Manual do Simulador GrubiX

Desenvolvedores do GrubiX

Julho - 2011

Capítulo 1

Tutorial GrubiX - QuickStart

1.1 O que é?

GrubiX é um ambiente de simulação *Open Source*, desenvolvidos pelo Grupo de Redes Ubíquas do Departamento de Ciência da Computação da Universidade Federal de Lavras (Grubi - UFLA). Esse ambiente é uma extensão de um conhecido simulador para rede móveis ad hoc, o ShoX [ShoX: A scalable ad hoc network simulator], que é utilizado inclusive para rede de sensores sem fios. Sua utilização deve-se para a tarefa de verificação e avaliação do comportamento da rede. O simulador GrubiX já apresenta uma versão estável e suas simulações fazem uso de eventos para exibir o comportamento da rede. Sua características, baseadas em [Lessmann, Heimfarth e Janacik 2008], estão descritas a seguir:

- O GrubiX foi implementado de forma modularizada, portanto funcionalidades (protocolos) podem ser incluídas ao simulador de maneira flexível.
- Para o desenvolvimento de aplicações, apenas tem que importar o ".jar" do projeto ou criar dependências entre projetos.
- É um simulador baseado na linguagem de programação Java (para implementação) e na de marcação XML (para configuração e saídas exportadas). Não é preciso, com isso, o aprendizado de linguagens desconhecidas, tais como TCL (J-Sim), NED (OMMNeT++) e OTcl (ns-2).
- Java elimina questões problemáticas de programação, como falha de segmentação, alocação e desalocação de memória, que estão presentes nas linguagens C e C++. Já o XML permite o uso de ferramentas padronizadas para análise (parsing) de arquivos e tradução de arquivos para outros formatos.
- Todo o conceito de camadas OSI (pilha de protocolos), pacotes, mobilidade, propagação de sinal e modelos de tráfego são definidos em classes abstratas. Para definir novos protocolos e outros modelos é suficiente à subclasse estender a classe abstrata, que provê frameworks e um padrão de como desenvolver novos conceitos.
- É um simulador orientado a eventos discretos. Para cada evento, que pode ser qualquer ação ou detecção de ações dentro da rede, geram-se outros eventos, controlados pelo projetista da aplicação. Assume-se também que eventos acontecem em qualquer instante de tempo durante a simulação da rede.
- Permite seguir o paradigma orientado a objetos para desenvolvimento das aplicações.
- O GrubiX possui suporte GUI (Graphical User Interface) para configuração de cenários, visualização da rede de sensores e ferramentas de estatística. Durante a simulação, todo

comportamento da rede de sensores é guardado em um arquivo de LOG para uma futura visualização e análise dos resultados.

1.2 Para que serve?

O GrubiX foi desenvolvido para ser utilizado em simulação de redes móveis ad hoc, que apresentam uma característica muito interessante: a não-existência de uma entidade central para gerenciar a comunicação das entidades da rede.

Redes de Sensores Sem Fios são caracterizadas como um tipo especial de rede móvel ad hoc (MANET - Mobile Ad hoc Network). Segundo [Campos 2003], essa arquitetura usa comunicação sem fio, não possui qualquer tipo de infra-estrutura e os dispositivos trocam informações diretamente entre si, ou através de seus vizinhos. De acordo com [Loureiro et al. 2003], a conexão entre os sensores é feita através de enlaces sem fios, porque a tarefa de interligar por meios guiados os nós é muito complicada e/ou inviável, em muitos casos. Por exemplo, essas redes podem estar presentes em locais nos quais cabos têm altos custos, como em oceanos, ou em locais onde não são aplicáveis, como em florestas e regiões vulcânicas.

Em uma rede sem fio tradicional, toda comunicação entre elementos da rede é realizada através de estações base, onde existem células que delimitam o alcance de cobertura de cada estação. É visto uma grande diferença em relação às MANETS, e por consequência, às RSSFs. Em termos de organização, RSSFs e MANETs são idênticas, porém, na questão funcional, enquanto as MANETs visam manter a comunicação entre os elementos computacionais que as compõem e podem executar tarefas distintas, as RSSFs geralmente realizam tarefas colaborativas, onde os nós-sensores fornecem os dados e nodos especiais os processam (sink nodes ou nodos sorvedouros). [Ruiz et al. 2004].

Os mais diversos tipos de comunicação dessa classe de redes e suas respectivas características podem ser modeladas pelo simulador GrubiX. Vários tipos de aplicações podem ser desenvolvidas a partir de simulações, como por exemplo:

- Militar: Detecção de intrusão em uma área determinada.
- Ambiental: Detecção de incêndios em florestas.
- Emergenciais: Análise de atividade vulcânica.
- Engenharia: Verificação de danos na estrutura de pontes e prédios.

Após feita a simulação dos diferentes tipos de aplicações, um arquivo de registro (LOG) é gerado, e através dele poderão ser visualizados a saída da simulação com os eventos ocorridos durante a simulação. O arquivo LOG mostra o passa-a-passo de toda simulação. Quando o visualizador lê o arquivo LOG gerado, ele faz a devida interpretação dos dados e mostra todos os eventos simulados através de uma animação gráfica.

1.3 Como instalar?

A instalação do GrubiX é bem simples. Para isto você deve ter instalado em sua máquina o programa Eclipse obtido em (*http://www.eclipse.org/downloads/*) ou outra IDE para programação na linguagem Java. Você deve fazer também o download do simulador GrubiX e do seu visualizador, obtidos em (*http://pesquisa.dcc.ufla.br/grub/dowloads*). Após obter tais softwares em seu computador, abra o Eclipse com o workspace vazio. Caso o workspace não esteja vazio feche os outros projetos do workspace para facilitar a configuração. Sua tela vai ficar parecida com a imagem na Figura 1.1.

Em seguida vá em File $\longrightarrow$ Import $\longrightarrow$ No campo General vá em Existing Projects into Workspace $\longrightarrow$ Marque o campo Select root directory $\longrightarrow$ Clique em Browse e localize o simulador GrubiX que você baixou $\longrightarrow$ Clique em Ok. Após feito isto verifique no campo Projects se o projeto

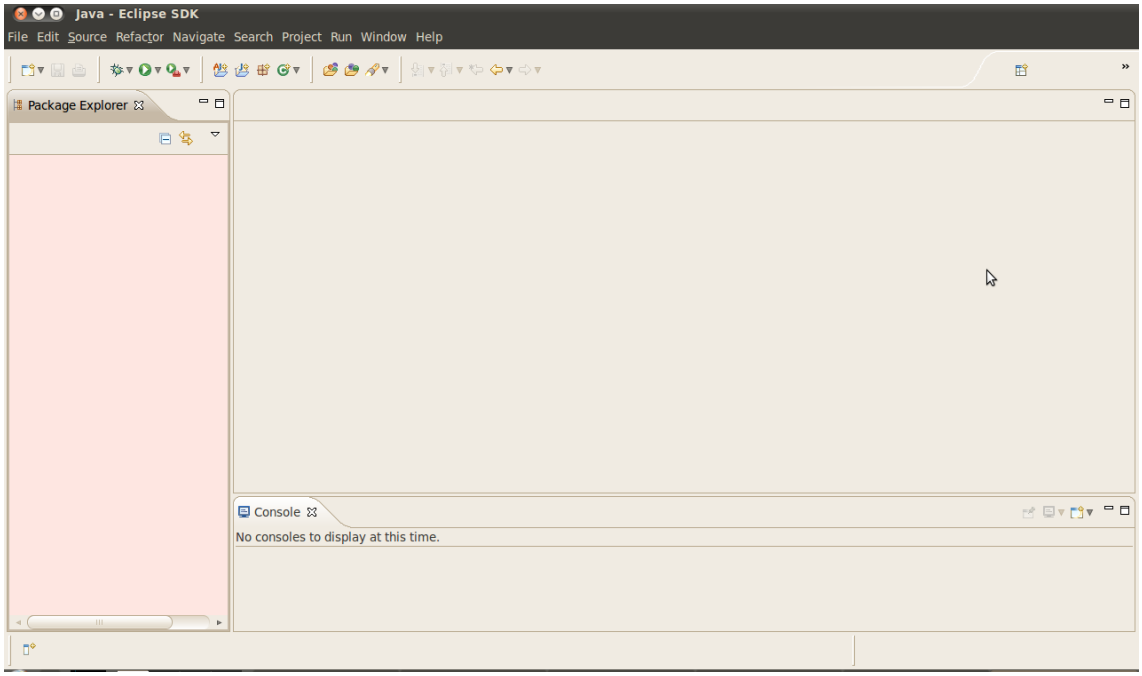

Figura 1.1: Tela inicial do Eclipse com workspace vazio

grubix está marcado. Se estiver clique em finish; senão marque-o e clique em finish. Pronto o GrubiX já esta pronto para você utilizá-lo.

Após carregado o projeto GrubiX você já pode fazer sua simulação. Para criar a sua simulação existem duas formas: criar um projeto no Eclipse ou modificar um projeto que já esteja configurado (recomendado).

Para criar um novo projeto no Eclipse, deve ser levado em consideração que o projeto deve ter ter toda configuração realizada (arquivos XML e log4j), além disso deve ser importado arquivos *.jar* que são necessários ao GrubiX. Para ver quais são esses arquivos e configurações, faça o download do projeto Aplicação Básica (BasicApplication), que também se encontra em *http://pesquisa.dcc.ufla.br/grub/dowloads*.

Em seguida, vá em File $\longrightarrow$ New $\longrightarrow$ Java Project $\longrightarrow$ Finish. Coloque um nome para a sua aplicação $\longrightarrow$ clique em Finish. Após feito isto sua tela deve estar conforme a tela da Figura 1.2 . Pronto seu projeto já esta criado mas ainda falta estabelecer uma relação de dependência com o GrubiX.

Crie os arquivos de configuração e importe as bibliotecas em *.jar* de acordo com o projeto Aplicação Básica.

Agora temos de estabelecer uma dependência de projetos. O projeto de aplicação que você criou deve ter uma relação de dependência com o GrubiX. Assim para fazer isto vá onde estão os projetos $\longrightarrow$ Clique com o botão direito do mouse em Properties da sua aplicação $\longrightarrow$ Vá no campo Project References $\longrightarrow$ Selecione o grubix $\longrightarrow$ Clique em Ok, para finalizar. Observe a Figura 1.3 e veja que estamos marcando a dependência do projeto Application em relação ao GrubiX.

Agora seu projeto já tem uma relação de dependência com o GrubiX e assim está pronto para inserir o código da simulação. Para ver como fazer isto leia o seção 1.8.

Contudo, esse trabalho de configuração e importação de arquivos não é necessário, porque já existe um projeto completamente configurado para isso. O projeto Aplicação Básica já atende as especificações do GrubiX e recomenda-se utilizá-lo para começar a desenvolver uma aplicação.

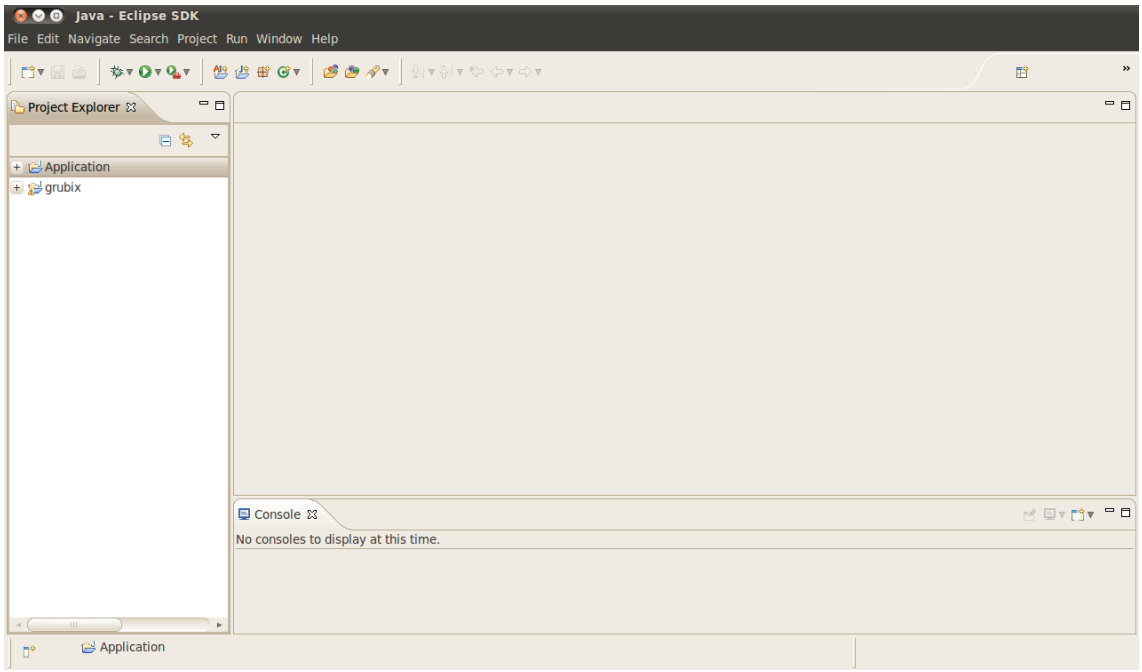

Figura 1.2: Tela após carregar o GrubiX e criar um novo projeto

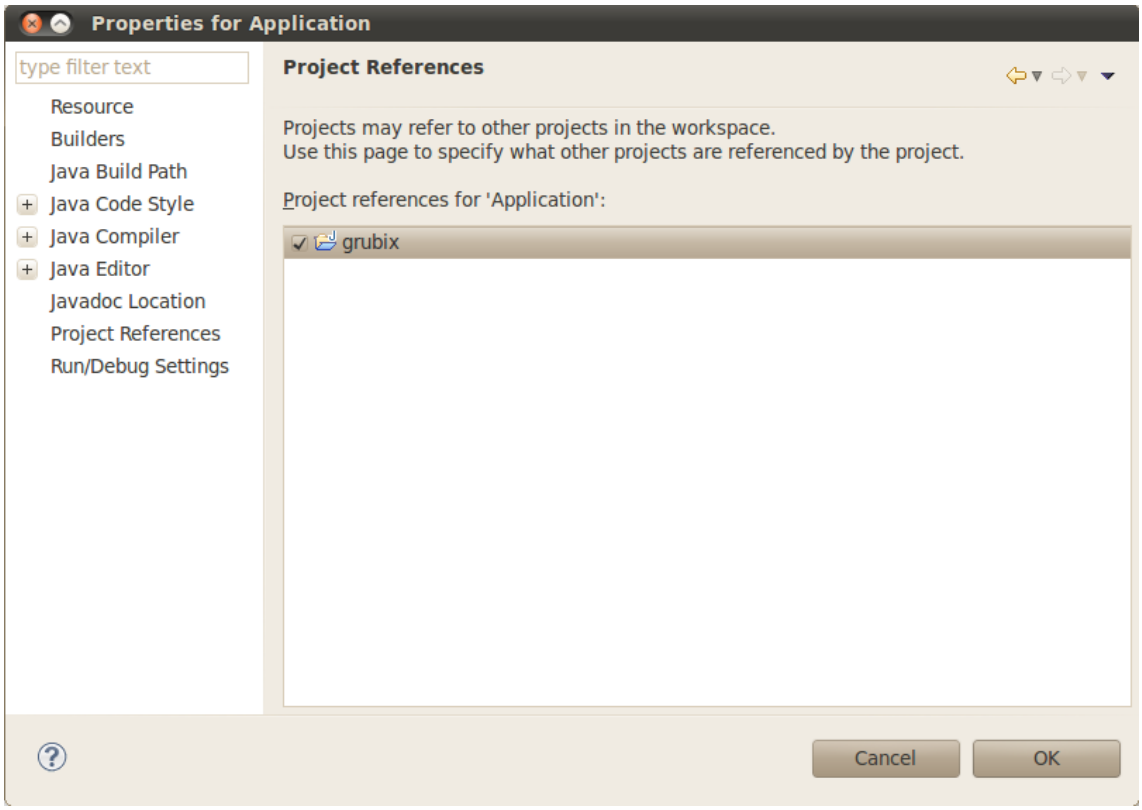

Figura 1.3: Estabelecendo relação de referência

1.4 O que é simulação orientada a eventos

O GrubiX é um simulador de eventos discretos que assume que um evento pode ocorrer em qualquer ponto do tempo. Cada evento tem uma marca indicando quando ele deve ser processado.

Na simulação orientada a eventos existe um procedimento associado com cada tipo de evento no sistema. O simulador ciclicamente escala eventos, atualiza o relógio para o próximo evento a ocorrer e executa o procedimento associado ao evento.

A execução no simulador envolve retirar o evento de uma lista global de eventos e processá-lo por um componente do sistema. O componente executará sua lógica e poderá gerar novos eventos. O tempo não é absoluto, mas uma quantia virtual.

Os eventos podem ser agendados para um determinado instante e esses serão ordenados por tempo de ocorrência. A ocorrência de um evento afeta o estado da simulação. Também são atualizados os contadores de estatísticas que permitem a geração dos relatórios da simulação.

1.5 Diagrama de classes

Para um melhor entendimento do simulador GrubiX, a sua arquitetura foi dividida em duas partes principais para a construção do diagrama de classes. A primeira a ser descrita é a arquitetura dos eventos, na qual mostra quais os tipos de eventos são processados pelo simulador. A estrutura dos nós-sensores na simulação ad hoc é apresentada posteriormente.

A Figura 1.4 mostra o diagrama de objetos com a estrutura dos eventos. Além da arquitetura dos eventos, a figura também apresenta a classe mais importante do framework de simulação: o SimulationManager. Essa classe tem o conhecimento de todos os objetos globais da simulação (como por exemplo, a Topologia) e controla todo o fluxo da rede. O controlador de eventos é feito nessa classe, que contém a referência para a fila de evento globais (fila de prioridades).

A fila de prioridade é responsável por manter todos os eventos agendados para a execução no sistema. Ao invés de colocar diretamente os eventos dentro da fila global, um objeto envelope é usado (classe EventEnvelope). Ele armazena uma informação futura sobre o evento e provê um método para processar o envento encapsulado no envelope. A classe Event é uma superclasse abstrata que representa todos os eventos na simulação. Eventos podem ser dos seguintes tipos: eventos internos, eventos que afetam todas as camadas do nó (classe ToNode) e eventos específicos de uma camada do nó-sensor (classe ToLayer).

Eventos de simulação são usados internamente pela simulação e não podem se desviar dos protocolos que estão sendo simulados. Exemplos de eventos internos são eventos de movimento, que são responsáveis por mudar a posição física de um dado nó, e consequentemente, a topologia da rede.

Eventos que afetam todas as camadas dos nós são modelados pela classe ToNode e afetam todas as camadas (pilha de protocolos) de um nó-sensor. Um exemplo de tal tipo de evento é o inicializar, que é utilizado para inicialização. Quando esse evento é lançado, todas as camadas dos nós recebem isso.

Já eventos que são específicos do nó-sensor são feitos pela classe ToLayer. Esses são sempre despachados para uma certa camda de um dado nó. O evento mais importante desse tipo é o pacote (Packet): pacotes são endereçados para uma camada específica de um nó alvo. Logicamente, toda camada pode se comunicar com o a mesma camada de outro nó. Entretanto, fisicamente os pacotes devem passa através das camadas inferiores antes de serem enviados através da rede física. Como é comum para o modelo de pilha de rede, um dado pacote contem todos os pacotes das camadas superiores. Isso é implementado no GrubiX por meio de uma agregação como é apresentada na classe Packet.

Outro evento muito importante presente em ToLayer é o WakeUpCall. Esse é utilizado para agendar eventos futuros no sistema. Por exemplo, depois de enviar um pacote, a camada de enlace, em algumas implementações, tem que esperar uma resposta (ACK). Para isso, um tempo limite é configurado para receber essa resposta e um evento WakeUpCall é emitido quando o tempo limite expira.

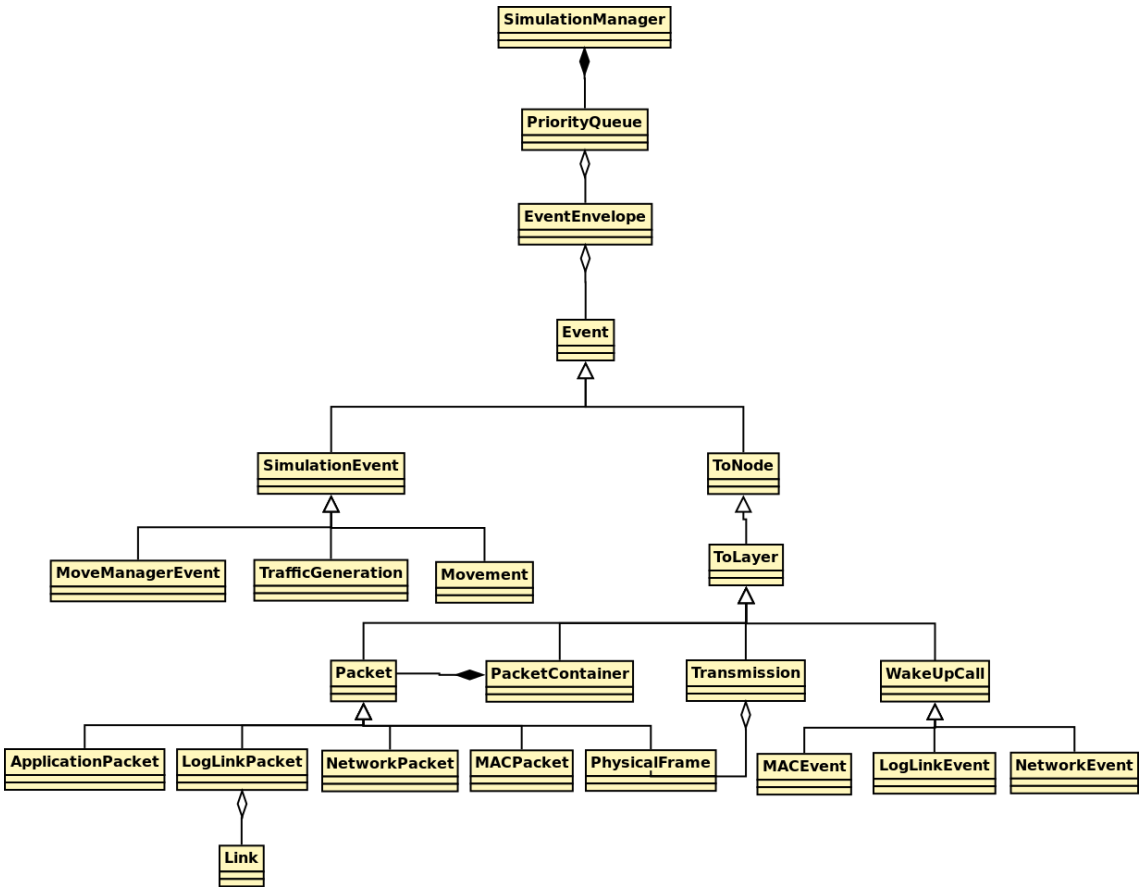

Figura 1.4: Diagrama de classes simplificado dos simulador GrubiX mostrando a arquitetura de eventos.

A segunda parte da arquitetura do GrubiX representa a estrutura dos nós. O diagrama de classes pode ser visto na Figura 1.5. O elemento central é o Node, ou seja, o nó-sensor. Um nó-sensor é especificado por suas camadas em um pilha de rede. As camadas de protocolos concretas são derivadas da classe abstrata Layer, que modela todo o tipo de camada. A gerenciador de simulador tem uma coleção completa dos nós do sistema. Cada nó da simulação corresponde a uma instância da classe Node e de todas as suas camadas.

Durante a fase de inicialização, todas as instâncias necessárias dos nós do sistema são geradas por um NodeGenerator. A classe Configuration é responsável por redefinir a configuração da simulação, feita pelo arquivo em XML, e por controlar, entre outras coisas, a a geração de um número correto de instâncias de nós com camadas especificadas. Por exemplo, para a implementação da métrica de enlace, a classe derivada da LogLinkLayer pode ser utilizada.

1.5.1 Descrição das classes e métodos

Nessa subseção, serão detalhadas, através de tópicos, todas as partes (classes e métodos principais) dos diagramas de classes simplificadas da Figura 1.4 e da Figura 1.5. A maioria dessas classes podem e devem ser utitlizadas em sua aplicação e algumas delas estão estabelecidas no arquivo de configuração.

As classes que determinam a arquitetura de eventos são:

1. **SimutionManager** - Classe central para todo o framework de simulação. Esse gerenciador

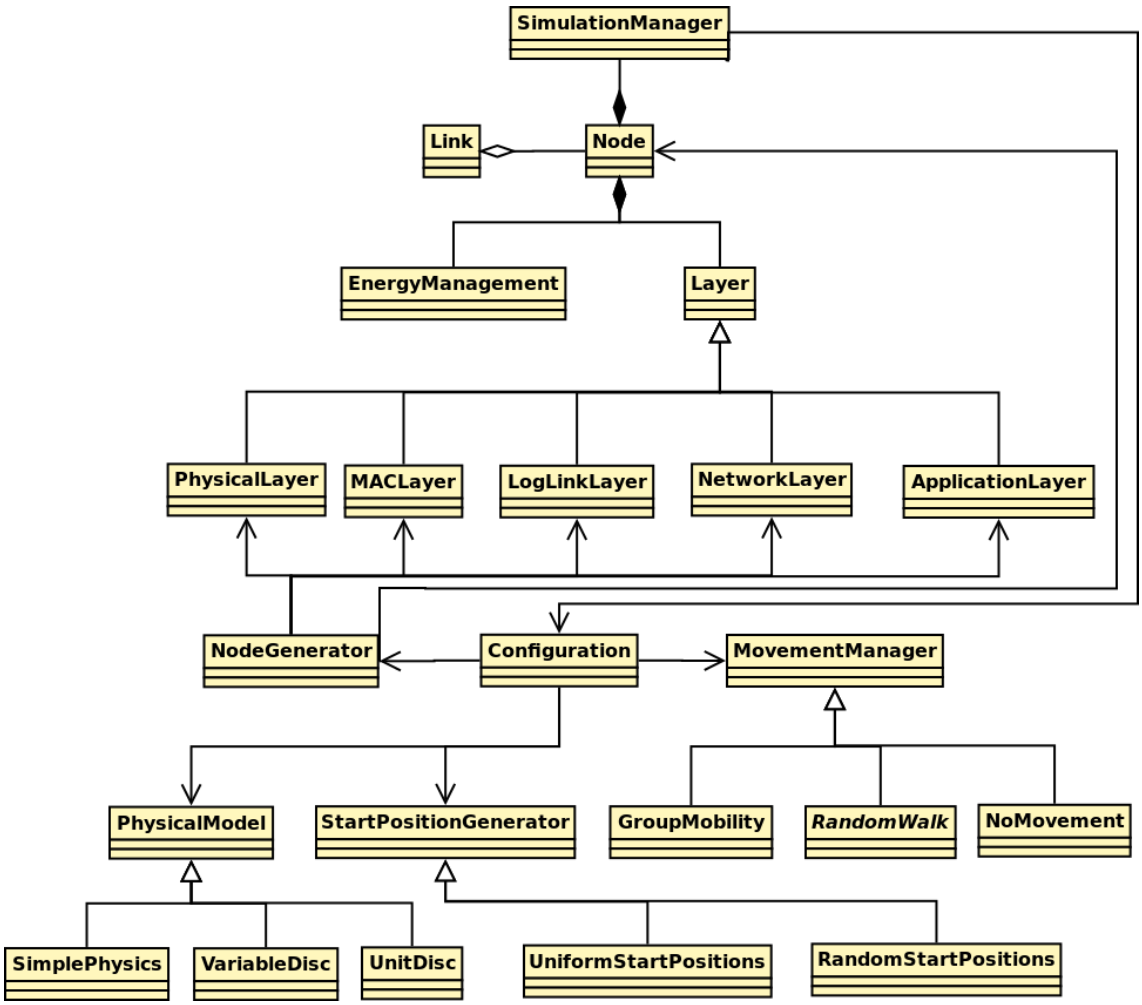

Figura 1.5: Diagrama de classes simplificado dos simulador GrubiX apresentando a estrutura dos nós-sensores.

conhece todos os objetos globais da simulação e controla todo o fluxo. A manipulação de eventos é feita nessa classe.

- **getInstance()** - Retorna a instância do SimulationManager.
- **runSimulation()** - Executa a simulação após todos os parâmetros estarem configurados. Continua os eventos.
- **endSimulation()** - Faz uma chamada interna no fim da simulação para que os nós possam executar limpezas finais.
- **cancelSimulation()** - Aborta a simulação corrente.
- **transmitPacket(Transmission senderTransmission)** - Método para transmissão de pacotes. Tem como parâmetro o pacote a ser transmitido e todos os parâmetros necessários para que isso seja feito.
- **logStatistic(parameters)** - Método estático para realizar LOG durante a simulação.
- **logNodeState(NodeId node, String name, String type, String value)** - Método que gera um LOG para determinar um estado do nó. Um estado do nó é um par (name, value), que é o nome do LOG para aquele evente e o valor desse LOG para isso. Esse

método pode determinar qualquer propriedade do nó para visualização no visualizador do GrubiX. O primeiro parâmetro determina o ID do nó que vai ter esse LOG e o terceiro determina o tipo do LOG (inteiro, real, string).

- **getCurrentTime()** - Eetorna o tempo corrente da simulação.
- **setCurrentTime(TimedObject obj)** - Configura o tempo de simulação para o especificado no parâmetro obj. O gerenciador da simulação toma a decisão se esse tempo é aceitável para a simulação.

2. **PriorityQueue** - Abstração feita pela classe SimulationManager para informar qual evento tem prioridade. O processamento de segue a lista de prioridades, que diz qual evento deve ser processado primeiro.

3. **EventEnvelope** - Classe abstrata para todos os envelopes. Envelopes guardam os eventos em um pacote para um futuro processamento, que é feito de acordo com uma lista global.

   - **EventEnvelope(Event event, double time)** - Método com acesso protegido qu encapsula um evento eventdurante um tempo time, para ser envidado para processamento.
   - **getEvent()** - Retorna o evento que foi encapsulado.

4. **Event** - Classe que representa os eventos na simulação. Eventos podem ser de vários tipos: como eventos internos, como movimentos (SimulationEvent), para todo o nó como inicializações (ToNode), para uma camada específica de um nó (ToLayer), etc.

   - **getId()** - Retorna a identificação (id) do evento requerido.

5. **SimulationEvent** - Classe abstrata para eventos internos de simulação.

6. **ToNode** - Classe abstrata para modelar todos os eventos enviados aos nós. Esses eventos devem ser enviados a todo o nó (camadas), não para uma camada específica.

   - **getReceiver()** - Retorna o nó receptor desse tipo evento.
   - **setReceiver(NodeId receiver** - Configura o nó receptor do evento.

7. **MoveManagerEvent** - Classe que modela eventos do gerenciador de movimento. A classe SimulatonEvent chama a classe MovementManager para gerar novos movimentos após um certo período.

   - **getDelay()** - Retorna o atraso para gerar novos movimentos que são requisitados.

8. **TrafficGeneration** - Essa classe representa um evento que é criado por um gerador de tráfego da rede e o entrega para instâncias da camada de aplicação de cada nó. A classe requisita à aplicação dos nós para enviar um pacote de dado para especificar o nó que recebeu o evento. Para ser hábil de distinguir tipos diferentes de pacotes que uma aplicação pode enviar, um identificador do tipo do pacote deve ser especificado na requisição de geração de tráfego.

9. **Movement** - Classe interna de simulação que representa o movimento dos nós. Criada pelo gerenciador de movimento (MovementManager). Depois que o movimento Movement é finalizado (tirado da lista global de eventos de simulação), a topologia é atualizada e um evento mudança de posição (Moved) é enviado para o nó-sensor.

10. **ToLayer** - Superclasse abstrata para todos os eventos que devem ser entregues a um nó específico e para uma camada específica no nó. A camada receptora é definida pelo tipo do objeto específico (por exemplo, LogLinkPacket). WakeupCalls são endereçados via enfileiramento.

    - **getSender()** - Retorna quem enviou o evento do tipo ToLayer.

- **setSender(Address Sender)** - Configura, caso seja requerido, o endereço do nó que está requisitando o evento.

11. **Packet** - Superclasse abstrata para todos os tipos de pacotes, ou seja, todos os demais pacotes herdam essa classe. Um pacote pode estar ou não encapsulado e a comunicação entre os nós da rede é possível somente entre as mesmas camadas.

    - **createReceiverLists()** - Esse método é chamado para transmitir para os nós desejados.
    - **addReceiver(boolean isValid, NodeId other)** - Adiciona um nó destino (other) a esse pacote para a lista apropiada do pacote original, somente se o nó seja válido para ser colocado na lista destino desse pacoto (isvalid).
    - **getDirection()** - Retorna a direção do pacote dentro do nó-sensor.
    - **getEnclosedPacket()** - Retorna o pacote encapsulado.
    - **getReceivers()** - Retorna a lista dos nós que receberão esse pacote.
    - **addExpectedReceiver(NodeId other)** - Método para adicionar um outro nó (other) a lista de nós destino.
    - **isExpectedReceiver(NodeId other)** - Método que informa se um dado nó (other) é considerado importante para a lista nós que receberão o pacote.
    - **getLayer()** - Retorna a camada no qual o pacote pertence atualmente.

12. **PacketContainer** - Abstração da classe Packet para mostrar que essa é parte de um contêiner de pacotes (PacketContainer), que é um tipo de encapsulamento utilizado pelo simulador para criar pacotes. No caso do exemplo do diagrama, um pacote forma uma parte essencial do contêiner. Caso o pacote seja destruído, o contêiner também será.

13. **ApplicationPacket** - Essa classe é uma classe abstrata para representar e modelar todos os tipos de pacotes da camada de aplicação. Como essa camada é a mais interna, nenhum pacote pode ser enscapsulado nessa camada.

    - **getHopCount** - Retorna a quantidade de saltos realizada pelo pacote.

14. **LogLinkPacket** - Classe abstrata de todos os pacotes gerados pela camada de enlace.

15. **NetworkPacket** - Classe implementada para fornecer uma abstração para todos os tipos de pacotes gerados pela camada de rede.

    - **getFinalNetworkPacketReceiver()** - Esse método retorna a identificação (NodeId) do nó destino do pacote de rede mais encapsulado.
    - **getInitialNetworkPacketSender()** - Esse método retorna a identificação (NodeId) do nó remetente do pacote de rede mais encapsulado.

16. **MACPacket** - Classe abstrata para os pacotes gerados pela camada de acesso ao meio (MAC).

17. **PhysicalPacket** - Essa classe representa moldes dos blocos de bits (frames) que são enviados via meio aéreo que é usado pela camda física para encapsular pacotes que foram enviados de um nó para outro. A principal proposta desse pacote é permitir a ligação lógica entre pacotes físicos.

    - **getSyncDuration()** - Retorna o tempo de sincronização de transmissão em passos de simulação.
    - **setSyncDuration(double headerDuration)** - Configura o tempo de sincronização de acordo com o cabeçalho.

- **getDuration()** - Retorna o tempo de duração em passos de simulação necessária para transmitir o pacote.
- **getBPS()** - Retorna a quantidade de bits por passos de simulação.
- **isTransitToWillSend()** - Retorna verdadeiro, se o nó destino enviou um ACK de resposta que a transmissão é possível. Assim, o rádio desse nó muda seu status para WILL-SEND.

18. **Transmission** - Classe para encapsular pacotes transmitidos durante o envio. Usada para tratamento adequado do pacote via módulo aéreo e armazenar dados relacionados com a comunicação (por exemplo, força do sinal).

    - **getSignalStrength()** - Retorna a força do sinal de transmissão.
    - **getSendingTime()** - Retorna o tempo do envio do pacote de acordo com os passos de simulação.
    - **setSenderPacket(PhysicalPacket senderPacket)** - Configura a referência para o pacote que será transmitido, para o pacote que se quer (senderPacket).
    - **getSenderPacket** - Retorna a referência para a instância do pacote que será usado na transmissão.

19. **WakeupCall** -Classe para eventos de "acordar" (despertador do nó-sensor). O nó remetente pode colocar esse evento para que o nó receptor acorde em um tempo específico.

20. **MACEvent** - Classe abstrata para tratar todos os eventos para serem processados pela camada MAC.

21. **LogLinkEvent** - Uma implementação de uma classe abstrata para todos os eventos que são processados pela camada de enlace (link lógico).

22. **Link** - Essa classe representa um enlace bidirecional entre dois nós. Um link é definido pelos IDs dos dois nós que estão conectados, e opcionalmente a taxa de bits e a potência de transmissão.

    - **createLink(Node u, NodeId v)** - Cria o enlace entre os dois nós, o nó que cria o link(u) e o destino desse link (v).
    - **createLink(Node node, NodeId v, BitrateAdaptationPolicy raPolicy, int bitrateIndex, double transmissionPower)** - Cria um enlace entre os dois nós, com nó que origina o enlace (node), o destino desse link(v), com a taxa de bits (raPolice, bitrateIndex) e a taxa de transmissão (transmissionPower).

23. **NetworkEvent** - Superclasse abstrata para todos os eventos enviados da camada de rede.

E as classes que estruturam os nós-sensores são:

1. **SimutionManager** - Classe descrita nos itens que descrevem a arquitetura de eventos.

2. **Node** - É uma classe abstrata para todos os tipos de nós. Um nó-sensor é especificado pelas suas camadas na pilha de rede. Nenhuma especialização para essa classe é permitida.

    - **processEvent(SimulationState simState)** - Processa eventos do tipo Simulation-State, através do simState, sendo que esse pode assumir instâncias do tipo Finalize (processa eventos de finalização), StartSimulation (processa eventos iniciais), Initialize (processa eventos de inicialização).
    - **processEvent(TrafficGeneration tg)** - Encaminha a solicitação de geração de tráfego para a instância da camada de aplicação. O parâmetro tg gera a requisição com todos os argumentos necessários.

- **getPosition()** - Retorna a posição real (x,y) do nó-sensor.
- **transmit(PhysicalPacket packet, double signalStrength)** - Transmite o pacote físico packet com uma força de sinal (signalStrength).
- **getId()** - Retorna o ID do nó-sensor.
- **getConfig()** - Retorna a configuração.
- **clearNeighbours()** - Limpa a lista de vizinhos.
- **addNeighbor(Node other)** - Adiciona um novo vizinho na lista de vizinhos. Esse método deve ser somente chamado pelo gerenciador de simulação (SimulationManager);
- **removeNeighbor(Node other)** - Remove um nó da lista de vizinhos.
- **getNeighbors()** - Retorna toda a lista de vizinhos.
- **getNeighborCount()** - Retorna o número de vizinhos do nó.

3. **Link** - Classe descrita nos itens que descrevem a arquitetura de eventos.

4. **EnergyManagement** - Classe responsável por modelar o gerenciador de energia do nó-sensor.

5. **Layer** - Superclasse abstrata para todos os tipos de camadas na pilha de rede. Uma camada tem um serviço geral de ponto de acesso (SAP), para que o pacote seja processado, seja ele advindo de uma camada superior (upperSAP) ou de uma camada inferior (lowerSAP). Todas as demais camadas estenderão a camada Layer e os métodos padronizados nessa camada são um padrão que deve ser sobrescrito para uma aplicação específica (protocolo).

   - **sendPacket(Packet packet))** - Implementado para que um pacote packet seja enviado ao longo da direção desejada para a próxima camada.
   - **sendEventUp(ToLayer event)** - Implementado para enviar um evento eventpara uma camada superior, em relação a camada que se está.
   - **sendEventDown(ToLayer event)** - Método implementado para enviar um evento event para uma camada inferior, em relação a camada que se está.
   - **endEventSelf(ToLayer event)** - Implementado para que um evento event seja enviado para a mesma camada, em relação a camada que se está.
   - **sendEventTo(ToLayer event, LayerType toLayer)** - Implementado para que um evento event seja enviado para uma camada específica toLayer.
   - **processEvent(ToLayer event)** - Implementado para processar um evento event que chegou à camada através de um serviço de ponto de acesso.
   - **processEvent(SimulationState simState)** - Método para inicializar/processar eventos iniciais/finalizar a camada correspondente, dependendo do tipo do parâmetro simState.

6. **PhysicalLayer** - Classe abstrata para a camada física de uma pilha de rede.

   - **processEvent(SimulationState simState)** - Método para inicializar/processar eventos iniciais/finalizar a camada correspondente, dependendo do tipo do parâmetro simState.
   - **standardPhyInit()** - Método padrão para ler os parâmetros comuns para todas as camadas físicas.

7. **MACLayer** - Classe abstrata para todos os tipos de camada MAC da pilha de rede.

   - **sendPacket(Packet packet)** - Com esse método, um pacote packet é enviado ao longo de uma direção desejada para a próxima camada. Esse método é usado para configurar o estado do rádio para WILL SEND, ou seja, para o estado "vai enviar", se o pacote é enviado para a camada física.

8. **LogLinkLayer** - Classe abstrata para todos os tipos de camada de enlace da pilha de rede. Uma camada de enlace pode configurar uma informação específica do enlace (força do sinal, etc.) para cada pacote.

    - **addLinkToPacket(Packet packet, Link link)** - Método de ajuda para adicionar um enlace link para um pacote packet.
    - **sendPacket(Packet packet)** - Método chamado para enviar um pacote packet. A implementação atual usa a implementação da classe Layer se o enlace existe, mas caso nenhum enlace esteja ajustado, a camada adiciona-o ao pacote.

9. **NetworkLayer** - Classe abstrata para todos os tipos de camada de rede da pilha de rede.

10. **ApplicationLayer** - Classe abstrata para todos os tipos de camada de aplicação da pilha de rede.

    - **initConfiguration(Configuration configuration)** - Inicializa a configuração com uma configuração padrão. A implementação corrente checa se o contador do tipo do pacote e gera uma exceção caso o tipo do pacote seja diferente do previsto.
    - **processWakeUpCall(WakeUpCall wuc)** - Evento do tipo ToLayer que é tratado pela superclasse Layer. Esse processamento é realizado quando quer-se esperar um tempo para que um evento desejado ocorra.

11. **NodeGenerator** -A classe geradora de nós para a simulação. Chamada somente durante o processo de inicialização.

    - **generateNodes(Configuration configuration, SortedMap≤NodeId, Node≥ allNodes)** - Método estático que gera os nós de um arquivo de configuração especificado. Os parâmetros são a configuração fornecida pelo arquivo XML e um mapa para comportar todos os nós da aplicação.
    - **log(Configuration configuration, ShoxLogger writer)** - Método para registar a geração dos nós (posições, IDs, etc.) em um arquivo. Os parâmetros são a configuração e o arquivo a ser escrito.

12. **Configuration** - Classe para ajustar todos os parâmetros de configuração.

    - **getInstance()** - Método para acessar a instância da configuração, utilizando-se do padrão de projetos Singleton.
    - **getNodeCount()** - Retorna a quantidade de nós.
    - **getStartPositions()** - Retorna um mapa com as posições dos nós geradas para cada nó.
    - **getSimulationSteps()**- Retorna o período em que são feitos os passos da simulação
    - **getXSize() e getYSize()** - Retornam as dimensões (largura e altura) da área de sensoriamento.

13. **MovementManager** - Superclasse abstrata para todos os tipos de geradores de movimento. O gerenciador de simulação (SimulationManager) chama o gerenciador de movimentos (MovementManager) para criar os movimentos repetidamente em um tempo constante.

14. **GroupMobility** - Possíveis implementações com diferentes tipos de padrões de mobilidade.

15. **RandomWalk** - Gerenciador para criar movimentos aleatórios dos nós. Cada nó é movido por uma distância aleatória entre zero e um limite. Sendo q a direção também é gerada de forma aleatória.

    - **createMoves(Collection≤Node≥ allNodes)** - cria movimentos randômicos para todos os nós configurados com esse tipo de movimento. Tem como parâmetro os nós que terão movimento aleatório. As distâncias e direções são sorteadas.

16. **NoMovement** - Classe que gera nenhum movimento para todos os nós. Todos os nós são estacionários, ou seja, nenhum evento de movimento é realizado pelo simulador

17. **PhysicalModel** - Classe para todos os tipos de modelos físicos. Modelos físicos são usados para prover diferentes modelos de comunicação. Eles devem decidir se uma mensagem alcança um nó receptor de um nó remetente. Para isso é verificado a força do sinal, a distância, etc.

18. **SimplePhysics** - Modelo físico simplificado . Alcance de transmissão global.

    - **apply(Node re, Node se, double signalStrength)** - Usa o modelo físico simplificado para calcular a capacidade de alcance para um nó receptor re e outro remetente se, dada a força do sinal signalStrength.

19. **VariableDisc** - É um modelo de propagação de sinal baseado no modelo de propagação espaço livre, que dita que um pacote alcança o receptor com força de sinal igual a: s(rx) = s(tx) / d^2 (d é a distância euclidianta entre o nó remetente e o nó receptor. Se o sinal recebido pelo receptor for menor que um limiar t, então o receptor é considerado não alcançável pelo nó remetente.

    - **apply(Node re, Node se, double signalStrength)** - Usa o modelo físico VariableDisc para calcular a capacidade de alcance para um nó receptor re e outro remetente se, dada a força do sinal signalStrength.

20. **UnitDisc** - É um modelo determinístico com uma constante de atraso de transmissão. Para esse modelo são necessárias um limite de transmissão e uma constante para o atraso. O pacote alcança um nó receptor se a distância euclidiana entre as posições dos nós envolvidos é menor ou igual ao limite de transmissão.

    - **apply(Node receiver, Node sender, double signalStrength)** - Usa o modelo físico UnitDisc para calcular a capacidade de alcance para um nó receptor receiver e outro remetente sender, dada a força do sinal signalStrength.

21. **StartPositionGenerator** - Superclasse abstrata parara todas as classes que geram posições iniciais dos nós-sensores. A posição inicial do nó pode ser ou não especificada explicitamente no arquivo de configuração (uma posição para cada nó), ou, eles podem ser gerados de acordo com alguma distribuição por uma subclasse da StarPoositionGenerator.

22. **UniformStartPositions** - Essa classe gera uniformemente posições iniciais dos nós-sensores sobre a área de sensoriamento. As posições são arranjadas como uma grade, sendo que os pontos de encontro de cada linha com coluna são as posições dos nós.

    - **newPosition(Node node)** - Gera uniformemente posições iniciais para o nó node.

23. **RandomStartPositions** - Essa classe gera aleatoriamente posições iniciais dos nós-sensores segundo as dimensões da área de sensoriamento.

    - **newPosition(Node node)** - Gera aleatoriamente posições iniciais para o nó node.

1.6 Filosofia de desenvolvimento

Podemos citar como vantagens do GrubiX em relação a outros simuladores a possibilidade de implementar protocolos novos de uma maneira simples e rápida utilizando orientação a objetos (o simulador é escrito na linguagem Java). Além disso, o simulador apresenta uma ferramenta de configuração compreensiva e flexível, além de uma ferramenta de visualização e verificação dos resultados.

Uma das ideias do simulador GrubiX que o torna interessante é a abstração de classes. Essa ideia traz rapidez e muita facilidade de desenvolvimento, porque qualquer desenvolvedor que queira

adicionar funcionalidades ao simulador, como protocolos por exemplo, apenas precisa estender uma camada abstrata existente, que modela a camada à qual esse protocolo pertence e começar a desenvolver. Essa camada contém variáveis, constantes e métodos já padronizados para redes móveis ad hoc. Isso agiliza, facilita e aumenta a confiabilidade do projeto de sistemas via recursos computacionais, visto que diminui o esforço do projetista porque aumenta o nível de abstração de modelagem, permitindo ao projetista preocupar-se com necessidades de maior importância, como por exemplo, a fidelidade do modelo em relação ao sistema real.

A Figura 1.6 exemplifica essa filosofia.

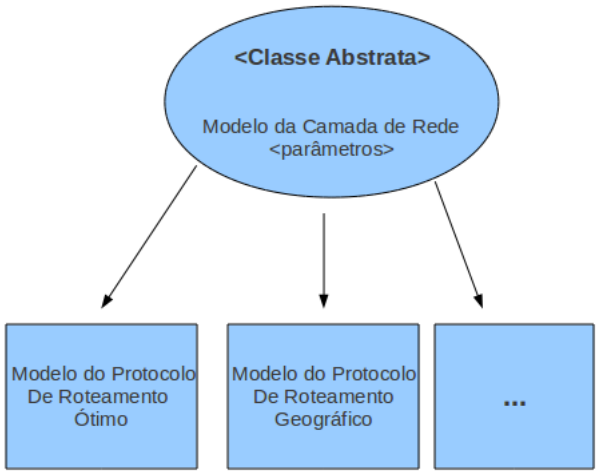

Figura 1.6: Filosofia de desenvolvimento do GrubiX.

1.7 Configurar simulações no GrubiX

Para que o GrubiX "entenda" quais protocolos, padrões e parâmetros devem ser passados e utilizados pela rede, a definição de um arquivo no formato XML tem que ser encontrado no mesmo diretório de desenvolvimento da aplicação. Esse arquivo XML será descrito na subseção 1.7.1. A classe principal "Main.java" é responsável por essa tarefa. Nela está contido o código 1.1 que passa ao simulador argumentos contidos no XML "application.xml" que são necessários à aplicação. A importação, pela cláusula *import* do *kernel* do simulador, é feito justamente para essa tarefa.

Listing 1.1: Classe principal utilizada no GrubiX.

```
import br.ufla.dcc.grubix.simulator.kernel.Simulator;


public class Main {

        public static void main(String[] args) {
                String path = "application.xml";
                args = new String[1];
                args[0] = path;
                Simulator.main(args);
        }

}
```

As definições e características desse arquivo XML serão descritas na subsecção seguinte.

1.7.1 Definições do arquivo XML de configuração

Nos tópicos seguintes serão apresentados todas as descrições (passo-a-passo) do arquivo XML responsável por configurar uma aplicação no GrubiX. Suas *tags* ou marcações que representam características do meio físico e do sensor sem fio que são implementadas por classes que foram descritas no Diagrama de Classes da seção 1.5.

1. As primeiras tags do XML representam os parâmetros e classes que irão modelar o meio físico. O código XML 1.2 é descrito pelos seguintes itens:

   - A tag *field* informa o tamanho da área em função da largura (x) e da altura (y).
   - A tag *physicalmodel* é implementada, no caso da figura abaixo, pela classe *br.ufla.dcc.grubix.simulator.physical.UnitDisc*. Essa tag delimita a distância alcançável pelos nós para realização da transmissão e distância miníma para que um nó não interfira na comunicação do outro.
   - A tag *movementmanager* define ou não a movimentação do nó.
   - Por fim, a tag *bitmanglingmodel* trata colisões, caso essas ocorram.

Listing 1.2: Tags para o meio físico.

```
<field>
    <dimension x="200" y="185"/>
</field>

<physicalmodel>
    <class>br.ufla.dcc.grubix.simulator.physical.UnitDisc</class>

    <params>
      <param name="reachableDistance">30</param>
      <param name="interferenceDistance">30</param>
    </params>
</physicalmodel>

<movementmanager>
    <class>br.ufla.dcc.grubix.simulator.movement.NoMovement</class>
</movementmanager>

<bitmanglingmodel>
    <class>br.ufla.dcc.grubix.simulator.physical.CollisionPacketMangler</
        class>
</bitmanglingmodel>
```

2. A segunda parte do arquivo XML é responsável pelos comportamentos específicos dos nós-sensores sem fios, os quais estão caracterizados na tag *nodes*. Primeiramente, para cada tipo de nó-sensor é definida uma tag, como nome *node*. Nela é dado um nome e uma quantidade para esse tipo de nó-sensor. Entrando nas marcações de *node*, existe em *layers*, diversas outras tags , nas quais são especificados os modelos de camadas (pilha de protocolos) para as redes que serão implementadas. As tags descritas implementam todas as características do nós-sensores sem fios presentes na rede. O código 1.3 e os itens abaixo irão descrever e exemplificar a tag *node*:

   - As tags *operatingSystem* e *energyManager* descrevem respectivamente abstrações para um sistema operacional e para um controlador de energia presentes no nó-sensor.
   - A tag *physical* especifica como a camada física será implementada. No exemplo da figura abaixo, parâmetros para especificar as características do nó são configurados.
   - A tag *mac* configura parâmetros e qual é o protocolo que irá controlar o acesso ao meio.
   - A tag *logLink* define qual classe implementará a camada de enlace da pilha de protocolos.

- A tag *network* é responsável por implementar as características da camada de rede, como por exemplo, os algoritmos de roteamento que a compõem.
- Para finalizar as tags *layers* e *node*, a classe que desenvolve a camada de aplicação do nó deve ser especificada dentro da tag *application*.

Listing 1.3: Tags para caracterizar o nó-sensor sem fio.

```
<nodes>
    <node name="REGULAR" count="30">
        <layers>
            <operatingSystem>
                <class>br.ufla.dcc.grubix.simulator.node.user.os.
                    NullOperatingSystemLayer</class>
            </operatingSystem>

            <energyManager>
                <class>br.ufla.dcc.grubix.simulator.node.energy.
                    BasicEnergyManager</class>
            </energyManager>

            <physical>
                <class>br.ufla.dcc.grubix.simulator.node.user.
                    PHY_802_11bg</class>
                <params>
                    <param name="standard">g</param>
                    <param name="preamble">short</param>
                    <param name="channelCount">1</param>
                    <param name="signalStrength">0.0</param>
                    <param name="minSignalStrength">0.0</param>
                    <param name="maxSignalStrength">100.0</param>
                    <param name="minFrequency">2.4e9</param>
                    <param name="maxFrequency">2.4e9</param>
                </params>
            </physical>

            <mac>
                <class>br.ufla.dcc.grubix.simulator.node.user.
                    MAC_IEEE802_11bg_DCF</class>
                <params>
                    <param name="retryLimit">10</param>
                    <param name="rateAdaption">AARF</param>
                    <param name="raUpLevel">10</param>
                    <param name="raDownLevel">2</param>
                    <param name="raUpMult">2</param>
                    <param name="raTimeout">10</param>
                </params>
            </mac>

            <logLink>
                <class>br.ufla.dcc.grubix.simulator.node.user.
                    LogLinkDebug</class>
            </logLink>

            <network>
                <class>br.ufla.dcc.grubix.simulator.node.user.
                    OptimalSourceRouting</class>
            </network>

            <application>
                <class>br.ufla.dcc.pp.node.RegularNode</class>
            </application>
        </layers>
    </node>
</nodes>
```

3. A terceira e última parte para descrição do arquivo XML de configuração e de suas respectivas tags serão feitas no código 1.4 e nos itens abaixo. Esses itens descrevem onde os resultados das informações serão salvos (Log), quais posições serão dadas aos nós e qual o tempo de simulação.

   - A tag *logging* define se o Log dos eventos ocorridos na rede será feito ou não, quais serão os nomes dos arquivos de Logs e de estatísticas, qual classe irá guardar as informações no arquivo e, por fim, quais eventos serão captados pelo arquivo, considerando as classes que implementam eventos.
   - A tag *position* configura qual classe será a geradora das posições dos nós-sensores na rede e qual o arquivo que será gerado para definir essas posições (ponto x,y). No caso do código 1.4 o arquivo onde estão definidas as posições dos nós-sensores é *startpositions.xml*.
   - Para finalizar o arquivo XML, a tag *simulationtime* mostra o tempo de simulação, juntamente com qual unidade de tempo será utilizada na simulação e os passos no qual os eventos acontecem dentro desse tempo, definido pelo desenvolvedor do GrubiX.

Listing 1.4: Tags para log em arquivos e posicionamento dos nós-sensores.

```
<logging>
    <log>true</log>
    <nameHistoryFile>LogAppPingPong</nameHistoryFile>
    <nameStatisticsFile>StatiticsAppPingPong</nameStatisticsFile>
    <logClass>br.ufla.dcc.grubix.debug.compactlogging.CompactFileLogger</
        logClass>
    <filter>
          <description>Sample filter</description>
          <logdata>true</logdata>
          <acceptedtypes>
              <class priority="off">br.ufla.dcc.grubix.simulator.event.user.
                  OptSrcRoutingPacket</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  PhysicalPacket</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  Transmission</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  TransmissionBeginIncoming</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  TransmissionEndIncoming</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  TransmissionEndOutgoing</class>
              <class priority="info">br.ufla.dcc.grubix.simulator.event.
                  Movement</class>
          </acceptedtypes>
    </filter>
</logging>

<positions>
    <generated>
        <generator>
            <class>br.ufla.dcc.grubix.simulator.movement.RandomStartPositions</
                class>
                <params>
                </params>
        </generator>
        <targetfile>startpositions.xml</targetfile>
    </generated>
</positions>

<simulationtime stepspersecond="100" base="seconds">600</simulationtime>
```

Uma característica importante desse arquivo de configuração em XML e, consequentemente, do simulador GrubiX é a possibilidade de ter nós-sensores implementados de formas diferentes.

Isso é possível porque dentro da tag *nodes* podem existir várias definições de nós-sensores, ou seja, pode haver heterogeneidade entre os nós-sensores, com respeito às suas camadas e características. Isso será explicado com mais detalhes na subseção 1.8.4.

1.8 Desenvolvendo e executando uma aplicação básica no GrubiX

1.8.1 Caracterização do Problema

O objetivo dessa aplicação será o incremento da contagem dos saltos, a medida que um pacote caminha sobre a rede. O primeiro nó que envia o pacote é o nó com id igual a 1. Esse nó-sensor envia o pacote ao nó com id igual a 2 um pacote que contém o contador de saltos. O nó 2, faz o incremento da informação do pacote e envia para o nó com id igual a 3. Isso é feito até que o contador atinja valor 30. Tudo é feito através de fluxo *unicast*. A Figura 1.7 mostra esse processo.

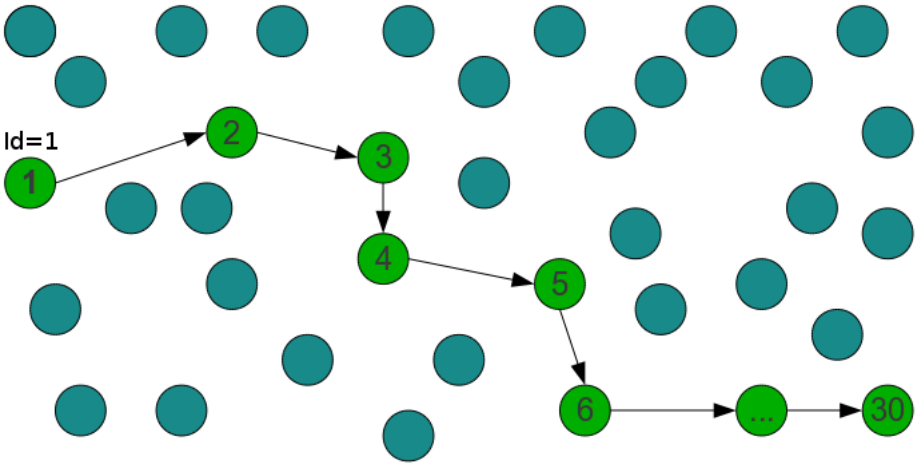

Figura 1.7: Tags para log em arquivos e posicionamento dos nós.

1.8.2 Desenvolvendo uma simulação no GrubiX

Para construir uma aplicação utilizando o simulador GrubiX, alguns cuidados devem ser tomados, em relação a seus padrões. O primeiro passo é criar um projeto Java, importar o projeto do simulador GrubiX e estender o projeto criado ao projeto GrubiX, como foi explicado. Em segundo, tem que ser pensado que toda a parte de comunicação entre os nós sensores, segue por tráfego de pacotes. Portanto, vamos começar com a criação desse pacote. Essa é uma classe simples, no nosso caso ela estará no pacote do projeto *br.ufla.dcc.packages*. Para que ele funcione corretamente, devem ser importados três classes do GrubiX:

1. **br.ufla.dcc.grubix.simulator.Address** - para conter o endereço de quem enviou o pacote.
2. **br.ufla.dcc.grubix.simulator.NodeId** - para identificar para qual nó-sensor o pacote vai ser enviado.
3. **br.ufla.dcc.grubix.simulator.event.ApplicationPacket** - para identificar que o pacote vai ser da camada de aplicação.

No código 1.5 é exibida a classe do pacote. Ele extende o pacote da camada de aplicação, por *ApplicationPacket*. Um contador foi utilizado para controlar o número de saltos do pacote durante

seu trajeto na rede de sensores sem fios. Para esse controle, métodos *get* e *set* foram criados. Além disso, um construtor desse pacote foi criado, com o endereço de quem o está enviando e qual nó-sensor vai recebê-lo. Todo o controle é feito pela classe *ApplicationPacket* utilizando a cláusula *super*.

Listing 1.5: Pacote da camada de aplicação utilizado para comunicação entre nós

```
package br.ufla.dcc.packages;

import br.ufla.dcc.grubix.simulator.Address;
import br.ufla.dcc.grubix.simulator.NodeId;
import br.ufla.dcc.grubix.simulator.event.ApplicationPacket;

public class Pacote extends ApplicationPacket{

        private int cont;

        public Pacote(Address sender, NodeId receiver) {
                super(sender, receiver);
        }

        public int getCont() {
                return cont;
        }

        public void setCont(int cont) {
                this.cont = cont;
        }
}
```

Da mesma forma que a classe Pacote, existe outra classe importante que é a responsável por gerar *Wakeupcalls*, que é o despertar do nó em um tempo definido para realização de alguma tarefa. Essa classe está presente no pacote *br.ufla.dcc.wuc*. Para o funcionamento correto, os seguintes pacotes devem ser importados:

1. **br.ufla.dcc.grubix.simulator.Address** - para conter o endereço de quem enviou o pacote.
2. **br.ufla.dcc.grubix.simulator.event.WakeUpCall** - classe responsável pelo controle dos *WakeUpCalls*.

No caso do problema que está sendo tratado, uma variável contadora e seus métodos *get* e *set* também foram criados, assim como na classe Pacote. Um construtor é necessário para que contenha o endereço do nó que está solicitando o *wakeupcall* e o tempo de atraso que o nó vai esperar até começar a iniciar o processamento. Todo o controle desses atrasos são feitos pela classe mãe *WakeUpCall*, que foi estendida. O controle é realizado com a cláusula *super*. O código 1.6 abaixo, mostra o que foi dito a respeito dessa classe.

Listing 1.6: Classe WakeUpCall responsável por gerar atrasos nos eventos.

```
package br.ufla.dcc.wuc;

import br.ufla.dcc.grubix.simulator.Address;
import br.ufla.dcc.grubix.simulator.event.WakeUpCall;

public class AppWuc extends WakeUpCall {
        private int cont;

        public AppWuc(Address sender, double delay) {
                super(sender, delay);
        }

        public int getCont() {
                return cont;
        }
```

```
        public void setCont(int cont) {
                this.cont = cont;
        }
}
```

Agora, depois das duas classes auxiliares terem sido criadas, a classe que representará o nó-sensor pode ser construída. Crie a classe *SensorNode* no pacote do projeto *package br.ufla.dcc.node*. Em seguida estenda a essa classe a classe *ApplicationLayer*. Isso identifica que a classe criada vai estar funcionando na camada de aplicação. Assim que essa classe for estendida, ocorrerá um erro, porque existem métodos padronizados da camada de aplicação que não foram implementados. Portanto, eles devem ser implementados. Se o ambiente utilizado é uma IDE otimizada, ela consegue completar esses métodos e as importações de classes do GrubiX necessárias à aplicação. O código 1.7 mostra a classe que modelará o nó-sensor já com os métodos implementados.

Listing 1.7: Classe que representa o nó-sensor sem fio.

```
package br.ufla.dcc.app.node;

import br.ufla.dcc.grubix.simulator.LayerException;
import br.ufla.dcc.grubix.simulator.event.Packet;
import br.ufla.dcc.grubix.simulator.event.TrafficGeneration;
import br.ufla.dcc.grubix.simulator.node.ApplicationLayer;
import br.ufla.dcc.pp.Pacote;

public class SensorNode extends ApplicationLayer {

        @Override
        public int getPacketTypeCount() {
                // TODO Auto-generated method stub
                return 0;
        }

        @Override
        public void processEvent(TrafficGeneration tg) {
                // TODO Auto-generated method stub

        }

        @Override
        public void lowerSAP(Packet packet) throws LayerException {
                // TODO Auto-generated method stub

        }
}
```

Como o GrubiX é um simulador baseado em eventos, um primeiro evento é necessário para iniciar os demais. Isso é feito por um método chamado *processEvent*, que tem como parâmetro do tipo evento de simulação inicial (*StartSimulation*). O método para o nosso problema, vai comparar dos 30 nós presentes, qual tem identificação igual a um. Assim, que achá-lo, vai disparar o primeiro evento, que será o de criar o pacote, configurar seu contador e enviá-lo para o nó que tem identificação igual a dois. Em seguida, é criada uma tag para representar esse evento no visualizador do GrubiX, sendo que isso é feito utilizando o método *SimulationManager.logNodeState*. Essa tag irá mostrar uma cor no visualizador, quando o evento ocorrer. Nesse código serão geradas três dependências que vão necessitar da importação de duas classes do GrubiX e uma que foi criada, que são: *br.ufla.dcc.grubix.simulator.event.StartSimulation*, *br.ufla.dcc.grubix.simulator.kernel.SimulationManager* e a *import br.ufla.dcc.packages.Pacote*. O código 1.8 mostra o método que foi criado com essas características.

Listing 1.8: Método que gera o evento inicial de simulação.

```
//Evento inicial:
protected void processEvent(StartSimulation start) {
```

```
        //Se a identificacao do noh eh igual a 1:
        if (this.node.getId().asInt() == 1){

                //Cria o pacote com id do noh que vai receber
                //e o endereco do noh que estah enviando:
                Pacote pk2 = new Pacote(sender,NodeId.get(2));

                // seta o valor do contador para 2:
                pk2.setCont(2);

                // envia o pacote da camada de aplicacao:
                sendPacket(pk2);

                // cria uma tag desse evento:
                SimulationManager.logNodeState(this.node.getId(), "Primeiro-envio",
                     "int", String.valueOf(10));
        }
}
```

O próximo passo é tratar a chegada de pacotes nos nós-sensores. Isso é feito pelo método *lowerSAP* que trata a chegada de pacotes que vieram das camadas de baixo. Esse método já foi criado, basta modificá-lo. Dessa forma, quando o pacote chegar, tem que ser visto seu tipo, se ele é do tipo *Pacote*, por exemplo, e em seguida realizar o tratamento. No código 1.9 é mostrado a verificação do tipo do pacote pela condição com *instanceof*. Logo após isso, é feito o controle do pacote recebido, criando-se uma tag para sua recepção, tendo sua cor no visualizador identificada pelo contador do pacote. Feito isso, um *WakeUpCall* é criado para enviar um outro pacote para um outro nó, sendo que isso será processado daqui a 1000 unidades de tempo do simulador. Uma condição é feita para que o próximo envio do pacote ocorra apenas se o número de saltos ser menor/igual a 30. Caso isso seja verdadeiro, o contador do *WakeUpCall* é configurado como o contador do pacote recebido e o evento de *WakeupCall* é enviado para que o próprio nó processar, utilizando o *sendEventSelf*. Vai ser gerada uma dependência com a criação desse *WakeUpCall* que pode ser tratada com a importação da classe *br.ufla.dcc.wuc.AppWuc*. Um outro conflito por dependência pode ser solucionado com a importação da classe *br.ufla.dcc.grubix.simulator.event.Packet*, que modela pacotes gerais advindos de outros nós.

Listing 1.9: Método para tratar chegada de pacotes.

```
// Processa um pacote que vem de uma camada de baixo:
@Override
public void lowerSAP(Packet packet){
        //Se o pacote recebido eh do tipo Pacote:
        if (packet instanceof Pacote){
                // Pega o pacote que ele recebeu;
                // Faz um cast;
                Pacote pac = (Pacote)packet;

                // cria uma tag que chegou o pacote:
                SimulationManager.logNodeState(this.node.getId(), "Chegou pacote",
                    "int", String.valueOf(pac.getCont()));

                // processa um evento daqui um tempo, cria um wakeup call para ser
                     iniciado daqui um tempo de 1000:
                AppWuc wuc = new AppWuc(sender, 1000);

                // se o contador do pacote for menor/igual que 30:
                if(pac.getCont() <= 30){

                        //seta o contador do wakeup call pro contador do pacote que
                             ele recebeu:
                        wuc.setCont(pac.getCont());

                        //Envia o evento para ele mesmo, para processar daqui um
                            tempo de 1000:
                        sendEventSelf(wuc);
```

```
                }
        }
}
```

Já que o *WakeUpCall* no nó-sensor foi criado quando o pacote foi recebido, agora ele deve ser tratado. Para isso existe no GrubiX um método padrão, que é o *processWakeCall*. Do mesmo modo do método *lowerSAP*, ele verifica o tipo do *WakeUpCall* para realizar um tratamento ou processamento específico àquele tipo de pacote. No caso do código 1.10 é feito essa verificação e em seguida é incrementado o contador do *WakeUpCall* em um e isso é armazenado em uma variável auxiliar. Isso é feito para se saber para qual nó-sensor o próximo pacote será enviado. Com esse incremento pode ser criado um novo pacote, configurando seu contador para o valor do auxiliar e, por fim, enviando o pacote para algum nó-sensor.

Listing 1.10: Método para tratar WakeUpCalls.

```
//Processa o WakeupCall dos nos-sensores:
public void processWakeUpCall(WakeUpCall wuc){

        // Verifica WakeupCall recebido eh do AppWuc:
        if (wuc instanceof AppWuc){
                // Pega o contador do WakeupCall e incrementa um:
                int aux = ((AppWuc) wuc).getCont()+1;

                // Cria um novo pacote e manda para o proximo
                Pacote pk = new Pacote(sender,NodeId.get(aux));

                // Configura o contador do pacote como o incremento do contador do
                    WakeupCall:
                pk.setCont(aux);

                //Envia o pacote para o proximo:
                sendPacket(pk);
        }
}
```

Todos os códigos criados estão em uma pasta chamada *Codigos-Utilizados* para retirada de eventuais dúvidas. Mas no fim, o código do seu nó terá a aparência de 1.11.

Listing 1.11: Código em Java que modela o nó-sensor sem fio.

```
package br.ufla.dcc.node;

import br.ufla.dcc.grubix.simulator.LayerException;
import br.ufla.dcc.grubix.simulator.NodeId;
import br.ufla.dcc.grubix.simulator.event.ApplicationPacket;
import br.ufla.dcc.grubix.simulator.event.Finalize;
import br.ufla.dcc.grubix.simulator.event.Packet;
import br.ufla.dcc.grubix.simulator.event.StartSimulation;
import br.ufla.dcc.grubix.simulator.event.TrafficGeneration;
import br.ufla.dcc.grubix.simulator.event.WakeUpCall;
import br.ufla.dcc.grubix.simulator.kernel.SimulationManager;
import br.ufla.dcc.grubix.simulator.node.ApplicationLayer;

import br.ufla.dcc.packages.Pacote;
import br.ufla.dcc.wuc.AppWuc;

//Classe que vai modelar os nos-sensores
//Simulara o comportamento dos nos-sensores
public class RegularNode extends ApplicationLayer {

        @Override
        public int getPacketTypeCount() {
                // TODO Auto-generated method stub
                return 0;
        }
```

```
@Override
public void processEvent(TrafficGeneration tg) {
        // TODO Auto-generated method stub

}

//Evento inicial:
protected void processEvent(StartSimulation start) {

        //Se a identificacao do noh eh igual a 1:
        if (this.node.getId().asInt() == 1){

                //Cria o pacote com id do noh que vai receber
                //e o endereco do noh que estah enviando:
                Pacote pk2 = new Pacote(sender,NodeId.get(2));

                // seta o valor do contador para 2:
                pk2.setCont(2);

                // envia o pacote da camada de aplicacao:
                sendPacket(pk2);

                // cria uma tag desse evento:
                SimulationManager.logNodeState(this.node.getId(), "Primeiro
                    -envio", "int", String.valueOf(10));
        }
}

// Processa um pacote que vem de uma camada de baixo:
@Override
public void lowerSAP(Packet packet){
        //Se o pacote recebido eh do tipo Pacote:
        if (packet instanceof Pacote){
                // Pega o pacote que ele recebeu;
                // Faz um cast;
                Pacote pac = (Pacote)packet;

                // cria uma tag que chegou o pacote:
                SimulationManager.logNodeState(this.node.getId(), "Chegou
                    pacote", "int", String.valueOf(pac.getCont()));

                // processa um evento daqui um tempo, cria um wakeup call
                    para ser iniciado daqui um tempo de 1000:
                AppWuc wuc = new AppWuc(sender, 1000);

                // se o contador do pacote for menor/igual que 30:
                if(pac.getCont() <= 30){

                        //seta o contador do wakeup call pro contador do
                            pacote que ele recebeu:
                        wuc.setCont(pac.getCont());

                        //Envia o evento para ele mesmo, para processar
                            daqui um tempo de 1000:
                        sendEventSelf(wuc);
                }
        }
}

//Processa o WakeupCall dos nos-sensores:
public void processWakeUpCall(WakeUpCall wuc){

        // Verifica WakeupCall recebido eh do AppWuc:
        if (wuc instanceof AppWuc){
                // Pega o contador do WakeupCall e incrementa um:
                int aux = ((AppWuc) wuc).getCont()+1;
```

```
                        // Cria um novo pacote e manda para o proximo
90                      Pacote pk = new Pacote(sender,NodeId.get(aux));

                        // Configura o contador do pacote como o incremento do
                            contador do WakeupCall:
                        pk.setCont(aux);

95                      //Envia o pacote para o proximo:
                        sendPacket(pk);
                    }
                }

100
}
```

1.8.3 Visualizando uma simulação no GrubiX

Para que isso seja realizado, tem que ser feito o *download* do visualizador do GrubiX, o VisualGrubix, que está disponível em *http://pesquisa.dcc.ufla.br/grub/dowloads*. Feito isso, você deve executar o Jar do visualizador. A aparência do VisualGrubix está de acordo com a Figura 1.8.

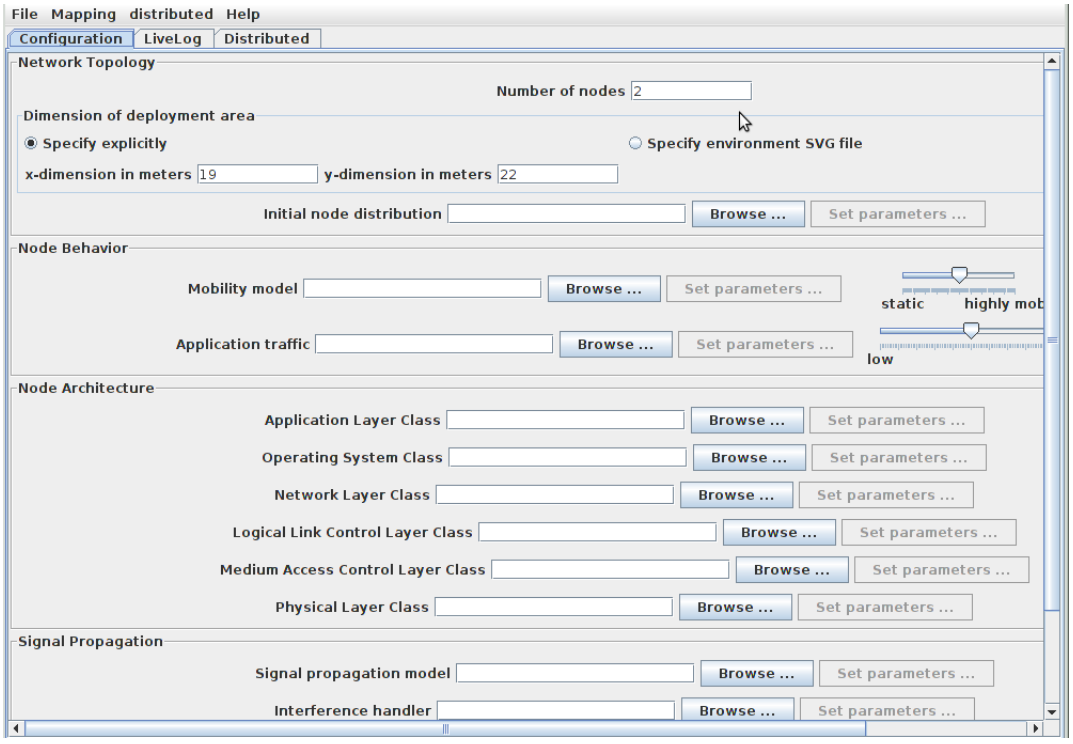

Figura 1.8: Visualizador do GrubiX.

Sua simulação será visualizada assim que o arquivo de Log for aberto. Vá em *File* $\longrightarrow$ *Open Log File* $\longrightarrow$ Selecione o diretório da sua aplicação $\longrightarrow$ Abra o arquivo *LogAppPingPongdefault.compact*, que foi definido no arquivo XML de configuração. Espere realizar o *Parsing* e quando esse acabar aparecerá uma nova aba, chamada *Monitor*. Clique na aba *Mapping* $\longrightarrow$ Clique em *Config Mapping...* $\longrightarrow$ Mapeie os eventos, como por exemplo, "Primeiro-Envio" como *Color* e "Chegou Pacote" como *Label*. Vá na aba *Monitor* e clique em *Start*. No final sua simulação terá a aparência da Figura 1.9.

O ponto verde indica o nó-sensor que iniciou os eventos e os números mostram para onde foi o pacote, salto por salto.

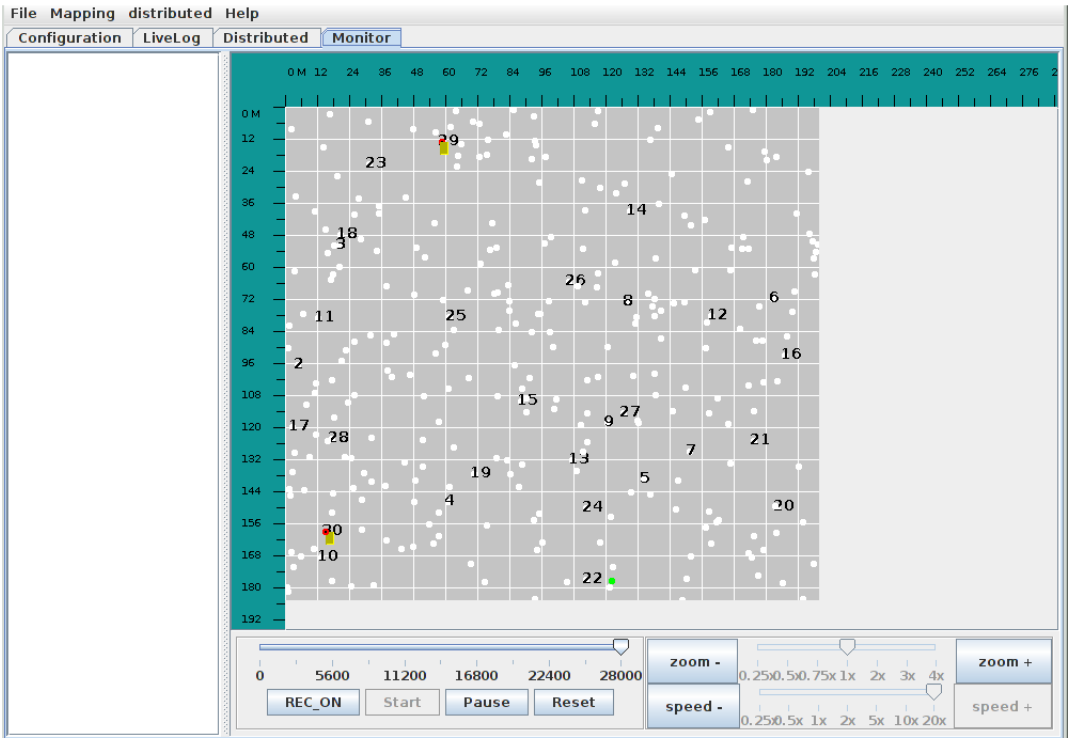

Figura 1.9: Resultado da simulação no simulador GrubiX.

1.8.4 Construindo aplicações com nós-sensores heterogêneos

Para construir aplicações com nós-sensores heterogêneos, ou seja, que apresentam características distintas, mas que possam se comunicar, é necessário tomar algumas precauções. A primeira é definir no arquivo de configuração em XML que a rede de sensores será caracterizada por nós heterogêneos. Isso é feito na tag *nodes*, criando-se mais instâncias da tag *node*. Isso pode ser visto no código em XML 1.12.

Listing 1.12: Caracterizando nós-sensores heterogêneos

```
    <nodes>
        <node name="SensorNode" count="30">
                <layers>
                        <operatingSystem>
5                               <class>br.ufla.dcc.grubix.simulator.node.user.os.
                                    NullOperatingSystemLayer</class>
                        </operatingSystem>
                        <logLink>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    LogLinkDebug</class>
                        </logLink>
10                      <energyManager>
                                <class>br.ufla.dcc.grubix.simulator.node.energy.
                                    BasicEnergyManager</class>
                        </energyManager>
                        <physical>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    PHY_802_11bg</class>
15                              <params>
                                <param name="standard">g</param>
                                <param name="preamble">short</param>
                                        <param name="channelCount">1</param>
                                        <param name="signalStrength">0.0</param>
```

```
                                        <param name="minSignalStrength">0.0</param>
                                        <param name="maxSignalStrength">100.0</
                                            param>
                                        <param name="minFrequency">2.4e9</param>
                                        <param name="maxFrequency">2.4e9</param>
                                </params>
                        </physical>
                        <mac>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    MAC_IEEE802_11bg_DCF</class>
                                <params>
                                        <param name="retryLimit">10</param>
                                        <param name="rateAdaption">AARF</param>
                                        <param name="raUpLevel">10</param>
                                        <param name="raDownLevel">2</param>
                                        <param name="raUpMult">2</param>
                                        <param name="raTimeout">10</param></params>
                        </mac>
                        <network>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    OptimalSourceRouting</class>
                        </network>
                        <application>
                                <class>br.ufla.dcc.MissionDissemination.SensorNode</class>
                        </application>
                </layers>
        </node>

        <node name="SinkNode" count="1">
                <layers>
                        <operatingSystem>
                                <class>br.ufla.dcc.grubix.simulator.node.user.os.
                                    NullOperatingSystemLayer</class>
                        </operatingSystem>
                        <logLink>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    LogLinkDebug</class>
                        </logLink>
                        <energyManager>
                                <class>br.ufla.dcc.grubix.simulator.node.energy.
                                    BasicEnergyManager</class>
                        </energyManager>
                        <physical>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    PHY_802_11bg</class>
                                <params>
                                <param name="standard">g</param>
                                <param name="preamble">short</param>
                                        <param name="channelCount">1</param>
                                        <param name="signalStrength">0.0</param>
                                        <param name="minSignalStrength">0.0</param>
                                        <param name="maxSignalStrength">100.0</
                                            param>
                                        <param name="minFrequency">2.4e9</param>
                                        <param name="maxFrequency">2.4e9</param>
                                </params>
                        </physical>
                        <mac>
                                <class>br.ufla.dcc.grubix.simulator.node.user.
                                    MAC_IEEE802_11bg_DCF</class>
                                <params>
                                        <param name="retryLimit">10</param>
                                        <param name="rateAdaption">AARF</param>
                                        <param name="raUpLevel">10</param>
                                        <param name="raDownLevel">2</param>
                                        <param name="raUpMult">2</param>
                                        <param name="raTimeout">10</param></params>
```

```
                    </mac>
                    <network>
80                      <class>br.ufla.dcc.grubix.simulator.node.user.
                            OptimalSourceRouting</class>
                    </network>
                    <application>
                        <class>br.ufla.dcc.MissionDissemination.SinkNode</class>
                    </application>
85              </layers>
        </node>
    </nodes>
```

Pode ser visto nessa listagem de código que existem dois nós, com características distintas (foque nas classes da camada de aplicação). O primeiro nó é um nó-sensor comum (SensorNode) e o segundo é o nó responsável pelo processamamento dos dados finais (SinkNode). Lembre-se, a listagem dos nós no XML deve ser feita alfabeticamente, potanto o nó de nome SinkNode não poderia vir antes do nó SensorNode.

A idéia que pode ser seguida para ter vários nós-sensores diferentes, considerando que eles vão estar na mesma camada é criando uma classe GenericNode que conterá informações básicas para todos os nós, como por exemplo, a herança da camada de aplicação. Isso pode ser visto na listagem 1.13

Listing 1.13: Nó-sensor genérico

```
package br.ufla.dcc.APP;

import br.ufla.dcc.grubix.simulator.LayerException;
import br.ufla.dcc.grubix.simulator.event.StartSimulation;
5 import br.ufla.dcc.grubix.simulator.event.TrafficGeneration;
import br.ufla.dcc.grubix.simulator.node.ApplicationLayer;
import br.ufla.dcc.grubix.simulator.event.WakeUpCall;
import br.ufla.dcc.grubix.simulator.event.Packet;

10 public class GenericNode extends ApplicationLayer {

        @Override
        public int getPacketTypeCount() {
                // TODO Auto-generated method stub
15              return 0;
        }

        @Override
        public void processEvent(TrafficGeneration tg) {
20              // TODO Auto-generated method stub

        }

        @Override
25      public void lowerSAP(Packet packet) throws LayerException {
                // TODO Auto-generated method stub

        }

30      protected void processEvent(StartSimulation start) {

        }

        public void processWakeUpCall(WakeUpCall wuc) {
35
        }

}
```

Assim, os demais nós, como os das listagens de códigos 1.14 e 1.15 podem estender esse nó-sensor genérico para serem modelados de acordo com as regras do simulador.

Listing 1.14: Nó-sensor genérico

```
package br.ufla.dcc.APP;

import br.ufla.dcc.grubix.simulator.LayerException;
import br.ufla.dcc.grubix.simulator.event.Initialize;
import br.ufla.dcc.grubix.simulator.event.Packet;
import br.ufla.dcc.grubix.simulator.event.StartSimulation;
import br.ufla.dcc.grubix.simulator.event.TrafficGeneration;
import br.ufla.dcc.grubix.simulator.event.WakeUpCall;
import br.ufla.dcc.grubix.simulator.node.ApplicationLayer;

public class SensorNode extends GenericNode {

        protected final void processEvent(Initialize init) {
        }


        protected void processEvent(StartSimulation start) {
        }

        @Override
        public void lowerSAP(Packet packet) throws LayerException {
        }



        @Override
        public void processWakeUpCall(WakeUpCall wuc) {
        }


        @Override
        public int getPacketTypeCount() {
                return 1;
        }

        @Override
        public void processEvent(TrafficGeneration tg) {
        }

}
```

Listing 1.15: Nó-sensor sink

```
package br.ufla.dcc.APP;

import br.ufla.dcc.grubix.simulator.LayerException;
import br.ufla.dcc.grubix.simulator.event.Initialize;
import br.ufla.dcc.grubix.simulator.event.Packet;
import br.ufla.dcc.grubix.simulator.event.StartSimulation;
import br.ufla.dcc.grubix.simulator.event.TrafficGeneration;
import br.ufla.dcc.grubix.simulator.event.WakeUpCall;
import br.ufla.dcc.grubix.simulator.node.ApplicationLayer;


public class SinkNode extends GenericNode {

        protected final void processEvent(Initialize init) {
        }

        protected void processEvent(StartSimulation start) {
        }

        @Override
        public void lowerSAP(Packet packet) throws LayerException {
        }
```

```
        @Override
25      public void processWakeUpCall(WakeUpCall wuc) {
        }


        @Override
30      public int getPacketTypeCount() {
                return 1;
        }

        @Override
35      public void processEvent(TrafficGeneration tg) {
        }
}
```

Cuidados devem ser tomados com relação a sobrescrição de método, porque isso pode impedir o nó-sensor modelado de funcionar no GrubiX.

Referências Bibliográficas

[Campos 2003]CAMPOS, C. A. V. *Uma Modelagem da Mobilidade Individual para Redes Móveis Ad hoc*. Dissertação (Mestrado) — Universidade Federal do Rio de Janeiro - COPPE/UFRJ, 2003.

[Lessmann, Heimfarth e Janacik 2008]LESSMANN, J.; HEIMFARTH, T.; JANACIK, P. Shox: An easy to use simulation platform for wireless networks. In: *UKSIM '08: Proceedings of the Tenth International Conference on Computer Modeling and Simulation*. Washington, DC, USA: IEEE Computer Society, 2008. p. 410–415. ISBN 978-0-7695-3114-4.

[Loureiro et al. 2003]LOUREIRO, A. A. F. et al. Redes de sensores sem fio. In: *Tutoriais do Simpósio Brasileiro de Redes de Computadores*. [S.l.: s.n.], 2003.

[Ruiz et al. 2004]RUIZ, L. B. et al. Arquiteturas para redes de sensores sem fio. In: *Tutoriais do Simpósio Brasileiro de Redes de Computadores*. [S.l.: s.n.], 2004.

[ShoX: A scalable ad hoc network simulator]SHOX: A scalable ad hoc network simulator. http://shox.sourceforge.net: [s.n.].