Simulador de Rede de Sensores Sem Fio Utilizando Sinalgo

Disciplina de Sistemas Distribuídos
Renan Vicentin Fabrão
Vinícius Aguiar Moraes
Universidade Tecnológica Federal do Paraná

12 de Dezembro de 2014

Abstract. *As RSSFs são redes que monitoram um ambiente através de dispositivos autônomos (nós-sensores) que coletam dados e cooperam entre si. Aplicações nesse tipo de rede impõem requisitos específicos relacionados ao consumo de energia e confiabilidade de entrega. Protocolos de roteamento para as RSSF devem possuir características de auto-configuração para descobrir qual a melhor rota para transmitir, com garantia de entrega e com consumo mínimo de energia, a informação entre os nós que compõem a rede. Este trabalho propõe implementar uma RSSF, observando o consumo de energia dos nós sensores, bem como o quantitativo de mensagens transmitidas, energia consumida e área de cobertura. O simulador utilizado para testar os experimentos deste trabalho foi o Sinalgo. No decorrer deste documento descrevemos os testes e resultados obtidos com o auxílio do Sinalgo. Os resultados apontaram um ganho crescente na autonomia da rede em cenários com maior quantidade de Sinks.*

Conteúdo

Capítulo 1

Introdução

Redes de Sensores Sem Fio (RSSFs) são redes com grande número de micro-sensores compactos com capacidade de transmissão de dados sem fio, chamados de nós sensores, estes dados são enviados para um ou mais *Sinks* dispostos pela rede [Ruiz et al. 2004]. Uma RSSF tem o potencial para um grande numero de aplicações, que variam desde coletar dados do meio-ambiente até aplicações militares.

Dentre os principais requisito da rede estão o consumo de energia e a área de cobertura, que é influenciado pelo consumo também, sendo estes requisitos responsáveis por afetar todas as fases de seu ciclo de vida independente da aplicação. A comunicação em RSSF consome mais energia do que o processamento e o sensoriamento realizado pelos nós da rede. Esta característica requer protocolos de roteamento que possibilite que os nós sensores se comuniquem de forma eficiente e eficaz com o mínimo de consumo de energia [RUIZ et al.]. Da mesma forma, é necessário o controle da área de cobertura, devido a impossibilidade de utilização de alguns sensores devido ao consumo de energia e a comum necessidade de utilização de baterias neste tipo de equipamento, gerando falhas na comunicação entre os sensores.

O objetivo deste trabalho foi desenvolver uma simulação baseado no *framework* SinalGo, onde foi simulada uma RSSF para monitorar um campo mil por mil *1000x1000* no qual foram distribuídos trezentos sensores, que foram responsáveis pela captura da temperatura e da umidade relativa do ar. Neste contexto, cada sensor detectava qual nó *Sink* se encontrava mais próximo, e assim então, enviando os dados de forma á obter menor consumo de bateria.

Neste documento, realizamos um estudo comparativo baseado na simulação desenvolvida, onde analisamos a eficiência energética, a quantidade de ciclos (turnos), a taxa de cobertura e taxa de recebimento de mensagens que a RSSF variando a quantidade de *Sinks* instalados.

Capítulo 2

Conceitos

2.1. Redes Ad Hoc

Rede Ad Hoc, como o próprio nome indica, é uma rede que nasce espontaneamente, sem intervenção técnica, ou até mesmo intencional, daqueles que dela usufruem ou beneficiam. Ao contrário do que ocorre em redes convencionais, não existe uma estrutura planeada e previamente definida. Neste tipo de rede os dispositivos, através de protocolos, são capazes de permutar informações diretamente entre si, formando uma rede de comunicação. Predominantemente, utilizam o ar como meio físico privilegiado para o envio e recepção de dados entre os pontos de comunicação que fazem parte da rede num dado lapso de tempo [Monteiro 2006].

A principal característica dessas redes é a ausência de infra-estrutura, como pontos de acesso ou estações-base, existentes em outras redes locais sem fio ou ainda nas redes de telefonia celular. A comunicação entre nós que estão fora do alcance de transmissão do rádio é feita em múltiplos saltos através da colaboração de nós intermediários. Além disso, a topologia da rede pode mudar dinamicamente devido á mobilidade dos nós. [Fernandes et al. 2006]

2.1.1. Redes de Sensores Sem fio

Rede de sensores sem fio (RSSF) pode ser descrita como uma rede que possui grande quantidade de nós-sensores com a capacidade de se comunicar. Esses nós podem ser colocados dentro do fenômeno a ser analisado ou próximo a ele, diferentemente das redes de sensores tradicionais. Normalmente as posições de cada nó não são predeterminadas ou pré-calculadas, são aleatórias, visto que muitas vezes a implantação de redes de sensores em locais de difícil acesso pode ocorrer pelo uso de helicóptero, apenas ”soltando”os nós sobre a região a ser analisada. [Johnson and Margalho 2006]

Os nós-sensores são tipicamente alimentados por baterias com comunicação e funções de computação limitadas. Cada nó pode ser equipado com uma variedade de modalidades de sensoriamento tais como acústico, sísmico, e infra-vermelho. RSSFs podem operar por períodos de tempo variando de semanas a anos de forma autônoma. Isso depende fundamentalmente da quantidade de energia disponível para cada sensor na rede. Em muitas aplicações nós sensores podem não estar facilmente acessíveis por causa da localização onde são empregados ou da escala da rede. Em ambos os casos, a manutenção da rede para reabastecimento de energia se torna impraticável. Mais ainda, caso seja necessário

substituir a bateria de um sensor frequentemente, as principais vantagens de uma RSSF seriam perdidas. [Loureiro et al. 2003]

Generalizando, uma rede de sensores sem fio é composta por vários nós-sensores pequenos e um ou mais nós-Sink, cada nó-sensor é equipado com um transmissor-receptor de rádio, um microprocessador e uma série de sensores. Estes nós são capazes de formar uma rede através da qual as leituras do sensor pode ser propagada. Cada nó tem uma capacidade de processamento autônomo, os dados colhidos pelos sensores são enviados para o nó-Sink que encontra-se mais próximo. O Sink é o elemento que recebe as mensagens de dados enviadas pelos nós-sensores e comunica-se com outras redes ou com um ou mais observadores. [RUIZ et al.]

2.1.2. Exemplos

As RSSF levam óbvias vantagens sobre as redes com fio, pois eliminam altos custos com cabeamento e podem ser implantadas em locais de difícil acesso, podendo, como já dito, ser implantadas através de um helicóptero, talvez, sobre a área a ser analisada. Por exemplo, pode-se interconectar sensores para fazer o monitoramento e controle das condições ambientais numa floresta, oceano ou até mesmo nas extremidades de um vulcão. Como é exposto de uma forma genérica na Figura 2.1.

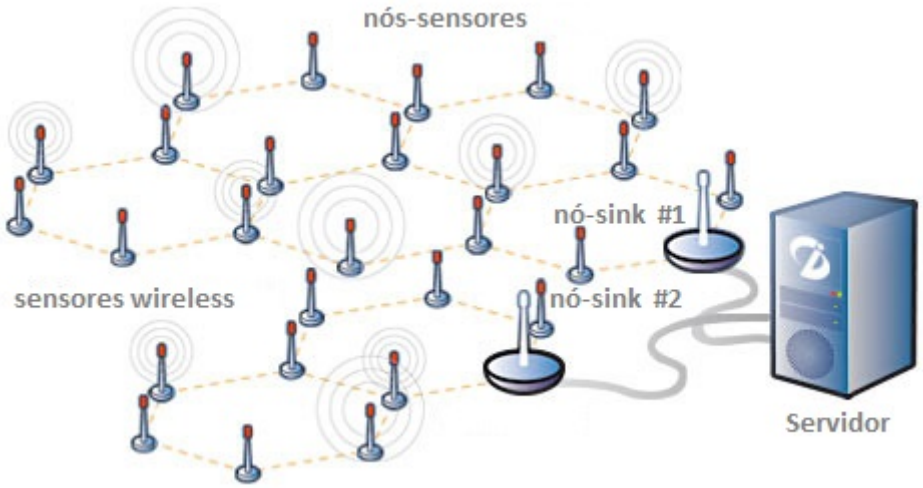


Figura 2.1. Exemplo genérico de uma RSSF

[1]

Neste exemplo é exibido uma estrutura comum de uma RSSF, onde é estabelecida uma rota entre os nós sensores que os ligam até os nós-Sinks, os quais são responsáveis por transmitir todos os dados coletados pelos nós para um servidor central. Como dito anteriormente, este conceito pode ser abstraído de um simples monitoramento de plantação de uvas, para até mesmo o monitoramento de temperaturas das proximidades de um vulcão.

Um outro conceito de aplicação de redes de sensores muito difundido hoje me dia é a aplicação *Healthcare*, onde são instalados sensores no corpo humano, normalmente de forma não evasiva como a *wearable*.

[2]

[1]Fonte: `http://www.isisingenieria.com/br/products.html`
[2]Fonte: `http://www.infotech.oulu.fi/Annual/2007/opme.html`

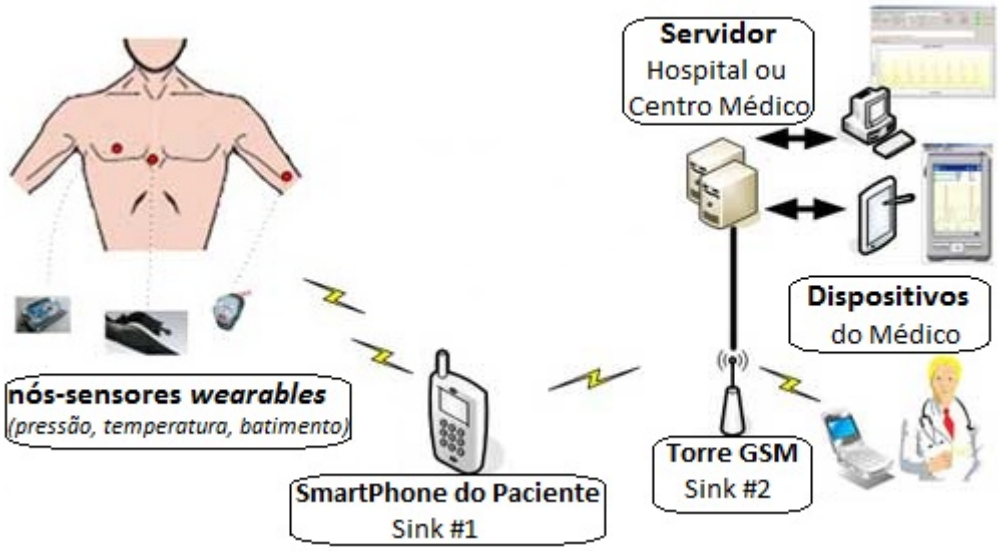


Figura 2.2. Exemplo de RSSF para uso médico

Neste exemplo os nós-sensores são instalados envolto ao corpo humano, em funcionamento, são responsáveis por coletar dados como pressão arterial, temperatura e batimentos cardíaco do paciente. Os dados coletados são transmitidos para o *smartphone* do paciente, neste contexto, representado como Sink 1. O *smartphone* envia o sinal até a torre GSM (Sink 2), a qual é responsável por enviar as informações dos sensores até o Servidor de dados. Neste servidor são tratadas as informações coletadas pelos sensores e posteriormente são acessadas pelos médicos responsáveis.

Além dos exemplos citados acima, hoje em dia já existem muitos projetos reais que fazem o uso de RSSF, como por exemplo, a York International, gigante do segmento de sistemas de ventilação, está instalando vários nós-sensores nos condicionadores de ar vendidos a seus clientes, facilitando a monitoração das temperaturas, que deverão ser enviadas aos escritórios da York, aliviando o trabalho dos técnicos e aumentando a produtividade. [Arampatzis et al. 2005]

A Intel já experimenta sistemas baseados nesses sensores em centros de saúde para ajudar pacientes com problemas de lapsos de memória, lembrando-lhes a hora de comer ou de beber água. [int]

2.1.3. Simuladores

Pela definição, ”a simulação é um processo de projetar um modelo computacional de um sistema real e conduzir experimentos com este modelo com o propósito de entender seu comportamento e/ou avaliar estratégias para sua operação [Pegden et al. 1995]. Desta maneira, podemos entender a simulação como um processo amplo que engloba não apenas a construção do modelo, mas todo o método experimental que se segue.

Testar fisicamente projetos de RSSFs quase sempre se torna algo inviável, adquirir centenas de dispositivos, gerenciar seu software e configurações, encontrar área física para realização dos experimentos e isolá-los de interferências são algumas das dificuldades que tornam esta tarefa custosa. O processo nesta área depende fundamentalmente da capacidade das ferramentas de simulação e mais especificamente da escalabilidade dos

simuladores RSSFs. [Loureiro et al. 2003]

Atualmente existem diversos simuladores de RSSF *open source* dentre os mais conhecidos podemos citar o *Network Simulator*, o OMNeT+ e o SinalGo. O desenvolvimento deste estudo foi realizado sob o *framework* SinalGo, o qual abordaremos suas características de forma mais abrangente no Capítulo 3.

2.2. Qualidade de Serviço

Qualidade de Serviço, ou simplesmente QoS (*Quality of Service*), pode ser conceituada como a qualidade que deve ser apresentada pelos serviços oferecidos pelo sistema, que é especificada pelo usuário destes serviços na forma de requisitos de qualidade no momento em que o serviço é requisitado. [Chao and Guo 2002]

Esse conceito, inicialmente focado na rede, evoluiu para uma noção mais ampla, contemplando as múltiplas camadas da interação usuário-sistema, onde pode ser implementado para definir a qualidade de qualquer serviço fornecido pelo sistema. Processamento de instruções e transmissão de dados são exemplos de serviços oferecidos pelo sistema, enquanto tempo máximo de processamento e máximo erro de transmissão são respectivamente exemplos de requisitos de QoS aplicados a estes serviços. Também podemos especificar restrições temporais relativas às características de confiabilidade e de segurança de uma aplicação, como o máximo tempo de falha (downtime), ou o método usado para autenticar o usuário de um serviço. [Siqueira 2002]

2.2.1. Qualidade de Serviço em Redes de Sensores sem Fio

Existe uma ampla variedade para o conceito de QoS em redes de sensores [Perillo 2003] [Iyer and Kleinrock 2003]. QoS em uma RSSF também é definida em termos da resolução espacial da rede. Segundo essa definição, um grande número de sensores que devem estar ativos na rede, dependendo do estímulo ambiental presente a cada momento.

Qualidade de serviço também se refere à fornecer confiabilidade de dados para a aplicação ao mesmo tempo em que os recursos de energia da rede são consumidos de modo eficiente [Iyer and Kleinrock 2003]. Alguns especialistas consideram que a confiabilidade dos dados está ligada ao número de sensores ativos na rede. Ambas as definições de QoS citadas estão bastante relacionadas à cobertura de sensoriamento, considerada em inúmeros trabalhos como o principal requisito de QoS em RSSFs. Outros trabalhos consideram a latência como um requisito crucial para várias aplicações. Outros ainda argumentam que, para alguns tipos de aplicações, o tempo de vida da rede é um requisito crítico.[Delicato 2005]

Capítulo 3

Simulador Sinalgo

Sinalgo é um *framework open source* desenvolvido em JAVA, que tem como finalidade realizar testes e validações de algoritmos de rede. Diferentemente de outros simuladores que gastam boa parte do tempo simulando diferentes camadas da pilha de rede (vide Apêndice A), Sinalgo foca nos algoritmos de rede e abstrai as camadas mais baixas. Apesar de ter sido desenvolvido para simulações de redes sem-fio, não é limitado apenas a estas.

O fato de ser prototipado em JAVA, ao invés da linguagem específica de hardware, não só é muito mais rápido e mais fácil, mas também simplifica a depuração do projeto, além de disponibilizar um amplo conjunto de condições da rede, sob o qual você pode-se testar inúmeros algoritmos.

Neste cenário é possivel simular algoritmos de redes sem fio em duas ou três dimensões. Quanto ao tipo de simulação, pode-se realizar simulações síncronas, onde todos os eventos têm seus inícios sincronizados com um *clock*, similar ao *clock* de um processador de computador, por exemplo; ou assíncronas, onde os eventos dependem uns dos outros para seus inícios e realizações.

3.1. Estrutura

O Sinalgo inclui 6 projetos de exemplos que podem ser utilizados para testar suas funcionalidades. Além disso, o código-fonte desses projetos, assim como os presentes no *framework* em si, são bem documentados, permitindo assim, serem utilizados como fonte de consulta para futuras implementações.

A implementação de um projeto utilizando o *framework* é realizada implementando diversas classes abstratas que realizam conceitos fundamentais em uma rede sem fio como modelos de conectividade, distribuição na área, interferência, transmissão de mensagens, deslocamento de nós, assim como os conceitos de nós, conexões entre estes, mensagens ou pacotes, além do timer ou temporizador para execução das ações. Porém, existem diversas classes já implementadas que podem ser utilizadas para estes fins, devendo apenas implementar as que são necessárias para conceitos específicos necessários para o problema em questão.

Analisando os exemplos disponíveis no *framework*, foi possível determinar quais seriam as classes fundamentais para a implementação deste projeto. Estas são: `Node`, `Message` e `Timer`. Ambas são classes abstratas, portanto, cria-se extensões para elas a

fim de assumirem um comportamento desejado nas simulações.

Além das três principais classes Abstratas, outras duas classes de suma importância para desenvolvimento de simulações são a classe `Edge` e a classe `InterferenceModel`.

Existem classes importantes também na estrutura do Sinalgo, como as classes de modelo de conectividade, modelo de mobilidade, modelo de transmissão de mensagens, que não serão abordadas neste documento, pois será utilizadas classes já implementadas nos pacotes projectDeafault.

3.1.1. Node

A classe `Node` representa qualquer entidade de rede, que no escopo deste trabalho, é representado pelos nós. Nós são qualquer elemento em uma rede, capaz de transmitir e/ou receber mensagens ou pacotes. Por ser uma classe abstrata se faz necessário a implementação de métodos abstratos contidos nesta classe, os quais são:

- *handleMessages:* método chamado após todas as mensagem serem recebidas pelos nós, este método é responsável por tratar e reencaminhar estas mensagens.
- *preStep:* este método é chamado antes da execução de cada round.
- *init:* método chamado uma única vez, logo após a inicialização do nó.
- *neighborhoodChange:* no início de cada round, os nós são movidos de acordo com a implementação da mobilidade.
- *posStep:* método chamado após a execução de cada round.
- *checkRequirements:* método verifica as configurações de acordo com a especificação de cada nós. Este método é executado imediatamente após a inicialização do nó, antes mesmo da execução do *init*.

Além dos métodos abstratos, a classe `Node` possui outros métodos que são de estrema importância para a simulação de uma RSSF, dentre todos, podemos citar dois principais métodos finais: O *broadcast* e o send. O *send* é responsável por encaminhar uma mensagem a um nó vizinho. Já o método *broadcast* envia uma mensagem a todos os nós vizinhos.

Um atributo de extrema importância nesta classe é o `outgoingConnections`, o qual é um objeto do tipo `Connections`, ou seja, permite que a partir de um nó, seja possível a verificação dos nós vizinhos.

Além destes atributos e métodos, existem diversos outros nesta classe utilizados para outros fins, os quais não são o foco deste trabalho.

3.1.2. Message

A classe abstrata `Message` é utilizada no conceito de mensagem e pacote para serem transmitidas pelos nós da rede. Como em uma situação real, nesta simulação, toda e qualquer informação trocada entre os nós foi por meio de mensagens. A classe `Message` não possui atributos, pois estes atributos correspondem aos cabeçalhos e informações contidas em um pacote de rede. Desta forma, estes atributos são específicos de cada aplicação e tipo de mensagem.

Para implementação desta classe é necessária a implementação do método abstrato clone, o qual realizar a clonagem das mensagens quando estas são enviadas a diversos destinatários, como utilizando o método *broadcast* da classe `Node`.

3.1.3. Timer

`Timer` é um classe abstrata que representa um temporizador, diferentemente dos nativos Java, possui métodos para se trabalhar em tempo de simulação, que no Sinalgo é tratado como *rounds*. A classe `Timer` possui três métodos significativos para o desenvolvimento do projeto, os quais, são citados a seguir:

- *startRelative:* Dispara uma ação em um tempo relativo a determinado nó
- *startGlobalTimer:* Este método dispara a ação de acordo com o tempo global de execução da aplicação
- *fire:* Este é um método abstrato que deve ser sobrescrito. Após o fim da contagem do temporizador, é executada a ação contida neste método.

3.1.4. Edge

A classe `Edge` é a classe responsável por definir como um nó está conectado ao outro. Cada `Edge` é uma conexão entre um nó e outro em uma das direções, sendo necessário a definição de duas `Edge` para envio e recebimento de mensagens.

3.1.5. DistribuitionModel

A classe abstrata `DistribuitionModel` é a classe responsável por realizar a distribuição dos nós no ambiente de simulação, a partir da implementação do método abstrato *getNextPosition*, o qual retorna a posição de cada nó inserido no ambiente de acordo com o algoritmo implementado.

3.1.6. InterferenceModel

A classe `InterferenceModel` é uma classe abstrata utilizada para implementação de técnicas de definição de interferência e perda de pacotes ou mensagens. Nela pode-se implementar o algoritmo de definição da interferência a partir de diversos aspectos, como a sobreposição de sinal dos nós, a quantidade de nós, a quantidade de mensagens transmitidas na rede ou apenas aleatoriamente a partir de uma probabilidade.

Para implementação desta classe é necessária a implementação do método *isDisturbed*, o qual verifica a partir do algoritmo implementado, se um determinado pacote ou mensagem, será entregue ao nó destino, sendo assim retornando apenas um `boolean`.

3.1.7. MobilityModel

A classe abstrata `MobilityModel` é uma classe utilizada para implementação de um tipo de mobilidade dos nós, de forma a definir uma nova posição, a partir do método abstrato *getNextPos* a sem implementado, gerando uma regra de mobilidade.

3.1.8. MessageTransmissionModel

A classe abstrata `MessageTransmissionModel` é uma classe utilizada para implementação de um tipo de transmissão de mensagens entre os nós, onde é definido o algoritmo de transmissão, implementando o método abstrato *timeToReach*, a partir do nó que envia, o nó que recebe e a mensagem a ser enviada.

3.1.9. ReliabilityModel

A classe abstrata `ReliabilityModel` é a classe utilizada para implementação de um modelo de confiança, de acordo com o método abstrato *reachesDestination*, que é utilizado para determinar se um pacote de dados, alcança ou não o seu destino, de acordo com o algoritmo implementado.

3.2. Modelo de Desenvolvimento

O desenvolvimento do projeto foi baseado na implementação de classes abstratas essenciais a execução do mesmo, como das classes abstratas `Node`, `Messages`, `Timer`, além dos modelos `DistribuitionModel` e `InteferenceModel`.

Além dessas classes desenvolvidas foram utilizadas classes já implementadas nos pacotes defaultProject, como `UDG` como modelo de conectividade, `Grid2D` e `Random` como modelos de distribuição, `ConstantTime` como modelo de transmissão de mensagens, `NoMobility` como modelo de mobilidade, `ReliableDelivery` como modelo de confiança e da classe `Edge`.

3.2.1. Pacote models

As classes implementadas neste pacote são diferentes modelos para implementação de atributos dos nós. Neste trabalho foram implementados modelos de distribuição e interferência. Os outros modelos como conectividade, transmissão de mensagens, mobilidade e confiabilidade, foram utilizados modelos já implementados nos pacotes defaultProject. No caso dos modelos de transmissão além da classe implementada, foi utilizada também os modelos Random e Grid2D, do defaultProject.

DistribuicaoSinks

Foi implementada a classe `DistribuicaoSinks` para realizar a distribuição dos Sinks no ambiente de simulação. Esta classe é uma implementação da classe abstrata `DistribuitionModel`. Esta classe abstrata possui um método abstrato, *getNextPosition*, o qual deve retorna um objeto `Position`, responsável por definir a distribuição de cada nó a ser implementado no ambiente de simulação.

Na implementação da classe `DistribuicaoSinks`, a implementação do método *getNextPosition* que retorna os pontos onde deve ser posicionado cada um dos Sinks de 1 a 4, como definido nos requisitos deste trabalho.

InterferenciaAleatoria

Foi implementada a classe `InterferenciaAleatória` para cálculo da interferência gerada na comunicação. Esta classe é uma implementação da classe abstrata `InterferenceModel`.

Como para este projeto não existe um algoritmo ou algum requisito com relação a implementação da interferência, o método *isDisturbed* verifica se um numero aleatório, gerado de 0 a 100, é maior que o atributo estático pcInterferencia da classe `Configuracoes`, se assim for, o pacote não é entregue.

3.2.2. Pacote Nodes

As classes implementadas neste pacote são nós, atributos e ferramente dos mesmos também,. Neste trabalhos foram implementados nós, mensagens, timers, assim como ferramentas para utilização neste projeto em específico. As características da classe `Edge` utilizada neste trabalho foi obtida a partir da utilização da respectiva classe no pacote do framework.

MensagemDados

A classe `MensagemDados` representa um pacote ou mensagem de dados. Esta classe é uma implementação da classe abstrata `Message` que representa os pacotes ou mensagens possíveis de serem enviados pelos nós.

A classe utiliza os atributos idAnterior e idOrigem para identificação do nó que mandou repassou a mensagem por ultimo e o nó que gerou e enviou a mensagem.

È criado um objeto `Random`, para gerar valores aleatórios para os atributos temperatura e umidade, de acordo com os valores máximos e mínimos definidos na classe `Configuracoes` os quais devem ser enviados através da mensagem.

Foram implementados dois construtores para essa classe, um para a criação de uma mensagem deste tipo pelos nós, ou para o reenvio de uma mensagem deste tipo pelos nós.

Esta classe implementa também diversos métodos *geters* e *seters*, além da implementação do método *clone*, que gera uma nova mensagem idêntica, e o *getEnvelopeColor*, que define a cor dos envelopes de envio desta classe como amarelos.

MensagemRota

A classe `MensagemRota` representa um pacote ou mensagem de roteamento. Esta classe é uma implementação da classe abstrata `Message`. Esta classe é utilizada para realizar o roteamento entre os nós da simulação. Ela é gerada a partir de um Sink e distribuída aos nós para gerar rotas de comunicação.

`MensagemRota` utiliza como atributos a idMensagem, a idAnterior e a idOrigem, para identificar a mensagem, o nó que retransmitiu pela ultima vez e o nó que gerou a mensagem, respectivamente. Além disso esta classe também faz uso dos atributos qntPassos e rota, onde o primeiro é responsável por verificar a distância de um determinado nó, quando realizado o roteamento, ao Sink, e o segundo é responsável por definir a rodada de roteamento da mensagem transmitida.

Esta classe implementa diversos métodos *geters* e *seters*, além dos métodos *clone* e *getEnvelopeColor*, como a classe `MensagemDados`, onde é utilizada a cor azul para estes pacotes enviados.

Os construtores desta classe, assim como da classe `MensagemDados`, devem possuir 2 métodos, sendo um para o reenvio das mensagens e o outro para a criação e envio de uma nova mensagem, por cada nó.

São implementados dois métodos para comparação de mensagens. O primeiro, *equals*, o qual é implementado verificando se o idMensagem, rota e o idOrigem são iguais em duas mensagens, garantindo assim que a mensagem seja da mesma rodada de roteamento e enviada pelo mesmo Sink. O segundo, *equalsBasic*, o qual é implementado verificando se apenas o idMensagem e a rota são iguais em duas mensagens.

RoundEnvio

A classe `RoundEnvio` representa um temporizador ou *timer* para execução de uma tarefa. Esta classe é uma implementação da classe abstrata `Timer`.

Os atributos utilizados nesta classe são mensagem, que é a mensagem a ser enviado em determinado tempo; isBroadcast, que identifica se a mensagem deve ser enviada a todos os nós ou a um nó específico, determinado pelo atributo destino.

Existem dois construtores para esta classe, um para ser utilizado quando envia uma mensagem em *Broadcast* e outra para um determinado nó.

Deve-se também reescrever o método *fire*, o qual é executado ao tempo determinado, definindo se o envio de uma mensagem sera realizado por *broadcast* ou a um nó específico.

Bateria

A classe `Bateria` representa uma bateria a ser utilizada em nós, como `Sensor`, para determinar um limite de energia ou de ações que podem serem executadas. A energia utilizada pelo nó é definida pelo atributo energiaGasta. Esta classe não é uma classe que deve ser implementada obrigatoriamente para um projeto, porém devido ao contexto deste projeto, foi necessária esta implementação, adicionando uma nova funcionalidade aos nós.

È implementado diversos métodos para realizar o consumo da energia da bateria, como *consomeEnvia*, *consomeRecebe* e *consomeLigado*, os quais realizam o consumo quando um nó envia, recebe e permanece ligado, respectivamente. Existe também o método *consomeBroadcast* o qual realiza o consumo de acordo com o numero de mensagens enviadas em *broadcast*.

Além disso é implementado o método *isActive* que verifica se ainda existe energia na bateria.

Os parâmetros de consumo ao envio, recebimento e ligado, assim como a quantidade de energia existente em uma bateria, são definidos nos atributos da classe `Configuracoes`.

Sink

A classe `Sink` representa um tipo de nó na rede simulada. Esta classe é uma implementação da classe `Node`, a qual já foi explicada na sub-seção 3.1.1. Um Sink, no contexto desse projeto de simulação, é um nó, o qual está conectado, não utiliza bateria, ou seja, contextualizando, esta conectado a energia, e é responsável por gerar as mensagens que realizam o roteamento, além de serem responsáveis por realizar a leitura das mensagens de dados enviadas pelos Sensores.

Como está é uma classe que implementa outra classe abstrata, é necessário a implementação dos métodos já demonstrados.

Ao ser criado um nó deste tipo, é iniciado um atributo chamado rota, o qual é responsável por informar, na criação de uma nova mensagem de rota, qual é a rodada deste roteamento, sendo assim, ao criar uma nova mensagem deste tipo, sendo incrementada.

A implementação do método *draw* é definida a cor do nó Sink como azul, na interface.

O método *realizarRoteamento* é responsável por iniciar um novo roteamento, in-

crementando a rodada da rota, criando uma novo mensagem de rota e enviando aos nós vizinhos, por *broadcast*. Este método é chamado na inicialização, a partir do método *init*, e após passado o tempo de contagem para criação de uma nova rota, a partir do método *contaRoteamento*, que também realiza esta contagem. Este método é chamado a cada *round* a partir do método sobrescrito *preStep*.

O método sobrescrito *handleMessagens*, que recebe as mensagens, realiza a chamada de dois método: *tratarMensagemRota* e *tratarMensagemDados*. O primeiro método não executa nada, apenas ignora, no caso de uma mensagem de rota ser recebida pelo Sink. O segundo método realiza a leitura da mensagem de dados, e exibe ao usuário, além de incrementar a estatística de mensagens recebidas.

Foi criado um método para aquisição dos dados estatísticos, *dadosSimulação*, que pode ser executado de acordo com a interação do usuário, clicando com o botão direito sobre um Sink e selecionando "Dados da Simulação".

Rotas

A classe `Rotas` foi criado no intuito de se desvincular o processo de criação de rotas de um determinado Sensor da classe `Sensor`.

Quando instanciado um objeto desta classe dentro da classe `Sensor`, ele inicia a contagem a partir do atributo esperaRota, o qual realiza a contagem utilizando a inicialização, *iniciaEspera*, a decrementação, *esperaNovaRota*, e a comparação se a rota é ativa ou não, a partir do método *rotaAtiva*.

O método *semRota* é utilizado a partir do objeto instanciado para permitir a inutilização de uma rota.

O atributo distanciaSink é utilizado para determinar a menor distância entre o nó que possui o objeto do tipo `Rota` e um Sink. O atributo active é utilizado para determinar se as rotas deste objeto estão ativas ou não.

È criada também uma lista de mensagens de rota, que é adicionada todas as novas mensagens de rotas recebidas por um nó que contenha um objeto `Rotas`. Existe uma lista dos nós vizinhos ao nó que contém o objeto, que possuem a mesma menor rota ao Sink, determinando a rota a ser percorrida.

O construtor da classe `Rotas`, quando executado, realiza a atribuição dos valores de distanciaSink, active e do nó Sink responsável por aquela mensagem de rota. È também criado uma nova lista nosRota. Após isso, é realizada uma verificação nos nós vizinhos ao nó onde é criado o objeto, percorrendo-os de modo a verificar os de menores distância ao Sink, e então adiciona-los ao nosRota. Além disso, a mensagem de rota recebida é adicionada a lista mensagensRotaRecebidas.

O método *adicionaNo* realiza a adição de um nó, nos nós da rota, quando não deve ser reiniciado uma nova rota.

O método *melhorSensor* realizar a verificação dos nós que são rota do nó que possui o objeto, e retorna o nó de rota com maior quantidade de energia em sua bateria.

O método *retornaRota* realiza a verificação da distância do nó ao Sink, e então determina o nó retornado para o enviado de uma mensagem, sendo que se distanciaSink

for igual a 1, é retornado o Sink.

Os métodos *passouRodada* e *passouRota*, realizam a verificação, a partir de uma mensagem de rota, se uma mensagem de rota já foi recebida naquela rodada e se aquela mensagem de rota, específica de um Sink, já foi recebida naquela rodada, respectivamente.

Sensor

A classe `Sensor` representa um nó do tipo sensor, o qual é capaz de gerar uma mensagem de dados, e envia-la ao Sink. Esta classe é uma implementação da classe abstrata `Node`, sendo necessário a implementação de seus métodos abstratos.

È criado um objeto `Random`, para gerar o valor para a variável cont, que realiza a contagem do tempo de espera para a geração de uma nova mensagem de dados.

È criado também um objeto do tipo `Bateria` e um `Rotas`. Além disso, há um atributo isActive, do tipo `boolean`, o qual define se o sensor está ativo ou não. Há também o atributo esperaMsg, o qual é utilizado para verificar se já foi possível que o Sink receba a mensagem enviada pelo Sensor, possibilitando a incrementação das estatísticas.

No método sobrescrito *init* é realizada a adição no número total de sensores quando criado um novo Sensor.

O método sobrescrito *draw* define a cor do nó a ser desenhado na interface, o qual é definido como preto, quando o Sensor está ativo, e vermelho, quando o sensor está desativado.

O método *desativarSensor* realiza a verificação se o Sensor está ativo, a partir da bateria e do rotas, realizando um decremento dos sensores ativos, desativando as rotas e desativando o Sensor, caso não esteja correto.

O método *recebeuMensagem* realiza a verificação, após o envio de uma mensagem pelo Sensor, se já foi possivel que o Sink receba a mensagem enviada pelo Sensor, incrementando as estatísticas de mensagens enviadas.

O método sobrescrito *preStep*, executado a cada *round*, realiza o consumo da bateria e a contagem a espera de uma nova rota, se o Sensor estiver ativo, além de realizar a chamada do método *recebeuMensagem*.

O método *geraMensagemDados* realiza a geração de uma nova mensagem de dados e envia esta a partir das rotas, realizando também a inicialização do temporizado esperaMsg e do contador cont, alem de realizar o consumo da bateria, de acordo com o envio.

O método *enviaMensagemDados* realiza a contagem do tempo para geração e envio de uma nova mensagem de dados, a partir do método *geraMensagemDados*, pelo Sensor, a partir da verificação da rota e se o Sensor está ativo.

O método sobrescrito *postStep*, executado a cada *round*, realiza a verificação do sensor a partir do método *desativasSensor* e da verificação e geração de uma nova mensagem de dados a partir do método *enviaMensagemDados*.

O método *tratarMensagemDados* realiza o reenvio de uma mensagem de dados a partir das rotas geradas e também realiza o consumo da energia da bateria.

O método *tratarMensagemRota* realiza as verificações para tratamento de mensagens de rota, verificando primeiramente se o nó continua ativo, verificando se o Sensor já recebeu aquela mensagem de rota. Senão, é reiniciada a espera por uma nova rota, além de realizar a verificação se o Sensor já recebeu uma mensagem de rota na mesma rodada daquela mensagem. Se já tiver recebido, é realizada apenas a verificação se existe algum nó vizinho com a mesma distância mínima de algum Sink, senão é realizada a construção de novas rotas, além de reenviar a mensagem de rota aos vizinhos, em *broadcast*, e realizar o consumo de bateria a partir da quantidade de envios. Quando um Sensor recebe sua primeira rota, ele é adicionado a estatística dos sensores ativos.

O método sobrescrito *handleMessages*, o qual recebe as mensagens, realiza o tratamento a partir do tipo das mensagens recebidas, utilizando os métodos *tratarMensagemRota* e *tratarMensagemDados*, além de realizar o consumo na bateria a partir do consumo para receber uma mensagem.

3.2.3. Pacote do Projeto

No pacote do projeto estão implementados duas classes de grande importância no aspecto deste projeto: `Configuracoes` e `Estatisticas`.

A classe `Estatisticas` será explicada na seção **??**.

Configuracoes

A classe configuração é uma classe final implementada no sentido de definir parâmetros de configuração no projeto e na simulação de forma facilitada, sem a necessidade de modificar diversos dados em diversas classes.

Os atributos desta classe são todos atributos públicos, finais e estáticos, o que permite o acesso em qualquer classe do projeto e não permite que outro código modifique seus valores. Estes atributos devem ser modificados pelo usuário quando este sentir necessidade.

Os valores utilizados para estes atributos nos testes foram definidos sem utilização de metodologia ou dados específicos.

No atributo pcInterferencia deve ser definido o valor da porcentagem dos pacotes a serem entregues, ou seja, não afetados por interferência. O valor deve variar de 0 a 100.

No atributo tempoRota deve ser definido o valor de espera para que cada Sink realize um novo roteamento, ou seja, que o Sink envie uma nova mensagem de rota.

No atributo qntEnergia deve ser definido a quantidade de energia disponível em cada bateria, utilizada nos sensores. Os atributos consumoEnvia, consumoRecebe e consumoLigado definem o consumo da bateria quando cada sensor realiza tais ações, enviar mensagem, receber mensagem e permanecer ligado.

Os atributos minEspera e maxEspera definem o tempo mínimo e máximo de espera de um Sensor, para que este possa gerar e enviar uma nova mensagem de dados.

O atributo esperaRotaSensor define o tempo máximo de espera, em que um Sensor é considerado ativo, mesmo sem que este recebe uma nova mensagem de dados neste tempo, após isto o Sensor é considerado inativo mesmo se houver energia em sua bateria.

Os atributo minTemperatura, maxTemperatura, minUmidade e maxUmidade definem os valores máximos e mínimos que um nó pode gerar para a temperatura e a umidade.

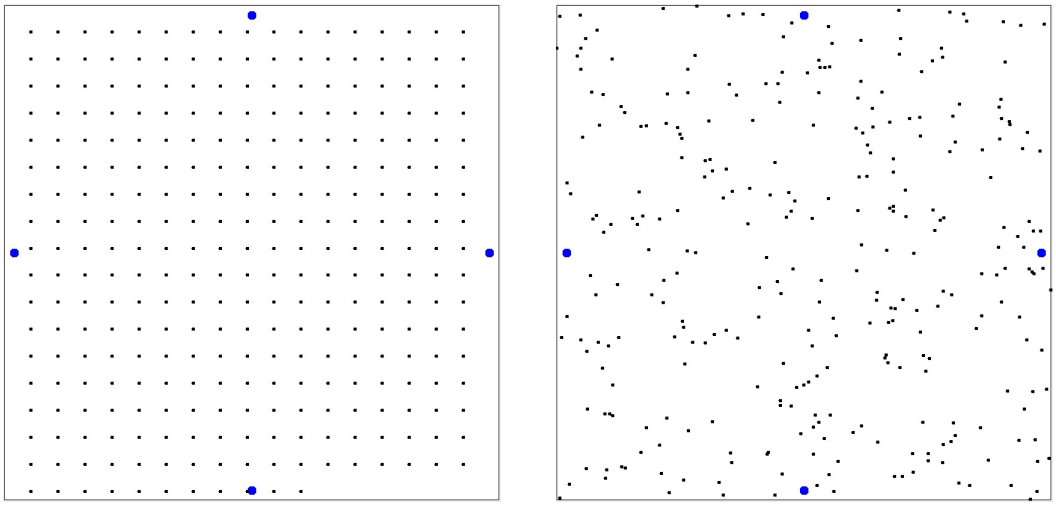
Capítulo 4

Proposta de Simulação

4.1. Cenários

Este projeto, tenta simular uma RSSF em uma área onde os sensores são responsáveis pela coleta de dados referentes à temperatura ambiente e umidade relativa do ar. Cada sensor deve enviar estes dados coletados para seus nós vizinhos até que chegassem a uma estação de recebimento, no caso, o nó-sink.

Neste contexto foi realizada a simulação utilizando o padrão bidimensional constituído por um espaço de 1000x1000 (*pixels*), onde foi construído dois diferentes cenários de simulação que permitiram uma melhor compreensão didática do problema proposto. O primeiro se trata de uma topologia em formato *Grid2D*, onde os nós-sensores se posicionam de forma uniforme na área proposta, pode ser vista melhor na figura 4.1(a). No segundo cenário foi utilizado a topologia *Random*, onde os nós são posicionados de forma aleatória na área da simulação, exposta na figura 4.1(b).



(a) Grid2D (b) Random

Figura 4.1. Topologias utilizadas no projeto

Em ambos os cenários foram utilizados 300 nós-Sensores e variação de 1 á 4 nós-Sinks.

Os nós-Sinks foram posicionados entre as extremidades da área de simulação, o posicionamento do sensores foram de acordo com as topologias já mencionadas.

De uma forma bem simples e didática, foi estipulado um 'tempo de vida' de 1000 pontos, para cada nó. A cada round, foi decrementado 1 ponto do nó, a cada mensagem que o nó-sensor recebesse eram decrementados 3 pontos e quando um nó transmitisse uma mensagem, era decrementados 5 pontos. Por exemplo, se um nó x recebesse uma mensagem do nó y e em seguida o nó x repassasse a mensagem para o sink, ele consumiria 10 pontos, pois além dos 3+8 pontos consumidos do encaminhamento das mensagens, levaria 2 rounds para que esta transmissão foce feita.

Com fins didáticos, simplificadamente, foi implementado interferência nos nós, onde cada nó tem uma probabilidade de 2% de não conseguir receber uma mensagem que outro nó vizinho está transmitindo. Para configuração deste parâmetro é utilizado o atributo pcInterferencia da classe `Configuracoes`.

Para realização dos testes, neste trabalho, foi utilizada a classe `Configuracoes` para configurar diversos parâmetros, como já informado anteriormente. O atributo tempoRota é utilizado para configuração do tempo que cada Sink demora para gerar e enviar uma nova mensagem de roteamento para os Sensores. Os atributos minEspera e maxEspera são utilizados para configurar o intervalo do tempo de espera entre o envio da mensagem anterior e uma nova mensagem de dados pelos Sensores. O atributo esperaRotaSensor define o tempo máximo de espera de recebimento entre uma mensagem anterior e uma nova mensagem de rota, passado este tempo o Sensor passa para o estado inativo. Os atributos minTemperatura, maxTemperatura, minUmidade e maxUmidade, definem os intervalos de dados gerados pelos Sensores, para envio de mensagens de dados, a respeitos dos valores de temperatura e umidade, respectivamente.

Os nós sensores são representados por um pequeno circulo preto, enquanto os nós-sink são representados por um circulo maior e de cor azul.

4.2. Parâmetros de Comparação

Nesta seção, abordaremos quais e como foram tratadas as variáveis comparadas a cada simulação. Falaremos sobre área de cobertura, sobre a quantidade de pacotes recebidos pelos sinks, sobre a quantidade de pacotes enviados pelos nós, o tempo de 'bateria' dos sensores e a energia total consumida pelos sensores. Estes dados foram obtidos através da classe `Estatisticas` a partir dos atributos msgsRecebidas, msgsEnviadas, sensoresAtivos, numeroSensores, energiaConsumida e dos métodos *txMsgs*, *pcSensores*, *txEnergia*.

Referente a área de cobertura foi comparado apenas a quantidade de nós ativos na rede, devido a grande complexidade desta comparação na topologia random. Por exemplo, simulamos um cenário com 300 nós-sensores, se apenas 200 nós estiverem ativos, a área de cobertura será de 66%. Esta cobertura foi calculada dividindo a quantidade total de nós-Ativos, sensoresAtivos, pela quantidade total de nós, numeroSensores, existentes na rede, com o método *pcSensores*.

A quantidade de mensagens recebidas pelo sink, msgsRecebidas, é uma variável que é incrementada a cada mensagem que chega até o sink a partir dos sensores que fazem conexão direta á ele. A quantidade de mensagens enviadas é a soma de todas as mensagens que foram enviadas pelos nós, msgsEnviadas, independente se foi ou não entregue ao sink. A partir destes valores calculamos a taxa de mensagens recebidas, dividindo a quantidade de mensagens recebidas pelos sinks pela quantidade de mensagens

enviadas por todos os sensores, a partir do método *txMsgs*.

Foi estipulado 1000 pontos de bateria para cada nó sensor. Esses pontos foram calculados da seguinte maneira: A cada turno é decrementado 1 ponto de cada nó, isto refere-se ao consumo de manutenção do sensor. A cada mensagem recebida por um nó, decrementamos 2 pontos de bateria. E por fim, é decrementado 5 pontos de bateria por cada mensagem que o nó-sensor transmitir. Essas configurações a respeito dos valores de configuração da bateria é realizada a partir dos atributos qntEnergia, consumoLigado, consumoRecebe e consumoEnvia da classe `Configuracoes`. Toda energia consumida pelos sensores são armazenadas e exibidas como Energia total consumida, energiaConsumida e é realizada o cálculo da taxa de energia gasta por mensagem recebido, a partir do método *txEnergia*.

4.3. Resultados

Os testes relatados a seguir foram executados em uma área de 1000x1000. A duração da aplicação simulada é de 240 *rounds* utilizando a topologia Grid2D e 300 *rounds* na topologia Random, frequência de emissão de mensagens pelos sensores variaram de 10 á 40 rounds, em uma rede com interferência de 2%, variando apenas o número de Sinks. A rede foi arranjada em nos esquemas Grid 2d (distribuição homogênea pelo campo) e Random (distribuição aleatória pelo campo).

Os resultados foram divididos em três etapas. A cada etapa o processo foi repetido três vezes, o primeiro teste utilizando a topologia Grid2d com variações de Sinks, seguir foram realizados os mesmos testes utilizando a topologia Random e por fim, também na topologia Random, foi realizado uma comparação utilizando quatro Sinks e alterando apenas a 'tempRota' para trinta rounds. A apresentação dos gráficos e discussão dos resultados divididos em: `Topologia Grid2D`, `Topologia Random` e `tempRota 50 vs tempRota 30`.

Em ambas as topologias `Grid2d` e `Random` foram coletados os dados referentes a *Área de Cobertura* dos sensores, o *Total de Mensagens Recebidas pelos Sinks*, a média de *Energia consumida por Mensagem* e por fim a *Energia Total* durante os turnos. Estes gráficos são gerados a partir da média dos resultados coletados em três testes.

4.3.1. Topologia Grid2D

A topologia grid2d normalmente apresenta resultados previsíveis. Esta topologia raramente é aplicada em uma RSSF, neste trabalho ela foi utilizada apenas para melhor compreensão do comportamento da rede.

Abaixo seguem os gráficos e discussão dos resultados alcançados nesta topologia.

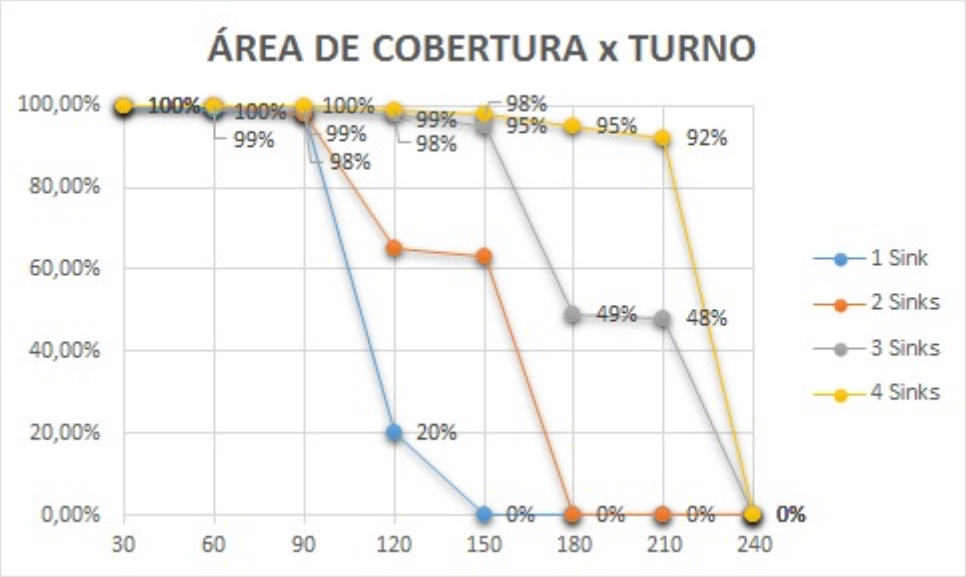


Figura 4.2. GRÁFICO: Grid2D - Área de cobertura

O gráfico 4.2 apresenta a área de cobertura alcançada ao longo dos rounds. Claramente pode-se notar uma grande discrepância nos resultados alcançados com um e com quatro Sinks. A cobertura realizada com o emprego de apenas um Sink passa de ser viável aproximadamente no round noventa. Este fato acontece pois na topologia grid2d, os Sinks possuem poucas conexões e todo tráfico realizado pela rede, obrigatoriamente deve passar por estas poucas conexões. Fato que ocasiona um alto consumo de bateria, o que proporciona a inutilização da rede em poucos rounds.

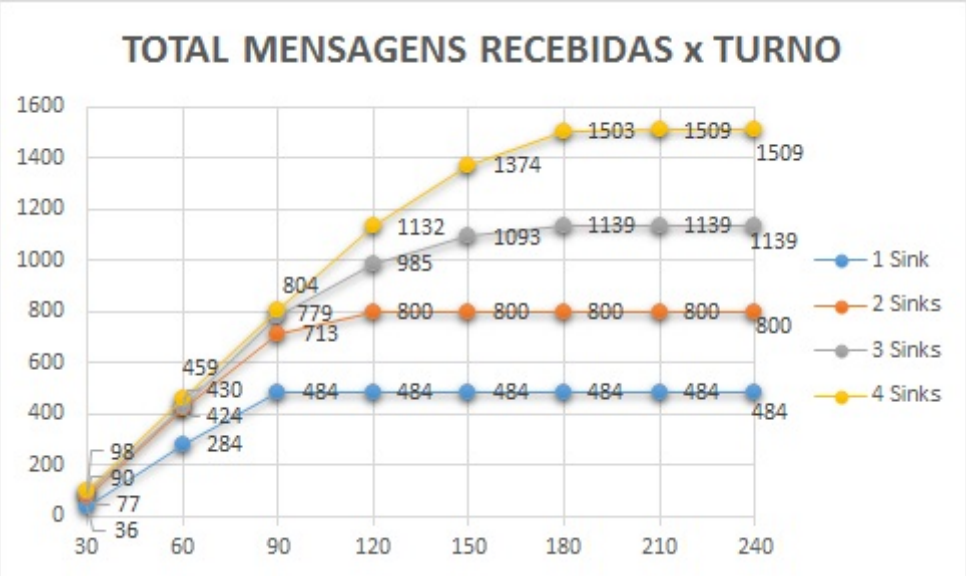


Figura 4.3. GRÁFICO: Grid2D - Total de Mensagens recebidas pelos Sinks

Observando gráfico 4.3, nota-se que os testes realizados com 2, 3 e 4 Sinks, tem um total de mensagens recebidas muito similar até o round 90 e após isto, de forma gradual, cada teste teve recebimento de mensagem congelado, devido ao fato dos nós que mantêm a conexão com o Sink terem sidos desativados por falta de bateria.

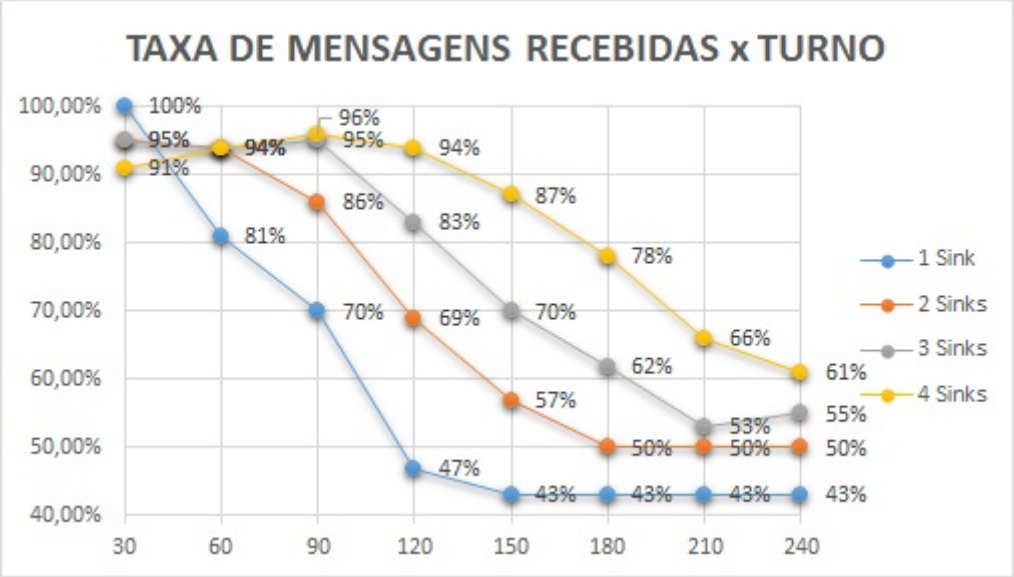


Figura 4.4. GRÁFICO: Grid2D - Porcentagem de Mensagens recebidas vs perdidas

O gráfico 4.4 elucida a taxa de mensagens recebidas. Neste contexto, com exceção ao teste com 1 Sink, nota-se que em todos os outros esta taxa persevera por alguns rounds e despenca ao longo do tempo. Este fato ocorre pois um sensor tem uma devida rota até que o Sink envie novos dados de rota, neste período, quando os sensores começam a ter a bateria esgotada, muitos pacotes são perdidos. Este fato é ainda mais gritante quando observado o teste apenas com 1 Sink.

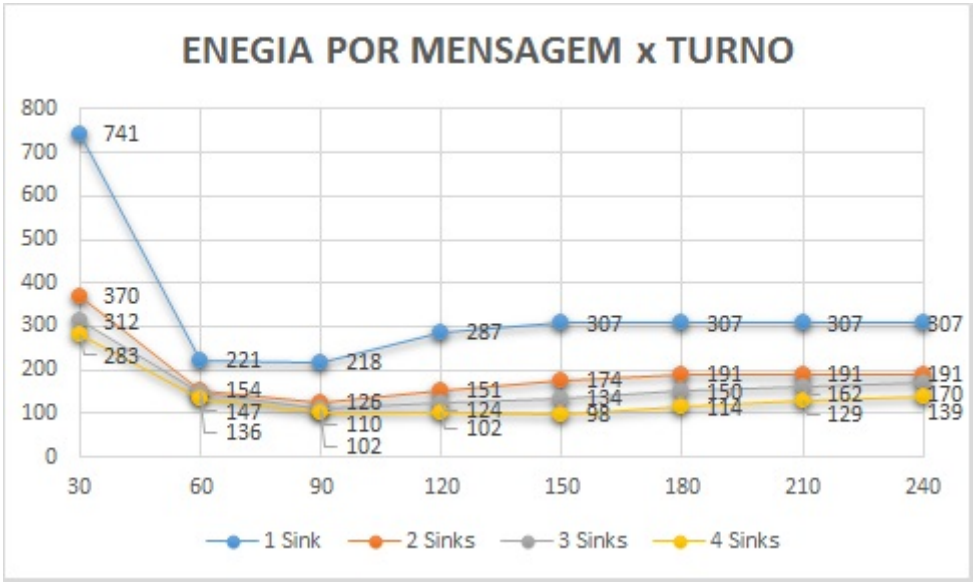


Figura 4.5. GRÁFICO: Grid2D - Média de Energia gasta por mensagem recebida

O gráfico 4.5 exibe a quantidade média consumida de energia por mensagem enviada. Observa-se que os testes com 2, 3 e 4 sinks mantiveram padrões muito similares. Porém nas simulações realizas com apenas um Sink observa-se uma grande diferença, ofato ocorre pois como o Sink é posicionado próximo á borda da área de simulação, quando um nós do lado oposto envia uma mensagem, esta mensagem deve percorrer toda

a área até que chegue ao Sink. Também é possível observar o alto custo no inicio das simulações, fato ocasionado pela inundação de roda realizado pelo Sink logo nos primeiros turnos.

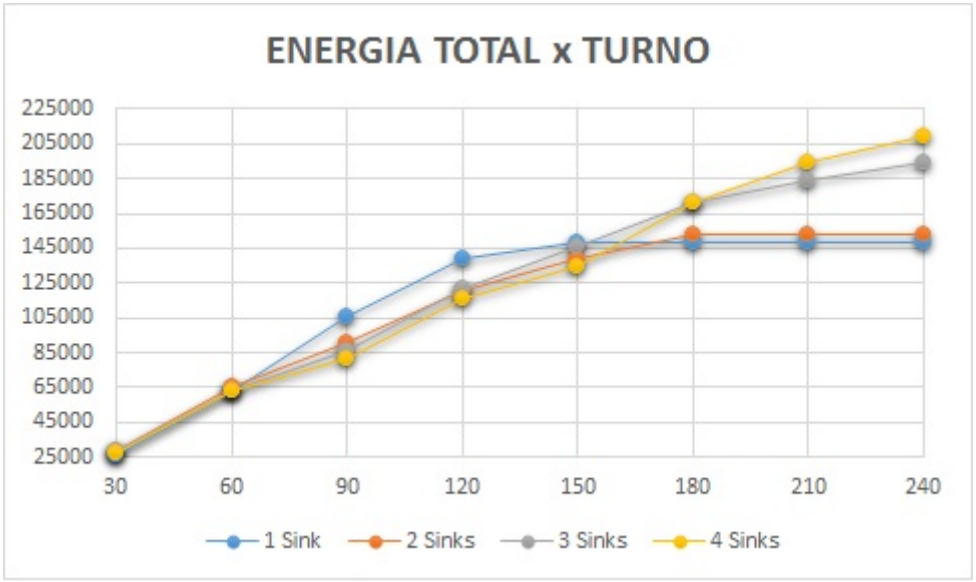


Figura 4.6. GRÁFICO: Grid2D - Energia total consumida pela Simulação ao longo dos rounds

E por fim, o gráfico 4.6, nele é possivel observar que o consumo das simulações com 1 e 2 sinks mantiveram um padrão bem similar, bem como as simulações com 3 e 4 sinks. Nota-se que com 3 e 4 sinks o valor de energia consumido é extremamente maior que quando utilizado 1 ou 2 sinks, o fato ocorre pois quando é alcançado em torno de 150 rounds, as simulações com menos sinks acabam sendo inviabilizadas, pois os nós responsável por toda a comunicação com os sinks acabam se esgotando, evento que tarda a acontecer com as simulações com 3 e 4 sinks.

4.3.2. Topologia Random

Diferente da topologia Gride2D, a topologia Random se assimilha mais com a realidade das RSSF. Nesta topologia os resultados são imprevisíveis, fato que levou o grupo a recorrer de vários teste para alcançar um valor consistente.

A seguir é apresentado os gráficos e as discussão dos resultados alcançados nesta topologia.

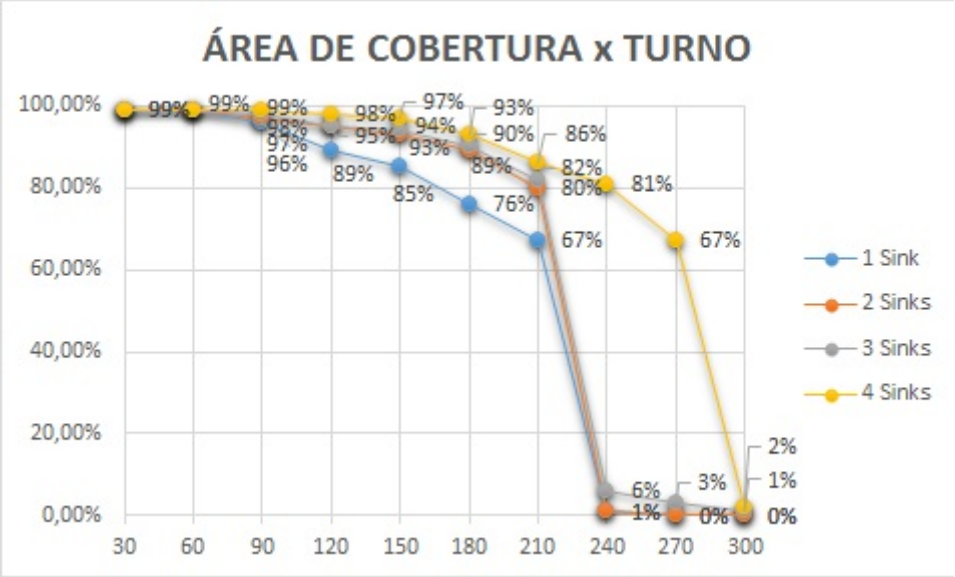


Figura 4.7. GRÁFICO: Random - Área de cobertura

O gráfico 4.7 apresenta a área de cobertura alcançada ao longo dos rounds. Diferente do modo grid, nesta topologia, nem sempre a área de cobertura inicial é 100%, pois nesta distribuição alguns nós ficam isolados. Mas através deste gráfico, é possível observar que até o round 210 os testes com 1, 2 e 3 sinks apresentaram um comportamento similar, porém o teste com 4 sinks perseverou durante alguns turnos, diferente do que pode-se utilizando menos sinks. Também é possível que em todos os casos, há uma queda repentina na área de cobertura, este fenômeno ocorre, pois os sensores se desligam quando não recebem um novo sinal de rota durante 60 turnos.

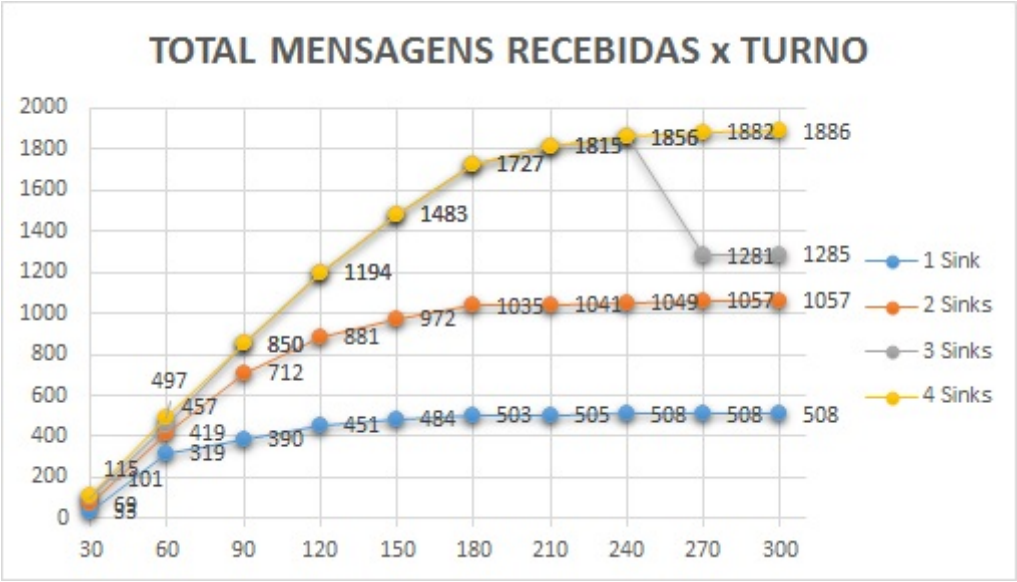


Figura 4.8. GRÁFICO: Random - Total de Mensagens recebidas pelos sinks

Observando o gráfico 4.8, encontramos certa semelhança com os dados alcançados no modo grid, representados no gráfico 4.3, porém no modo random, os testes realizados com 3 sinks apresentaram um padrão um pouco diferente dos outros. Isto ocorre pois em

todos os testes, os sensores mais próximos aos sinks, tinham suas baterias esgotadas, o que acabava desconectando o restante da rede.

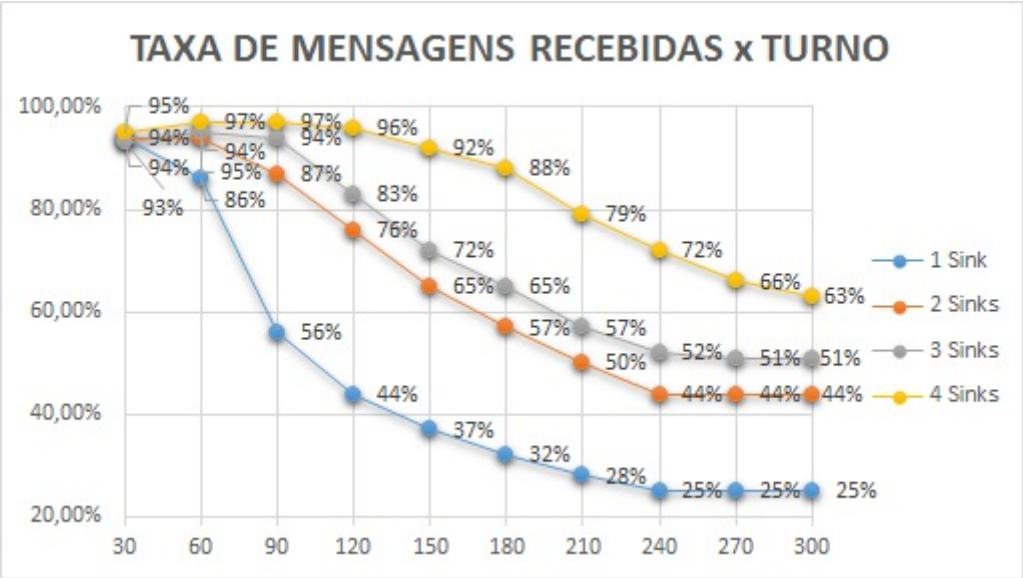


Figura 4.9. GRÁFICO: Random - Porcentagem de Mensagens recebidas vs perdidas

Os dados apresentados no gráfico 4.9 exibem a porcentagem calculada entre mensagens enviadas pelos sensores vs mensagens recebidas pelos sinks. Observando o comportamento do gráfico, fica nítida a diferença desta taxa conforme a variação da quantidade de sinks. Ao longo dos rounds, este valor fica muito comprometido, pois muitos nós começam a se desligar por falta de bateria, então todas as mensagens enviadas a eles são perdidas. Isto é corrigido apenas quando uma nova rota é enviada pelo Sink.

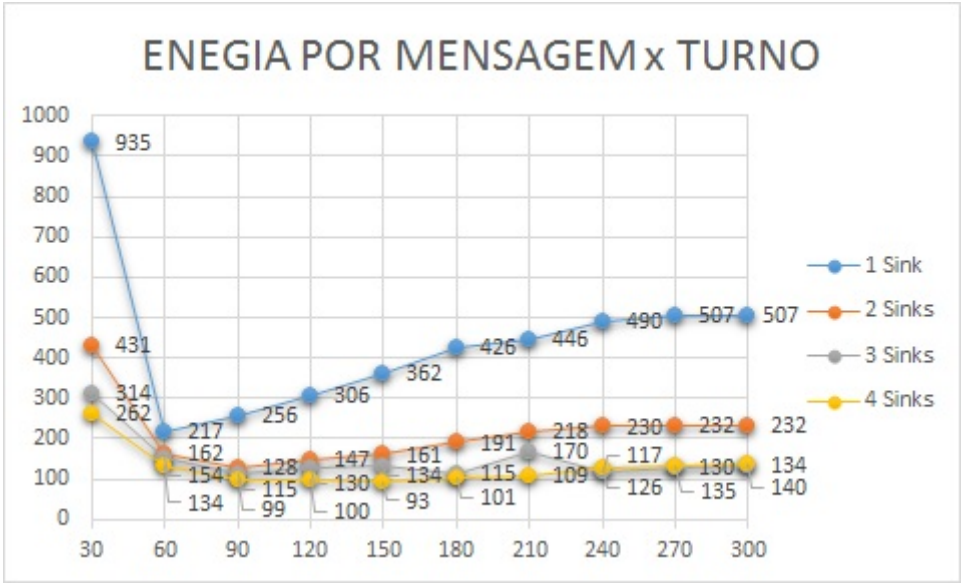


Figura 4.10. GRÁFICO: Random - Média de Energia gasta por mensagem recebida

O grafico 4.10 apresenta um comportamento muito similar ao gráfico 4.5 da topologia grid. Porém os testes com apenas 1 Sink é ainda mais custoso no modo random do que no modo grid, pois a distancia máxima de nós até o Sink no modo random é ainda

mais alto que a distancia máxima no modo grid. Também é possível observar que após uma determinada quantia de rounds, os valores se mantém constantes, isso é ocasionado pele desligamento da rede, na maioria das vezes causada pela exaustão de energia dos nós mais próximos aos sinks.

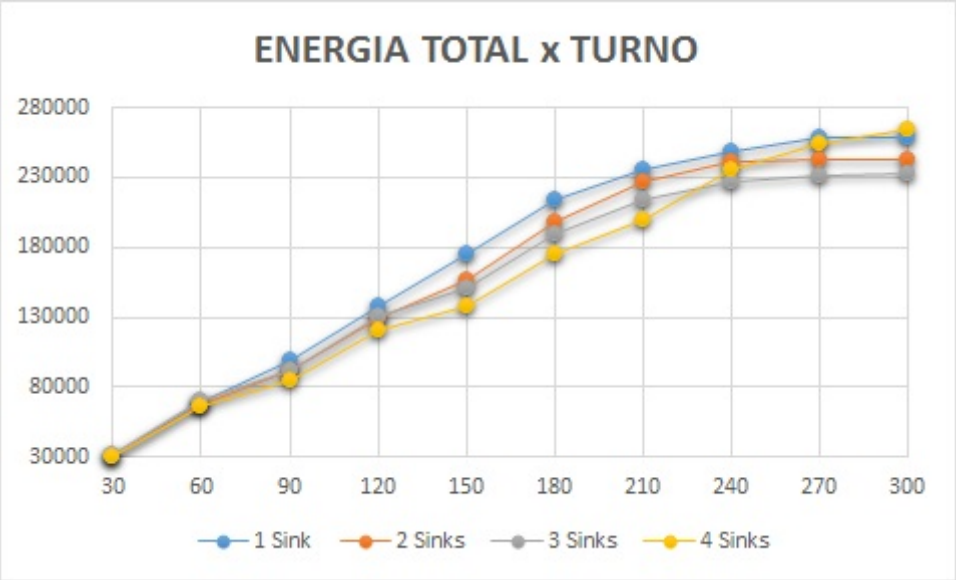


Figura 4.11. GRÁFICO: Random - Energia total consumida pela Simulação ao longo dos rounds

E por fim o gráfico 4.11, que também se assemelha muito ao gráfico 4.6 da topologia grid, exceto pelo fato do maior consumo de energia total. Isso pode ser esclarecido facilmente observando a quantidade de rounds. Na topologia random, os testes alcançavam até 300 rounds em comunicação, diferente do que ocorreu no modo grid, que alcançaram um máximo de 240 rounds.

4.3.3. tempRota 50 vs tempRota 30

Nesta seção dos testes, comparamos o desempenho da RSSF quando o tempRota fosse alterado do valor padrão dos testes (50) para 30. O tempRota é uma varável que estipula á quantos rounds os sinks devem enviar uma nova rota.

Neste contexto, a comparação foi feita apenas para fins de curiosidade. Os testes foram realizados 2 vezes e foram comparados apenas simulações com topologia random e 4 sinks.

A seguir é apresentado os gráficos e as discussão dos resultados alcançados nos testes.

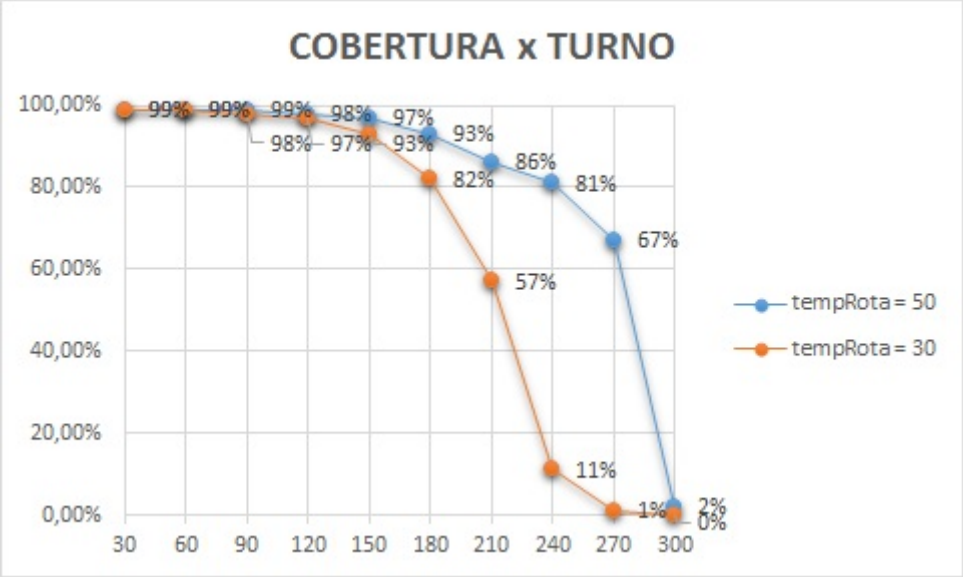


Figura 4.12. GRÁFICO: tempRound - Área de cobertura

Observando o gráfico 4.12 notamos o comportamento típico da topologia random, em ambos os casos a média da cobertura inicial se deu por 99%. Prosseguindo a análise, notamos que há uma queda precoce na curva do teste feito com tempRound 30. Este comportamento ocorre pois a rede é inundada com pacotes broadcast do Sink de forma muito frequente. Isso gera um alto consumo de energia dos sensores, ocasionando um menor tempo de vida da rede.

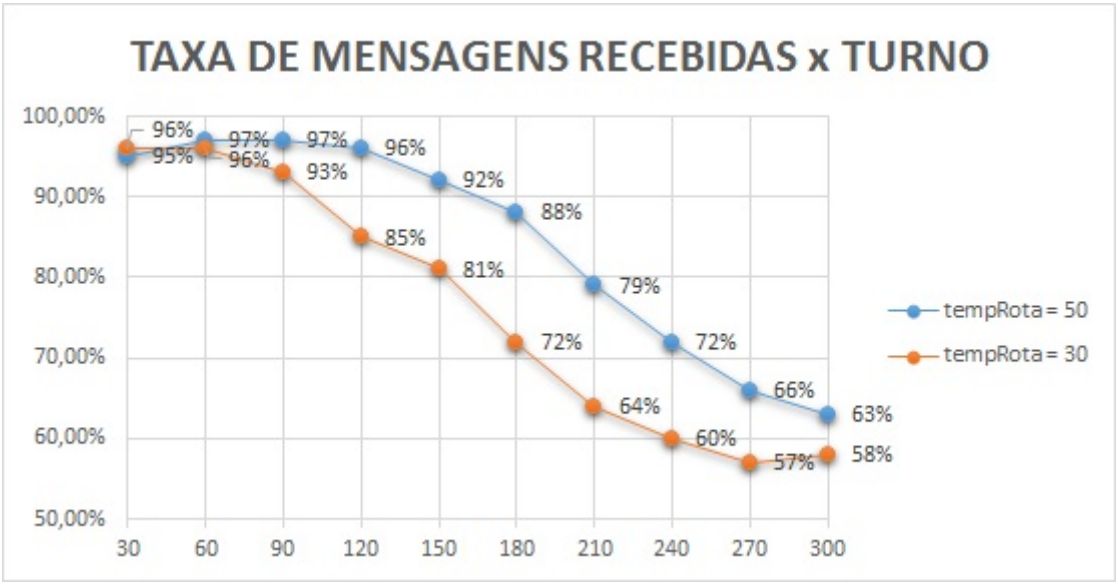


Figura 4.13. GRÁFICO: tempRound - Porcentagem de Mensagens recebidas vs perdidas

Levando em consideração os dados do gráfico anterior, no gráfico 4.13 a taxa de mensagens recebidas pelo Sink está diretamente ligado ao que foi dito anteriormente. No modelo com tempRound 30, os nós desligam mais rapidamente por falta de bateria, todos os pacotes enviados a estes nós são perdidos, o fato ocorre até uma nova rota ser traçada.

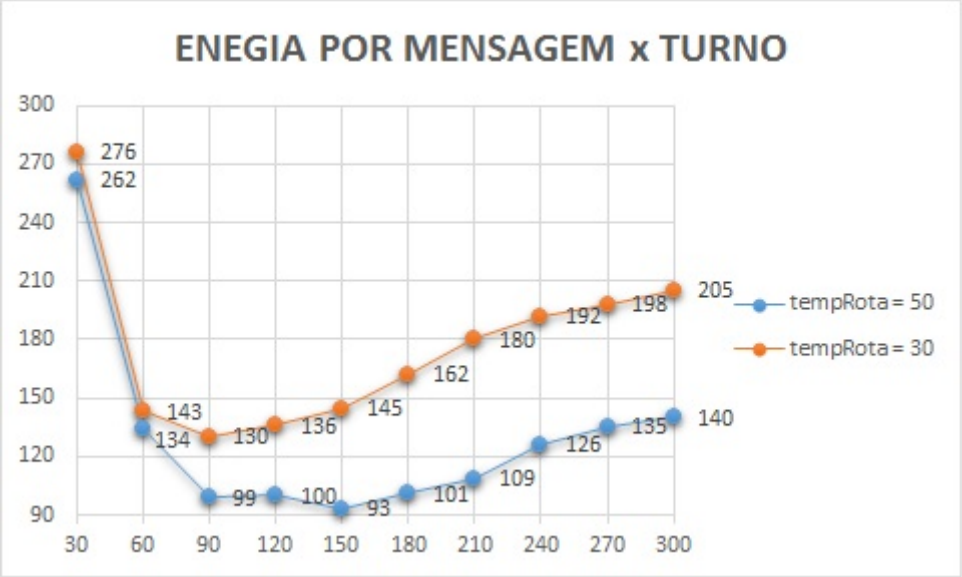


Figura 4.14. GRÁFICO: tempRound - Média de Energia gasta por mensagem recebida

O gráfico 4.14 elucida o alto consumo de energia por mensagem no modelo com tempRota 30. Em alguns pontos da coleta, esta discrepância chega até a 35%. Este alto custo por mensagem está diretamente ligado á inundação de pacotes de rota na rede. O alto custo para cada nó retransmitir estes dados afeta diretamente a viabilidade deste modelo.

4.3.4. Análise dos Resultados

Observando todos as informações representadas pelos gráficos, podemos fazer uma breve conclusão. No que se trata de topologias, o modelo Grid2D se comporta de forma muito similar em todos os testes. Neste modelo a rede manteve-se ativa por 240 rounds no máximo, em seu melhor caso com baixa QoS. Já no modelo random esta uniformidade de resultados não acontece. Podemos observar que a autonomia de uma RSSF espalhada de forma randômica é superior ao modelo grid, alcançando até 300 rounds, também com baixa QoS. Neste resultado também verificamos que nem sempre manter a rota dos nós constantemente atualizada é uma boa técnica, pois isso demanda de uma grande capacidade energética dos nós.

Capítulo 5

Conclusão

Neste trabalho estudamos a transmissão de dados em RSSFs. Este é um assunto bastante investigado já que a forma como é desenvolvida pode causar grande impacto na eficiência da rede. Neste contexto a utilização de simuladores se faz de extrema importância.

Este estudo foi realizado utilizando o simulador SinalGo, que é um framework desenvolvido em java. Graças ao Sinalgo, foi possível que os alunos adquirissem muito conhecimento sobre diversos fatores que englobam a RSSF, desde Qualidade de Serviços até diferentes simuladores. Para sedimentar esses conhecimentos foram realizados testes que contaram com a alteração da organização dos sensores e a variação da quantidade de sinks.

Através dos resultados obtidos pelos testes, podemos perceber com mais clareza a importância de diversos fatores que comprometem viabilidade da implantação de uma RSSF. O fator que mais nos chamou a atenção foi a forma como é implementado o roteamento da rede. Uma rota mal implementada é sem sombra de dúvidas um dos maiores desafios do problema. Porém outro fator muito importante é a quantidade de sinks, como e onde são posicionados. A frequência com que os nós-sinks enviam mensagens de roteamento por broadcast na rede é algo que deve-se ter cautela. A inundação da rede com pacotes de roteamento causa grande consumo de bateria dos sensores, que além de muito limitadas são caras.

Ao longo da realização deste trabalho podemos verificar a importância de se investigar as peculiaridades e requisitos de uma RSSF antes de projetar-la. O desempenho da rede está totalmente ligado á forma que os sensores são implementadas, logo o projetista deve procurar e testar as melhores maneiras de adaptar os nós ao contexto desejado.

Pode-se concluir que este trabalho alcançou as metas propostas e poderá servir em trabalhos futuros, sejam estes trabalhos com o simulador SinalGo ou trabalhos com sensores reais.

Como trabalhos futuros, sugere-se a implementação de um método de interferência mais condizente com a realidade. Outro trabalho futuro seria a implementação otimizada do 'tempRota', onde este tempo pode variar de acordo com o estado da rede. Também sugere-se como trabalhos futuros o estudo mais aprofundado de formas de otimização de energia dos sensores.

Referências

[int] Saúde digital: Novos modelos de cura.

[Arampatzis et al. 2005] Arampatzis, T., Lygeros, J., and Manesis, S. (2005). A survey of applications of wireless sensors and wireless sensor networks. In *Intelligent Control, 2005. Proceedings of the 2005 IEEE International Symposium on, Mediterrean Conference on Control and Automation*, pages 719–724. IEEE.

[Chao and Guo 2002] Chao, H. J. and Guo, X. (2002). *Quality of service control in high-speed networks*. John Wiley & Sons.

[Delicato 2005] Delicato, F. C. (2005). *Middleware baseado em serviços para redes de sensores sem fio*. PhD thesis, UNIVERSIDADE FEDERAL DO RIO DE JANEIRO.

[Fernandes et al. 2006] Fernandes, N. C., Moreira, M. D., Velloso, P. B., Costa, L., and Duarte, O. (2006). Ataques e mecanismos de segurança em redes ad hoc. *Minicursos do Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais (SBSeg'2006)*, pages 49–102.

[Iyer and Kleinrock 2003] Iyer, R. and Kleinrock, L. (2003). Qos control for sensor networks. In *Communications, 2003. ICC'03. IEEE International Conference on*, volume 1, pages 517–521. IEEE.

[Johnson and Margalho 2006] Johnson, T. M. and Margalho, M. (2006). Wireless sensor networks for agroclimatology monitoring in the brazilian amazon. In *Communication Technology, 2006. ICCT'06. International Conference on*, pages 1–4. IEEE.

[Loureiro et al. 2003] Loureiro, A. A., Nogueira, J. M. S., Ruiz, L. B., Mini, R. A. d. F., Nakamura, E. F., and Figueiredo, C. M. S. (2003). Redes de sensores sem fio. In *Simpósio Brasileiro de Redes de Computadores (SBRC)*, pages 179–226.

[Monteiro 2006] Monteiro, J. A. (2006). Redes ad hoc. *ISPGAYA - Instituto Superior Politécnico Gaya*, 13:23–30.

[Pegden et al. 1995] Pegden, C. D., Sadowski, R. P., and Shannon, R. E. (1995). *Introduction to simulation using SIMAN*. McGraw-Hill, Inc.

[Perillo 2003] Perillo, H. (2003). Providing application qos through intelligent sensor management. In *Sensor Network Protocols and Applications, 2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, pages 93–101. IEEE.

[Ruiz et al. 2004] Ruiz, L. B., Correia, L. H. A., Vieira, L. F. M., Macedo, D. F., Nakamura, E. F., Figueiredo, C. M., Vieira, M. A. M., Bechelane, E. H., Camara, D., Loureiro, A. A., et al. (2004). Arquiteturas para redes de sensores sem fio1.

[RUIZ et al.] RUIZ, L. B. et al. Arquiteturas para redes de sensores sem fio-departamento de ciência da computação da ufmg. *Departamento de Informática da PUCPR, Departamento de Ciência da Computação da UFLA, Fundação Centro de Análise, Pesquisa e Inovação Tecnológica–FUCAPI, Departamento de Engenharia Elétrica da UFMG http://www. sensornet. dcc. ufmg. br/pdf/mc-sbrc2004. pdf*.

[Siqueira 2002] Siqueira, F. (2002). Especificaçao de requisitos de qualidade de serviço em sistemas abertos: a linguagem qsl. In *Proceedings of the 20th Brazilian Symposium on Computer Networks, Búzios-RJ, Brazil*.

Apêndice A

Protocolos de Comunicação

No contexto de redes de computadores, um protocolo é um conjunto de regras definido para controlar uma comunicação entre duas máquinas quaisquer de forma que as mesmas possam trocar informações de maneira íntegra.

Tecnicamente, é um conjunto de regras-padrão que caracterizam o formato, a sincronização, a sequência e, ainda, a detecção de erros e falhas na comutação de pacotes, isto é, na transmissão de informação entre dispositivos computacionais.

A.1. Modelo OSI

No modelo OSI, a comunicação é dividida em sete camadas sobrepostas, onde cada camada trata de um aspecto específico da comunicação, essa divisão objetiva reduzir a complexidade da implementação de um protocolo de rede. Cada camada fornece uma interface para a camada imediatamente acima. Essa interface consiste de um conjunto de operações que uma camada pode prestar para a camada acima, ocultando detalhes da implementação desse recurso. Cada camada da pilha de protocolos de uma máquina se comunica com a mesma camada de outra máquina, e para isso existe um protocolo para comunicação entre as mesmas camadas, tomando possível a interpretação do que a camada da máquina remota expressa.

A figura A.1 mostra as camadas de protocolos da pilha OSI e seus respectivo protocolos entre as camadas.

[1]

[1] Fonte: `http://www.teleco.com.br/imagens/tutoriais/`

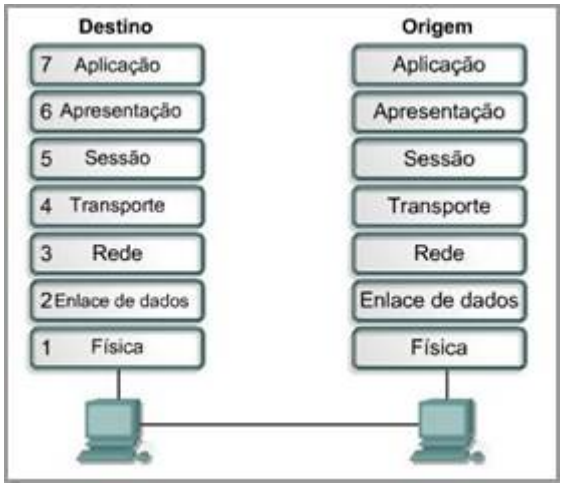


Figura A.1. Camadas do modelo OSI

A cada nível descendente na pilha OSI é adicionado um cabeçalho referente ao protocolo da camada anterior e, ao chegar na camada física, um pacote a ser transmitido toma a forma ilustrada na figura A.2.

1	2	3	4	5	6	7	Dados

Figura A.2. Pacote com os cabeçalhos do modelo OSI

A camada `Física` é responsável por converter em sinais os quadros da camada de `Enlace de dados` em sinais compatíveis com o meio de transmissão.

A camada de `Enlace de dados` é responsável por receber os pacotes da camada de `Rede` e transforma-los em quadros (*frames*).

A camada de `Rede` é responsável por efetuar o endereçamento dos pacotes da camada de `Transporte` de forma que chequem ao destino.

A camada de `Transporte` é responsável por efetuar a divisão dos dados em pacotes de forma a serem transmitido pela rede.

A camada de `Sessão` é responsável por controlar a transmissão de dados entre duas aplicações tratando os erros e registros.

A camada de `Apresentação` é responsável por efetuar a tradução e conversão de dados oriundos da camada de `Aplicação` de forma a serem transmitidos.

E por fim, a camada de `Aplicação` do modelo OSI é responsável por efetuar a interface entre as aplicações e os protocolos de rede de forma a apresentar os dados ao utilizador.

A.2. Modelo TCP/IP

O modelo TCP/IP pode ser visto como um modelo de camadas (Modelo OSI) evoluído, onde cada camada é responsável por um grupo de tarefas, fornecendo um conjunto de serviços bem definidos para o protocolo da camada superior. As camadas mais altas estão logicamente mais perto do usuário (chamada camada de aplicação) e lidam com dados mais abstratos, confiando em protocolos de camadas mais baixas para tarefas de menor nível de abstração.

Na prática não se usam todas as sete camadas do modelo OSI, é o que acontece com o protocolo TCP/IP que possui apenas quatro camadas: *Aplicação* (Aplicação, Apresentação e Seção), *Transporte*, *Rede*, e camada *Física de enlace*. A figura A.3 faz uma comparação entre o modelo TCP/IP e o modelo OSI das sete camadas.

Figura A.3. Modelo TCP/IP vs Modelo OSI

A camada *Física de Enlace* especifica os detalhes de como os dados são enviados fisicamente pela rede, inclusive como os bits são assinalados eletricamente por dispositivos de hardware que estabelecem interface com um meio da rede, como cabo coaxial, fibra óptica ou fio de cobre de par trançado.

A camada de *Rede* empacota dados em datagramas IP, que contêm informações de endereço de origem e de destino usadas para encaminhar datagramas entre hosts e redes. Executa o roteamento de datagramas IP.

A camada de *Transporte* fornece gerenciamento de sessão de comunicação entre computadores host. Define o nível de serviço e o status da conexão usada durante o transporte de dados.

A camada de *Aplicação* define os protocolos de aplicativos TCP/IP e como os programas host estabelecem uma interface com os serviços de camada de transporte para usar a rede.