de roteamento DSR (classe `Dsr`) em cima dos protocolos do padrão IEEE 802.11 (classe `WiFiNode`).

Na simulação existe apenas um nó móvel (nó "0") que gera tráfego com destino para um segundo nó da rede (nó "1"), que irá consumir este tráfego (classe `RoutingCbr`). O primeiro nó tem uma taxa de envio (`sendingRate`) de 4 pacotes por segundo, e envia pacotes de tamanho (`packetSize`) de 64 bytes.

Todos os nós da rede se movimentam segundo o modelo de mobilidade *Random Way-Point* (classes `RandomWayPoint` e `RandomWayPointFactory`). A velocidade média dos nós da rede é baseada na velocidade média de uma pessoa caminhando, que é de 1,4 m/s (5,04 Km/h). A velocidade do nó varia num intervalo de 20% em torno da velocidade média, isto é, a velocidade mínima é 1,12 m/s (4,032 Km/h), e a velocidade máxima é 1,68 m/s (6,048 Km/h). E o tempo de pausa do movimento é sempre de 30 segundos.

Nesta simulação o nível de energia do nó não é levado em consideração. Portanto usamos o modelo de energia básico (classes `EnergyModel` e `EnergyFactory`) e alteramos o valor do atributo `ignoringConsume` para `true`, isto é, a energia foi mantida no nível máximo durante toda a simulação.

O modelo de conectividade do canal utilizado na simulação foi o `SimpleRadioPropagation`, com o atributo `radioRange`, da classe `WiFiPhy`, ajustado para o valor de 250 m para todos os nós.

Os únicos objetos simulados que geraram informação de *trace*, foram as instâncias do protocolo `RoutingCbr` dos dois nós ("0" e "1").

Após definido o cenário da simulação, criamos o código da simulação. Para isso, criamos a classe `FirstSimulation`, derivada da classe `Simulation`, e em cujo único método estático `main()`, foi programada toda a simulação. A Figura 13 ilustra o código do método `main()`, devidamente comentado.

```
public static void main(String[] arg) {
    //Variaveis auxiliares
    int i = 0;
    RandomWayPoint rwp = null;
    RoutingCbr rcbr = null;
    Protocol p = null;

    //Constantes para referenciar os nos de origem e destino do trafego CBR
    final int SOURCE = 0;
```

PUC-Rio - Certificação Digital Nº 0016060/CA

```
    final int DESTINATION = 1;

    //Constante com o numero de nos da simulacao
    final int TOTAL_NODES = 50;
    //Cria um vetor de nos da rede
    WiFiNode[] node = new WiFiNode[TOTAL_NODES];

    //Informa o tamanho da regiao da simulacao
    setRegion(new R3CartesianCoordinate(1500,300));

    //Agenda o termino da simulacao. Duracao de 300 segundos de tempo simulado.
    stopAt(300);

    //Criacao de todos os nos da rede
    for (i = 0; i < TOTAL_NODES; i++) {
        //Cria um no da rede com modelo de mobilidade RandomWayPoint e
        //com modelo de energia EnergyModel
        node[i] = new WiFiNode("node" + i, RandomWayPoint.randomX(),
            RandomWayPoint.randomY(), RandomWayPointFactory.getInstance(),
            EnergyFactory.getInstance());

        //Cria o Protocolo de Roteamento DSR e o conecta
        //ao restante da pilha de protocolos do no da rede.
        node[i].connectRoutingProtocol(new Dsr(node[i]));

        //Configura o RandomWayPoint e inicia a mobilidade
        rwp = (RandomWayPoint)node[i].getMobility();
        rwp.setParameters(1.12, 1.68, 30, 30);
        rwp.start();

        //Configura o modelo de energia para ignorar o consumo de energia
        node[i].getEnergy().setIgnoringConsume(true);

        //O alcance do sinal de radio e' ajustado para 250m
        ((WiFiPhy)node[i].getPhysical(0)).setRadioRange(250);
    }

    //Cria RoutingCbr para no de origem
    rcbr = new RoutingCbr(node[SOURCE]);
    //Conecta ao restante da pilha
    p = node[SOURCE].getRoutingProtocol();
    rcbr.addProvider(p);
    p.addClient(rcbr);
    //Habilita a geracao de informacao de trace do RoutingCbr
    rcbr.getTracer().setTraceOn(true);
    //Informa os parametros do CBR
    rcbr.setParameters(node[DESTINATION].getNetIf(0).getIpAddress(), 64, 4);
    //Inicia a geracao de trafego CBR
    rcbr.startCbr();

    //Cria RoutingCbr para no de destino
    rcbr = new RoutingCbr(node[DESTINATION]);
    //Conecta ao restante da pilha
    p = node[DESTINATION].getRoutingProtocol();
    rcbr.addProvider(p);
    p.addClient(rcbr);
    //Habilita a geracao de informacao de trace do RoutingCbr
    rcbr.getTracer().setTraceOn(true);

    //Informa o modelo de conectividade do canal
    Air.getInstance().getChannel(0)
        .setChannelConnectivityModel(SimpleRadioPropagation.getInstance());

    //Inicia a simulacao
    start();
}
```

Figura 13 – Implementação da simulação `FirstSimulation`

PUC-Rio - Certificação Digital Nº 0016060/CA