DANNIEL HUGO RIBEIRO

Orientador: Joubert de Castro Lima
Co-orientador: Ricardo Augusto Rabelo

JSENSOR: UMA PLATAFORMA PARALELA PARA SIMULAÇÃO DE REDES DE SENSORES

Ouro Preto
Dezembro de 2012

Universidade Federal de Ouro Preto
Instituto de Ciências Exatas
Bacharelado em Ciência da Computação

JSENSOR: UMA PLATAFORMA PARALELA PARA SIMULAÇÃO DE REDES DE SENSORES

Monografia apresentada ao Curso de Bacharelado em Ciência da Computação da Universidade Federal de Ouro Preto como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

DANNIEL HUGO RIBEIRO

Ouro Preto
Dezembro de 2012

UNIVERSIDADE FEDERAL DE OURO PRETO

FOLHA DE APROVAÇÃO

JSensor: Uma Plataforma Paralela para Simulação de Redes de Sensores

DANNIEL HUGO RIBEIRO

Monografia defendida e aprovada pela banca examinadora constituída por:

Dr. Joubert de Castro Lima – Orientador
Universidade Federal de Ouro Preto

Dr. Ricardo Augusto Rabelo – Co-orientador
Universidade Federal de Ouro Preto

Me. Reinaldo Silva Fortes
Universidade Federal de Ouro Preto

Ouro Preto, Dezembro de 2012

Resumo

Redes de sensores sem fio (RSSFs) com milhares de nós é um assunto recorrente nas discussões em termos de sua viabilidade em ambientes reais. Aplicações em redes veiculares, cidades inteligentes e celulares tornam tal discussão ainda mais presente. Pesquisadores da área utilizam simulações para avaliar o impacto de novos modelos de distribuição de nós, roteamento, comunicação ou redução de dados, nos mais variados tipos de Redes de Sensores. Tais aplicações têm demandado novas ferramentas de simulação que permitam e suportem o desenvolvimento de modelos de RSSF com milhares de nós. Este trabalho implementa o simulador de modelos de redes de sensores chamado JSensor. O **JSensor** é capaz de realizar simulações de forma paralela, específicamente para arquiteturas de computadores com memória compartilhada ou multicores. Na literatura temos vários simuladores, dentre eles [Lee et al. (2009)], [Niazi e Hussain (2009)], [Fernekess et al. (2009)] e [Gao et al. (2009)], e temos o **Tossim** [Lewis et al. (2002)] e o **OMNet++** [OMNet++ (2012)] com vasta gama de usuários, porém em todos temos alguma limitação grave, seja no número de nós suportados pelo simulador, seja na característica da simulação, seja no suporte ao paralelismo. O **JSensor** permite a simulação de redes de sensores síncronas e assíncronas com aproximadamente quinhentos mil nós que repassam mensagens via inundação. De uma forma geral, o **JSensor** se mostrou eficiente, com um *runtime* proximo ao linear e possuindo um *speedup* que se assemelha muito ao descrito por Gustafson para um nível de concorrência na aplicação de aproximadamente 90%.

Abstract

Wireless sensor networks (WSNs) with thousands of nodes is an important theme in terms of its feasibility in real environments. Intelligent cities and houses, as well as mobile applications, turn the theme massive simulations even more actual. Researchers adopt simulations to observe the impact of new models, including node distribution, routing, communication and data reduction, in many different sensor networks. Such applications require simulators designed to support WSNs models with huge amount of nodes. This work implements a WSN simulator named **JSensor**. **JSensor** is able to parallel simulate huge amount of nodes using shared memory or multicore computer architectures. In the literature, there are many simulators [Lee et al. (2009)], [Niazi e Hussain (2009)], [Fernekess et al. (2009)] and [Gao et al. (2009)], and there are also simulators with many users **Tossim** [Lewis et al. (2002)] e o **OMNet++** [OMNet++ (2012)], but unfortunately all of them has a severe limitation. In general, they do not support huge amount of nodes, parallelism and properties such as parallel mobility. **JSensor** supports synchronous or asynchronous simulations with approximately five hundred nodes sending messages using a flooding strategy. The experiments demonstrate that **JSensor** is efficient, with a linear runtime and a good speedup, achieving 90% of code parallelism using the Gustafson Law.

Dedico este trabalho a minha noiva Aline Ramos Sampaio, "a flor que perfuma meu pequeno planeta".

Agradecimentos

Agradeço primeiramente a Deus por guiar meus passos, sem ele não conseguiria chegar onde cheguei.
Agradeço aos meus pais José Atanásio Ribeiro e Lélia Rosa Ribeiro, por não medirem esforços em me ajudar a realizar esta conquista. À minha irmã Damaris que eu amo muito, pela amizade e companheirismo. À minha noiva Aline por sempre estar comigo, me dando força e coragem para seguir em frente. Ao Bruno, à Sabrina, Bebelle e Lívia pelos momentos de alegria que sempre me proporcionaram. À Vicente, Elza, Marina e Rodrigo por me acolherem como membro da família. Aos meus companheiros de DoTA, em especial ao Kayram dos Santos, Luiz Henrique, Fernando Filipe, Matheus Henrique e Thallys Borges, aos meus companheiros da GroubBy Sandra Daniela e Marcos Bastos e a toda galera da Computação. Ao Thiagão (in memorian). Aos professores do DECOM por todo o ensinamento. Em especial Joubert e Rabelo pela orientação e ao David Menotti por toda ajuda durante o curso. E também ao ALLA pela força durante e após sua estadia no DECOM.
Obrigado a todos que confiaram em mim e me apoiaram durante esta jornada.

Sumário

1 Introdução **1**
1.1 Definição do Problema 1
1.2 Objetivos 3
1.2.1 Objetivo Geral 3
1.2.2 Objetivos Específicos 3

2 Trabalhos Correlatos **4**
2.1 Redes de Sensores Sem Fio (RSSF) 4
2.2 Simuladores na literatura 5
2.3 Simuladores amplamente usados 6

3 Desenvolvimento **8**
3.1 JSensor 8
3.1.1 Arquitetura Geral 8
3.1.2 Arquitetura do núcleo 9
3.1.3 Simulações no **JSensor** 12
3.1.4 Interface Gráfica (GUI) 17
3.1.5 Projetos 19
3.2 Prototipação da Versão Distribuída 20
3.2.1 O `Client` 20
3.2.2 O `Master` 20
3.2.3 O `Slave` 21
3.2.4 O `Handler` 21
3.2.5 Troca de mensagens entre os módulos 23
3.2.6 Requisições feitas pelos Slaves 23

4 Avaliação de Desempenho **25**

5 Conclusões e Trabalhos Futuros **28**

Referências Bibliográficas **29**

Lista de Figuras

1.1 Comunicação inteligente entre os dispositivos de uma cadeia de conversão de energia 2

2.1 Exemplo de uma rede de sensores sem fio. 4

3.1 Arquitetura do **JSensor**. 9
3.2 Exemplo de uma *thread* em uma simulação assíncrona pararela 15
3.3 Interface de configuração do **JSensor** 18
3.4 Interface de simulação do **JSensor** 18
3.5 Pasta `projects` no **JSensor** 19
3.6 Interação entre os módulos da versão distribuída do **JSensor** 23

4.1 Runtime do **JSensor** 26
4.2 SpeedUp do **JSensor** 26

Capítulo 1

Introdução

1.1 Definição do Problema

O mundo ao nosso redor possui uma variedade de fenômenos que podem ser descritos por algumas grandezas, como temperatura, pressão e umidade. Estas grandezas podem ser monitorados por dispositivos com poder de sensoriamento, processamento e comunicação. O conjunto desses dispositivos é conhecido na literatura como Rede de Sensores [Akyildiz et al. (2002)]. Pesquisas nesta área utilizam simulações síncronas e assíncronas para avaliar o impacto de novos modelos de distribuição, roteamento, comunicação ou redução de dados nos mais variados tipos de redes. Esse assunto tem sido amplamente pesquisado nos dias atuais. Recentes avanços nesta área têm permitido que cada vez mais aplicações de grande porte sejam simuladas. Como exemplo temos cidades [Exame.com (2012)] e casas inteligentes [G1 (2012)] e redes veiculares [Alves et al. (2009)].

A Figura 1.1 ilustra a comunicação entre diversos dispositivos com o objetivo de se aproveitar energia em um modelo de cidade inteligente. Nesta direção, o desenvolvimento e validação de novos modelos de comunicação entre os dispositivos, distribuição dos nós sensores, roteamento, redução de dados, entre outros, necessitam de um simulador ou plataforma que suporte a movimentação, por exemplo , de centenas de milhares, senão milhões de sensores ou nós.

Há diversos simuladores disponíveis atualmente. Entre eles podemos destacar [Lee et al. (2009)], [Niazi e Hussain (2009)], [Fernekess et al. (2009)] e [Gao et al. (2009)] que são alguns dos simuladores apresentados na literatura. Temos ainda o [Sinalgo (2012)], [OMNet++ (2012)] e o [Lewis et al. (2002)] que são frequentemente utilizados no mercado e na academia. De uma forma geral, os simuladores encontrados não consideram aspectos importantes que devem ser levados em conta para realizar-se simulações em paralelo de tais redes. Eles não usam eficientemente o sistema de memória compartilhada e múltiplos cores, com isto não executam simulações que consideram mobilidade, cálculo de vizinhança e, principalmente, um elevado número de nós gerando muitos eventos simultaneamente.

Figura 1.1: Comunicação inteligente entre os dispositivos de uma cadeia de conversão de energia

Com base nas limitações das ferramentas existentes, foi desenvolvido o **JSensor**[1] que é uma plataforma para simulação de redes de sensores sem fio em larga escala em arquiteturas de computadores com memória compartilhada ou multicores. **JSensor** permite a execução de simulações síncronas e assíncronas, permitindo que o usuário configure eventos como fogo, tempestades, entre outros, durante a simulação, assim como a disposição inicial dos sensores na rede, seu mecanismo de comunicação, mobilidade e o cáculo da regra de vizinhança entre os sensores. De uma forma geral, o **JSensor** mostrou-se eficiente, com um *runtime* próximo ao linear e *speedup* de 95% numa máquina com 44GB de RAM e 8 cores físicos e 8 cores lógicos com a tecnologia *hyperthreading*. A curva de *speedup* do **JSensor** sugere um nível de concorrência de aproximadamente 90% do simulador. Estes resultados foram obtidos considerando um cenário de aproximadamente quinhentos mil nós enviando dados do tipo inundação e sem a consideração de mobilidade e consequentemente a utilização de alguma estratégia para o cálculo de vizinhos de um determinado sensor.

[1] `http://www.joubertlima.com.br/JSensor/`

1.2 Objetivos

1.2.1 Objetivo Geral

- Desenvolver um simulador para redes de sensores que possa executar simulações massivas de forma paralela em arquiteturas *multicore*, considerando nas simulações modelos de mobilidade, comunicação e distribuição de nós e fatos tais como incêndios, tempestades, enchentes, entre outros.

1.2.2 Objetivos Específicos

- Desenvolver estratégias que permitam executar simulações do tipo síncronas e assíncronas.
- Criar um módulo paralelo para simulações síncronas e assíncronas.
- Permitir a visualização dos experimentos em tempo real.
- Executar simulações em larga escala para validar a eficiência do simulador.
- Prototipar um módulo distribuído para simulações síncronas e assíncronas.

Capítulo 2

Trabalhos Correlatos

2.1 Redes de Sensores Sem Fio (RSSF)

Uma rede de sensores sem fio (RSSF) é um tipo especial de rede que tem a capacidade de monitorar e processar dados coletados do meio ambiente [Akyildiz et al. (2002)]. Basicamente, estas redes são compostas de dispositivos autônomos chamados nós sensores. Variáveis como temperatura e luminosidade podem ser monitoradas e então convertidas em dados que serão continuamente enviados pela rede até o sorvedouro. Nós sensores podem enviar dados ao observador ou sorvedouro tanto via protocolo de roteamento *multihop*, ou soluções baseadas em agrupamentos.

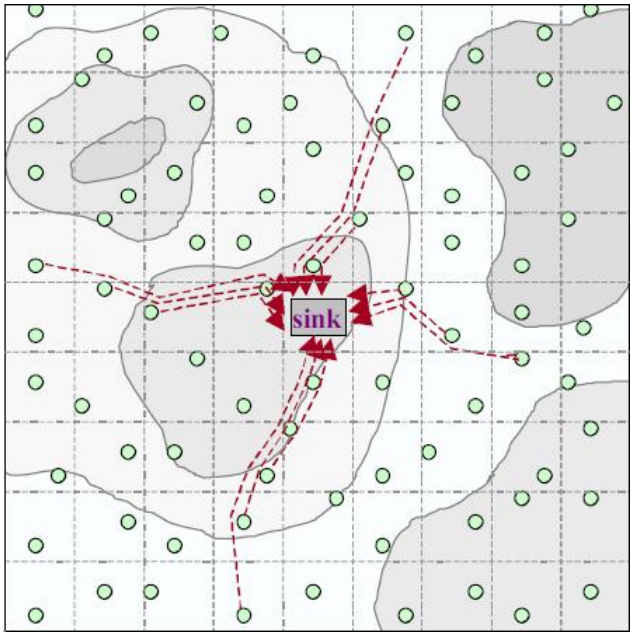

Figura 2.1: Exemplo de uma rede de sensores sem fio.

A Figura 2.1 mostra um exemplo de uma rede de sensores sem fio. Nela temos representa-

dos os nós (círculos) e estes estão escoando dados monitorados do ambiente via saltos ou *hops* (linhas pontilhadas) até o sorvedouro (*sink*).

Redes de sensores são comumente usadas também para monitorar a ocorrência de eventos específicos, como fogo, qualidade do ar, ou presença de inimigos militares. Uma tarefa importante para redes de sensores direcionadas a eventos é enviar eficientemente os dados ao sorvedouro [Nakamura et al. (2009)]. O caminho que a mensagem irá tomar obedece a regras de roteamento, amplamente estudado na literatura.

Há ainda aplicações recentes que mostram um grande interesse em casas [G1 (2012)] e cidades inteligentes [Exame.com (2012)]. Cidades inteligentes são cidades que utilizam de recursos tecnológicos para amenizar os problemas advindos do adensamento populacional. Estas cidades procuram criar ambientes sustentáveis, eficientes e com alto grau de conectividade, o que acarreta em um ganho nos níveis de qualidade de vida dos cidadãos.

Para se criar este tipo de solução, deve-se lançar mão de redes de sensores sem fio, que possibilitam o monitoramento de diversos fatores ligados à estas cidades, como por exemplo o monitoramento do trânsito, da qualidade da água e do ar, entre outros. Outra aplicação é a chamada casa inteligente que é uma residência onde procura-se integrar os componentes domésticos, como por exemplo televisores, aparelhos de som, iluminação da residência e até mesmo o celular. Assim, é possível detectar se o dono do imóvel entrou na residência e assim a casa escolhe uma das música previamente escolhida e a inicia, ou liga a televisão no canal preferido, por exemplo. Por último temos redes veiculares [Alves et al. (2009)]. Estas utilizam de sensores instalados nos próprios veículos para informarem aos motoristas as melhores rotas possíveis para seus trajetos, tentando evitar sempre engarrafamentos e trânsitos desnecessários.

Estes cenários são corriqueiros e frequentemente utilizam centenas de milhares, senão milhões de sensores para se obter os resultados desejados. Levando-se em consideração o custo de se instalar tais sensores, não é viável realizar testes com sensores reais. Neste caso a utilização de simuladores é de extrema importância no desenvolvimento de soluções para redes de sensores.

Os modelos de arquitetura de computadores *multicore* propiciam o uso do sistema de memória compartilhada e múltiplos *cores*, sendo possível executar simulações que geram muitos eventos simultâneos. O paralelismo deve, no entanto, ser abstraído do usuário final, que geralmente não é familiarizado com este tipo avançado de programação. Assim, o uso de *sockets*, *threads*, gerenciamento de grande volume de objetos e balanceamento de carga não devem ser delegados ao modelador e sim ao simulador.

2.2 Simuladores na literatura

Niazi e Hussain (2009) apresenta um simulador baseado na simulação de agentes em uma

rede sem fio. Esta abordagem é defendida para simulações com muitos nós na rede. O modelo de agentes permite uma maior abstração dos elementos da rede, porém os cenários testados atingiram apenas 1000 nós e sem considerar características da simulacao como mobilidade por exemplo. O simulador também não possui escalabilidade para arquiteturas *multicore*.

Gao et al. (2009) mostra um *framework* composto de vários simuladores, com diferentes modelos de redes sem fio, desde a configuração das interfaces de rede. O *framework* é composto por vários algoritmos já consolidados, porém a solução não simula muitos nós, pois é centralizada em detalhes mais abrangentes da rede, tais como as várias camadas da mesma.

Fernekess et al. (2009) introduz um simulador baseado em snapshots temporais da troca de pacotes em uma rede celular. Como parâmetro de entrada, as características da célula e o tráfego esperado e o simulador particiona a carga de acordo com os eventos gerados. O simulador pode ser configurado com as especificações da rede a ser emulada. A configuração de entrada permite um detalhamento das interfaces, contudo não possui uma implementação paralela.

No trabalho de Lee et al. (2009) é apresentado uma técnica para o processamento paralelo de simulações orientadas a eventos temporais, em redes com grande quantidade de nós. A técnica utilizada possui três princípios para o paralelismo: particionamento da carga, sincronização entre os múltiplos bancos de dados e a estrutura de dados usada no simulador. O particionamento é baseado na distribuição geográfica dos nós da rede. As simulações tiveram um ganho de desempenho proporcional até trinta e dois processadores. Contudo, os testes não atingiram a mesma quantidade de nós que os aqui apresentados.

2.3 Simuladores amplamente usados

O **Sinalgo** [Sinalgo (2012)] é um *framework* para teste e validação de algoritmos de redes. Ao contrário de outros simuladores disponíveis, o **Sinalgo** foca na verificação de novos algoritmos. Sendo assim, as simulações não implementam protocolos específicos das camadas de rede. Desenvolvido em **Java**, ele é também um software livre e de código aberto, sob a licença **BSD**. Suporta simulações do tipo síncrona e assíncrona e tem suporte à visualização em 2D e 3D. No entanto, o **Sinalgo** também não tem suporte a arquiteturas multiprocessadas e distribuídas, mas seus idealizadores garantem que simulações com até cem mil nós podem ser executadas em tempo hábil. O **JSensor** é fortemente baseado no **Sinalgo**, porém o estende em diversas características, tanto de simulação, como por exemplo a possibilidade de manipulação de eventos (os chamados Fatos no **JSensor**) como também suporte a arquiteturas *multicore*.

O **TOSSIM** [Lewis et al. (2002)] é um simulador para o **TinyOS** que é o sistema operacional utilizado na maioria dos sensores comerciais. Ele pode executar até milhares de nós, e compila diretamente do código **TinyOS**. Não existe uma versão paralela do **TOSSIM**,

limitando a simulação de cenários muito grandes.

O **OMNet++**[OMNet++ (2012)] é um *framework* orientado a objeto de simulação discreta, escrito totalmente na linguagem **C++**. Por conter uma arquitetura genérica, abrange vários domínios de problemas:

- Modelagem de redes de comunicação cabeada e não cabeada;
- Modelagem de protocolos;
- Modelagem de sistemas de *hardware* multi-processados e distribuídos;
- Validação de arquiteturas de *hardware*;
- Avaliação de aspectos de desempenho de sistemas complexos de *software*;
- Modelagem de qualquer sistema onde uma aproximação discreta é possível;

O **OMNet++** não disponibiliza suporte à simulações síncronas. Ele possui interface gráfica 2D. De uma forma geral, tanto o simulador quanto as interfaces são altamente portáveis, podendo ser executados no **Linux**, **Windows** ou **MAC**. Possui ainda uma extensão chamada **Horizon**[Horizon (2012)] que permite executar simulações em arquiteturas multiprocessadas e as extensões **Castalia** [Castalia (2012)] e **MiXiM** [MiXiM (2012)] que permitem simular redes de sensores sem fio no **OMNet++**. Infelizmente o **Castalia** executa sobre o kernel **Omnet++** e não sobre o **Horizon**, com isto propriedades como mobilidade não podem ser executadas em paralelo. O mesmo ocorre para o **Mixim**. Sendo assim, podemos afirmar que o **Omnet++** não permite mobilidade e paralelismo, por exemplo. Caso o usuário possua tais requisitos ele deve programar diretamente no **Horizon**.

Capítulo 3

Desenvolvimento

3.1 JSensor

3.1.1 Arquitetura Geral

O simulador **JSensor** foi desenvolvido como um *framework*, possibilitando que usuários possam construir seus modelos à partir de um conjunto de componentes de *software* básicos, tais como estruturas de dados, mecanismo de comunicação entre processos, balanceamento de carga, entre outros. O **JSensor** é fortemente baseado no simulador de algoritmos de redes **Sinalgo**. Adotamos muitas das estratégias de visualização, simulação e de modelagem presentes no **Sinalgo**. O **JSensor** é totalmente escrito em Java, o que o torna portável para qualquer plataforma. Além disso, o **JSensor** permite executar dois tipos de simulação: síncrona e assíncrona. Uma simulação do tipo síncrona é baseada em *rounds*. A cada *round* a plataforma move os nós de acordo com um determinado modelo de mobilidade. Depois, atualiza suas conexões com os outros nós da rede. Por fim, cada nó executa uma determinada ação, seja ela enviar mensagens, monitorar o ambiente ou entrar em modo de espera. Em contrapartida, uma simulação dita assíncrona, ou discreta, é puramente baseada em eventos. A plataforma possui um conjunto de eventos de envio/recepção de mensagens e de temporizadores, os quais são ordenados pelo tempo em que devem ocorrer. Repetidamente, o simulador retira o evento mais recente da lista e o executa.

O simulador **JSensor** é dividido em três camadas:

- O Núcleo: Considerada a principal camada do simulador, nele tem-se toda a lógica de simulação, incluindo conceitos como sensor, escalonador, vizinhança, temporizadores, entre outros. No núcleo também é definido os modelos de conectividade, mobilidade e distribuição da simulação, explicados em detalhes mais a frente nesta seção.

- A Interface Gráfica do Usuário (GUI): Responsável pela interação do usuário com os elementos do sistema a ser simulado. Na GUI é possível criar novos sensores, os remover,

controlar o fluxo de simulação, acompanhar os resultados da simulação e definir tipos de simulação (síncrona e assíncrona), entre outras funcionalidades.

- Os Projetos: Camada onde o usuário define seus próprios modelos de simulação. Um modelo deve estar associado a um projeto e um projeto pode ter vários modelos implementados. Na camada projetos o usuário pode especificar, por exemplo, os tipos de sensores, temporizadores ou mesmo algumas especificidades relacionadas à comunicação, mobilidade ou distribuição dos sensores numa rede. O **JSensor** possui exemplos de sensores, temporizadores, mecanismo de comunicação, mobilidade e distribuição, porém o usuário pode estendê-los e com isto aumentar o catálogo de opções ou estratégias de simulação **JSensor**.

As camadas do **JSensor** e suas interações são mostradas na Figura 3.1.

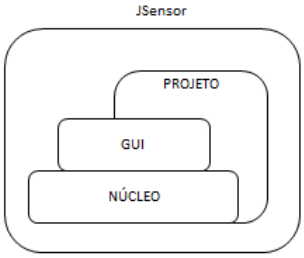

Figura 3.1: Arquitetura do **JSensor**.

3.1.2 Arquitetura do núcleo

O **JSensor** permite que simulações síncronas e assíncronas sejam executadas em paralelo. Para que ambas as simulações ocorram, definiu-se uma classe Runtime que possui os métodos que implementam simulações do tipo síncrona e assíncrona, encapsulando as particularidades de cada tipo.

Para implementar modelos síncronos ou assíncronos, o **JSensor** provê um conjunto de entidades, modelos e eventos que o usuário pode estender e integrar ao seu projeto.

3.1.2.1 Entidades

A primeira entidade a ser compreendida é a classe `Node`. Esta é a classe que representa os nós ou sensores da rede. Todo novo tipo de nó criado deve estender a classe `Node`. No **JSensor** já possui os seguintes tipos de nó:

1. `DummyNode`: Nó modelo, que não executa nenhuma operação específica;
2. `SensorNode`: Nó que detecta eventos e coleta dados sobre ele;

3. `FloodNode`: Nó que repassa mensagens via inundação;

4. `SinkNode`: Nó que recebe as mensagens coletadas.

Como padrão o simulador **JSensor** envia mensagem de duas formas: via inundação ou via unicast. A primeira opção envia uma mensagem a cada um de seus vizinhos enquanto a segunda envia a um nó específico. No unicast, é feita uma verificação antes do envio da mensagem para garantir que o nó destino está no raio de comunicação do nó que envia.

Um conjunto de métodos é disponibilizado para que o usuário possa implementar suas simulações. Durante a criação do nó, o método `onCreation` é chamado. As ações do `preStep` ocorrem antes de cada passo de uma simulação síncrona. O `postStep` define as ações que irão ocorrer depois de cada passo. A vizinhança será atualizada e o método `neighbourhoodChange` será executado. Por fim, as ações do método `handleMessages` são executadas. As configurações padrão servem para facilitar o desenvolvimento no **JSensor** e estão presentes nos tipos de nós citados acima.

Além da classe `Node`, o **JSensor** define a classe `Message` . Esta classe representa as mensagens que serão enviadas na rede. Na classe `Message` define-se também o tipo de mensagem e a forma como elas são criadas. O único método da classe Message é o clone . Este método deve retornar uma cópia idêntica da mensagem anterior, para evitar que vários nós alterem a mesma mensagem. Note que o usuário poderá descrever qualquer mensagem no **JSensor**, apenas herdando a classe `Message`.

A terceira entidade é a classe `Fact`. Essa classe é responsável por representar os fatos ocorridos no ambiente, como a variação da temperatura, umidade do ar, nível de erosão ou qualquer outro fenômeno natural que se deseje monitorar. Os fatos são tratados de diferentes formas dependendo do tipo de simulação. Na simulação síncrona, o *framework* itera sobre os fatos (também de forma paralela) atualizando seus valores e tempo de vida no início de cada *round*. Se o tempo de vida de um fato for atingido, este é removido automaticamente da simulação e não poderá mais ser detectado por nenhum nó. Na simulação assíncrona, a cada fato é associado um ou mais AbstractFactEvent que serão responsáveis por atualizar os valores dos fatos.

3.1.2.2 Modelos

O **JSensor** permite que o usuário possa facilmente especificar modelos, ou seja, como seus nós irão se mover pela rede, como os nós deverão ser posicionados inicialmente na rede e qualquer relação de vizinhança entre eles. Para especificar como os nós são dispostos inicialmente na área simulada, o modelo de distribuição requer que o usuário implemente apenas o método `getNextPosition()`, que retorna uma posição inicial para um dado nó. Como padrão já possuímos os seguintes tipos de modelos de distribuição no **JSensor**:

- `RandomDistribution`: Distribui aleatoriamente os nós na grade;

- `GridDistribution`: Distribui os nós seguindo o modelo de grade.

O modelo de conectividade descreve quando dois nós tem ou não uma conexão entre si. Mesmo que dois nós estejam perto um do outro devido ao seu raio de comunicação, o modelo de conectividade é que define se estes nós possuem ou não uma conexão. O **JSensor** possui o modelo de conectividade `NearConnected`, que se conecta a qualquer nó na rede, desde que raio de comunicação definido na simulação.

Por fim, cabe ao usuário definir ou não seu modelo de mobilidade, responsável por descrever como um nó irá se movimentar. Na simulação síncrona a movimentação ocorre a cada novo *round*. Ao início de cada *round* cada nó pode se mover para uma nova posição na área simulada seguindo este modelo. No modelo de mobilidade o usuário deve implementar apenas o método `getNextPosition`, que retorna a próxima posição que um determinado nó irá assumir no próximo *round*. Como padrão o seguinte modelo de mobilidade já está implementado no **JSensor**

- `RandomDirection` onde movimenta-se o nó para uma posição aleatória na grade.

3.1.2.3 Eventos

Os eventos são mais comumente utilizados durante a simulação assíncrona. Um evento é uma ação que pode ser executada pelos nós a fim de se produzir um conjunto de reações dos outros elementos da rede. Mais especificamente no **JSensor**, os eventos são classificados em quatro grandes categorias: os eventos Temporizadores, eventos de Envio, Movimentação e Atualização de Fatos. A classe `Timer` representa os temporizadores. Estes temporizadores despertam em determinados momentos especificados pelo usuário durante uma simulação. Após despertarem os temporizadores executam determinadas ações especificadas também pelo usuário. A classe `Timer` permite que a classe `Node` agende uma tarefa que será executada no futuro. No devido momento no tempo, o nó é escalonado para executá-la imediatamente. Todo temporizador é uma subclasse de `Timer`. Os eventos do tipo Envio são utilizados para enviar mensagens. No entanto, não é necessário que o usuário crie eventos do tipo Envio, pois o **JSensor** os cria toda vez que um nó envia mensagem a outro, seja via `unicast` ou inundação, disponíveis na classe `Node`. A classe que representa os eventos do tipo Envio é a chamada `SendingEvent`.

No modo assíncrono, a mobilidade dos nós é representada como um evento, assim como o `Timer` e o Envio. Cada nó pode movimentar-se individualmente dependo do evento de mobilidade associado a ele. A classe abstrata que representa este tipo de evento é a `AbstractMovimentEvent`. Cada evento deste tipo permite mobilidade para um determinado nó. A movimentação dos nós ocorre da seguinte forma:

1. Cria-se uma classe que estende de `AbstractMovimentEvent`.

2. O usuário define quando a movimentação do nó terá início, por quanto tempo o nó continuará a se movimentar e também o passo em que a movimentação vai ocorrer.

3. O *framework* cria tantos eventos quanto necessários para a movimentação do nó e os coloca na lista de eventos a serem tratados.

4. Quando é chegada a hora de serem executados os eventos de movimentação, o *framework* os retira da lista de eventos e modifica a posição do nó para a sua nova posição.

Por fim, temos os eventos do tipo Atualização de Fatos. Estes são responsáveis por atualizar o valor do fato associado a eles. Assim como o `Timer`, estes eventos acontecem somente em determinados momentos no tempo. Os eventos de Atualização de Fatos tem um tempo de início e término e são executados segundo um passo pré-definido. O fato independe do evento, ou seja, mesmo que o evento termine, ele ainda pode continuar existindo, dependendo apenas do seu próprio tempo de vida.

3.1.3 Simulações no JSensor

3.1.3.1 Particionamento dos nós

O particionamento dos nós na rede ocorre de duas maneiras no núcleo do **JSensor**: coleções e mapas. O particionamento dos nós é fundamental para garantir a escalabilidade ao **JSensor**.

1. Simulação baseada em Coleções
   Neste tipo de simulação, os nós são agrupados por meio de coleções. Cada *thread* implementa uma coleção de nós como uma lista encadeada. Cada lista é então percorrida por uma determinada *thread*. Desta forma, cada *thread* pode modificar apenas sua lista, o que exime cada lista de ser *thread safe*, simplificando a solução. A cada *round* as threads percorrem esta lista movendo, descobrindo vizinhos e executando o método Step de cada nó. Os dois primeiros passos citados anteriormente é que determinam o quão rápida uma simulação pode ser. Para mover os nós, o núcleo necessita percorrer toda a coleção de nós delegada a ele uma única vez. Sendo assim, ele executa $O(n/m)$ operações, onde n é a quantidade de nós existentes na rede e m é a quantidade de threads existentes. No entanto, a segunda operação de encontrar vizinho é um pouco mais cara computacionalmente. Para se localizar todos os vizinhos de um nó, cada *thread* deve varrer não só a sua lista de nós como também a lista das outras threads à procura de vizinhos. Esta tarefa consome $O(n^2)$ operações, o que torna esta etapa mais lenta quando a rede é composta por um número muito grande de nós. A vantagem desta estratégia é que nenhuma estrutura extra é necessária, reduzindo o consumo de memória.

2. Simulação baseada em mapas concorrentes
   Nesta implementação os nós são agrupados não só por meio de listas, mas também são

indexados segundo sua posição relativa. A cada nova *thread* é delegada uma lista de nós e um mapa global de coordenadas `x, y, z`, totalmente *thread safe*. Este mapa é definido da seguinte forma:

```
Map<String,Map<int,Node> >
```

Onde a chave do `Map` mais externo é uma `String` representando a posição relativa do nó e o `Map` mais interno possui uma chave de inteiros representando os *IDs* dos nós. A chave do tipo `String` é construída da seguinte forma:

Dado um nó N, pegamos sua posição `(x,y)` ou `(x, y, z)` na área simulada. Calculamos a posição relativa deste nó dividindo as coordenadas, por exemplo, `(x,y)` pelo raio de comunicação do nó. Suponha um nó na posição `(100,100)` simulada e raio de comunicação igual a 50. Neste cenário, a posição relativa é equivalente a `(100/50, 100/50)` ou `(2,2)`. O valor da chave do `Map` mais externo é uma `String` que representa as coordenadas da posição relativa separados por vírgula. Para o exemplo em questão, a chave do Map externo seria a String "2,2"no mapa de coordenadas 2D. Este mapeamento nos permite determinar os possíveis vizinhos de um nó varrendo apenas as nove posições relativas em torno da posição relativa do nó. Para um nó na posição `(x,y)` as posições onde os possíveis vizinhos podem estar localizados no Map são: `(x,y)`, `(x-1,y)`, `(x,y-1)`, `(x-1,y-1)`, `(x+1,y)`, `(x,y+1)`, `(x+1,y+1)`, `(x-1,y+1)` e `(x+1,y-1)`. Isso só é verdade porque cada nó pode se comunicar com outro que está a no máximo o raio de comunicação unidades de distância. Uma vez que cada coordenada pode estar associada a vários nós, implementamos ambos os mapas sendo *thread safe*. A imposição de uma seção crítica, mesmo que mínima, é a desvantagem desta solução. Em suma, tornamos sequencial a manipulação de uma coordenada por múltiplas threads e também tornamos sequencial a manipulação da coleção de coordenadas quando há necessidade de adição ou remoção de uma coordenada no espaço. Apesar de consumir mais espaço que a Simulação baseda em Coleção, a Simulação baseada em mapas concorrentes é muito mais rápida na tarefa de localização de vizinhos, garantindo um custo constante de $O(1)$ verificações (nove ao todo) contra $O(n)$ da `List-Based` para cada nó.

3.1.3.2 Simulação paralela

1. Simulação Síncrona Paralela
   Uma simulação do tipo síncrona é um tipo de simulação onde o fluxo de execução é baseado em rodadas (*rounds*). A cada rodada é realizada uma iteração sobre todos os nós da rede executando ações pré-definidas. Basicamente, o núcleo divide o número de nós da rede pelo numero de processos leves, também chamados *threads*, a serem criados numa simulação paralela. De uma forma geral, usamos a decomposição dos dados para

garantir uma simulação síncrona, paralela e escalável. O núcleo distribui todos os nós e fatos da rede entre n *threads* criadas, onde n é, por exemplo, o número de núcleos de processamento da máquina. Em seguida cada *thread* varre seu subconjunto de fatos atualizando seus valores e em seguida varre seu subconjunto de nós como descrito no Algoritmo 1.

Algoritmo 1 Algoritmo de Simulação Síncrona

Entrada: *_nodes* – Conjunto de todos os nós da rede, *_rounds* número de *rounds* a serem executados

1: **para** $j \leftarrow 0$ até *_rounds* **faça**
2: Incremente o tempo global em 1
3: **para todo** *node* em *_nodes* **faça**
4: Mova *node*. "De acordo com seu respectivo modelo de mobilidade"
5: Atualize conexões de *node*. "De acordo com seu modelo de conectividade"
6: Execute o método Step de *node*.
7: **fim para**
8: **fim para**

A cada *round* o algoritmo incrementa o tempo global (linha 2) e para cada nó pertencente à rede ele executa três etapas:

a) Move o nó de acordo com a implementação do usuário de um modelo de mobilidade (linha 4). Este modelo pode ser simples, como por exemplo, encontrar uma posição aleatória no terreno ou até mesmo algo mais complexo baseado em velocidade, tempo e distância.

b) Atualiza as conexões do nó (linha 5). Nesta etapa o nó descobre quem são seus vizinhos e atualiza suas conexões, respeitando as regras expressas no modelo de conectividade.

c) Executa o método `Step` do nó (linha 6). Este método executa uma sequência de ações pré-definidas e implementadas pelo usuário.

As ações que podem ser implementadas pelos usuários são:

- `Pré-Step` : Primeira ação a acontecer quando o método `Step` é invocado.
- `Neighborhood Change` : Acontece quando a vizinhança do nó sofreu alterações.
- `Handle Timers` : Executa os eventos de `Timer` associados ao nó.
- `Handle Messages` : Executa ações quando há mensagens na caixa de entrada do nó.
- `Post Step` : Ação que o corre ao final do método `Step`.

O Algoritmo 2 define como o núcleo particiona os nós e como aplica as execuções necessárias a cada nó da rede de sensores de forma paralela.

Algoritmo 2 Algoritmo do fluxo de execução síncrono paralelo

Entrada: *_nodes* – Conjunto de todos os nós da rede, *_n* quantidade de nós ,
_cores – Quantidade de núcleos ou de Threads a serem criadas, *_rounds* número de *rounds* a serem executados

```
1: para j ← 0 até _rounds faça
2:     Incremente o tempo global em 1
3:     para i ← 0 até _cores faça
4:         Crie Thread T[i]
5:         Delege à Thread T[i] o subconjunto S[i] ∈ _nodes de tamanho (_n/_cores)
6:     fim para
7:     para i ← 0 até _cores faça
8:         Movimente os nós de S[i] na thread T[i]
9:     fim para
10:    para i ← 0 até _cores faça
11:        Atualize conexões dos nós de S[i] na thread T[i]
12:    fim para
13:    para i ← 0 até _cores faça
14:        Execute o método Step dos nós de S[i] na thread T[i]
15:    fim para
16: fim para
```

Na linha 4 do Algoritmo 2 cria-se um conjunto de threads de tamanho especificado pelo usuário. É associado a cada nova *thread*, um subconjunto de nós da rede (linha 5). Cada *thread* executa então, os mesmos passos do Algoritmo 1 a seu conjunto de nós (linhas 7 a 15) sequencialmente.

2. Simulação Assíncrona Paralela

   A simulação do tipo assíncrona é puramente baseada em eventos. Tais eventos são ordenados pelo tempo em que devem ocorrer. Repetidamente o simulador retira o evento mais recente da lista e o executa. Em uma simulação do tipo assíncrona um nó reage sob quatro tipos de eventos: *Timer*, Envio, Movimentação e Atualização de Fatos. Em uma simulação assíncrona cada *thread* possui um conjunto de nós. Cada nó deste conjunto possui uma pilha de eventos a serem executados. Estes eventos são gerados à medida que o usuário os cria em sua simulação. Esta pilha de eventos é ordenada pelo tempo no qual o evento em questão deve ocorrer. Assim, eventos que possuem o menor tempo ficam no topo da pilha.

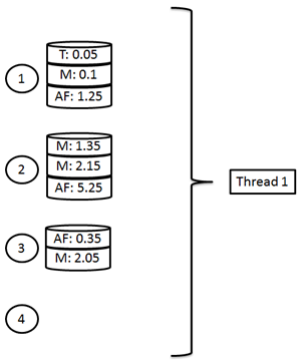

Figura 3.2: Exemplo de uma *thread* em uma simulação assíncrona pararela

A Figura 3.2 mostra um exemplo do funcionamento de uma *thread* em uma simulação assíncrona. Os círculos à esquerda representam os nós que estão sendo gerenciados pela *Thread* 1. O cilindro ao lado de cada nó representa a pilha de eventos do mesmo. Cada parte do cilindro representa um evento a ser realizado. As letras T, M, E e AF representam respectivamente eventos de Timer, Movimentação, Envio e Atualização de Fatos e os números em frente a elas representa o tempo em que o evento deve ocorrer. Na simulação assíncrona não utiliza-se o conceito de *rounds*. Neste caso, a unidade básica da simulação assíncrona no **JSensor** é uma unidade de tempo. A cada incremento de tempo global do simulador a *thread* retira os eventos da pilha de cada nó com o tempo de ocorrência igual e os executam. Assim, para cada incremento, o simulador executa os eventos contidos na faixa de tempo entre o momento presente e o incremento anterior e os distribuem entre as diversas threads. Para o exemplo da figura anterior, vamos supor que o simulador está no tempo 1. A *Thread* 1 vai percorrer o conjunto de nós e irá executar os eventos cujo tempo de ocorrência é menor que 1 e maior que 0. Para o exemplo em questão, serão executados os dois primeiros eventos do nó 1 e o primeiro evento do nó 3. Nos nós 2 e 4 nenhum evento será executado, dado que no nó 2 todos os eventos possuem tempo de ocorrência maior que 1 e no nó 4 não existe nenhum evento na pilha. O restante dos eventos nas pilhas será executado em incrementos posteriores.

O Algoritmo 3 mostra o fluxo de execução paralelo do modo assíncrono do simulador **JSensor**.

Algoritmo 3 Algoritmo do fluxo de execução assíncrono paralelo

Entrada: *_nodes* – Conjunto de todos os nós da rede,*_n* quantidade de nós ,*_cores* – Quantidade de núcleos ou de *Threads* a serem criadas,*_seconds* número de segundos a serem executados

```
1: para ti ← 0 até _seconds faça
2:     para i ← 0 até _cores faça
3:         Crie Thread T[i]
4:         Delege à Thread T[i] o subconjunto S[i] ∈ _nodes de tamanho (_n/_cores)
5:     fim para
6:     para i ← 0 até _cores faça
7:         Avalie caixa de entrada dos nós de S[i] na thread T[i]
8:         se Caixa de entrada não está vazia então
9:             Manipule as mensagens recebidas do nó
10:        fim se
11:    fim para
12:    para i ← 0 até _cores faça
13:        Execute o método Step(ti) dos nós de S[i] na thread T[i]
14:    fim para
15:    Incremente o tempo global em 1
16: fim para
```

De forma semelhante ao Algoritmo 2, cria-se um conjunto de *threads* e a elas são associados subconjuntos de nós (linhas 2 a 5). Cada *thread* avalia a caixa de entrada de seus respectivos nós e manipula as mensagens recebidas através do método `handleMessages` se necessário (linhas 6 a 11). Por fim, o método `step` é invocado, tendo como parâmetro `ti` que é o tempo corrente (linhas 12 a 14). O Algoritmo 4 representa como os eventos

são tratados. Na linha 1 é retirado um evento da pilha de eventos do nó. Somente são retirados eventos cujo tempo de ocorrência é menor ou igual ao tempo máximo. Descobre-se o tipo do evento, e então ele é executado (linhas 2 a 8).

Algoritmo 4 Algoritmo do método Step na execução assíncrona

Entrada: *_node* – Nó avaliado, *_ti* – Tempo de ocorrência máximo dos eventos
1: Retire um evento E de *_node* tal que o tempo de ocorrência $T(E)$ é menor ou igual a *_ti*
2: **se** E é um evento tipo Timer **então**
3: Execute E como tipo Timer
4: **senão se** E é um evento tipo Movimento **então**
5: Execute E como tipo Movimento
6: **senão se** E é um evento tipo Atualização de Fato **então**
7: Execute E como tipo Atualização de Fato
8: **senão**
9: Execute E como tipo Envio
10: **fim se**

Como dito anteriormente, na versão atual o **JSensor** realiza incrementos de uma unidade no tempo. Acontece que em algumas ocasiões, no tempo avaliado, não há nenhum evento para ser executado, fazendo com que o **JSensor** execute diversos incrementos desnecessários. Uma possível melhoria para futuras versões do **JSensor** seria marcar, através de uma variável global que pudesse ser vista e modificada por todas as threads, o evento de menor tempo. Assim, os incrementos dependeriam desta variável global e não de incrementos unitários.

3.1.4 Interface Gráfica (GUI)

A interface gráfica do **JSensor** é subdividida em duas. A primeira é a interface de configuração enquanto a segunda é a interface onde a simulação é representada de fato. A Figura 3.3 mostra a interface de configuração.

A Interface de configuração oferece ao usuário a possibilidade de configuração dos seguintes parâmetros:

- Projeto:
  Nesta configuração é possível definir qual projeto será executado. O **JSensor** automaticamente lista todos os projetos contidos na pasta `Project`. Na Figura 3.3, esta configuração está representada em `Project Name`.

- Área:
  Permite definir a largura e a altura da área simulada. Na Figura 3.3 é representada em `Area Setup`.

- Tipo de Simulação:
  Permite escolher entre Simulação do tipo Síncrona e Assíncrona. É representado em `Project Configuration`.

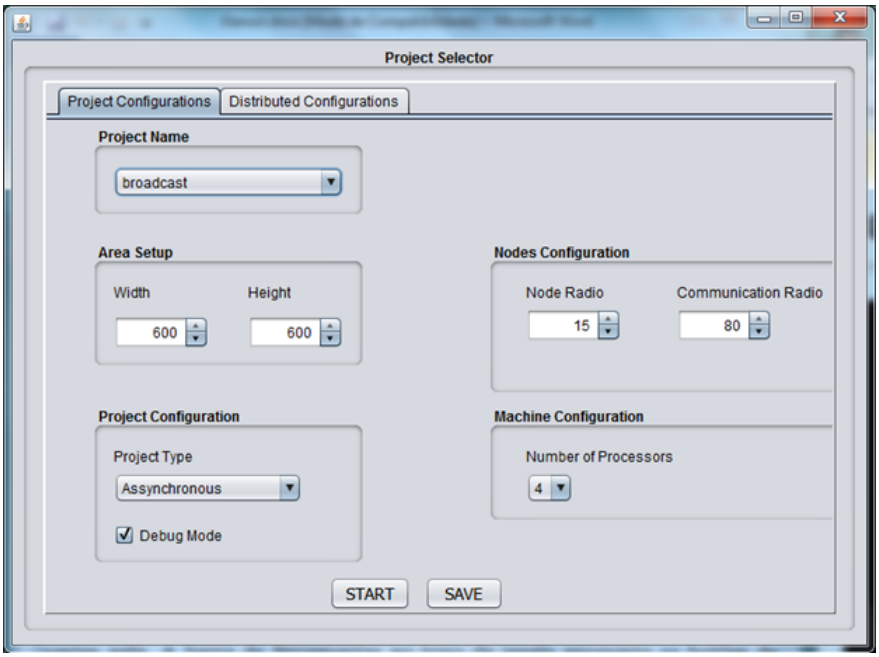

Figura 3.3: Interface de configuração do **JSensor**

- Configuração de nós: Permite definir o raio de comunicação e o raio do nó. Na Figura 3.3 está sendo representado em `Node Configuration`.
- Configuração da máquina:
  Define quantos processadores serão utilizados. Representado em `Machine Configuration`.

A próxima interface é a de simulação.

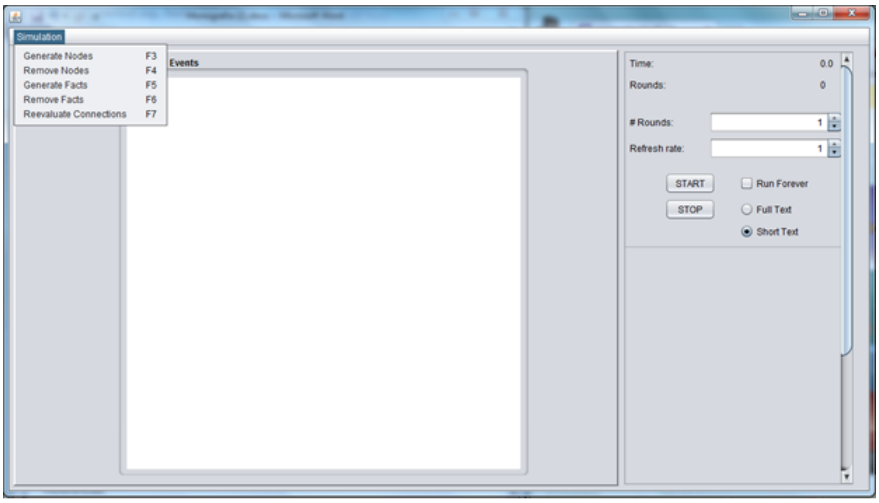

Figura 3.4: Interface de simulação do **JSensor**

A Figura 3.4 mostra a interface de simulação do **JSensor**. No menu superior temos a opção de controlar a criação e remoção dos nós e fatos. No canto direito temos as opções de controle de simulação. Podemos por exemplo iniciar e parar a simulação, controlar o número de *rounds*/tempo que serão executados, taxa de atualização da tela, definir se a simulação vai executar indefinidamente e se a interface vai mostrar apenas a informação do número do *round*/tempo corrente ou exibir todos os eventos que estão sendo executados. No centro da tela temos a área onde os eventos serão mostrados. Diferentemente das versões anteriores onde eram mostrados os nós, nesta versão do **JSensor** são mostrados apenas informações textuais do *round*/tempo corrente (opção *Short Text*) ou informações do *round*/tempo somadas à informações sobre os eventos que estão sendo executado no momento (opção *Full Text*). Faz parte dos trabalhos futuros simulações 2D e 3D paralelas.

3.1.5 Projetos

A criação de novos projetos no **JSensor** é simples. É necessário apenas copiar uma cópia da pasta defaultProject situada na pasta `projects`, e renomeá-la. A pasta `projects` é mostrada na Figura 3.5.

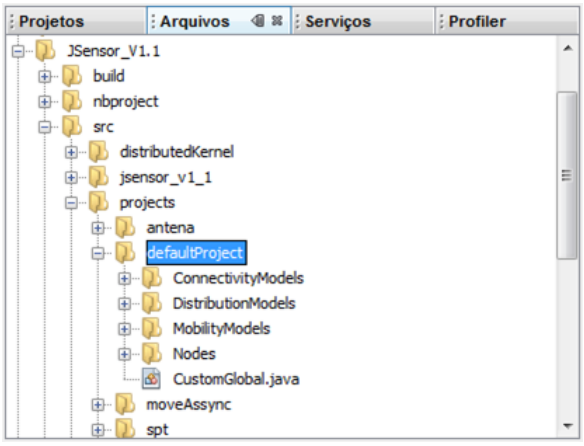

Figura 3.5: Pasta `projects` no **JSensor**

O próximo passo é implementar os tipos de nó, modelos de conectividade, distribuição e mobilidade. Também, pode-se implementar novos modelos de Mensagem e de Fatos. Ainda, é possível que o usuário defina a classe `CustomGlobal`, que permitem que ele controle ações gerais da simulação. Se o usuário optar por realizar a simulação do tipo síncrona, ele pode implementar modelos de conectividade, distribuição e mobilidade. Caso contrário, ele pode optar por implementar os modelos de conectividade e distribuição, mas não deve implementar os modelos de mobilidade, uma vez que a mobilidade do modo assíncrono se dá por meio de eventos. Ainda na simulação assíncrona, pode-se considerar implementar *Timers*, eventos de mobilidade e de atualização de fatos, colocando-os na pasta `Nodes.Timers`. O **JSensor** dispõe

diversos projetos por padrão, os quais o usuário pode abrir e estudar em caso de dúvidas. Mais informações sobre instalação, criação e projetos e casos de estudo podem ser encontradas na página do **JSensor**.

3.2 Prototipação da Versão Distribuída

Nesta seção apresentamos um protótipo da versão com memória distribuída que vem sendo desenvolvida no **JSensor**. Para o tal adotamos uma arquitetura híbrida que se comporta como cliente-servidor para alguns serviços e ponto-a-ponto para outros. Na atual arquitetura o **JSensor** possui quatro grandes componentes disponíveis: O `Client`, o `Master`, o `Slave` e o `Handler`. Cada um destes componentes é detalhado nas seções que se seguem.

3.2.1 O `Client`

O Cliente (cliente) é o componente responsável por realizar requisições advindas do usuário final. Mais especificamente no caso do **JSensor**, o cliente é responsável por enviar pedidos ao Servidor para que este execute simulações. O fluxo de execução de uma simulação utilizando memória distribuída se inicia quando o `Client` envia ao Servidor um pedido de simulação. A partir daí, o `Client` aguarda o término do seu pedido e é notificado a cada mudança de estado da simulação.

3.2.2 O `Master`

O `Master` é responsável por atender à requisições do `Client` e repassá-las ao `Slave`. Cabe ao `Master` fazer o mapeamento de um `Client` à um conjunto específico de `Slaves`. É ele quem controla quais `Slaves` estão associados à determinado `Cliente` e garante que o número de `Slaves` requisitados não ultrapasse a quantidade máxima disponível. Para permitir conexões de clientes, o `Master` é implementado como uma *Thread*. Esta permanece ativa recebendo conexões vindas dos clientes e tratando-as devidamente. Na versão atual, apenas um `Client` é atendido por vez. Isto significa que quando há dois ou mais clientes a serem atendidos, o segundo deve esperar que o primeiro cliente seja atendido e só então ele poderá realizar suas requisições ao `Master`. Além das conexões com o `Client`, o `Master` permite que `Slaves` se conectem a ele para que possam ser utilizados durante a simulação. Os `Slaves` se conectam através de uma *thread* à parte chamada `Slave Recruiter`. Esta tarefa permanece ativa enquanto o `Master` permanecer ativo. O `Master` também é responsável por gerenciar o conjunto `Client-Slaves`. Quando a atribuição de `Slaves` é feita com sucesso, o Master cria uma *thread* `Handler` para manipular as requisições do cliente e transformá-las em tarefas para os `Slaves`. No `Handler` está contido o mapa da área simulada, o mapa dos nós que cada `Slave` irá cuidar e o conjunto de posições dos nós. O chaveamento de `Client` para um conjunto de Slaves ocorre seguindo o conceito de **Produtor-Consumidor**. Este se dá da seguinte forma:

- Os `Slaves` se conectam ao `Slave Recruiter` e geram um `Socket`.
- Este *socket* é adicionado a uma lista chamada `Recurso`.
- O cliente se conecta ao `Master` utilizando um `Socket`.
- O `Master` retira `Slaves` do recurso e os atribui ao `Client`, desde que a quantidade de `Slaves` requisitados seja menor ou igual à quantidade disponível.
- O `Master` cria um `Handler` para gerenciar a comunicação do `Client` com os `Slaves` e para gerenciar a simulação como um todo.
- Os `Slaves` podem se conectar ao `Slave Recruiter` a qualquer momento durante este processo.

Esta estratégia foi adotada para permitir que mais tarde novos `Masters` sejam adicionados ao processo sem que seja necessário realizar grandes alterações.

3.2.3 O `Slave`

O `Slave` é o componente responsável por realizar de fato as simulações. Ao serem atribuídos a um `Client` um conjunto de `Slaves` deverá receber a requisição do cliente, trata-la e convertê-la em uma simulação utilizando diversas máquinas. O conjunto destas máquinas comunicando e cooperando entre si é o que faz com que a simulação ocorra de forma distribuída. Como citado anteriormente, o `Slave` se recruta a um Master e aguarda até que ele seja atribuído a um cliente. Quando isso ocorre, o cliente envia ao `Master` os parâmetros necessários para a execução da simulação. O `Master` avalia estes parâmetros e atribui um conjunto de nós a cada `Slave`, dependendo da quantidade de nós requisitada e do total de `Slaves` disponíveis para aquele `Client`. Ao receber os nós, o `Slave` inicia a criação dos mesmos. Uma vez criados, tem-se início a simulação. Durante a simulação é possível que o um Slave necessite de dados localizados em outro `Slave`. Neste caso, o `Handler` é notificado e este o informa a localização exata de onde a informação se encontra. A partir daí o `Slave` estabelece uma comunicação com o `Slave` que possui a informação.

3.2.4 O `Handler`

O `Handler` é responsável por gerenciar o conjunto `Client-Slaves`. Ele é criado no momento em que a atribuição de Slaves ao `Client` é feita com sucesso. O `Handler` mantém o controle sobre algumas variáveis da simulação. São elas:

- O mapa da área simulada (mapa concorrente): Na versão atual o mapa foi centralizado no `Handler`. Ele detém o controle sobre a área total da simulação. Nele estão contidos os nós (seguindo o conceito de mapas concorrentes citado na Seção 3.1.3.1) e onde estão

localizados. A decisão de centralização do mapa no `Handler` e não nos `Slaves` foi adotada por ser mais fácil de ser implementada. Versões futuras do **JSensor** tendem a migrar o controle da área simulada para os próprios Slaves.

- Conjunto de posições iniciais dos nós: O `Handler` mantém o conjunto das posições iniciais dos nós, pois dependendo da implementação do usuário pode ser necessário ter o conhecimento das últimas posições geradas para o cálculo de uma nova. Desta forma, para evitar que sejam feitas muitas comunicações entre os `Slaves`, centralizou-se a geração de posições no `Handler`.

- O Conjunto `Slave-Sensores`: O `Handler` detém também o controle de qual `Slave` está cuidando de certo conjunto de nós. Este controle é necessário, pois muitas vezes a plataforma necessita saber em qual `Slave` está localizado determinado nó, seja para enviar mensagens ou para tomar decisões de controle.

Além disso, o `Handler` controla a simulação. Este processo se dá da seguinte forma:

1. Preparação dos Slaves: o `Handler` notifica os `Slaves` sobre quais são os parâmetros utilizados na simulação.

2. Criação do conjunto de nós: Neste momento, os `Slaves` recebem uma notificação do `Handler` ordenando a criação do seu conjunto de nós. Cada `Slave` por sua vez, notifica o `Handler` quando o processo foi terminado.

3. Início da simulação: Uma vez que todos os nós foram criados, o `Handler` dá início à simulação. Ele inicia o incremento do tempo global da simulação e a cada nova iteração os `Slaves` são notificados do início de um novo *rounds*/segundo. Assim, no `Segundo 1`, todos os slaves executam os passos de simulação e ao fim avisam ao `Handler`. Quando todos terminaram e estão no mesmo passo, o `Handler` dá início a um novo segundo. Este processo se repete até que a quantidade de *rounds*/segundos requeridos pelo usuário tenham terminado.

A Figura 3.6 resume a interação dos módulos da versão distribuída do **JSensor**. Inicialmente, um conjunto de `Slaves` se candidata através do `Slave Recruiter`. Este por sua vez povoa o Resource com os Sockets de comunicação recém-criados. O `Client` envia uma requisição ao `Master` pedindo por uma simulação. O `Master` por separa um conjunto de Slaves, retirando-os do `Recurso` e os atribui ao `Client`. A seguir, o `Master` cria um `Handler` que irá gerenciar a comunicação `Client-Slave` e `Slave-Slave`, bem como a simulação como um todo. Durante este processo, os Slaves podem estabelecer comunicação entre si para troca de informações.

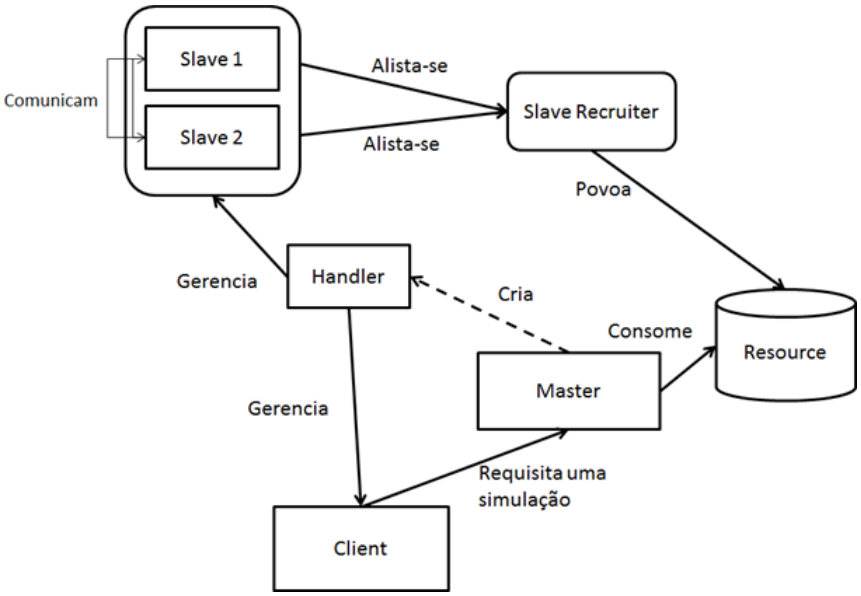

Figura 3.6: Interação entre os módulos da versão distribuída do **JSensor**

3.2.5 Troca de mensagens entre os módulos

Para que os módulos possam se comunicar foi adotado um padrão de comunicação. Este padrão é definido pela classe `Message`. Esta classe não possui relação alguma com a classe `Message` citada na versão de memória compartilhada do **JSensor**. Esta Mensagem possui três atributos principais:

- `serialVersionID`: Variável para auxiliar o **Java** no processo de Serialização da Mensagem que irá trafegar na rede.
- `messageType`: Tipo da mensagem que será enviada.
- `Message`: Objeto contendo a mensagem a ser enviada.

3.2.6 Requisições feitas pelos Slaves

É comum os `Slaves` necessitarem de informações de um nó que não está na sua lista. Neste caso os `Slaves` enviam uma requisição ao `Handler` que a processa e envia uma resposta contendo o dado requisitado ou uma notificação de sucesso ou falha. Estas requisições podem ser dos seguintes tipos:

- Vizinhança: Ocorre quando um nó do Slave necessita saber quem são seus vizinhos para continuar o seu processo. Neste caso o `Slave` bloqueia a sua simulação e envia uma requisição ao `Handler`. Por ser o único a possuir a informação do mapa da rede, é necessário que o `Slave` faça essa requisição ao `Handler`. Ao receber este pedido, o

`Handler` consulta o seu mapa e envia ao `Slave` que requisitou a informação sobre o conjunto de nós que são possíveis vizinhos do nó em questão. O `Slave` por sua vez processa a resposta e continua a simulação do ponto onde parou.

- Mover nós: Ocorre quando um nó troca de posição na grade. Neste caso é necessário enviar ao `Handler` uma notificação da mudança de posição para que este possa atualizar o mapa da rede.

- Envio de Mensagem via *Unicast* ou *Flooding*: Ocorre quando um nó precisa enviar uma mensagem a um vizinho (ou a um conjunto de vizinhos) que não está sendo gerenciado pelo mesmo `Slave` que ele. Neste caso o `Slave` envia uma requisição ao `Handler` que a recebe e envia a mensagem ao seu destinatário. O processo de envio de mensagens não está totalmente terminado no protótipo atual do **JSensor**.

Capítulo 4

Avaliação de Desempenho

Neste capítulo avaliamos o **JSensor** considerando simulações assíncronas. Utilizamos até quinhentos mil nós. Destes, escolhemos aleatóriamente mil nós que vão gerar mensagens a serem entregues à outros nós escolhidos aleatóriamente na rede. Estas mensagens são passadas via inundação. Quando um nó recebe a ele a envia a todos os seus vizinhos. Assim, para cada nó que participa do percurso da mensagem, é gerado um total de novas mensagens igual ao número de vizinhos que este nó possui. As mensagens são enviadas via *flooading* em intervalos aleatórios. A simulação é limitada a dez mil segundos. Em cada segundo todos os sensores são ativados independente deles estarem participando da troca de mensagens.

A estratégia de inundação habilita cada nó a receber, computar e enviar pelo menos uma mensagem após alguns rounds de simulação. Além disso, permite a inserção de dados redundantes, uma vez que muito nós irão receber e processar a mesma mensagem. Este problema, chamado de *storage placement* em RSSFs, tem sido estudado usando simuladores como o **JSensor** desde seu primeiro trabalho Sheng et al. (2006). O envio de dados do tipo inundação também permite a análise de cenários extremos, onde 100% dos nós precisam participar da simulação.

Os experimentos foram executados em duas máquinas com as seguintes configurações: (i) dois processadores **Intel Xeon E5405 quad-core** com 2 GHz; 16 GB de *RAM* compartilhada; quatro drives *SATAII* com 7200 *rpm*; e sistema operacional **Windows Server 2003 R2 64 -bit**. (ii) 16 *cores* de 3 GHz: 8 físicos e 8 lógicos com 12 MB de cache em cada core; 48 GB de *RAM DDR3* 1.03 GHz; dois discos de 500 GB, 10k rpm e um buffer de 64 MB; e sistema operacional **Linux Ubuntu 64-bit versão 11.10**. Todos os algoritmos foram codificados com o **java 64-bit (JRE 7.0 update 9)**.

As Figuras 4.1 e 4.2 mostram o desempenho do **JSensor** em cada máquina. Os resultados com a máquina com 16-cores são ilustrados pela linha escura do gráfico e os resultados da máquina com 8-*cores* são ilustrados pela linha cinza. Os resultados apresentam a média de dez execuções. Durante cada simulação observamos a troca de cerca de dez milhões de mensagens.

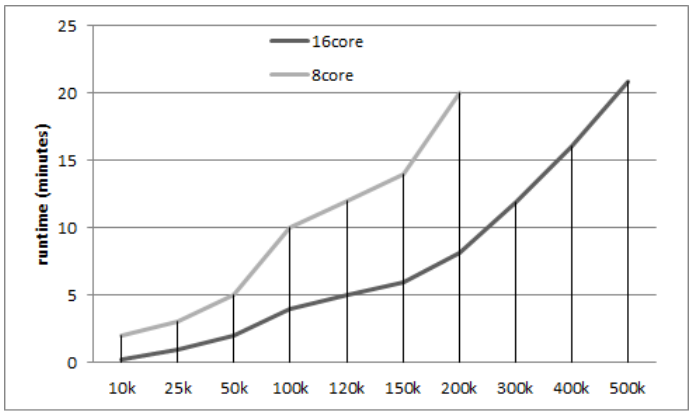

Figura 4.1: Runtime do **JSensor**

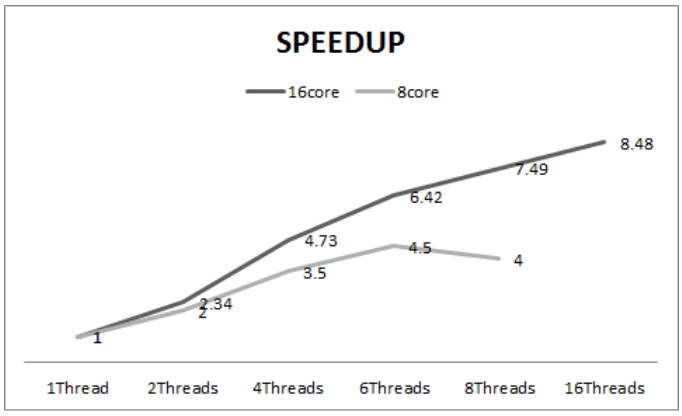

Figura 4.2: SpeedUp do **JSensor**

Figura 4.1 ilustra o runtime do **JSensor**. Na máquina 8-*core*, fixamos o número de *threads* em 8 e variamos o número de nós em 10-200 mil. Na máquina 16-*core*, fixamos o número de *threads* em 16 e variamos o número de nós em 10-500 mil. A curva que representa a máquina 8-*core* mostra um *runtime* quase linear, ou seja, a entrada sendo incrementada de 10 para 200 (20x) e o runtime de 1 para 20 minutos (igualmente 20x). Com dezesseis *threads* na máquina 16-*core*, os resultados tornam-se melhores, uma vez que a entrada dos dados incrementa em 50x e o *runtime* incrementa apenas de 20x.

Simulamos dez mil segundos com meio milhão de nós gerando tráfego do tipo inundação em aproximadamente 21 minutos, utilizando uma máquina com 8 *cores* físicos e 8 lógicos, como observado na Figura 4.1. Nessa simulação foi consumido 40-44 GB *RAM*. Vale salientar que esse tipo de cenários, com milhares de nós, pode facilmente ser encontrado se considerarmos aplicações com sensores em celulares, veículos, sistemas de transporte entre outros.

A Figura 4.2 mostra o *speedup* do **JSensor**. Na máquina 8-*core*, fixamos o número de nós em 200 mil e variamos o número de *threads* em 1-8. Na máquina 16-*core*, fixamos o número de nós em 500 mil e variamos o número de *threads* em 1-16. A curva de *speedup* para a

máquina 8-*core* é muito próxima a 90% de paralelização de código. O máximo *speedup* é 4.5, obtido quando utilizamos 6 *threads*. Ocorre um decremento do *speedup* (4.5 para 4) quando 8 *threads* são utilizadas. Este comportamento pode ser justificado pela disputa que ocorre pelo barramento da memória compartilhada quando mais *threads* são utilizadas. O *speedup* é um pouco melhor na máquina 16-*core*, alcançando um paralelismo de 95% considerando meio milhão de nós. Este comportamento é explicado pela lei de Gustafson e Albuquerque (1988). Em resumo, a lei de Gustafson define que uma porcentagem sequencial do algoritmo é ajustável e não fixa como supõe Amdahl. A consequência é que quando a entrada aumenta o speedup do **JSensor** também aumenta, como apresentado pela Figura 4.2.

JSensor alcança um *speedup* super-linear quando utilizamos 2, 4 e 6 threads. O paralelismo implícito é a principal justificativa para esse comportamento. **JSensor** utiliza 2, 4 e 6 *threads*, mas existem 16 cores para executar cada instrução, portanto se existe uma baixa dependência entre as instruções, as técnicas de *pipeline* e multi-escalar podem produzir um *speedup* super linear.

Finalmente, os *cores* lógicos foram fundamentais para obtermos um *speedup* de 7.49 usando 8 *threads*, porém existe uma clara saturação quando utilizamos 16 *threads*. O *speedup* foi 13% melhor, porém o número de *threads* duplicaram de 8 para 16 . Isto acontece devido à contenção no barramento do sistema de memória da máquina. O paralelismo implícito não incrementou o tempo de execução, uma vez que não existiam *cores* ociosos

Capítulo 5

Conclusões e Trabalhos Futuros

Neste trabalho de conclusão de curso apresentamos um simulador de RSSFs em larga escala chamado **JSensor**. Mostramos técnicas de paralelização de simulações do tipo síncrona e assíncrona. Além disso, mostramos o processo de criação de novos modelos, sejam eles para avaliar comunicação, distribuição dos nós ou movimentação. O reuso destes modelos pode ajudar a produzir rapidamente aplicações reais. Ao testarmos o simulador com uma rede de aproximadamente 200 mil nós mostramos que o *runtime* se aproxima muito do ideal e o *speedup* nos mostrou uma paralelização do código de cerca de 90%. Ao simularmos meio milhão de nós, em aproximadamente 21 minutos, obtivemos 95% de paralelismo. Vale salientar que algumas áreas, como telefonia móvel, sistemas de transporte e redes veiculares, demandam a utilização de milhares de sensores.

Para trabalhos futuros, pretendemos terminar o desenvolvimento da versão com memória distribuída do **JSensor**, o que nos permitirá executar uma simulação em diversas máquinas e utilizando diversos núcleos. Além disso, temos que testar o **JSensor** com modelos mais complexos onde mobilidade, fatos e todas as demais capacidades do **JSensor** estão presentes. Testes com modelos já utilizados com o **Omnet++**, por exemplo, são fundamentais. Por fim, testes comparativos com o **Sinalgo** para avaliarmos o ganho que o paralelismo trouxe ao **JSensor** que tem sua solução muito próxima ao **Sinalgo** em termos conceituais.

Referências Bibliográficas

Akyildiz, I. F.; Su, W.; Sankarasubramaniam, Y. e Cayirci, E. (2002). A survey on sensor networks. *IEEE Communications Magazine*, 40(8):102–114.

Alves, R. S.; Campbell, I. V.; Couto, R. S.; Campista, M. E. M.; Moraes, I. M.; Rubinstein, M. G.; Costa, L. H. M. K.; Duarte, O. C. M. B. e Abdalla, M. (2009). Redes veiculares: Princípios, aplicações e desafios. *Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pp. 199–254.

Castalia (2012). Castalia - a simulator for wsns. `http://castalia.npc.nicta.com.au/index.php`. Visitado em Dezembro de 2012.

Exame.com (2012). Nascem as cidades inteligentes. `http://exame.abril.com.br/revista-exame/edicoes/0988/noticias/aonde-o-futuro-ja-chegou`. visitado em Dezembro de 2012.

Fernekess, A.; Klein, A.; Wegmann, B. e Dietrich, K. (2009). Modular system-level simulator concept for ofdma systems. *IEEE Communications Magazine*, 47(3):150–156.

G1 (2012). 'casas inteligentes' atraem cada vez mais clientes no mercado capixaba. `http://g1.globo.com/espirito-santo/mercado-imobiliario/noticia/2011/09/casas-inteligentes-atraem-cada-vez-mais-clientes-no-mercado-capixaba.html`. visitado em Dezembro de 2012.

Gao, Y.; Zhang, X.; Yang, D. e Jiang, Y. (2009). Unified simulation evaluation for mobile broadband technologies. *Communications Magazine, IEEE*, 47(3):142 –149.

Gustafson, J. L. e Albuquerque, N. M. (1988). Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533.

Horizon (2012). Horizon parallel network simulator. `http://www.comsys.rwth-aachen.de/research/projects/horizon/`. Visitado em Dezembro de 2012.

Lee, H.; Manshadi, V. e Cox, D. (2009). High-fidelity and time-driven simulation of large wireless networks with parallel processing. *IEEE Communications Magazine*, 47(3):158–165.

Lewis, P.; Lee, N.; ; Welsh, M. e Culler, D. (2002). Tossim: Accurate and sacalable simulation of entire tinyos applications. In *5th Symposium on Operating Systems Design and Implementation*.

MiXiM (2012). Mixim project. `http://mixim.sourceforge.net/`. Visitado em Dezembro de 2012.

Nakamura, E. F.; Ramos, H. S.; Villas, L. A.; de Oliveira, H. A. B. F.; de Aquino, A. L. L. e Loureiro, A. A. F. (2009). A reactive role assignment for data routing in event-based wireless sensor networks. *Comput. Netw.*, 53:1980–1996.

Niazi, M. e Hussain, A. (2009). Agent-based tools for modeling and simulation of self-organization in peer-to-peer, ad hoc, and other complex networks. *Communications Magazine, IEEE*, 47(3):166 –173.

OMNet++ (2012). Omnet++. `http://www.omnetpp.org/`. visitado em Dezembro de 2012.

Sheng, B.; Li, Q. e Mao, W. (2006). Data storage placement in sensor networks. In *7th ACM International symposium on Mobile ad hoc networking and computing*, pp. 344–355.

Sinalgo (2012). Sinalgo - simulator for network algorithms. `http://www.disco.ethz.ch/projects/sinalgo/`. Visitado em Dezembro de 2012.