3
Descrição do Framework

Segundo Mattsson (1996), quando se faz a documentação de um framework deve-se ter em mente que a sua consulta será feita por diferentes públicos, com diferentes necessidades. Pelo menos três públicos diferentes podem ser identificados:

- Engenheiros de software que têm que decidir qual framework deverão usar. Neste caso apenas uma breve descrição das capacidades do framework é suficiente. A documentação tem que explicar as características mais importantes do framework, podendo usar exemplos;
- Engenheiros de software que já tenham decidido usar o framework. Neste caso a documentação tem que descrever como o framework deve ser usado. Uma boa abordagem é descrever um tipo de "livro de receitas" (do inglês *cookbook*) que poderá ensiná-los rapidamente em como usar o framework;
- Engenheiros de software que desejem ir além do uso normal do framework e queiram adicionar novas características à este. Para este público é necessário uma documentação com mais detalhes que possibilite um entendimento mais profundo do framework;

Nesta dissertação foi desenvolvido um framework orientado a objeto, chamado MobiCS2, cujo domínio de aplicação é a simulação de redes móveis ad hoc. O objetivo deste framework é definir uma arquitetura para prototipação de protocolos e para a criação de simulações para testar, depurar e avaliar o desempenho desses protocolos num ambiente de redes móveis ad hoc. Este framework foi implementado em Java.

Para atingir este objetivo foram identificadas as características comuns e as diferenças dos diversos protocolos de roteamento, dos protocolos de enlace de

dados e das especificações do nível físico utilizados nas redes móveis ad hoc. Com isto foram modeladas abstrações destas entidades, num paradigma orientado a objeto, criando um ambiente de simulação extensível e flexível para a prototipagem de protocolos para MANETs.

Normalmente um framework pode ser dividido apenas em duas partes. Uma parte onde ficam disponíveis os *hot-spots* (pontos de flexibilização), que é a parte do framework que está aberta para customização e extensão, e que expressam aspectos do domínio do framework que não podem ser antecipados. E uma outra parte composta pelos *frozen-spots*, ou *kernel*, que são as partes fixas do framework, o seu núcleo, que são partes de código já implementadas que usam um ou mais *hot-spots* definidos em cada instância. Mas eventualmente o framework pode disponibilizar já algumas implementações dos seus pontos de flexibilização de modo a facilitar a sua instanciação. No caso do MobiCS2 foram incluídas algumas implementações dos pontos de flexibilização.

Acreditamos que existam três perfis de usuários para este framework. Um primeiro perfil é do usuário que deseja apenas utilizar as classes concretas já disponíveis no framework, estendendo algumas poucas classes, de modo a instanciar rapidamente uma simulação, apenas configurando o ambiente e criando um script de simulação, para um certo protocolo que deseja testar.

Um segundo perfil seria o do usuário que deseja desenvolver um protótipo de um novo protocolo, e que além de executar as atividades do primeiro perfil, também precisa ser capaz de testar, de usar e analisar o protocolo durante o processo de seu desenvolvimento.

E por último, um terceiro perfil seria do usuário que deseja testar um determinado protocolo em um ambiente simulado com outras características além daquelas disponibilizadas pelas instaciações “padrão” fornecidas com o framework, como por exemplo, criando um novo modelo de mobilidade, ou então um novo modelo de propagação do sinal de rádio.

3.1. Requisitos

Durante as fases de análise e de projeto do desenvolvimento do framework, foram definidos os requisitos que o MobiCS2 deveria atender para atingir os seus objetivos. Nesta seção são descritos esses requisitos.

Um dos primeiros requisitos é que o framework deveria ser de fácil aprendizagem e uso. Um dos focos é seu uso para fins didáticos, em disciplinas de algoritmos distribuídos e computação móvel, bem como ferramenta base para a pesquisa em protocolos para redes móveis.

Os principais requisitos foram:

- Criar um modelo cujas abstrações tenham uma analogia quase que direta com o domínio das redes de computadores, mais especificamente das redes móveis ad hoc, de forma a facilitar o entendimento do framework por uma pessoa que conheça o assunto.
- Oferecer uma interface e um modelo de programação simples, que facilitasse o desenvolvimento de protocolos.
- A possibilidade de um nó móvel da rede poder executar mais de um protocolo, e possibilitar a estruturação de protocolos em pilhas de protocolos.
- Permitir o conceito de posição no espaço de um nó da rede. P.ex., adotar um sistema de coordenadas cartesianas no espaço R^3.
- Criar a abstração de modelo de mobilidade associado aos nós da rede, através do qual fosse possível criar várias extensões e refinamentos de padrões de mobilidade, mas que permitisse a extensão destas abstrações (mobilidade e nó) de forma separada.
- Criar a abstração de modelo de energia associado aos nós da rede, através do qual fosse possível criar várias extensões e refinamentos de padrões de energia, mas que também permitisse a extensão destas abstrações (energia e nó) de forma separada.
- Disponibilizar no framework algumas implementação dos pontos de flexibilização (*hot-spots*), de forma a facilitar a instanciação do

framework. Por exemplo, já incluir no framework alguns modelos de mobilidade, modelos de energia e protocolos.

- Criar um facilitador para a geração dos logs de simulação, e uma padronização para o seu formato.
- Permitir a troca do algoritmo para escalonamento de eventos usado no núcleo do simulador.
- Criar um modelo que permita criar descrições da conectividade dos nós da rede móvel com variados graus de refinamento, indo desde a representação e manipulação explícita do grafo representado a topologia de interconexão, até uma inferência da conectividade a partir de variáveis tais como a posição dos nós e o alcance do sinal de rádio.

Durante o processo de desenvolvimento do framework foi colocado sempre em primeiro lugar o atendimento destes requisitos, eventualmente as vezes até em detrimento de um melhor desempenho do simulador.

No desenvolvimento deste trabalho existiu um compromisso de manter a mesma filosofia do framework MobiCS (para redes sem fio com infra-estrutura – ver seção 2.3), que é a de preservar ao máximo a independência entre o protocolo sendo prototipado e os modelos de simulação adotados.

3.2. Os Pacotes: Uma Visão Geral

Como pode ser observado na Figura 2, o framework MobiCS2 é constituído de quatro pacotes principais. São estes: `nodes`, `commsys`, `simulation` e `util`.

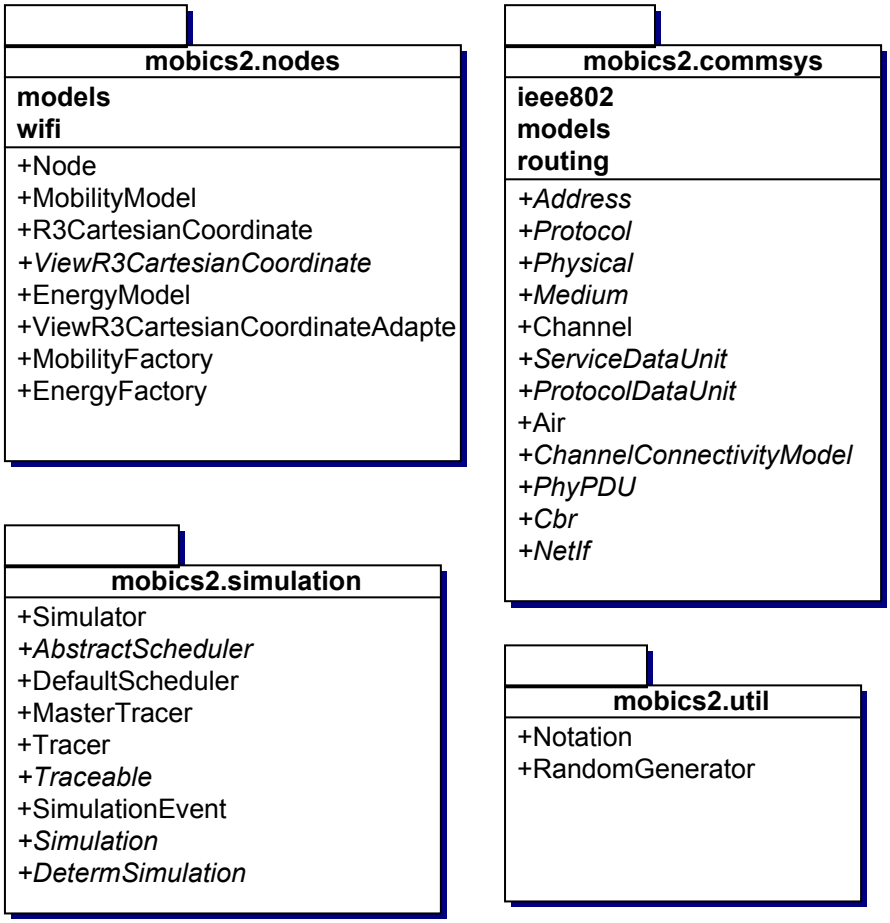

Figura 2 – O Framework MobiCS2

O pacote `nodes` contém as classes e interfaces que estão diretamente relacionadas com o nó de rede, como por exemplo, o seu modelo de mobilidade e de energia.

O pacote `commsys` (do inglês *Communication System*) dá suporte ao sistema de comunicação, e que possui abstrações para representar os meios físicos de transmissão, os canais de comunicação, e principalmente os protocolos.

O pacote `simulation` contém as classes usadas na máquina de simulação e que lidam com o escalonamento de eventos, tais como a chegada de um pacote em um protocolo e o término de energia em um nó da rede.

E por último o pacote `util` (do inglês *Utility*) onde se encontram classes utilitárias do framework, como por exemplo, a classe `RandomGenerator`, responsável pela geração de números pseudo-aleatórios.

A seguir faremos uma descrição mais detalhada de cada um desses quatro pacotes.

3.2.1.
Nós da Rede (Pacote `nodes`)

A principal classe do pacote `nodes` é a classe `node`, que representa o nó da rede móvel ad hoc. As suas principais características são: possuir uma posição no espaço, que só pode ser alterada pelo seu modelo de mobilidade; possuir recursos de energia, representado pelo seu modelo de energia; e possuir um conjunto de protocolos, podendo formar uma pilha de protocolos, possibilitando o acesso ao sistema de comunicação. Na Figura 3 podemos ver a classe `node` e as demais classes, interfaces e relacionamentos deste pacote.

A classe `R3CartesianCoordinate` representa um sistema de coordenadas cartesianas ortogonais no espaço R^3. As coordenadas (que são representadas nesta classe pelos atributos `x`, `y`, `z`) podem representar tanto a posição de um ponto, quanto podem representar um vetor no espaço de três dimensões. Esta classe possui dois construtores, um que só aceita os atributos `x` e `y`, e outro que aceita as três coordenadas. Ambos os construtores criam um objeto (ponto ou vetor) no espaço R^3, porém o primeiro construtor cria um objeto que está no plano `z=0`, e que por definição nunca pode sair deste plano. Caso se deseje queira um ponto ou vetor que possa ter a sua coordenada `z` com qualquer valor, deve-se construir o objeto com o segundo construtor. Com isto consegue-se criar simulações que são modeladas tanto para o espaço R^2 quanto para o R^3, e ambas utilizando esta única classe.

Para justificar esta preocupação em modelar simulações tanto para o espaço R^2 quanto para o R^3, gostaria de exemplificar dois cenários de simulação. Num primeiro cenário poderíamos estar querendo simular uma rede móvel ad hoc composta por vários dispositivos móveis, onde queremos testar o desempenho de um certo protocolo de roteamento, e queremos simplificar o nosso modelo e tratarmos tudo como se estivéssemos no espaço R^2, pois sabemos que para o que queremos testar não faz diferença se a dimensão do espaço é dois ou três. Então o usuário do MobiCS2 vai poder criar a sua simulação com toda a simplificação de como se realmente estivesse no espaço R^2, mas na verdade vai estar no plano `z=0` do espaço R^3.

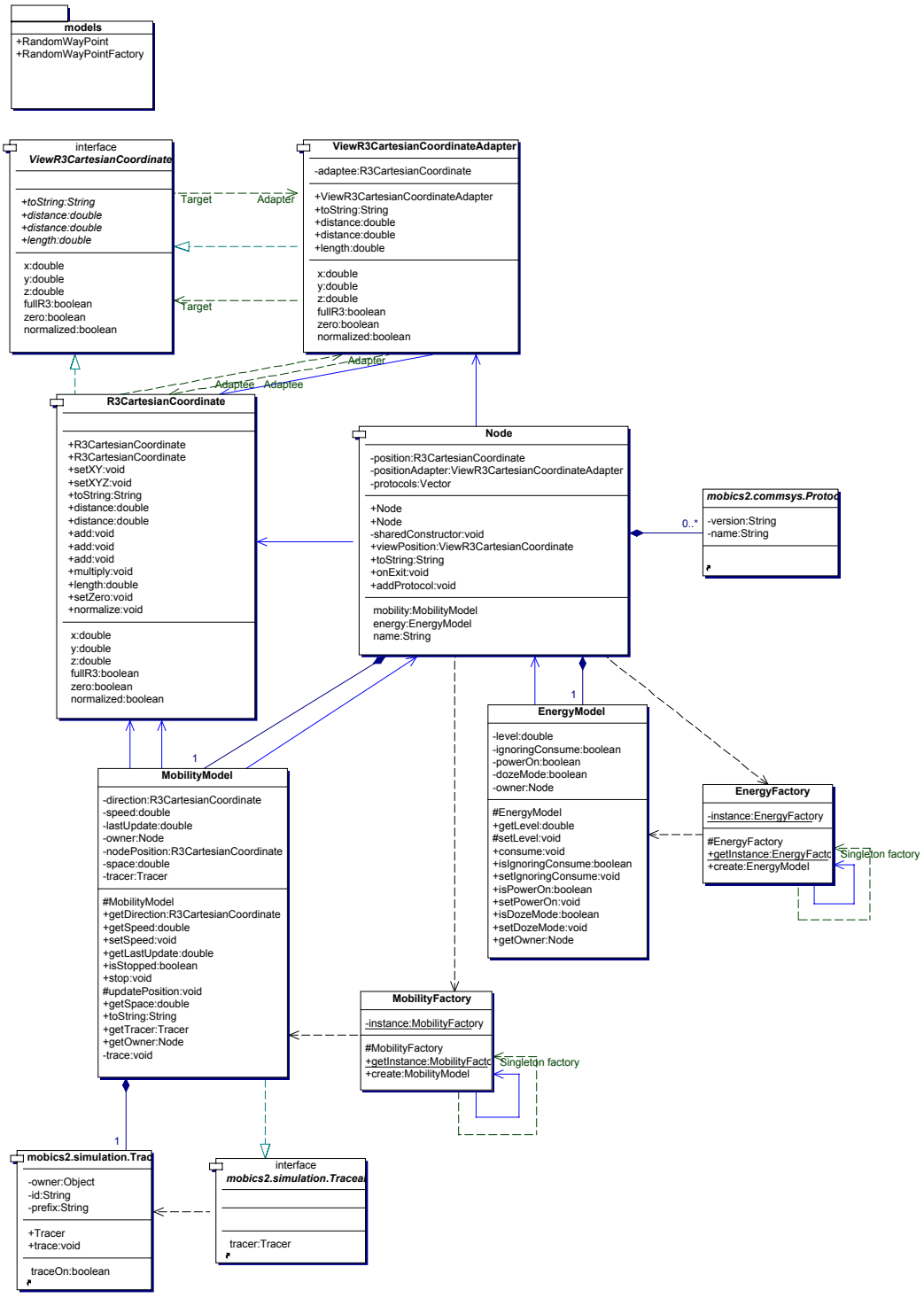

Figura 3 – Diagrama de Classes do Pacote `nodes`

Num segundo exemplo poderíamos imaginar um cenário militar, composto por soldados com dispositivos móveis, por sensores, por veículos terrestres, como tanques e carros blindados, todos espalhados num terreno plano. E ao mesmo tempo vários veículos aéreos, como helicópteros e aviões, poderiam estar sobrevoando a mesma área, formando uma grande rede móvel ad hoc, com nós

terrestres e aéreos. Neste exemplo, para simplificar o modelo e também para garantir que não venhamos a cometer um erro de programarmos um soldado ou tanque "voador", poderíamos modelar todos os dispositivos terrestres como se pertencessem apenas ao espaço R^2, e todos os veículos aéreos seriam modelados no espaço R^3, sendo o atributo `z` tratado como se fosse a altura do veículo em relação ao solo. Como os nós terrestres estariam na verdade no plano `z=0` do espaço R^3, então todos os nós da rede estariam compartilhando o mesmo espaço tridimensional.

A interface `ViewR3CartesianCoordinate` possui um subconjunto dos métodos da classe `R3CartesianCoordinate` que possibilita apenas realizar consultas a esta classe, sem alterar os seus atributos. Esta interface é implementada pelas classes `ViewR3CartesianCoordinateAdapter` e pela própria `R3CartesianCoordinate`.

A classe `ViewR3CartesianCoordinateAdapter` utiliza o padrão de projeto *Adapter*, que converte a interface da classe `R3CartesianCoordinate` para a interface `ViewR3CartesianCoordinate`, disponibilizando apenas métodos de consulta a esta classe. A classe `ViewR3CartesianCoordinateAdapter` é utilizada pela classe `node` para expor consultas a sua posição, sem que esta possa ser alterada, desta forma atendendo a um dos requisitos, que é possibilitar a consulta da posição de um nó, por exemplo, por seus protocolos, sem que estes possam alterá-la.

As classes `MobilityModel` e `MobilityFactory` são responsáveis respectivamente pelo modelo de mobilidade e pela criação deste modelo, ambas utilizadas pelo nó da rede. O modelo de mobilidade será discutido mais adiante na seção 3.4.1.

As classes `EnergyModel` e `EnergyFactory` são responsáveis respectivamente pelo modelo de energia e pela criação deste modelo, ambas utilizadas pelo nó da rede. O modelo de energia será discutido mais adiante na seção 3.4.2.

As classes `node`, `MobilityModel`, `MobilityFactory`, `EnergyModel` e `EnergyFactory` são pontos de flexibilização do framework. Estas classes podem ser estendidas e configuradas para atender as necessidades do usuário do MobiCS2. Já as classes `R3CartesianCoordinate` e

`ViewR3CartesianCoordinateAdapter`, e a interface `ViewR3CartesianCoordinate` fazem parte do *kernel* do framework.

3.2.2. Sistema de Comunicação (Pacote `commsys`)

A classe abstrata `Protocol` representa uma abstração do protocolo de comunicação. Esta classe serve de base para todos os protocolos a serem simulados na rede móvel ad hoc. Todo protocolo tem dois atributos de identificação, que são o nome e a versão do protocolo. A utilização da classe `Protocol` para a programação de protocolos será discutida no capítulo 4.

As classes abstratas `ServiceDataUnit` e `ProtocolDataUnit` representam, respectivamente, a Unidade de Dados do Serviço (SDU) e a Unidade de Dados do Protocolo (PDU), ambas baseadas no modelo de referência OSI da ISO. A classe `ServiceDataUnit` deve ser estendida por um usuário que pretende utilizar os serviços de um protocolo. O usuário deve incluir os seus dados na SDU e entregá-la para o seu protocolo para que este possa transmiti-la. Já a classe `ProtocolDataUnit` deve ser estendida pelo protocolo, que deve incluir nesta PDU a SDU recebida pela entidade de nível superior, e deve também incluir as informações de controle do protocolo. Uma vez que a PDU esteja montada, esta deve ser encaminha para um outro protocolo de nível mais baixo, ou caso este protocolo já esteja no nível físico, os dados devem ser transmitidos diretamente pelo meio físico.

Na modelagem orientado a objeto do conceito de SDU e PDU da RM-OSI, fizemos com que a classe `ProtocolDataUnit` estendesse a classe `ServiceDataUnit`, e também incluísse uma referência para a SDU. Desta forma um protocolo sempre espera receber uma SDU de uma entidade acima, o que por polimorfismo, pode ser uma PDU do protocolo que está acima. Toda SDU e PDU possui um atributo que é o tamanho da unidade de dados em bits, porém é oferecida uma interface para manipular este atributo que pode tratá-lo tanto em bits como em octetos.

A classe abstrata `Address` representa uma abstração de um endereço. Um endereço pode ser de três tipos: *unicast*, *broadcast* ou *multicast*. Esta classe deve ser estendida para representar endereços da camada de rede, da subcamada MAC,

etc. Um protocolo pode possuir vários endereços, por exemplo, um protocolo da sub-camada MAC tem o seu endereço MAC e pode também pertencer a um grupo *multicast*, possuindo assim um segundo endereço.

A classe abstrata `Physical` define uma abstração do protocolo do nível físico do modelo de referência OSI. Esta classe é responsável pela conexão do nó de rede ao meio físico para transmissão e recepção de dados entre os nós. Esta classe deve ser especializada para refletir as características do modelo pretendido, como por exemplo, as especificações de nível físico do padrão IEEE 802.11.

A classe abstrata `Medium` define uma abstração de uma mídia, como p.ex.: par trançado, fibra óptica, ar, etc. Esta classe representa o meio físico por onde acontecem as transmissões entre os nós da rede. A classe `Physical` de um nó deve estar conectada a uma mídia para que este nó possa transmitir para outros nós da rede. Esta conexão da classe `Physical` com a mídia (classe `Medium`) é feita através do método `attach()`.

A classe `Air` é uma representação da mídia "ar", por onde acontecem as transmissões sem fio. Esta classe concreta é uma extensão da classe `Medium` e é um *Singleton*, isto é, só pode existir uma mídia ar na simulação.

A classe `Channel` representa um canal de comunicação, dentro da mídia, por onde os nós de rede têm a possibilidade de se comunicarem. No modelo, uma mídia pode possuir vários canais, o que possibilita, por exemplo, que um nó possa alocar e utilizar dois canais simultaneamente, utilizando um canal para transmissão de dados e o outro canal para controle.

A classe abstrata `ChannelConnectivityModel` é um modelo abstrato da conectividade do canal. Este modelo determina a cada momento (instante simulado) quais outros nós da rede estão ao alcance de transmissão de cada nó. Este modelo define também qual é o tempo de propagação desta transmissão, e também é possível alterar o dado que é entregue ao destinatário, por exemplo, para simular erros durante a transmissão. Todo canal (classe `Channel`) para poder ser usado para transmissão tem que possuir um modelo de conectividade associado (classe `ChannelConnectivityModel`), sendo que o mesmo modelo pode ser compartilhado por mais de um canal. Esta classe será discutida em detalhes na seção 3.4.3.

Quase todas as classes descritas nesta seção são pontos de flexibilização do framework. A única exceção é a classe `Air`, que já é uma implementação de um ponto de flexibilização (classe `Medium`) disponível junto com o framework.

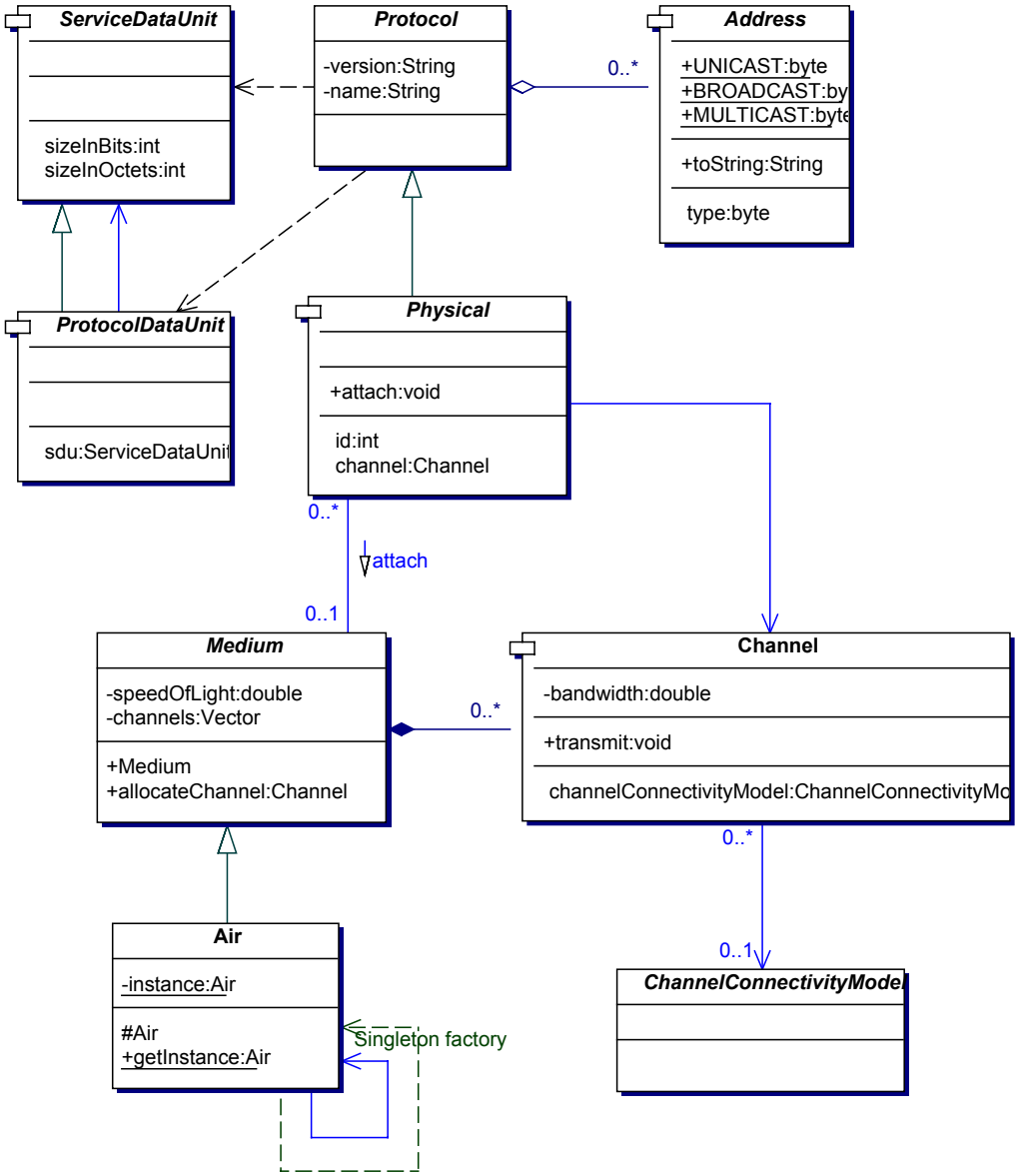

Figura 4 – Diagrama de Classes das Principais Classes do Pacote `commsys`

3.2.3. Simulação (Pacote `simulation`)

A principal classe do pacote `simulation` é a classe `Simulator`. Esta classe representa o simulador, e é a principal classe do *kernel* do framework. É uma classe *Singleton*, disponibilizando uma acesso único e global às

funcionalidades do simulador. É através desta classe que se comanda o início e o término de uma simulação. É também através desta classe que se configura o ambiente de simulação, como por exemplo, através do método `setRegion()` é especificada a região de simulação por onde os modelos de mobilidade dos nós devem movimentá-los. Esta classe é acessada por várias classes do framework, inclusive de outros pacotes, como por exemplo, pelo modelo de mobilidade `RandomWayPoint`, que será discutido na seção 3.4.1. A classe `RandomWayPoint` agenda eventos de pausa no seu movimento, através do método `scheduleRT()` da classe `Simulator`. Na Figura 5 podemos ver os relacionamentos da classe `Simulator` com as demais classes e interfaces do pacote.

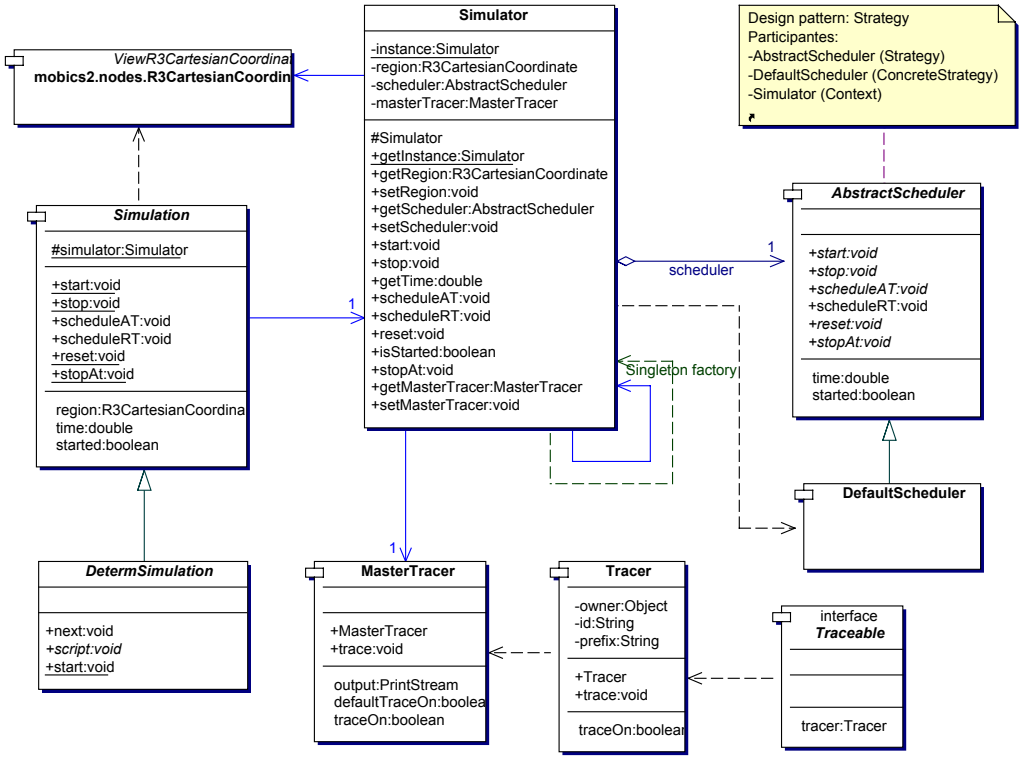

Figura 5 – Diagrama de Classes das Principais Classes do Pacote `simulation`

Nesta seção está sendo mostrada uma visão geral das classes do pacote `simulation`. Na seção 3.3 será discutida o funcionamento do simulador, sendo então visto mais detalhes das classes deste pacote.

A classe abstrata `AbstractScheduler` define as abstrações de um escalonador (do inglês *scheduler*). Esta classe, juntamente com a classe

`Simulator`, participam do padrão de projeto *Strategy*. Isto significa que o algoritmo utilizado pelo simulador para implementar o escalonador pode variar independentemente das classes clientes do simulador, tornando o framework mais flexível.

A classe `DefaultScheduler` estende a classe `AbstractScheduler`, implementando um escalonador concreto, que é a implementação default escolhida pela classe `Simulator`. Esta classe implementa um escalonador para simulação seqüencial de eventos discretos, utilizando para isto uma fila de eventos implementada com a classe `java.util.TreeSet`.

A classe abstrata `Simulation` foi criada apenas para facilitar a criação de simulações. A idéia é que para criar uma nova simulação deve-se estender esta classe e em seus métodos deve-se escrever o código para a criação e configuração do ambiente e do cenário de simulação, e também deve incluir o script de simulação. O uso desta classe é recomendado, mas não é obrigatório, isto é, é possível criar simulações com o MobiCS2 sem utilizar esta classe.

A classe abstrata `DetermSimulation` estende a classe `Simulation`, e serve para criar simulações onde se deseja criar passos de simulação que só podem ser executados ao término do passo anterior, criando assim uma ordem determinística dos acontecimentos na simulação. Ao contrário da classe `Simulation`, a classe `DetermSimulation` tem o seu uso obrigatório, caso o usuário queira desenvolver simulações deste tipo. Este tipo de simulação é interessante quando se deseja ter um controle maior sobre os acontecimentos da simulação, para facilitar, por exemplo, o teste e a depuração de um protocolo.

As classes `MasterTracer` e `Tracer`, e a interface `Traceable` dão suporte na geração de informação de *trace* da simulação. O uso destas classes e interface será discutido na seção 3.5.

3.2.4. Utilitário (Pacote `util`)

O pacote `util` possui apenas duas classes, denominadas `Notation` e `RandomGenerator`.

A classe `Notation` contém métodos para conversão de números em alguns tipos de notações (p.ex. notação científica) para números reais (`double`).

O seu uso é recomendado para facilitar a legibilidade do código programado. Todos os dois métodos da classe são estáticos e possuem várias constantes de classe para facilitar o seu uso.

O método `scientific()` converte um número em notação científica para um número real (`double`), por exemplo, a velocidade da luz no vácuo é de 3 x 10^8 m/s, utilizando esta classe escreveria o código: `Notation.scientific(3,8);` ou então para representar uma taxa de 2 Mb/s poderia se escrever o código: `Notation.scientific(2, Notation.MEGA).`

Outro método da classe `Notation` é o `binary()` que serve para representar números que utilizam potência de dois, por exemplo, para representar o tamanho de um arquivo a ser transferido pela rede de 50KB (50 x 2^{10}) escreveria o código: `Notation.binary(50, Notation.B_KILO).`

A outra classe do pacote é a `RandomGenerator` que é responsável pela geração de números pseudo-aleatórios. Esta classe foi desenvolvida baseada no exemplo encontrado em Horstman & Cornell (2001a), que segundo o seu autor a linguagem Java usa um gerador simples de convergência que pode ser não aleatório em certas situações, exibindo uma regularidade indesejada, o que pode ser verdadeiro quando é usado para plotar pontos ao acaso no espaço ou em certos tipos de **simulações**. Com esta classe espera-se ter fornecido junto com o framework um gerador de números pseudo-aleatórios melhor do que o fornecido pela linguagem Java para o uso em simulações.

3.3. O Simulador

Nesta seção será discutida a máquina de simulação do framework. Serão detalhadas as classes responsáveis pelo simulador, pela simulação e pelo escalonador. Vamos iniciar esta seção descrevendo os eventos de simulação.

3.3.1. Os Eventos de Simulação

No framework existem três tipos de eventos de simulação:

- **Síncronos**: estes eventos são tratados no mesmo instante simulado em que foram disparados. Não é contabilizado na simulação o tempo de execução da rotina de tratamento. Para a simulação é como se fossem instantâneos.
- **Assíncronos**: estes eventos são agendados no escalonador para serem tratados em um tempo futuro. Não é contabilizado na simulação o tempo de execução da rotina de tratamento.
- **Assíncronos com duração**: estes eventos são agendados no escalonador para serem tratados em um tempo futuro, porém o tempo de execução da rotina de tratamento é contabilizado na simulação.

Os tratadores de eventos síncronos são métodos que são chamados por outros métodos do framework (padrão de projeto *Template Method*), e possuem o prefixo "`on`" formando a seguinte assinatura de método: `on<Event>`. Por exemplo, ao terminar uma simulação, o Simulador (classe `Simulator`) dispara o evento *Exit* chamando o método `onExit()` de todos os nós de rede simulados. Desta forma, caso um usuário deseje tratar este evento, deve estender a classe `Node` e sobrescrever o método `onExit()`. O usuário deve tomar o cuidado para incluir no seu código uma chamada ao método original com o comando `super.onExit()`, afim de manter o tratamento original do evento e apenas acrescentar um novo código.

Para o tratamento de eventos assíncronos (com ou sem duração) existem várias opções de como implementar "funções de *callback*" para o tratamento desses tipos de eventos. Neste projeto foram consideradas quatro opções diferentes.

Uma primeira opção é o uso de "ponteiros de funções" (métodos), que em Java para conseguir isto é utilizado reflexão. Esta foi a opção escolhida pelo framework MobiCS (rede sem fio estruturada). Outra opção é o uso de Interfaces Java, sendo que o método de tratamento dos eventos teria que implementar uma estrutura de decisão (`if` ou `switch`) para determinar o evento a ser tratado.

O padrão de projeto *Command* é uma outra opção. Neste caso teria que existir uma classe para cada evento a ser tratado. A quarta opção seria o refinamento do padrão *Command* de forma a utilizar Classes Internas Java. Estas

duas últimas opções têm um excelente desempenho, porém torna complexo o modelo de programação de eventos no framework.

Baseado nos requisitos deste trabalho foi escolhido utilizar reflexão Java, por ser a opção que oferece um modelo de programação de eventos mais elegante e simples, mesmo em detrimento do seu desempenho quando comparada às outras três opções.

Os tratadores de eventos assíncronos (sem duração) são métodos que possuem o prefixo "`at`" formando a seguinte assinatura: `public void at<Event>(Object arg)`. Estes métodos necessariamente têm que ser públicos, não retornam valor, e recebem um único parâmetro do tipo `Object`. Caso necessite passar um parâmetro de tipo básico é necessário utilizar uma classe empacotadora ("*wrapper*"), por exemplo, se o parâmetro for do tipo básico `int`, este deve ser empacotado por um objeto da classe `Integer`. Caso necessite passar mais de um parâmetro deve-se criar uma classe para este fim, e caso não seja necessário nenhum parâmetro, deve-se passar o valor `null`.

Um exemplo de uso deste tipo de evento pode ser visto na classe `RandomWayPoint`, que é um modelo de mobilidade, que será discutida na seção 3.4.1. Nesta classe existe um evento de pausa do movimento, que é tratado pelo método `atPause()`. A forma como é agendado o tratamento deste tipo de evento será visto mais adiante quando for descrita a classe `Simulator` (seção 3.3.2).

Os tratadores de eventos assíncronos com duração são métodos que possuem o prefixo "`when`" formando a seguinte assinatura: `when<Event>`. Este tipo de evento é utilizado pelos protocolos de comunicação, e será discutido no capítulo 4.

3.3.2. A Classe `Simulator`

A classe `Simulator` representa o simulador do framework. Esta classe é um *Singleton*, pois só deve existir um único simulador, e este deve ser acessado de forma global pelas outras classes do framework. Desta forma, uma classe que deseje acessar as funcionalidades do simulador, não precisa receber uma

referência do mesmo, basta chamar o método estático `Simulator.getInstance()`.

A classe `Simulator` é a responsável por instanciar o escalonador default da simulação. É possível alterar o escalonador utilizado na simulação através do método `setScheduler()`. É possível acessar diretamente o escalonador em uso, através do método `getScheduler()`. Este método se faz necessário, por exemplo, caso se deseje acessar um método diferente do que é disponível pela classe abstrata `AbstractScheduler`.

A classe `Simulator` também é a responsável por instanciar a classe `MasterTracer`. Esta classe faz parte do esquema de suporte à geração de *trace*, que será discutida na seção 3.5. Através do método `getMasterTracer()` é possível acessar esta classe. O método `setMasterTracer()` pode definir um novo `MasterTracer` para ser utilizado pelo simulador.

A classe `Simulator` também serve para armazenar algumas informações do ambiente de simulação, como por exemplo, a região do espaço por onde os nós móveis devem movimentar durante a simulação é especificada através do método `setRegion()`. Este método recebe como parâmetro uma coordenada do espaço (`R3CartesianCoordinate`), que define a coordenada limite da região. A região é o espaço tridimensional, ou bidimensional, definido entre a coordenada informada e a origem, o ponto (0, 0, 0). Este parâmetro é consultado pelo método `getRegion()`, e pode ser utilizado pelos modelos de mobilidade para confinar os movimentos dos nós em um espaço limitado. A configuração deste parâmetro pode definir se a simulação vai ser somente no plano Z=0 ou se vai estar livre no R^3.

O método `start()` serve para iniciar uma simulação. Basicamente, este método inicia o escalonador para tratar os eventos de simulação. Este método também pode ser usada para reiniciar uma simulação a partir do ponto onde havia parado.

Existem três formas de parar uma simulação. Uma forma é utilizando o método `stop()`. Este método ao ser chamado provoca o término imediato da simulação. Outra forma é o método `stopAt()`, que serve para agendar o término da simulação para um determinado tempo simulado. E por último, uma

simulação chega ao seu fim quando não existem mais eventos para serem tratados pelo escalonador.

O método `reset()` serve para zerar o estado do escalonador, isto é, o tempo de simulação é zerado, e os eventos que existirem na fila do escalonador são descartados, deixando o escalonador pronto para iniciar uma nova simulação a partir do zero. Este método só pode ser chamado se o escalonador estiver parado, caso contrário não tem efeito.

O método `isStarted()` retorna um booleano que informa se o escalonador está processando uma simulação no momento.

O método `getTime()` retorna o tempo simulado. Este método, como os demais, é de acesso público, isto significa que até os protocolos podem ter acesso ao tempo simulado. Isto pode ser útil, por exemplo, para o protocolo de roteamento TORA (vide seção 8.3.4), que precisa de um sincronismo entre os nós para que o protocolo funcione. Na prática isto é conseguido através do uso de um GPS.

O método `scheduleAT()` serve para agendar eventos assíncronos, que são tratados por métodos com a assinatura: `public void at<Event>(Object arg)`. Este método espera quatro parâmetros: `absoluteTime` (`double`), `handler` (`Object`), `eventName` (`String`), `arg` (`Object`). O parâmetro `absoluteTime` indica o tempo em que o evento deve ser tratado. O parâmetro `handler` é o objeto que possui o método de tratamento do evento, que se chama `at<eventName>`. E `arg` é o parâmetro passado para o método de tratamento do evento. O método `scheduleAT()` utiliza reflexão Java para criar uma espécie de ponteiro pro método de tratamento do evento.

O sufixo "AT" do nome do método `scheduleAT()` é um acrônimo de *Absolute Time* (Tempo Absoluto). Existe o método `scheduleRT()`, que tem a mesma função que o método `scheduleAT()`, porém em vez de agendar um evento assíncrono num tempo absoluto é usado um tempo relativo ("RT" é o acrônimo de *Relative Time*). Neste método o tratamento do evento é agendado para ser executado após a passagem de um intervalo de tempo informado pelo parâmetro `relativeTime`.

Por exemplo, a classe `RandomWayPoint` agenda o evento de pausa do movimento, que é tratado pelo método `atPause()`, com o código: `Simulator.getInstance().scheduleRT(dt, this, "Pause", null);` onde `dt` é uma variável que informa o intervalo de tempo para o tratamento do evento de pausa.

3.3.3. O Escalonador

A classe abstrata `AbstractScheduler` define as abstrações de um escalonador de eventos discretos. Esta classe apresenta vários métodos abstratos, como por exemplo, `start()`, `stop()`, `stopAt()`, `getTime()`, etc. Todos os métodos desta classe são utilizados pelos métodos homônimos da classe `Simulator`. Por exemplo, o método `Simulator.stopAt()` chama diretamente o método `AbstractScheduler.stopAt()`. Outros métodos da classe `Simulator` podem acrescentar algum código, mas sempre chamam os métodos homônimos da instância do escalonador, derivada da classe `AbstractScheduler`.

O único método concreto da classe `AbstractScheduler` é o `scheduleRT()`, que é implementado baseado nos métodos abstratos `scheduleAT()` e `getTime()`.

A classe abstrata `AbstractScheduler` deve ser estendida para implementar escalonadores concretos. No framework é disponibilizado apenas um escalonador concreto, que é a classe `DefaultScheduler`. Esta classe implementa um escalonador seqüencial de eventos discretos, que é o escalonador default instanciado pela classe `Simulator`.

Quando o `DefaultScheduler` é iniciado pelo método `start()`, ele entra num loop, onde retira o primeiro evento da fila (evento com o menor tempo), avança o tempo de simulação para o tempo do evento e executa a rotina de tratamento do evento. Quando termina o tratamento do evento passa então para o próximo evento da fila. O `DefaultScheduler` é *single-threaded* e só pode existir um evento em execução a cada momento.

A classe `DefaultScheduler` utiliza a classe `java.util.TreeSet` para implementar a fila de eventos, e a classe `SimulationEvent` para implementar o evento de simulação.

A classe `SimulationEvent` basicamente armazena os dados do evento, como: o tempo, o objeto e o método que vai tratá-lo e o argumento que vai ser passado. Estes atributos são informados no construtor da classe. Esta classe tem mais um atributo chamado `id`, que é um número do tipo `long` que serve como identificador único da instância. Todos os atributos desta classe são apenas para consulta, não podem ser alterados.

A classe `SimulationEvent` implementa a interface `Comparable`, que serve para ordenar cronologicamente os eventos na fila implementada pelo `TreeSet`. A classe `TreeSet` é uma adaptação do algoritmo do *Red-Black tree* encontrado em Cormen et al (1990), que garante uma complexidade no tempo de *O(log(n))* para as operações básicas (adicionar, remover e acessar). O atributo `id` da classe `SimulationEvent` garante que possam existir na fila eventos com o mesmo tempo simulado, e que estes vão ser tratados seqüencialmente, baseados na ordem crescentes do seu `id`, respeitando a ordem em que os eventos de simulação foram gerados.

A classe `SimulationEvent` possui o método protegido `execute()` que serve para executar a rotina de tratamento do evento.

Se o usuário do framework precisar, pode criar outros escalonadores concretos, como por exemplo, um escalonador concorrente ou paralelo para melhorar o desempenho do simulador. Ou um escalonador que implemente a fila de eventos com uma outra estrutura de dados. Ou até um escalonador de tempo real, para o uso em emulação de redes móveis ad hoc.

3.3.4. A Simulação

No MobiCS2, para iniciar a programação de uma simulação, deve-se estender a classe abstrata `Simulation`. Por exemplo, vamos chamar esta nova classe de `MySimulation`.

Na classe `MySimulation` deve-se criar o método estático `main()`, onde deve ser programada a parte estrutural da simulação, como por exemplo a

configuração do ambiente e a criação e configuração do cenário de simulação. Fazem parte da configuração do ambiente atividades como: a definição da região de simulação e a habilitação de geração de *trace*. Fazem parte da criação e configuração do cenário atividades como: instanciação dos nós, protocolos, mídias, etc; e a configuração de todos estes elementos.

Vamos chamar de script de simulação, o código que define a parte comportamental da simulação, como por exemplo, como vão ser os movimentos dos nós e a geração do tráfego de dados, e principalmente a determinação do início e fim da simulação. O script de simulação também deve ser incluído no método `main()`.

A classe `Simulation` implementa os principais métodos da classe `Simulator`, com isso simplifica a programação da classe `MySimulation`. Por exemplo, em vez de iniciar a simulação com o comando "`Simulator.getInstance().start()`", pode simplesmente escrever "`start()`".

A classe `Simulation` possui um atributo protegido chamado `simulator` que aponta para a instância da classe `Simulator`. Com isso os métodos da classe `Simulator` que não foram implementados na classe `Simulation` podem ser acessados através desse atributo. Por exemplo, para alterar o escalonador, em vez de escrever "`Simulator.getInstance().setScheduler()`", pode simplesmente escrever "`simulator.setScheduler()`".

Os únicos métodos da classe `Simulation` que são um pouco diferentes da classe `Simulator` são os métodos `scheduleAT()` e `scheduleRT()`. A diferença é que nesses métodos não é preciso informar o objeto que vai tratar o evento, é assumido que é a própria instância do `MySimulation`. Por exemplo, caso queira que no tempo 50 a simulação execute um evento chamado "Fault", que vai simular a falha de alguns dos nós da rede, é só escrever "`scheduleAT(50,"Fault",null)`" e criar o método `atFault()` na classe `MySimulation`. Neste caso a classe `MySimulation` tem que ser instanciada. Quando estes dois métodos da classe `Simulation` não forem utilizados não precisa instanciar, pode só usar o método estático `main()`.

O motivo da criação da classe `Simulation` não foi só para simplificar a programação da simulação, foi também para servir de base para a classe `DetermSimulation`.

A classe abstrata `DetermSimulation` serve para criar simulações com ordem determinística de eventos. Neste tipo de simulação são estabelecidos passos de simulação, onde só pode iniciar a execução do próximo passo quando termina de executar todos os eventos do passo anterior. A classe derivada de `DetermSimulation` precisa sempre ser instanciada, diferentemente do exemplo anterior da classe `MySimulation`.

A classe `DetermSimulation` sobrescreve o método `start()` da classe `Simulation` e acrescenta dois novos métodos: `next()` e `script()`. O método `script()` é um método abstrato onde deve ser escrito o script da simulação determinística. O método `next()` define pontos de sincronização dentro do script de simulação, determinando onde termina um passo e inicia outro.

```
public void script() {
        //Passo 1
        ...
        next();

        //Passo 2
        ...
        next();

        //Passo 3
        ...
        next();
}
```

Figura 6 – Esquema de um Script de Simulação Determinística

Na Figura 6 vemos um esquema de um script de simulação determinística. No primeiro `next()` serão tratados os eventos gerados no “Passo 1”, e a simulação só vai seguir para o “Passo 2” quando todos os eventos forem tratados, garantindo assim que ao iniciar o “Passo 2” não existe nenhum evento pendente do passo anterior. Neste tipo de simulação, os eventos gerados em cada passo devem ser finitos, caso contrário a simulação nunca seguirá para o próximo passo. Por exemplo, nesse tipo de simulação não se pode usar um modelo de mobilidade como o `RandomWayPoint`, que está sempre gerando eventos de movimento e pausa.

A simulação determinística é interessante para ser usada em teste e depuração de protocolos, onde é possível ter um controle maior sobre os acontecimentos da simulação.

3.4. Modelos de Simulação

O framework MobiCS2 apresenta três modelos de simulação, são estes: o modelo de mobilidade, o modelo de energia e o modelo de conectividade do canal. Estes três modelos serão discutidos nas subseções seguintes.

3.4.1. Modelo de Mobilidade

O modelo de mobilidade, representado no framework pela classe `MobilityModel`, é responsável pela mobilidade do nó da rede, isto é, é responsável por atualizar o posição do nó segundo um certo modelo.

O modelo implementado na classe `MobilityModel` é bem simples, representa um vetor velocidade, que a cada instante simulado pode ser consultado ou alterado, atualizando assim a posição do nó.

O vetor velocidade para ser representado na classe foi dividido em dois atributos: um é o módulo do vetor, quantificando a velocidade em metros por segundo (atributo `speed`); e o outro representa apenas a direção e o sentido do vetor velocidade, não é considerado o seu módulo (atributo `direction`). Esta decomposição em dois atributos foi feita para simplificar o uso do modelo.

No construtor da classe `MobilityModel`, o vetor velocidade é iniciado com zero, isto é, sem movimento. O atributo `direction` é uma instância da classe `R3CartesianCoordinate`, e na sua instanciação é levado em conta o grau de liberdade de movimento do nó baseado no atributo `position` da classe `Node` (posição do nó), que também é da classe `R3CartesianCoordinate`, isto é, se o nó só pode ficar no plano `z=0`, o atributo `direction` também só pode assumir valores do plano `z=0`.

Apesar do conceito de mobilidade estar ligado diretamente ao nó, em vez de representar a mobilidade diretamente na classe `Node`, foi criada uma classe separada para representar o modelo de mobilidade, utilizando o padrão de projeto

Bridge, desta forma desacoplando as duas abstrações, de modo que as duas possam variar independentemente, isto é, pode-se estender a classe `Node` e a classe `MobilityModel` independentemente, desta forma evitando generalizações encaixadas (Gamma et al, 2000).

Na construção do framework existiram duas preocupações em relação à posição do nó. Uma preocupação era que somente o modelo de mobilidade poderia modificar a posição do nó, e a outra preocupação é que fosse fornecida uma interface de consulta da posição do nó para quem quisesse.

Para conseguir atender a estes dois requisitos foram tomadas várias decisões de projeto. A posição inicial do nó é passada através das suas coordenadas para o construtor da classe `Node`, esta por sua vez é quem cria o atributo o `position`. A posição do nó é adaptada pela classe `ViewR3CartesianCoordinateAdapter` para oferecer uma interface de consulta pela classe `Node`.

A classe `MobilityModel` só é instanciada no construtor da classe `Node` através da classe `MobilityFactory`, que implementa o padrão de projeto *Abstract Factory*, porém em vez de criar uma família de objetos, cria apenas objetos da classe `MobilityModel`. O construtor da classe `MobilityModel` é protegido (modificador `protected`), isto significa que só pode ser instanciado por outra classe do mesmo pacote, ou por classes derivadas. O uso da classe `MobilityFactory` serve para garantir que cada nó possua o seu próprio modelo de mobilidade, isto é, que uma mesma instância do `MobilityModel` não possa ser compartilhada por mais de um nó. O uso desta classe serve também para que se de um maior grau de proteção na passagem do atributo `position` para a classe `MobilityModel`.

As classes `MobilityModel` e `Node` em nenhum momento expõem a posição do nó para alteração direta, nem para suas classes derivadas, desta forma só é possível atualizar a posição do nó através da manipulação do vetor velocidade.

A classe `MobilityModel` oferece o método `getSpace()` que retorna o espaço (distância) percorrido em metros, como se fosse um hodômetro. Uma possível utilização deste método pode ser para calcular em um determinado instante a velocidade média do nó.

Outro método da classe `MobilityModel` é o `updatePosition()` que serve para atualizar a posição do nó. Este método é protegido e é chamado automaticamente sempre que são atualizados os atributos do vetor velocidade, e também sempre que é feita uma consulta sobre algo referente à posição corrente de um nó.

A classe `MobilityModel` gera informação de *trace* toda vez que a posição do nó é atualizada, mostrando o estado corrente do modelo de mobilidade, como por exemplo, o valor do vetor velocidade, a posição do nó, o espaço percorrido, etc.

A classe `MobilityModel` é um ponto de flexibilização do framework que deve ser estendido quando se desejar sofisticar o modelo de mobilidade para atender os desejos do usuário.

No MobiCS2 está disponibilizado uma implementação desse ponto de flexibilização, que é uma implementação um pouco modificada do modelo *Random Way-Point* de Johnson & Maltz (1996), que é um modelo bastante utilizado em simulações de redes móveis ad hoc.

A classe `RandomWayPoint` implementa o modelo de mobilidade *Random Way-Point* estendendo a classe `MobilityModel`. A classe `RandomWayPointFactory` estende a classe `MobilityFactory` implementando a fábrica abstrata do `RandomWayPoint`. A Figura 7 mostra o diagrama de classes destas classes.

A classe `RandomWayPoint` para funcionar precisa que sejam informados quatro parâmetros: velocidade mínima (m/s), velocidade máxima (m/s), pausa mínima (s), pausa máxima (s). Estes parâmetros são informados pelo método `setParameters()`, e só após o fornecimento destes parâmetros o modelo pode ser iniciado através do método `start()`.

Resumidamente, o modelo funciona escolhendo um ponto aleatório de destino, uniformemente distribuído dentro da região de simulação. É então escolhida uma velocidade aleatória, uniformemente distribuída dentro do intervalo especificado, e o nó se dirige para o seu destino nesta velocidade. Uma vez chegado ao seu destino o nó para, ficando em "pausa" num tempo aleatório, uniformemente distribuído dentro do intervalo especificado, e então todo o processo recomeça novamente.

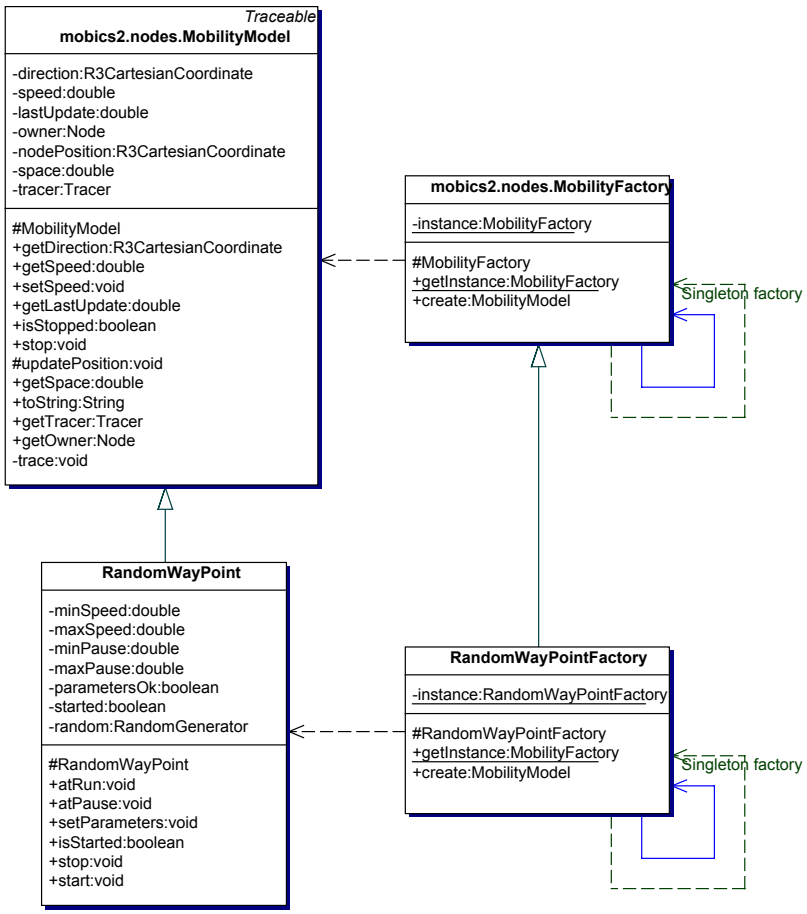

Figura 7 – Diagrama de Classes do modelo de mobilidade *Random Way-Point*

3.4.2. Modelo de Energia

O modelo de energia, representado no framework pela classe `EnergyModel`, é responsável por definir o recurso de energia do nó da rede. Para medir o nível de energia disponível no nó da rede, a classe `EnergyModel` possui o atributo chamado `level`. Este nível de energia é adimensional e normalizado, isto é, o nível de energia é um número real (`double`) que varia entre 0 e 1. Quando o nível de energia é 1 significa que o nó está com o máximo de energia disponível. E quando o nível de energia assume o valor 0 significa que o nó está sem energia, interrompendo todo o processamento do nó, ou seja, todo o processamento dos protocolos são interrompidos e o nó entra no estado "desligado" (*power off*).

A classe `EnergyModel` possui apenas dois métodos para manipular o nível de energia. O método `consume()` serve para que os recursos do nó que utilizam energia possam especificar a quantidade de energia que consomem num determinado evento de simulação. Por exemplo, o protocolo `Physical` pode ser estendido para que a cada transmissão ou recepção este possa chamar o método `consume()` para informar a quantidade de energia consumida na atividade.

O segundo método que manipula o nível de energia é o `setLevel()`. Este método permite alterar o nível de energia diretamente, informando um novo valor. Este método é protegido, isto significa que só pode ser usado por classes que estendam a classe `EnergyModel`. A intenção é oferecer um meio para que outros modelos de energia possam aumentar o nível de energia.

Existe um atributo chamado `ignoringConsume` que serve para ignorar o consumo de energia fazendo com que o nível de energia não diminua. Este atributo é útil quando uma simulação utiliza algum recurso que consome energia, mas numa determinada simulação não quer que isto seja considerado.

A classe `EnergyModel` possui o método `setPowerOn()` que serve para ligar e desligar o nó. Outro método é o `setDozeMode()` que serve para informar quando o nó entrou no estado de *doze mode*, isto é, quando o nó entrou num estado de menor atividade e conseqüentemente de menor consumo de energia. Relacionado a estes dois métodos existem quatro eventos na classe `Node`: `onPowerOn`, `onPowerOff`, `onDozeModeOn` e `onDozeModeOff`.

Existe um atributo chamado `lowEnergyThreshold` que serve para informar o valor limiar de energia no qual nó entra no estado de "baixa energia" (*low energy*). Quando o nó entra no estado de baixa energia é disparado o evento `onLowEnergy` na classe `Node`. E quando o nó sai desse estado de baixa energia é disparado o evento `onHighEnergy` na classe `Node`. Este atributo é iniciado com 0, o que significa que esta função está desativada. Para alterar este atributo é utilizado o método `setLowEnergyThreshold()`. Para exemplificar o uso desta função, podemos citar o trabalho de Machado et al (2002), onde foram realizadas simulações para avaliar o conceito de classes de nós em protocolos de roteamento para MANETs. A diferença entre as classes de nós era expressa por distintas capacidades de baterias. Um determinado nó poderia funcionar como

roteador caso o seu nível de energia fosse superior a um limiar. Caso contrário, o nó não seria mais capaz de propagar pacotes de terceiros pela rede.

A classe `EnergyModel` utiliza o padrão de projeto *Bridge* para desacoplar a abstração do modelo de energia da abstração do nó, podendo estender as duas classes independentemente.

A classe `EnergyModel` possui o seu construtor protegido, podendo só ser instanciada pela classe `EnergyFactory` (padrões de projeto *Abstract Factory* e *Singleton*), desta forma tenta garantir que o modelo de energia não seja compartilhado por mais de um nó.

A classe `EnergyModel` ao ser instanciada, começa com o seu nível de energia com o valor 1. Caso o usuário do framework queira iniciar com um valor diferente, este deve chamar o método `consume()` para diminuir o valor.

É importante observar que não existe uma relação de dependência entre a energia e a mobilidade do nó, isto é, caso o nó seja desligado, o nó pode continuar se movimentando. Caso o usuário do framework queira simular o comportamento de dependência entre energia e mobilidade, fazendo com que o nó pare de se movimentar quando for desligado, deve-se então programar o evento `onPowerOff` para chamar o método `stop()` da classe `MobilityModel`.

A classe `EnergyModel` é um ponto de flexibilização do framework. O MobiCS2 apresenta uma implementação deste ponto de flexibilização que é a classe `LinearConsumeBattery`.

A classe `LinearConsumeBattery` modela uma bateria (recurso de energia limitado), que possui um tempo de vida, no qual a sua energia vai sendo consumida linearmente até zerar a energia, desligando o nó.

Esta classe precisa de quatro parâmetros para iniciar sua função: `normalLifeTime`, `dozeLifeTime`, `exponential` e `consumeQuantum`. Os parâmetros `normalLifeTime` e `dozeLifeTime` representam o tempo de vida da bateria, respectivamente, no modo normal de funcionamento e no modo *doze*. O parâmetro `exponential` é booleano e determina se os dois primeiros parâmetros devem ser considerados como valores determinísticos, ou se são valores médios de uma distribuição exponencial. E o parâmetro `consumeQuantum` informa a quantidade mínima de energia que deve ser consumida repetidamente até zerar a energia disponível no tempo determinado.

3.4.3. Modelo de Conectividade do Canal

O modelo de conectividade do canal define como os nós da rede estão conectados através do canal de comunicação. O modelo define três características para determinar a conectividade entre dois nós durante uma transmissão. A primeira característica é se a transmissão do nó de origem alcança o nó de destino ou não. Caso o nó de destino seja alcançado, a segunda característica define o tempo de propagação da transmissão. E a terceira característica define a qualidade da transmissão, ou seja, se o dado que foi transmitido chega ao nó destino igual, ou se sofre alguma alteração. Com este modelo é possível controlar vários aspectos relacionados com a transmissão no canal de comunicação.

Este modelo é representado pela classe abstrata `ChannelConnectivityModel`. Esta classe apresenta apenas um método público e concreto, o método `transmit()`. Este método recebe como parâmetros os nós de origem e de destino, representados por seus protocolos da camada física, e recebe também o dado sendo transmitido. Esta classe utiliza o padrão de projeto *Template Method*. O método `transmit()` é o método *template* que implementa o algoritmo para transmissão de dados no canal de comunicação, chamando alguns métodos abstratos da própria classe. Estes métodos abstratos também são protegidos (`protected`), e devem ser implementados por subclasses da classe `ChannelConnectivityModel`.

São três os métodos abstratos: `isReachable()`, `getPropagationTime()` e `processData()`. O método `isReachable()` retorna um booleano que informa se a transmissão do nó de origem alcança o nó de destino. O método `getPropagationTime()` retorna o tempo de propagação da transmissão em segundos. E por último, o método `processData()` processa o dado sendo entregue ao destino, podendo alterar os seus atributos.

Todo canal (classe `Channel`) tem que possuir um modelo de conectividade associado para poder ser usado para transmissão. Um mesmo modelo de conectividade pode ser compartilhado por mais de um canal de comunicação.

A classe `ChannelConnectivityModel` é um ponto de flexibilização do framework. O MobiCS2 apresenta duas implementações deste modelo: `Topology` e `SimpleRadioPropagation`.

A classe `Topology` representa um modelo simples para definir a topologia da rede, sem levar em consideração características físicas do ambiente simulado, como por exemplo, as posições dos nós da rede e suas velocidades. Com este modelo pode-se criar simulações onde são abstraídas as características físicas que influenciam na conectividade dos nós da rede no canal de comunicação, com o objetivo de simplificar e oferecer maior controle da simulação.

A classe `Topology` implementa um grafo dirigido, onde os nós do grafo representam os protocolos da camada física (classe `Physical`) dos nós da rede, e as arestas do grafo representam os enlaces de comunicação entre os nós da rede. Cada aresta do grafo possui um peso que representa o tempo de propagação da transmissão. Como o grafo é dirigido, consegue-se representar enlaces simétricos (bidirecionais) e assimétricos (unidirecionais). Esta classe oferece métodos para manipular este grafo, criando e removendo arestas, e alterando os pesos das arestas. Neste modelo, os dados transmitidos sempre chegam inalterados no seu destino.

A classe `Topology` é um *Singleton* porque só precisa existir uma instância desta classe para representar toda a topologia da rede em uma simulação. A mesma instância pode ser compartilhada por diferentes meios físicos, em vários canais.

A outra implementação do modelo de conectividade do canal é a classe `SimpleRadioPropagation`, que representa um modelo de propagação de sinal de rádio bem simplificado. Este modelo leva em consideração apenas as posições dos nós de origem e de destino no instante da transmissão, e o raio de alcance do sinal de rádio do nó de origem.

Esta classe trabalha em conjunto com a classe `WiFiPhy`, que é uma implementação bem simplificada da camada física do padrão IEEE 802.11b, que será vista com mais detalhes no capítulo 4. Esta classe possui o atributo `radioRange`, que representa o raio de alcance do sinal de rádio em metros.

Considerando as grandezas da velocidade do sinal eletromagnético no meio físico em relação às velocidades dos nós da rede e do tempo de duração da

transmissão de um quadro, são assumidas simplificações no modelo por considerar que certos detalhes podem ser desprezados nestas condições. Estas simplificações significam que é considerada apenas a distância entre os nós de origem e de destino no instante que se inicia a transmissão do quadro no meio físico, não considerando as posições dos nós quando o "primeiro bit" do sinal chega no destino até o "último bit", por considerar esta diferença desprezível. Caso a distância entre os nós seja menor ou igual ao atributo `radioRange` do nó de origem, então o dado é entregue ao destino sem nenhuma alteração, caso contrário é informado que o nó de destino não é alcançável.

A classe `SimpleRadioPropagation` também é um *Singleton*, por não haver necessidade de existirem várias instâncias desta classe, pois ela apenas oferece um serviço, não importando o seu estado interno.

A classe `ChannelConnectivityModel` pode ser estendida para representar outros modelos de propagação de sinal de rádio, como por exemplo, os modelos *Free-space* e *Two-ray ground reflection*.

Para criar estes modelos, além de estender a classe `ChannelConnectivityModel`, é necessário também estender outras classes para incluir novos atributos, como por exemplo, para o modelo *Free-space* deve-se estender o protocolo da camada física para incluir atributos como, potência de transmissão, ganho da antena de transmissão e ganho da antena de recepção.

3.5. Suporte à Geração de *Trace*

As classes `MasterTracer` e `Tracer`, e a interface `Traceable` dão suporte à geração de informação de *trace* da simulação. A Figura 8 mostra um diagrama de classes de um exemplo de suporte à geração de *trace* para a classe `MobilityModel`.

Todo objeto simulado, para o qual se necessite gerar informações de *trace*, deve instanciar a classe `Tracer` e implementar a interface `Traceable`. A classe `Tracer` é responsável pelo auxílio na geração de informações de *trace* aos objetos simulados. A interface `Traceable` serve para disponibilizar um acesso à instância da classe `Tracer` do objeto simulado.

Todas as instâncias da classe `Tracer` se comunicam com a única instância da classe `MasterTracer` para poder realizar os seus trabalhos. O `MasterTracer` é a classe centralizadora responsável pelo gerenciamento de todas as informações de *trace*. Esta classe é acessível pelo método `getMasterTracer()` da classe `Simulator`.

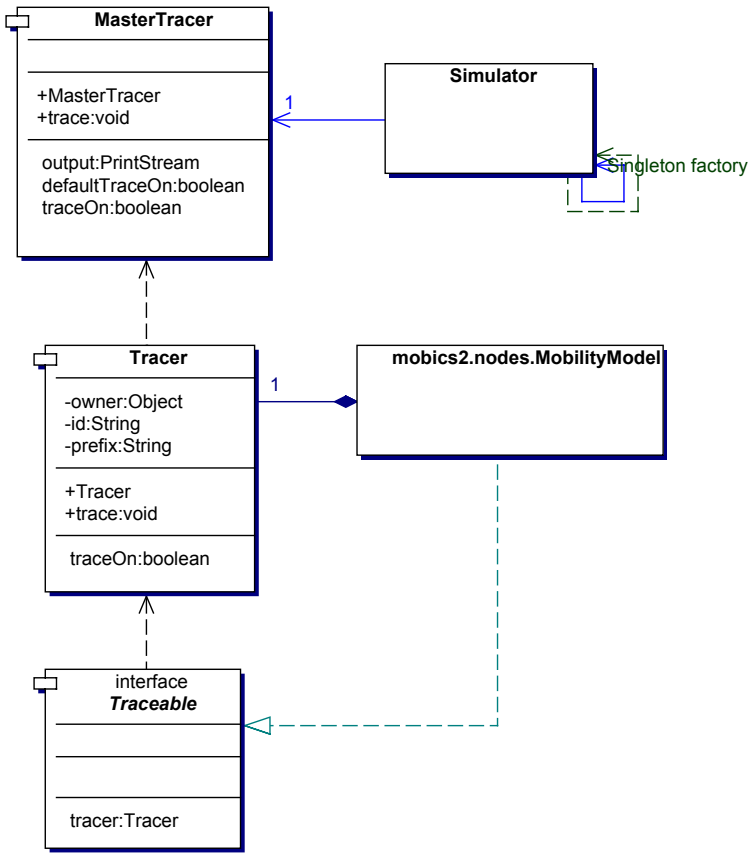

Figura 8 – Exemplo de Suporte à Geração de *Trace* para a Classe `MobilityModel`

A classe `MasterTracer` possui o método `setOutput()`, que serve para redirecionar o fluxo de saída da informação de trace. Por exemplo, com este método pode redirecionar o fluxo de saída para um arquivo. Inicialmente o fluxo de saída está direcionado para a saída padrão do Java (`System.out`).

Com o método `setTraceOn()` da classe `MasterTracer` é possível desabilitar e habilitar a geração de informação de *trace* de todo o framework.

O método `trace()` da classe `MasterTracer` é responsável por escrever uma linha de informação de *trace* no fluxo de saída. Este método recebe como parâmetro uma *string*, a qual é escrita no fluxo de saída logo após duas

informações. A primeira informação é a *string* "MOBX2>", que aparece no início de toda linha de *trace* gerada pelo MasterTracer. A segunda informação é o tempo simulado de quando o *trace* foi gerado. Na Figura 9 podemos ver um exemplo de informação de *trace* gerada pela classe RandomWayPoint.

```
MOBX2> 0.0 RandomWayPoint Node=n | speed=0.0 direction=(0.0,0.0) lastUpdate=0.0
nodePosition=(50.0,50.0) space=0.0

MOBX2> 19.403205949430237 RandomWayPoint Node=n | speed=2.1581737745249177
direction=(-4.885119500988566,-41.58956947089375) lastUpdate=19.403205949430237
nodePosition=(45.114880499011434,8.41043052910625) space=41.87549022176619

MOBX2> 32.17547764297839 RandomWayPoint Node=n | speed=0.0 direction=(0.0,0.0)
lastUpdate=32.17547764297839 nodePosition=(45.114880499011434,8.41043052910625)
space=41.87549022176619

MOBX2> 76.52946027619103 RandomWayPoint Node=n | speed=2.1546892500008292
direction=(31.455376992073212,90.24412720427341) lastUpdate=76.52946027619103
nodePosition=(76.57025749108465,98.65455773337966) space=137.44453979627295

MOBX2> 97.81569094582895 RandomWayPoint Node=n | speed=0.0 direction=(0.0,0.0)
lastUpdate=97.81569094582895 nodePosition=(76.57025749108465,98.65455773337966)
space=137.44453979627295

MOBX2> 100.0 RandomWayPoint Node=n | speed=1.2416884095754273 direction=(-
40.734660017302716,-4.675127127896957) lastUpdate=100.0
nodePosition=(73.87571473903061,98.3453043871616) space=140.15677103176782
```

Figura 9 – Exemplo de Informação de Trace Gerada pela Classe RandomWayPoint

O método trace() da classe Tracer é responsável por enviar a informação de *trace* do objeto simulado para o MasterTracer. Este método recebe como parâmetro uma *string*, a qual é concatenada a um prefixo antes de ser enviada para o MasterTracer. Este prefixo é formado pelo nome da classe do objeto simulado, mais uma identificação deste objeto, criada pelo próprio, e termina com o símbolo "|" (*pipe*). Na Figura 9 podemos observar que o nome da classe do objeto simulado é RandomWayPoint e que a sua identificação é "Node=n".

Como cada classe de objeto simulado pode ter um conjunto diferente de informações para o *trace*, fez-se necessário padronizar o formato destas informações. Todas as informações são separadas por um espaço em branco. Os dados que vem antes do sinal de *pipe* são introduzidos pelas classes Tracer e MasterTracer. O que vem depois do sinal de *pipe* é introduzido pelo objeto simulado no formato de pares de variável e valor, separados pelo símbolo de igualdade.

A classe `Tracer` oferece o método `setTraceOn()` para desabilitar e habilitar a geração de informação de *trace* para um determinado objeto simulado. O comportamento default é não gerar informação de *trace*. Quando a classe `Tracer` é instanciada, ela se baseia no atributo `defaultTraceOn` da classe `MasterTracer` para inicializar o seu atributo `traceOn`. Com isso é possível, no início de uma simulação, habilitar a geração de informação de *trace* para todos os objetos simulados.