DETECTORES DE DEFEITOS: DESAFIOS EM REDES MÓVEIS SEM FIO

Miguel Angelo Baggio

CENTRO DE TECNOLOGIA

CURSO DE CIÊNCIA DA COMPUTAÇÃO

DETECTORES DE DEFEITOS: DESAFIOS EM REDES MÓVEIS SEM FIO

TRABALHO DE GRADUAÇÃO

Miguel Angelo Baggio

Santa Maria, RS, Brasil

2008

DETECTORES DE DEFEITOS: DESAFIOS EM REDES MÓVEIS SEM FIO

por

Miguel Angelo Baggio

Trabalho de Graduação apresentado ao Curso de Ciência da Computação, da Universidade Federal de Santa Maria (UFSM, RS), como requisito parcial para obtenção do grau de

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Raul Ceretta Nunes (UFSM)

Santa Maria, RS, Brasil

Trabalho de Graduação nº. 253
2008

Universidade Federal de Santa Maria
Centro de Tecnologia
Curso de Ciência da Computação

A Comissão Examinadora, abaixo assinada,
aprova o Trabalho de Graduação

DETECTORES DE DEFEITOS:
DESAFIOS EM REDES MÓVEIS SEM FIO

elaborado por
Miguel Angelo Baggio

como requisito parcial para obtenção do grau de
Bacharel em Ciência da Computação

COMISÃO EXAMINADORA:

_____
Prof. Dr. Raul Ceretta Nunes (UFSM)
(Presidente/Orientador)

_____
Prof. Dr. João Baptista dos Santos Martins (UFSM)

_____
Prof. Msc. Rogério Corrêa Turchetti (UNIFRA)

Santa Maria, 28 de Janeiro de 2008

DEDICATÓRIA

Aos meus pais, que me incentivaram a lutar pelos meus objetivos, eles que mostraram o exemplo de força de vontade, persistência. São verdadeiros trabalhadores, detentores de características que marcam qualquer filho.

Aos meus irmãos, que sempre me apoiaram em tudo e que são e serão sempre os meus melhores amigos.

Aos meus amigos, que sempre estiveram por perto para as horas boas e as horas ruins.

A minha namorada, que viveu todo este tempo comigo me transmitindo muito carinho e atenção, compreensão e muito mais do que eu poderia imaginar de apenas uma pessoa.

A toda a minha família, que fazem com que o meu dia a dia seja sempre repleto de alegria e felicidade.

Obrigado!

AGRADECIMENTOS

Gostaria primeiramente de agradecer a minha família pelo apoio dado durante esse trabalho.

Agradeço também as pessoas que me deram a oportunidade de realizar este trabalho, ao professor e orientador Raul Ceretta Nunes, aos professores da banca: professor João Baptista dos Santos Martins e professor Rogério Corrêa Turchetti, e especialmente aos amigos e colegas Cezar Bilaco, Giovani Gracioli, Rodrigo Dewes.

Também agradeço a professora Iria Brucker Roggia, que possibilitou a realização de diversos trabalhos realizados na minha formação acadêmica.

RESUMO

Trabalho de Graduação
Curso de Ciência da Computação
Universidade Federal de Santa Maria

DETECTORES DE DEFEITOS: DESAFIOS EM REDES MÓVEIS SEM FIO

AUTOR: MIGUEL ANGELO BAGGIO
ORIENTADOR: RAUL CERETTA NUNES
Data e Local da Defesa: Santa Maria, 28 de Janeiro de 2008.

Sistemas distribuídos são caracterizados por computação concorrente e descentralizada, realizada sobre dois ou mais computadores ou dispositivos móveis que se comunicam através de uma rede de comunicação. Atualmente as redes sem fio são uma alternativa para as redes convencionais cabeadas, fornecendo as mesmas funcionalidades e facilidades de configuração. Essas redes combinam a mobilidade do usuário com a conectividade, abrindo um novo conjunto de aplicações.

Em redes sem fio falhas são inevitáveis, porém as conseqüências indesejadas podem ser evitadas pelo uso adequado de técnicas de tolerância a falhas. Para tolerar falhas, é importante detectar a presença de defeitos (percepção de um erro gerado por uma falha). Um detector de defeitos nada mais é que um algoritmo que fornece informações sobre suspeitas de defeitos em nodos ou processos monitorados. Em redes sem fio o desafio é distinguir a mobilidade dos nodos de um defeito.

O objetivo deste trabalho é explorar as principais características dos algoritmos de detecção de defeitos para redes móveis sem fio e colaborar com a melhora da precisão das informações fornecidas pelos detector de defeitos. Para tal é proposto um detector de defeitos que além de fornecer informações sobre sua lista de vizinhos fornece informações sobre o tempo total de falha dos nodos.

Palavras-chave: Tolerância à Falhas; Detectores de Defeitos; Sistemas Distribuídos.

ABSTRACT

Final Work
Computer Science
Federal University of Santa Maria

FAILURE DETECTORS: CHALLENGES IN WIRELESS MOBILE NETWORKS

AUTHOR: MIGUEL ANGELO BAGGIO
ADVISOR: RAUL CERETTA NUNES
Date and Place of Presentation: Santa Maria, January, 28th, 2008.

Distributed Systems are characterized by concurrent and decentralized computing, carried out by two or more computers or mobile devices communicating over a network. Today, wireless networks are an alternative to the conventional wire networks, offering the same functionalities and configuration facilities. These networks combine user mobility with its connectivity, opening new application fields.

On wireless networks faults are inevitable, but the unwanted consequences can be avoided by proper use of fault tolerance techniques. In fault tolerance detect failures is an important task. A failure detector is nothing more that an algorithm that provides information about monitored nodes or process suspected to be failed. On wireless network the challenge is distinguish node mobility from node failure.

The objective of this work is explore the main features of failure detector algorithms designed to wireless networks and cooperate to improve the accuracy of the information provided by failure detector. Thus, we propose a failure detector algorithm that show besides information on the list of its neighbors the total time of failure of the nodes.

Keywords: Distribute Systems; Failure Detectors; Wireless Mobile Networks; Fault Tolerance

LISTA DE FIGURAS

LISTA DE QUADROS

SUMÁRIO

1 INTRODUÇÃO

Sistemas Distribuídos é uma coleção de computadores independentes que se apresenta ao usuário como um sistema único e consistente. Esta coleção de computadores geralmente é interligada através de uma rede e os computadores são equipados com software que permite o compartilhamento dos recursos do sistema, sendo eles: software, hardware e dados (Tanenbaum e Steen, 2002).

Este sistema distribuído consiste em adicionar o poder computacional de diversos computadores, interligados por uma rede, para processar de forma cooperativa uma determinada tarefa. Como se fosse apenas um computador centralizado executando. Este sistema deve ter um cuidado especial para que as mensagens sejam entregues corretamente e as mensagens inválidas sejam rejeitadas. Também deve ter um cuidado especial para que os computadores que não estão mais disponíveis no sistema sejam identificados, seja por falhas ou por estarem fora de alcance.

Uma das áreas de pesquisa, em sistemas distribuídos, é a tolerância a falhas e monitoramento.

1.1 Contextos e Desafios

De acordo com WEBER (2001), falhas são inevitáveis, mas a conseqüência das mesmas, como por exemplo, o colapso do sistema ou interrupção no fornecimento do serviço e a perda de dados podem ser evitados pelo uso adequado de técnicas de tolerância a falhas. Tolerância a falhas é a propriedade que permite que sistemas continuem a operar adequadamente mesmo após falhas em alguns de seus componentes.

A tolerância a falhas é propriedade inerente em sistemas de alta disponibilidade ou aplicações críticas como as dedicadas à medicina. A tolerância a falhas não é uma propriedade somente para máquinas individuais, e pode caracterizar também a maneira como as máquinas interagem entre si. Um bloco básico para a construção de um mecanismo de tolerância a falhas em sistemas distribuídos é o detector de defeitos.

Segundo GRACIOLI e NUNES (2007), um detector de defeitos é uma parte importante para sistemas tolerantes a falhas. FELBER et al. (1999), diz que um detector de defeitos é um algoritmo distribuído que fornece informações sobre suspeita de defeitos em

componentes monitoráveis. Essas informações são fornecidas através de troca de mensagens entre os nodos do sistema que contem o detector de defeitos acoplado. Existem vários algoritmos de detecção de defeitos, como o algoritmo proposto por Hutle (2004) e Sridhar (2006) que surgiram com as redes móveis sem fio. Esses detectores de defeitos são adaptados para os desafios dessas redes, tais como manter uma informação correta sobre a situação dos nodos, se os nodos estão fora da área de alcance, ou se a mensagem não foi entregue corretamente. Algumas propriedades dos detectores de defeitos são abrangência (*completeness*) e exatidão (*accuracy*). A abrangência refere-se a cada nodo detectar todos os nodos que estão com defeitos, e a exatidão refere-se ao nodo não ter uma falsa suspeita, detectando um nodo que está em seu estado normal, como um nodo falho.

1.2 Objetivos do trabalho

Os objetivos deste trabalho são:

a) estudar algoritmos conhecidos de detectores de defeitos;

b) desenvolvimento de nova proposta corrigindo lacunas de soluções anteriores;

c) validar a proposta;

1.3 Contribuições deste trabalho

Neste trabalho estudaram-se os principais algoritmos de detecção de defeitos existentes para obter um melhor domínio do estado da arte em detecção de defeitos em redes ad hoc. Com base neste estudo são propostas alterações no algoritmo *Gossip Básico* (descrito na seção 3.2.1.1) para que seja possível obter o tempo total que cada nodo ficou como falho, possibilitando que as aplicações tenham mais informações no momento de escolha de qual nodo utilizar.

1.4 Estrutura do Texto

Este texto está organizado em seis capítulos.

O segundo capítulo fala sobre as redes móveis sem fio, que são os ambientes com foco principal dos detectores de defeitos estudados neste trabalho, este capítulo mostra alguns exemplos de tecnologias de redes móveis sem fio.

O capitulo três descreve os detectores de defeitos estudados, suas principais características e vantagens de cada solução.

O quarto capítulo apresenta o simulador escolhido para as simulações feitas com os detectores de defeitos, e mostra algumas vantagens sobre os outros simuladores, demonstrando porque da escolha do mesmo para a realização deste trabalho.

O quinto capítulo mostra o desenvolvimento que foi realizado, e alguns casos demonstrando os ganhos que podem ser atingidos com esta nova implementação.

O sexto capítulo descreve a validação experimental onde foram realizados testes e demonstrados os resultados do algoritmo.

A conclusão é apresentada no capítulo sete, juntamente com os trabalhos futuros.

2 REDES MÓVEIS SEM FIO

GRACIOLI e NUNES (2007) relataram que redes móveis sem fio são caracterizadas por obter acesso a informações de maneira simples e direta através de uma rede de comunicação sem fio. Essa comunicação é feita por uma transmissão de dados através de ondas eletromagnéticas (Silva, 1998).

A figura 2.1 mostra um sistema de rede móvel sem fio típica, que é geralmente utilizada em prédios e empresas para fazer a conectividade dos vários dispositivos internamente.

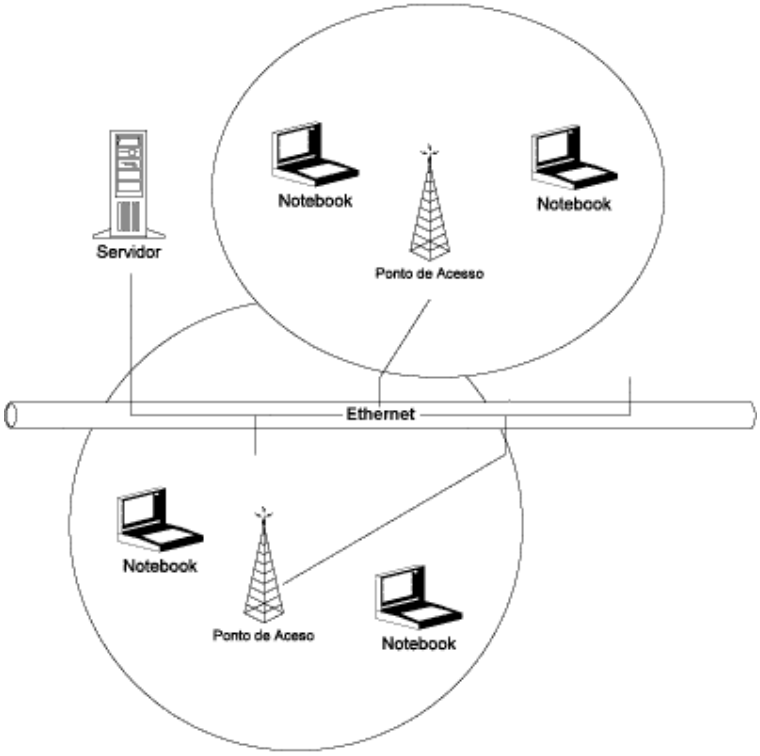

Figura 2.1 – Rede Wireless Típica. (Silva, 1998)

Nesta figura existem diferentes nodos (*notebooks*) por ponto de acesso que compartilham a rede *Ethernet*.

Existem várias tecnologias envolvidas nas redes sem fio, cada um com suas particularidades, limitações e vantagens.

2.1 Sistemas Narrowband

Sistemas Narrowband (banda estreita) são utilizados para fazer comunicação via ondas de rádio, essas ondas operam em uma freqüência de rádio específica configurada pelo usuário. Essa freqüência especifica é a freqüência do sinal. A transmissão dos dados é feita mantendo o sinal de rádio o mais estreito possível, sendo suficiente para conseguir transmitir as informações. É importante o conhecimento da utilidade do sinal para poder fazer a configuração correta para que seja possível utilizar a menor banda possível desde que seja utilizada uma largura de banda suficiente para transmitir os dados. O *crosstalk* indesejado entre os vários canais de comunicação pode ser evitado coordenando os diferentes usuários nos diferentes canais de freqüência (Silva, 1998).

2.2 Sistemas Spread Spectrum

Os sistemas Spread Spectrum utilizam a técnica de espalhamento espectral com sinais de rádio freqüência de banda larga, provendo maior segurança, integridade e confiabilidade, em troca de um maior consumo de banda. Há dois tipos de tecnologias spread spectrum: a FHSS (*frequency-hopping spread spectrum*) e a DSSS (*direct-sequence spread spectrum*).

A FHSS usa uma portadora de faixa estreita que muda a freqüência e o código conhecido pelo transmissor e pelo receptor, tendo como efeito a manutenção de um único canal lógico.

A DSSS gera um *bit-code* (também chamado de chip) redundante para cada bit transmitido. Quanto maior o chip transmitido maior será a probabilidade de recuperação da informação original, e também maior será a largura da banda necessária para que esse chip seja transmitido. Este mecanismo utiliza técnicas estatísticas embutidas no aparelho de transmissão, mesmo com falhas de um ou mais bits no chip sejam danificados durante a transmissão, não é necessário retransmitir os dados para recuperar os dados originais (Silva, 1998).

2.3 Sistemas Infrared

Este sistema utiliza técnicas de transmissão que utilizam freqüências muito altas, um pouco abaixo da luz visível no espectro eletromagnético. Igualmente a luz, o sinal infravermelho não pode penetrar em objetos opacos. Assim as transmissões em infravermelho ou são diretas ou difusas. Os sistemas infravermelhos diretos de baixo custo fornecem uma distância muito limitada, em torno de 1,5 metros e são comumente utilizados em PAN (P*ersonal Area Network*) (Silva, 1998).

2.4 Redes Ad Hoc

Redes ad hoc são caracterizadas pelos nodos terem a comunicação direta com os outros nodos sem a necessidade de criação de uma infra-estrutura. Os nodos comunicam-se sem uma conexão física, formando uma rede onde os dispositivos fazem parte apenas durante a comunicação, ou para dispositivos móveis, apenas quando estão dentro do alcance de transmissão. Essas redes podem ser formadas por qualquer aparelho que tenha um dispositivo *wireless*, como PDAs, *notebooks* ou celulares (Gracioli e Nunes, 2007).

Um dos maiores desafios nas redes ad hoc é encontrar uma rota para a entrega das mensagens, o que pode levar a uma mensagem não conseguir ser transmitida devido a não existência de pontos de acesso ou estações centrais. Isto faz com que cada nodo seja importante, pois cada nodo representa um ponto de acesso ou um caminho caso este ponto seja visível para dois ou mais nodos. Com a mobilidade dos nodos, existe uma grande dificuldade para conseguir criar rotas, pois com a movimentação dos nodos, o caminho que uma mensagem leva para trafegar de um nodo N1 até o nodo N2, pode mudar a todo instante.

O quadro 2.1 mostra algumas vantagens e desvantagens das redes ad hoc.

Quadro 2.1: Vantagens e desvantagens das redes ad hoc (Gracioli e Nunes, 2007)

Vantagens	**Desvantagens**
Mobilidade	Baixa banda passante
Fácil instalação	Maior perda de mensagens e erros
Comunicação direta entre os nodos	Problema em localizar um nodo
Recuperação rápida em caso de perda de um nodo	Topologia variável e consumo de energia

Essas redes locais sem fio já são realidade em vários ambientes de redes, principalmente nos que requerem mobilidade dos usuários.

Existem diversas aplicações para sistemas que utilizam as redes sem fio, como sistemas em hospitais onde o médico pode ter informações de diversos pacientes através de equipamentos que conseguem fazer uma comunicação sem fio, como outros diversos tipos de sistemas como fábricas e sistemas de monitoramento.

As redes móveis sem fio são caracterizadas pela escassez de recursos, como alcance de transmissão e consumo de energia (Gracioli e Nunes, 2007). Falhas de hardware e de comunicação são freqüentes (Macedo et al., 2005). As falhas ocorrem em virtude de eventos como a destruição de nós, degradação da qualidade do enlace, falha de comunicação devido a interferências ocorridas por movimentação dos nodos que podem ser bloqueados por objetos, bem como agentes maliciosos que tem como objetivo degradar o serviço da rede.

Utilizar algoritmos que são detectores de defeitos é importante para a implementação de técnicas e tolerância a falhas. Os detectores de defeitos são responsáveis sobre a informação fornecida do estado atual (está funcionando ou não) dos nodos em sistemas monitoráveis.

3 DETECTORES DE DEFEITOS EM SISTEMAS MÓVEIS

Conforme AGUILERA et al. (1997) os detectores de defeitos são uma importante abstração para viabilizar a implementação de protocolos tolerante a falhas em sistemas distribuídos. Diferentes autores fizeram implementações, cada autor procurou melhorar e deixar o seu algoritmo de acordo com um determinado objetivo.

Neste capítulo, será descrito as características dos diferentes algoritmos propostos pelos autores Renesse et al. (1998) , Hutle (2004), Hayashibara et al. (2005) e Sridhar et al. (2006). O estudo deste capítulo justifica-se para o entendimento do funcionamento de detectores de defeitos.

3.1 Definições

Segundo Hayashibara et al. (2005) os detectores de defeitos podem ser decompostos em três tarefas básicas: monitoramento, interpretação e ação.

A camada de monitoramento permite que o detector de defeitos obtenha informações sobre outros *hosts* e seus nodos. Isto é feito geralmente pela rede, por uma amostra de *heartbeats* que chegam ou por consultas que aguardam respostas.

A camada de interpretação é necessária para fazer sentido às informações obtidas através da camada de monitoramento. A camada de interpretação analisa o resultado obtido pelo monitoramento, e é responsável pela definição das ações a serem tomadas.

A camada de ação é executada como uma resposta às suspeitas causadas. Geralmente isto é feito pelas aplicações. Um exemplo disto é quando um nodo é detectado como falho pela camada de monitoramento. A camada de interpretação analisa o resultado e passa a informação para a camada de ação que executa a tarefa relacionada com o resultado que foi obtido pela camada de monitoramento.

3.1.1 Detectores de Defeitos Binários

Conforme Hayashibara et al. (2005) os detectores de defeitos binários são caracterizados por combinar duas tarefas básicas, que são as camadas de monitoramento e de

interpretação, proporcionando as informações já interpretadas para as aplicações. Assim, as aplicações têm o único papel de reagir com as suspeitas já interpretadas.

Com o detector de defeitos binário, a camada de interpretação é frequentemente realizada através de configurações de *timeouts* e geradores de suspeita. Os parâmetros utilizados (timeouts) modificam o resultado.

A figura 3.1 mostra como funciona um sistema com detector de defeitos binário.

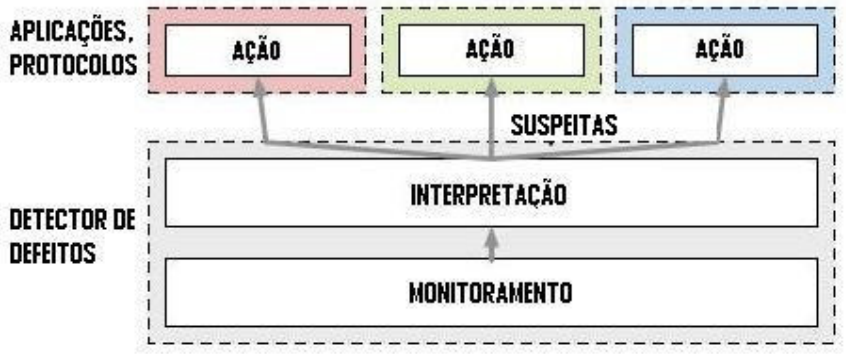

Figura 3.1 – Detector de Defeitos Binário

Podemos ver que o detector de defeitos faz o monitoramento e a interpretação e entrega para a aplicação a lista de suspeitos. A aplicação pode então tomar uma decisão do que fazer com esta lista de suspeitos.

3.1.2 Detectores de Defeitos Agregados

Os detectores de defeitos agregados deixam a tarefa de interpretar o nível de suspeita para as aplicações. Assim as diferentes aplicações podem ter diferentes tratamentos para nodos suspeitos de acordo com as suas necessidades, ou mesmo usar diretamente o nível de suspeita como um parâmetro para as ações que as aplicações irão tomar. Os detectores de defeitos agregados podem atuar da mesma forma que os detectores de defeitos binários, utilizando uma biblioteca, caso as aplicações preferirem este modelo de funcionamento (Hayashibara et al., 2005).

A figura 3.2 mostra como funciona um sistema com detector de defeitos agregados.

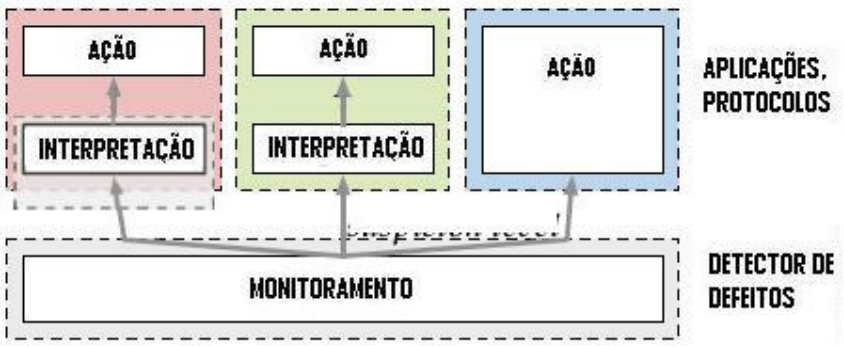

Figura 3.2 – Detector de Defeitos Agregado

Podemos ver que o detector de defeitos agregado possui três diferentes situações. Na primeira situação, o detector de defeitos faz o monitoramento sozinho e a parte de interpretação dos possíveis suspeitos ele faz junto com a aplicação. A segunda situação é quando o detector de defeitos faz a parte do monitoramento e deixa para a aplicação fazer a interpretação independente do detector de defeitos. A terceira situação é quando a aplicação recebe as suspeitas direto do detector de defeitos, que monitora e interpreta o nível de suspeita, atuando assim como um detector de defeitos binários. A aplicação apenas tem o papel de fazer a ação determinado para o resultado da saída do detector de defeitos.

O detector de defeitos agregado tem como saída números reais não negativos, onde cada valor corresponde a um nodo e representa o atual nível de suspeita do nodo.

3.2 Soluções Existentes

Esta seção apresenta as soluções existentes de detectores de defeitos, com as principais características e vantagens de cada solução.

3.2.1 Renesse

Renesse et al. (1998) propuseram um algoritmo baseado no protocolo Gossip. Este serviço fornece uma detecção de falha precisa com uma conhecida probabilidade de uma falsa detecção, e é resiliente contra a perda transiente de mensagens e com as divisões permanentes da rede, assim como as falhas do host. Este serviço utiliza dois protocolos separados que

fazem automaticamente uma análise da vantagem de topologia da rede, e tem um bom funcionamento com a escalabilidade.

O algoritmo de Renesse é baseado em *gossiping* (fofocar) randômico e segue as seguintes propriedades.

1. A probabilidade de um membro ser falsamente reportado (falsas suspeitas) é independente do número de nodos.
2. O algoritmo funciona bem quanto à perda de mensagens (particularmente, tempo de falha de entrega de mensagens) e falhas de nodos, e nisto uma pequena porcentagem de perda de mensagens ou pequena porcentagem de membros falhos não iniciar uma falsa detecção.
3. Se o movimento do clock local não influencia, o algoritmo detecta todos os nodos falhos ou os nodos não alcançados perfeitamente, com conhecimento probabilístico de erro.
4. O algoritmo escalona em tempo de detecção, e esse tempo de detecção aumenta *O* (nlogn) com o número de nodos.
5. O algoritmo escalona quando a rede é sobrecarregada, isto requer que a largura de banda seja aumentada linearmente com o aumento do número de nodos. Para redes extensas (grandes), a largura de banda usada nas sub-redes é aproximadamente constante.

3.2.1.1 Protocolo básico (*Basic Gossip*)

O protocolo básico *Gossip* funciona da seguinte maneira: um membro passa adiante uma nova informação para membros escolhidos aleatoriamente. Cada membro tem uma lista com os nodos conhecidos com seus endereços e um inteiro que vai ser usado para a detecção de defeitos e o último tempo que o contador de *heartbeat* foi alterado. Este inteiro é chamado de contador *heartbeat*. A cada T_{gossip} segundos, cada membro aumenta seu próprio contador de *heartbeat*, e seleciona outro membro aleatoriamente para enviar sua lista. Quando o respectivo membro recebe a mensagem, ele combina com a sua lista adotando o maior valor de *heartbeat* para cada membro encontrado.

Inicialmente os membros enviam uma mensagem para todos os outros membros com o objetivo de reconhecer ou detectar os membros participantes da rede ou sub-redes a que pertencem.

Se um membro não tiver seu contador de *heartbeat* alterado em T_{fail} segundos, então o membro é considerado falho. T_{fail} é escolhido de uma maneira que nenhum nodo faça uma detecção de defeitos errônea ou maior do um pequeno percentual $P_{mistake}$.

Quando um nodo é detectado como falho, ele não pode ser retirado da lista de membros imediatamente. Se um membro B for retirado da lista de nodos A porque foi detectado como falho e depois A recebe uma mensagem *gossip* de B, então A vai adicionar novamente o B como se tivesse visto ele pela primeira vez. O nodo A continua mandando uma mensagem *gossip* com o membro B ativo na sua lista novamente, e isto faz com que o membro B nunca seja retirado da lista de nodos.

Para lidar com isto, o detector de defeitos não remove o membro falho da sua lista até $T_{cleanup}$ segundos, com $T_{cleanup} > T_{fail}$. $T_{cleanup}$ é escolhido de uma maneira que a probabilidade que uma mensagem *gossip* recebida sobre um membro, após ele ser detectado como falho, é menor que um pequeno percentual $P_{cleanup}$.

Em outras palavras, considere a seguinte situação: digamos que B é um membro que falhou, e consideramos A um membro que escutou o último *heartbeat* de B no tempo t. Com probabilidade P_{fail}, qualquer outro membro vai ouvir o último *heartbeat* de B no tempo $t+T_{fail}$, e então qualquer nodo irá ter B com falho no tempo $t+2\times T_{fail}$. Então, se configurarmos o $T_{cleanup}$ para $2\times T_{fail}$, com a probabilidade P_{fail} , nenhum membro que falhou vai reaparecer na lista de A uma vez que este membro tenha sido removido.

3.2.1.2 Protocolo Modificado

Renesse et al. (1998) também sugeriram uma modificação para o algoritmo funcionar em sistemas de grande escala, chamado de *multi-level gossiping.* O protocolo básico é ineficiente para os sistemas de grande escala, e pode ser melhorado facilmente, com uma importante melhora na escalabilidade. O problema é que os membros escolhem aleatoriamente os outros membros, sem conhecimento da topologia da rede na qual eles estão conectados. Como resultado, os hosts que são *bridge* (pontes) entre as redes físicas são sobrecarregados com muita informação redundante. Os membros podem detectar automaticamente os limites das sub-redes de domínios da internet, reduzindo as mensagens *gossip* que atravessam esse limite.

Existem dois aspectos neste protocolo modificado. Primeiro o tamanho das sub-redes e o tamanho do número de hosts para cada domínio é passado na mensagem *gossiping* junto

com o contador *heartbeat* de cada host. Segundo, as mensagens são feitas em grande parte nas sub-redes, com poucas mensagens entre as redes e muito menos entre os domínios.

Sobre o primeiro aspecto, a maioria dos *hosts* mantém duas listas. Uma é a mesma do protocolo básico, e contem a lista dos hosts e seus contadores de *heartbeat*. A segunda lista contém o número de domínios e de sub-redes que determinam o tamanho da sub-rede e o número dos hosts.

Detalhe importante deste segundo protocolo é que para cada *host* encontrado na primeira lista, ele tem que estar na segunda lista também, que corresponde ao domínio do host e a sub-rede. Então como tipicamente muitos hosts estão na mesma sub-rede, a segunda lista vai ser muito menor. Esta segunda lista é enviada junto com a primeira, e é misturada na chegada.

Nas sub-redes, é utilizado o protocolo básico. Para mensagens que vão através das sub-redes e domínios é utilizada uma versão modificada do protocolo. Este protocolo modificado melhora a probabilidade de transmissão de uma mensagem *gossiping* para cada rodada entre as sub-redes. Em média, um nodo por sub-rede vai fazer uma transmissão *gossiping* para outro domínio ou sub-rede. Com isso, a largura de banda utilizada entre cada nível da sub-rede vai ser proporcional ao número de entidades contidas em outros níveis. Por exemplo, a largura de banda utilizada através da sub-rede vai depender apenas do número de sub-redes contidas no domínio. Para manter este comportamento, cada nodo gera um valor maior a cada rodada, quando ele envia uma mensagem *gossip*.

Depois de n tempos, onde n é o tamanho da sub-rede, ele pega aleatoriamente outra sub-rede do seu domínio, e depois de selecionar a sub-rede, ele seleciona aleatoriamente um nodo para enviar a mensagem *gossip*. O membro que tem o maior valor, mas desta vez com probabilidade $1/(n \times m)$, onde m é o número da sub-rede do seu domínio, ele pega aleatoriamente outro domínio, e então uma sub-rede aleatória com este domínio, e um *host* aleatório desta sub-rede para enviar a mensagem *gossip*.

Este protocolo reduz significativamente a largura de banda utilizada que é desperdiçada com as mensagens que seriam transmitidas através das ligações entre as sub-redes, desde que a mensagem *gossip* seja concentrada nas sub-redes. O protocolo permite também um tempo acelerado de detecção de falhas com sub-redes, e é mais resiliente (elástico) em oposição a partições de redes.

3.2.2 Hutle

Hutle (2004) tem um algoritmo que implementa um detector de defeitos para redes particionáveis esparsamente conectadas. Os nodos podem comunicar-se apenas com os seus vizinhos, e entre os vizinhos existe um limite superior no movimento irregular. Existe um número conhecido de vizinhos que é assumido ser Δ, o qual é um modelo adequado para redes wireless ad-hoc. Este algoritmo não requer um conhecimento a priori dos nodos no sistema, apenas precisa conhecer o limite superior do atraso de comunicação entre os nodos arbitrários, chamado de *jitter*. Cada nodo envia apenas Δ+1 mensagens para seus vizinhos por rodada, e sob a suposição de um constante tamanho de sub-rede e de domínio, essas mensagens têm tamanho constante.

Os nodos possuem informação precisa sobre os outros nodos mais próximos e informação não tão precisa sobre os nodos que estão à maior distância. É possível também alterar o algoritmo para os sistemas onde as ligações (links) podem ser recuperadas. Entretanto, em tais sistemas a definição de acessibilidade não é tão obvia, desde que uma aplicação do detector de defeitos possa usar um algoritmo de roteamento para a comunicação. A relação de informação mais precisa sobre os nodos que estão em outros níveis de sub-redes também dependeria do comportamento deste algoritmo de roteamento. Uma solução seria procurar algoritmos que dependem do detector de defeitos, mas operam diretamente nas redes esparsas, outros para derivar condições de conectividade na topologia para vários algoritmos de roteamento.

Neste protocolo, cada nodo possui uma tabela *heartbeat* para cada nodo conhecido:

- um contador *heartbeat* $\mathrm{hbc}_p[q]$ que contem o mais recente heartbeat de q.

- um contador de distância $distance_p[q]$ que contêm a estimativa de p´s sobre a atual distância para q.

- um *time-stamp* $last_p[q]$ que mantêm a ultima rodada que p recebeu um novo contador de q.

Estas tabelas aumentam dinamicamente de acordo com cada novo nodo aprendido por p. Por razões de simplicidade eles são usados no algoritmo como se fossem alocados estatisticamente. Existe um conjunto chamado $detected_p$ que contêm todos os nodos que o detector de defeitos não suspeita, e isto é o resultado do detector de defeitos.

Este algoritmo envia uma lista para seus atuais vizinhos, e recebe uma tarefa que é atualizar a lista local quando ele recebe um novo *heartbeat* dos vizinhos. Inicialmente, p

conhece apenas ele mesmo, e a cada T etapas, p aumenta o seu próprio contador, que é também usado como um o número do *round* local. A cada Δ^k rounds, todos os nodos conhecidos com distância k são colocados no conjunto $unsent_p$ para que sejam enviadas apenas as mensagens deste conjunto. Com isso ele garante que cada *heartbeat* de um nodo com distância k é transmitido no mínimo a Δ^k rounds.

Deste conjunto $unsent_p$, o id do nodo, *heartbeat*, e a distância Δ+1 dos nodos com menor distância de p são enviados e removidos de $unsent_p$ em cada rodada. Se p não recebe uma atualização de outro nodo por um tempo muito longo, um nodo que ele previamente detectou, ele então irá suspeitar do nodo.

O receptor da tarefa de p aumenta o contador de distância para um a cada mensagem que ele recebe. Se o contador de distância é menor que sua estimativa, ele adapta a nova distância. Quando ele recebe um *heartbeat* mais novo que os seus, ele adapta este *heartbeat*.

Um detector de defeitos para sistemas particionáveis é eventualmente perfeito, se sua saída de detecção cumpre as seguintes propriedades:

- Forte Integralidade: para qualquer dois nodos que fiquem desconectados (incluindo o caso de um deles ter falhado), onde um nodo suspeita de outro, existe um tempo que o nodo ativo fica desconfiando do nodo que detectou como falho.

- Eventual Precisão Forte: para qualquer dois nodos que ficam permanentemente conectados, existe um tempo que eles permanentemente suspeitam um do outro.

Esta classe de detectores de defeitos é denotada por $\Diamond$P.

Como a maioria dos outros algoritmos, o algoritmo de Hutle tem como saída uma lista de nodos. Visto que não requer que os nodos conheçam todos os outros nodos do sistema, ou mesmo n a priori, o detector de defeitos não terá como saída todos os nodos suspeitos. Ao invés disso, ele tem como saída uma lista de nodos, que é complementar e equivalente a lista de suspeitos, no sentido de que é possível determinar se um nodo específico esta atuando ou não. Assim, um nodo p suspeita de outro nodo q, se ele não está na sua lista de detecção. Formalmente, o histórico do detector de defeitos é uma função H(p, t) onde se um nodo q está em H(p, t) em um tempo t é dito que p detecta q, senão p suspeita de q.

3.2.3 Sridhar

Sridhar et al. (2006) propuseram um algoritmo detector de defeitos na presença de mobilidade em sistemas distribuídos. Este algoritmo é voltado para detecção de defeitos em redes locais, sendo um perfeito detector de defeitos eventual que tolera a mobilidade dos nodos, desde que esta mobilidade divida a rede. Este algoritmo tem integridade forte e exatidão eventual forte local. Este detector de defeitos mantém apenas informação sobre seus vizinhos imediatos, reduzindo a quantidade de memória necessária para armazenar dados sobre os nodos vizinhos (que é um recurso escasso em alguns sensores), em comparação com detectores de defeitos que armazenam informações sobre todos os nodos da rede.

Este protocolo é composto por duas camadas independentes. A primeira camada é o detector de defeitos local, responsável por construir a lista de suspeitos entre os vizinhos de um dado nó, que é chamada de Camada de Detector de Falhas Local (LFD – *Local Failure Detection Layer*). A segunda camada é a que detecta a mobilidade dos nodos através da rede, e esta camada é chamada de Camada de Detecção de Mobilidade (MB – *Mobility Detection Layer*). Juntas, essas duas camadas satisfazem as especificações de redes móveis, que são: forte integridade local, exatidão eventual forte local e localização suspeita.

A forte integridade local (*strong local completeness*) diz que tem que existir um tempo depois que cada nodo p que falha é permanentemente suspeito por qualquer outro nodo vizinho e correto q.

A exatidão eventual forte local (*eventual strong local accuracy*) refere que deve existir um tempo onde depois que nodos que estão ativos não são mais suspeitos de nenhum outro nodo ativo da vizinhança, os nodos atualizam sua visão de quem são os seus vizinhos.

A localização suspeita (*suspicion locality*) refere-se há um tempo depois que nodos corretos apenas suspeitam de nodos que estão na vizinhança local.

3.2.3.1 Camada de detecção local

A camada de detecção de falha local funciona da seguinte maneira: em cada nó p, a camada de detector de defeitos mantém informações de quais nodos em um conjunto $nbrs_p$ *(neighboards* de p) *são* suspeitos de falhar. Esta camada também monitora quando cada um destes nodos foi recentemente suspeitado. Este momento é marcado como td_q para nodos suspeitos de q, e este *timestamp* é um *timestamp* local. A camada LFD pode ser implementada

usando qualquer algoritmo de detectores de defeito da classe ◊P. Isto depende muito da necessidade e dependência da aplicação, o projetista pode escolher uma de varias estratégias construir a lista de suspeitos da lista de vizinhos. Esta camada não tem nenhuma preocupação com os vizinhos. As estratégias mais populares para garantir a detecção de falhas são:

- *Heartbeats*: cada nodo envia uma mensagem "*I am alive"* para cada um de seus vizinhos. Cada nodo também mantêm uma lista de seus vizinhos que ele já verificou, por meio da mensagem recebia "*I am alive"*. Por exemplo: Se um nodo p não obtém informação sobre outro nodo q por um período específico, p assume que q falhou, e então adiciona q a sua lista de suspeitos. Se depois de colocar q na sua lista de suspeitos, p receber um *heartbeat* de q, ele vê que cometeu um erro e retira o q da sua lista de suspeitos.

- *Adaptive Timeouts* (Timeouts Adaptáveis): cada nodo p mantém um timeout adaptável para cada nodo q vizinho. Se p suspeita de q indevidamente após o tempo *Wait Time* $WT_{p,q}$ e depois escuta q com um atraso $delay_q$, então p atualiza o período de *timeout* de q para ser longo o suficiente para que este engano não seja repetido. O novo tempo $WT_{p,q}$ é agora no mínimo $WT_{p,q} + delay_q$. Com isso cada nodo mantém estendido com o tempo que o *timeout* é suficiente longo para manter todas as formas de atrasos acidentais.

- *Pinging*: quando o nodo pergunta para o módulo detector de defeitos sobre a lista de suspeitos, o detector de defeitos envia para cada nodo vizinho a mensagem "*Are you alive"* perguntando se o nodo está vivo, esta mensagem é chamada de mensagem ping. Se ele recebe uma mensagem de resposta "*I am alive"* antes de um tempo específico, o vizinho não é adicionado a lista de suspeitos, caso contrário ele é adicionado a lista de suspeitos. A complexidade deste tipo de estratégia é o dobro de mensagens trocados se comparada com a estratégia *heartbeat*, levando em consideração que o número de vezes que o ciclo de detecção tem que ocorrer pode ser reduzido.

- *Lease* (arrendar ou alugar): em aplicações onde os nodos dormem a maioria do tempo (como sensores de rede), nenhumas das estratégias listadas anteriormente fazem sentido. Neste contexto, a função pode ser inversa, cada nodo p envia para cada vizinho q a mensagem "*I am alive*". Além disso, p também envia para q um pedido para alugar por um tempo ld_p. Agora p pode ir dormir, e tudo que ele tem que fazer é acordar antes que o tempo ld_p acabe, e enviar um pedido de renovação para alugar novamente para todos os seus vizinhos.

3.2.3.2 Camada de detecção de mobilidade

Cada nodo p executa esta camada para compartilhar sua visão de nodos falhos com o resto da rede e corrigir suas suspeitas baseadas nas informações sobre o que os outros nodos conseguem detectar. Devemos lembrar que existe apenas uma mensagem *gossip* na rede em um dado momento. No momento de posicionamento, algum nodo é escolhido como o iniciador. O iniciador para rodadas posteriores é nomeado no final de cada rodada.

Quando um nodo ocioso p recebe uma mensagem *gossip* de um nodo q, p coloca q como seu guardião (gerador) e vai para o estado ativo. Depois de acordar, é examinado o grupo suspeito que seja contido na mensagem. Ele compara a nova lista recebida de suspeitos com sua própria lista de suspeitos para ver se tem muito conflito de entradas. Caso a lista recebida contenha qualquer vizinho de p que não esteja na lista de suspeitos de p, então p procura ver a quanto tempo o nodo está sendo marcado como suspeito. Baseado neste tempo, o módulo detector de falhas de q decide se o nodo suspeito deve permanecer na lista de suspeitos ou não. Se o tempo decorrido desde a última comunicação de q é menor que o tempo de suspeita de q então exonera q, este nodo é removido da lista de suspeitos.

Uma situação que pode ocorrer é quando um nodo p suspeita de um nodo r que é vizinho de q, onde p e r não estão diretamente ligados, assim p não enxerga r. Quando p recebe a lista de q ele não suspeita mais de r, pois tem informação de que r está vivo, e sai da lista de suspeitos de p.

Quando um nodo recebe uma mensagem *gossip* e não tem nenhum vizinho para mandar a mensagem, então ele manda a mensagem de volta para quem o enviou. Mais uma vez cada nodo atualiza sua lista de suspeitos baseado na lista de suspeitas recebida pela mensagem *gossip*. Quando um nodo recebe de volta as mensagens de todos os seus filhos, ele envia de volta a mensagem para o seu guardião. Lembrando que cada nodo só espera a resposta dos vizinhos conhecidos. O round de *gossip* termina quando o nodo que iniciou escuta a mensagem de volta de todos os seus vizinhos. Assim no final de cada round, cada nodo tem uma visão atualizada dos seus vizinhos, se estão ativos ou não.

A fase de expansão do *gossip* é usada para exonerar nodos. Um nodo r inferior na árvore de propagação exonera um nodo q que é suspeito por um de seus ancestrais p. Na fase de encolhimento, p corrige sua lista de suspeitos e sua lista de vizinhos para remover q. Porém, mesmo se r não foi exonerado, p ainda vai precisar remover q da sua lista de vizinhos, senão a camada de detector de defeitos de p vai continuar suspeitando de q, e p vai continuar

em um estado ruim, impossibilitado de fazer um progresso local, mesmo tendo todos os seus vizinhos ativos.

Lorenzi et al. (2007), chegou a conclusão que o detector de defeitos de Sridhar teve um desempenho melhor que os outros detectores de defeitos para detecção de falhas.

3.2.4 Hayashibara

Hayashibara et al. (2005), sugeriram um detector de defeitos agregado como um serviço básico, que combina monitorização e interpretação. O detector de defeitos agregado separa essas duas tarefas e tem como saída um nível de suspeita ao invés de um valor binário, e deixa para as aplicações interpretarem esses valores. Idealmente, o acompanhamento é feito por um único serviço sendo executado em cada máquina, enquanto que a interpretação do nível de suspeita é deixada para cada nodo da aplicação.

Este tipo de serviço pode ser implementado como um *daemon* (um programa de computador executado em *background*), uma biblioteca ou um serviço do kernel, dependendo da troca desejada entre desempenho e intromissão.

Hayashibara também apresenta importantes condições para que o nível de suspeita sob a qual um detector de defeitos agregado é computacionalmente equivalente a um eventual perfeito detector de defeitos binário.

Esta equivalência é importante porque mostra que o detector de defeitos agregado não esconde nenhuma suposição de sincronia adicional com respeito a suas contrapartes binárias. Entretanto, a equivalência não implica que o detector de defeitos agregado não seja mais eficiente ou expressivo que os detectores de defeitos binários. Na verdade, Hayashibara argumenta extensivamente as vantagens de arquitetura do detector de defeitos agregado, e apresenta uso de padrões que são difíceis de lidar com a utilização de um detector de defeito binário.

3.2.4.1 Limitações dos Modelos Binários

Os modelos binários têm algumas limitações quando se trata de proporcionar um detector de defeitos como um serviço genérico. Um modelo binário de iteração binário torna difícil suportar diversas aplicações rodando simultaneamente. Na prática, é preciso perceber que há uma troca natural entre detector de defeito conservativo (lento e com maior precisão) e

agressivo (rápido e com menor precisão), pois as diferentes aplicações são suscetíveis de diferentes necessidades com relação ao esperado resultado do detector de defeitos, além de que diferentes níveis de resultados podem ser úteis dentro de um mesmo aplicativo.

Um exemplo é um aplicativo poder tomar medidas preventivas contra falhas catastróficas quando confiança em um suspeito atinge um determinado nível, e pode tomar medidas mais drásticas quando o nível de confiança atinge um segundo nível.

Detectores de defeitos binários são bem adaptados para atender as necessidades de muitos algoritmos, e seus modelos de iteração não podem lidar facilmente com alguns padrões de uso que podem ser atingidos na prática. A seguir na seção 3.5.2 são apresentadas duas situações desses padrões de uso.

3.2.4.2 Exemplos: BoT (Bag of Tasks) – Saco de Tarefas

Para mostrar que nem todos os exemplos de detectores de defeitos binários podem atender as necessidades, Hayashibara executou uma série de computações que chamou de BoT (*Bag of Tasks*) na sua plataforma OurGrid. Este exemplo ajuda a mostrar dois padrões interessantes de utilização de detectores de defeitos.

Consideramos um ambiente simplificado com um nodo mestre e uma coleção de nodos trabalhadores. O mestre mantém uma lista de tarefas independentes que necessitam ser executadas, envia essas tarefas para os trabalhadores disponíveis e colhe os resultados. Por simplicidade, assumimos que o nodo mestre nunca é falho, mas alguns dos trabalhadores podem falhar. O mestre deve ser capaz de detectar a falha de um trabalhador e re-alocar a tarefa que foi designada para este trabalhador, ou então a computação nunca poderá ser completada.

Consideramos agora as seguintes situações onde o mestre precisa usar a informação sobre a possível falha dos trabalhadores:

Situação um: Quando o mestre designa tarefas para os trabalhadores, o mestre deve evitar enviar as tarefas para os trabalhadores que tenham falhado. Com isto, o mestre precisa saber selecionar trabalhadores de acordo com a probabilidade de que eles ainda estejam operacionais.

Situação dois: Quando uma tarefa está sendo executada por um trabalhador, e este trabalhador falha, a falha deste trabalhador deve ser detectada e a tarefa deve ser reiniciada. Consideramos este custo para a tomada de decisão errada: se uma tarefa é abortada

erroneamente, todos os ciclos de CPU que foram gastos na computação da tarefa são desperdiçados. Podemos notar que o custo de abortar uma tarefa devido a uma suspeita errada aumenta com o passar do tempo.

Estas duas situações escritas acima são difíceis de lidar com detector de defeito binário. Embora soluções ad hoc certamente existam, uma abstração mais adequada pode simplificar o projeto e então aumentar a qualidade do sistema. Já existe uma tentativa de definir tal abstração, chamada *slowness oracle*, que lida com a primeira situação ordenando os nodos de acordo com a percepção de velocidade dos nodos. Porem, *slowness oracle* não lida bem com a segunda situação.

3.2.4.3 Detector de Defeitos Agregado

Para lidar com as situações descritas na seção 3.4.2, Hayashibara defendeu um modelo de ação mais flexível para os detectores de defeitos, que podem ser construídos em cima dos detectores de defeitos binários e outros tipos de detectores de defeitos.

Hayashibara criou uma família de detectores de defeitos chamada Detectores de Defeitos Agregados, com o qual cada associação de nodos monitora, para cada um dos nodos monitorados, um número real que muda ao longo do tempo. Este valor é o nível de suspeita do nodo. Quanto maior o valor, maior é a suspeita do nodo, e quanto menor o valor, menor é o nível de suspeita. Para exemplificar, quando o valor de suspeita é zero, isto quer dizer que o nodo não é suspeito por ninguém.

Detectores de defeitos agregados garantem que o nível de suspeita de um nodo monitorado P aumenta para o infinito se P é falho, e determina se P é correto.

3.3 Análise as soluções anteriores

O detector de defeitos proposto por Renesse tem a vantagem reduzir a troca de mensagens entre os membros da mesma sub-rede, e também tem a modificação do protocolo para funcionar com os detectores de defeitos que tem que mandar mensagem entre as sub-redes. Ele reduz a quantidade de mensagens que é trocada entre os nodos em cada rodada.

O Hutle propõem um detector de defeitos que envia um número conhecido de mensagens por rodada, o que torna fácil de configurar a quantidade de banda necessária para a

transmissão de mensagem de cada nodo, e ele também traz a vantagem de ter um *timeout* que é adaptado quando um nodo não responde no tempo necessário.

Sridhar demonstra uma solução dividida em duas camadas, que são a camada de detecção local e a camada de mobilidade.

O detector de defeitos de Hayashibara traz uma nova característica para os detectores de defeitos, que é a chamada arquitetura de detectores de defeitos agregados. Este novo detector de defeitos pode operar de três maneiras diferentes, e traz também a informação da porcentagem de um membro estar ativo.

Os detectores de defeitos propostos, sem incluir a proposta de Hayashibara et al. (2005), mantêm informações sobre os nodos apenas para saber se o seu estado atual está ativo ou falho. Estes detectores de defeitos não estão preocupados em saber qual foi o tempo que o nodo que falhou ficou detectado como falho, ou em qual momento que teve sua última falha, deixando a desejar neste requisito.

Estas informações podem ser muito importantes para as aplicações que fazem uso dos detectores de defeitos. Sabendo o tempo que um nodo ficou como falho e o momento que ele teve sua última falha, podem ser criadas diferentes regras de ação que podem ser tomadas baseadas nessas novas informações.

4 SIMULADOR JIST/SWANS – JAVA IN SIMULATION TIME / SCALABLE WIRELESS AD HOC NETWORK SIMULATOR

O JiST é um simulador de eventos de alto desempenho que é executado sobre uma máquina virtual Java padrão. Foi desenvolvido para construir simulações de eventos e seu uso foi bastante difundido devido a algumas características do seu projeto que trouxeram os seguintes benefícios:

- utilizar uma linguagem já consolidada para a plataforma de simulação;
- não criar uma biblioteca de simulação;
- não desenvolver um novo *kernel* de sistema para simulação;

O JiST cria sistemas de simulação que podem executar com eficiência (sistemas de simulação altamente otimizados), transparência (simulações são automaticamente transformadas para rodar com semânticas de tempo) e padronização (capacidade de escrever simulações com um sistema convencional de linguagem de programação) utilizando apenas as linguagens e ambientes de execução já desenvolvidas, como o Java (Barr e Renesse).

O quadro 4.1, mostra as soluções para diferentes aproximações de construções de simulações.

Quadro 4.1: Comparações do JiST e outros Simuladores

	Kernel	Library	Linguagem	JiST
Transparência	✓		✓	✓
Eficiência		✓, X	✓, X	✓
Padrão	✓	✓		✓

Legenda da tabela:

✓ - Possui.

X – Não possui.

Nesta tabela, a eficiência refere-se à execução que compara favoravelmente o existente e altamente otimizado sistema de simulação. A transparência implica que simulações são automaticamente transformadas para executar com semânticas de tempo de simulações. Padrão denota que as simulações são escritas em um convencional sistema de linguagem de programação, por oposição a um domínio específico de linguagem designada explicitamente

para simulação. Com base nas características e facilidades do simulador, foi escolhido utilizar o JiST/SWANS como o simulador para testar o funcionamento dos algoritmos.

5 ALGORITMO CONSISTENTE PARA A DETECÇÃO DE DEFEITOS EM REDES MÓVEIS

Este capítulo apresenta o algoritmo proposto para detecção de defeitos em redes móveis ad hoc. O capítulo está dividido em 4 seções. A primeira explica o modelo de sistema, a segunda explica o algoritmo proposto, a terceira mostra exemplos e a quarta seção mostra as classes e o diagrama de classes.

O objetivo deste trabalho foi oferecer uma nova abordagem para os detectores de defeitos. Incluindo dois novos atributos para a lista de vizinhos, chamado TempoFalho e ÚltimaFalha, este algoritmo pode dar uma maior área de escolhas para as aplicações lidarem com os resultados dos detectores de defeitos. O atributo TempoFalho, é designado para que cada nodo tenha um valor que determine o total de tempo que este nodo ficou falho. O atributo ÚltimaFalha, é para manter a informação da última rodada que o nodo falhou.

O nodo agora tem estas informações, podendo alterar a maneira de seleção de nodos confiáveis ou não de uma aplicação.

5.1 MODELO DE SISTEMA

O sistema baseia-se em um conjunto de 30 nodos, conectados por uma rede não totalmente conectada. A topologia da rede são grafos que podem ser descritos por nodos $G(N)=(V(N),t)$, onde $V(N)$ é o conjunto de todos os nodos N que são vizinhos e estão ativos dentro de um intervalo t. Essa ligação é direta, ou o nodo estão no raio de transmissão ou está como inativo. São assumidas apenas falhas de omissão de mensagens, que abrange a categoria de falhas de transmissão. Um nodo deve emitir uma resposta em um determinado intervalo, se a resposta não chegar o nodo é considerado como falho. Os nodos que são conectados diretamente são chamados vizinhos. Pode existir mais de um grupo de grafos formando diferentes grupos, sendo possível um nodo estar presente em mais de um grafo.

É considerado na simulação um canal TCP e UDP para a camada de transporte e MAC IEEE 802.11b, onde todas as mensagens são entregues, sem falha de comunicação ou modificação de mensagem.

Os processos fazem a comunicação com os seus vizinhos através de troca de mensagens via *broadcast.* Este processo é feito através de uma mensagem enviada, e então se

espera a resposta em um determinado intervalo de tempo. Caso um nodo não responder no determinado intervalo, ele é adicionado na lista de suspeitos do nodo, se o nodo responder a mensagem, são comparadas as listas de vizinhos, e é mantido o valor mais atual do contador de *heartbeats* de cada vizinho.

Se um nodo estava na lista de suspeito, e volta ao campo de transmissão do nodo, é então adicionado o tempo que este nodo esteve fora ao seu contador de tempo falho.

5.2 ALGORITMO PROPOSTO

Com as características já existentes nos algoritmos estudados, pode-se também juntar a informação do último *heartbeat* escutado, que geralmente é utilizado um atributo chamado *timestamp* (contido na maioria dos detectores de defeitos estudados) com o atributo TempoFalho e ÚltimaFalha, para poder fazer uma nova análise e tomar uma decisão com mais dados sobre os nodos.

Juntando estas duas informações, podem-se criar regras para confiar num nodo e podem-se ter mais opções na escolha que as aplicações devem fazer a partir dos resultados dos nodos se estes atributos forem utilizados.

Utilizando o algoritmo *Gossip* Básico (seção 3.2.1.1) foi desenvolvido um novo algoritmo onde cada nodo envia sua lista via broadcast para todos os outros nodos no seu raio de transmissão. Foram feitas alterações no *Gossip* Básico e inserido atributos novos (TempoFalho e ÚltimaFalha) na lista de atributos dos nodos vizinhos. Desta forma o algoritmo pode obter um resultado diferenciado do algoritmo original. Para a implementação funcionar de maneira correta, outro campo foi necessário ser adicionado, o último tempo de atualização do TempoFalho. Sem este campo, o controle da informação TempoFalho estava sendo contado incorretamente, devido ao um nodo atualizar o tempo de falha e enviar este tempo para outro nodo vizinho. Este nodo vizinho, depois de receber a mensagem com a informação de falha já atualizada, estava somando novamente no mesmo tempo de detecção uma nova falha para este nodo, e às vezes os nodos tinham mais tempo como falho do que o tempo total de execução do algoritmo. Com este informação, foi possível controlar este tipo de situação.

5.2.1 Pseudo Código

Segue o pseudo-código do algoritmo:

1. **Variáveis**
2. Heartbeat
3. Lista de Vizinhos
4. Lista de Suspeito
5.
6. **Inicializa**
7. **A cada T intervalo de tempo faz**
8. Aumenta o contador de heartbeat local
9. Envia mensagem para vizinhos
10. Recebe mensagem dos vizinhos
11. Se Lista de Suspeitos contem Endereço do Nodo que enviou Mensagem
12. ENTAO:
13. Se *heartbeat* local <= *Heartbeat* recebido então
14. Atualiza Heartbeat.
15. Remove o suspeito.
16. Atualiza o tempo de falha
17. FIM ENTAO
18 Detecta Suspeitos ()
19.
20. **Detecta Suspeitos**
21. Para cada vizinho da lista local faz:
22. Se TempoAtual - *Timestamp* do vizinho < Tgossip
23. ENTAO
24. Adiciona o vizinho na lista de suspeitos.
25. TempoUltimafalha = TempoAtual

5.3 Exemplos

Uma aplicação tem como regra que não designará tarefa para nodos que estejam com o nível de falha maior que 20 segundos.

Esta aplicação, depois de rodar o algoritmo com 100 rodadas, obteve como resultado dois nodos: P com 10 segundos de falha Q com 25 segundos de falha. Com isto, o nodo Q é descartado e o nodo P é escolhido. Essas regras podem ser aplicadas em casos onde é necessário mandar tarefas para nodos, e obter o resultado das tarefas, se estas foram completadas ou não. Caso algum nodo apresentar alguma falha, ele não precisa refazer todas as tarefas que foram designadas para ele, e pode utilizar o tempo que ele ficou como falho para ver as tarefas que precisam ser executadas por ele novamente.

Pode ser criadas regras como: mínimo de 70 segundos ativo e o tempo de ÚltimaFalha sendo menor que os 20 segundos restante como ativo para acabar a execução.

Com duas informações a mais na lista de atributos dos nodos vizinhos, é possível ter diferentes tipos de situações para definir a qual nodo designar uma tarefa.

Foram realizados testes do algoritmo, que não teve nenhum desempenho melhor na detecção de defeitos ou no tempo de execução, mas obteve mais informações sobre os nodos. Estas novas informações são uma boa fonte para avaliar a confiabilidade de um nodo no seu nodo vizinho. O resultado do algoritmo *Gossip* Básico, pode ser configurado para retornar uma lista dos vizinhos falhos do nodo ou dos vizinhos ativos do nodo. Este resultado é muito simples.

Se quisermos ter uma avaliação do comportamento total do nodo em um determinado intervalo de tempo, não será possível ter está informação, podendo ter apenas a informação do último intervalo de tempo que foi visto como ativo (*timestamp*).

Se um nodo ficar 70 segundos iniciais do tempo inativo, e nos últimos 30 segundos do tempo este nodo ficar ativo, o algoritmo *Gossip* Básico pode retornar este nodo como confiável.

A seguir, está à saída de um dos testes realizados, com tempo total de execução de 100 segundos.

Sou o nodo 0.0.0.1 e tenho os seguintes vizinhos:
- |0.0.0.15 e tenho o tempo de falha igual a: 0
- |0.0.0.4 e tenho o tempo de falha igual a: 25
- |0.0.0.19 e tenho o tempo de falha igual a: 10

- |0.0.0.8 e tenho o tempo de falha igual a: 15
- |0.0.0.11 e tenho o tempo de falha igual a: 25
- |0.0.0.16 e tenho o tempo de falha igual a: 0
- |0.0.0.18 e tenho o tempo de falha igual a: 20
- |0.0.0.3 e tenho o tempo de falha igual a: 20
- |0.0.0.12 e tenho o tempo de falha igual a: 5
- |0.0.0.17 e tenho o tempo de falha igual a: 20
- |0.0.0.2 e tenho o tempo de falha igual a: 15
- |0.0.0.13 e tenho o tempo de falha igual a: 30
- |0.0.0.9 e tenho o tempo de falha igual a: 20
- |0.0.0.6 e tenho o tempo de falha igual a: 20
- |0.0.0.14 e tenho o tempo de falha igual a: 15
- |0.0.0.10 e tenho o tempo de falha igual a: 15
- |0.0.0.5 e tenho o tempo de falha igual a: 30
- |0.0.0.0 e tenho o tempo de falha igual a: 10

Analisando os resultados, existem nodos que tem até 30 segundos de tempo em um estado de falha para o nodo 0.0.0.1. Também pode ser visto que existem apenas dois nodos que ficaram 100 segundos como ativo, que foram os nodos 0.0.0.15, que foi o nodo 0.0.0.16 que apresentaram o tempo de falha igual a zero.

5.4 Classes

O algoritmo tem duas classes: a classe chamada StartUpGossip e a classe SH.

5.4.1 Classe StartUpGossip

A classe StartUpGossip é a classe responsável pela criação do ambiente de virtual de execução do algoritmo. Nesta classe é definida a quantidade de nodos que vão ser utilizados, o tamanho do campo da simulação, o tempo total de execução do algoritmo, a potência da transmissão, o tempo Tgossip, o tempo Tcleanup e o tempo Tfail. É onde são definidos o tipo de mobilidade dos nodos, são os nodos se mexem ou não e o tipo de perda.

5.4.2 Classe SH

Essa classe é onde esta o algoritmo de detecção de defeitos. Com os parâmetros recebidos pela outra classe, o detector de defeitos segue as idéias do algoritmo, que são: aumentar seu contador, transmitir mensagem, atualizar a lista de nodos vizinhos e detectar nodos falhos.

A classe SH possuí uma estrutura chamada *NeighbourEntry* que é responsável por armazenar os dados dos vizinhos. A estrutura guarda as informações que os detectores de defeitos precisam manter sobre os seus vizinhos, com as seguintes informações: *NetAddress* que é a informação do endereõ do vizinho, *heartbeatCounter* que é o contador de *heartbeats, timestamp* que é o tempo do último incremento do contador, conterror que é responsável por armazenar a quantidade de tempo que o nodo ficou como falho e timestampfalha que é responsável por guardar o momento que o nodo teve sua última falha.

Cada nodo mantém uma lista de vizinhos e uma lista de suspeitos. A mensagem que vai ser enviada é controlada pela classe SH, e ela contêm a lista de nodos vizinhos.

Existe uma função para o recebimento das mensagens. Esta função faz a verificação da lista de vizinhos recebida com a sua lista de vizinhos local.

Se o endereço da lista recebida estiver na sua lista de suspeitos, então o nodo é removido, pois o nodo não está mais como falho, e se ele recebe um endereço que não está na sua lista de suspeitos e também não está na lista de vizinhos local, ele adiciono o novo nodo na sua lista de vizinhos local.

Se o nodo foi removido da lista de nodos suspeitos, então é atualizado o tempo que o nodo ficou como falho. Para isso ser feito, é necessário ver qual o tempo que ele teve seu último *timestamp* de falha atualizado. É adicionada no contador de tempo de falha total (conterror) a diferença de tempo de última falha atualizado (timestampfalha) menos o tempo da rodada atual (timestamp).

Existe o caso também de o nodo receber a lista de vizinhos e esta possuir o valor mais atual do contador de *heartbeats* do vizinho. Nesse caso, é atualizado o *heartbeat* da lista de vizinho local.

Para a detecção dos nodos suspeitos, existe uma função chamada *DetectSuspects* que verifica se um nodo esta falho ou não. Nesta função, é feita a comparação dos tempos, se o

tempo atual menos o *timestamp* do nodo vizinho é maior ou igual ao tempo de falha, então o nodo vizinho é adicionado na lista de nodos falhos.

Após a rodada, o nodo imprime a sua lista de vizinhos e o tempo falho de cada um. Se um nodo estiver com o tempo falho igual a zero, significa que o nodo nunca falhou durante a execução dos testes.

5.4.3 Diagrama de Classes

Este diagrama de classes tem o objetivo de mostrar os diversos objetos e classes do sistema e o relacionamento entre eles. As funcionalidades de cada função pode ser vista no Apêndice B.

No diagrama de classes, podemos ver que a classe StartUpGossip possui as funções de criar um nodo e de criar a simulação. A função *CreateSim* é responsável por chamar o algoritmo *BasicGossip*, que possui a estrutura *NeighbourEntry* que é a estrutura que contêm as informações de um vizinho e as funções *Receive* (responsável pelo recebimento da mensagem), *RemoveSuspeito* (responsável por retirar o nodo ativo da lista de suspeitos), *SendGossipMessage (*responsável por enviar a mensagem com a lista de vizinhos), *detectSuspects* (responsável por ver se um nodo está ativo ou falho) e a função *ListaTudo* (responsável por imprimir na tela o tempo de falha total da lista de vizinhos de cada nodo no final da simulação).

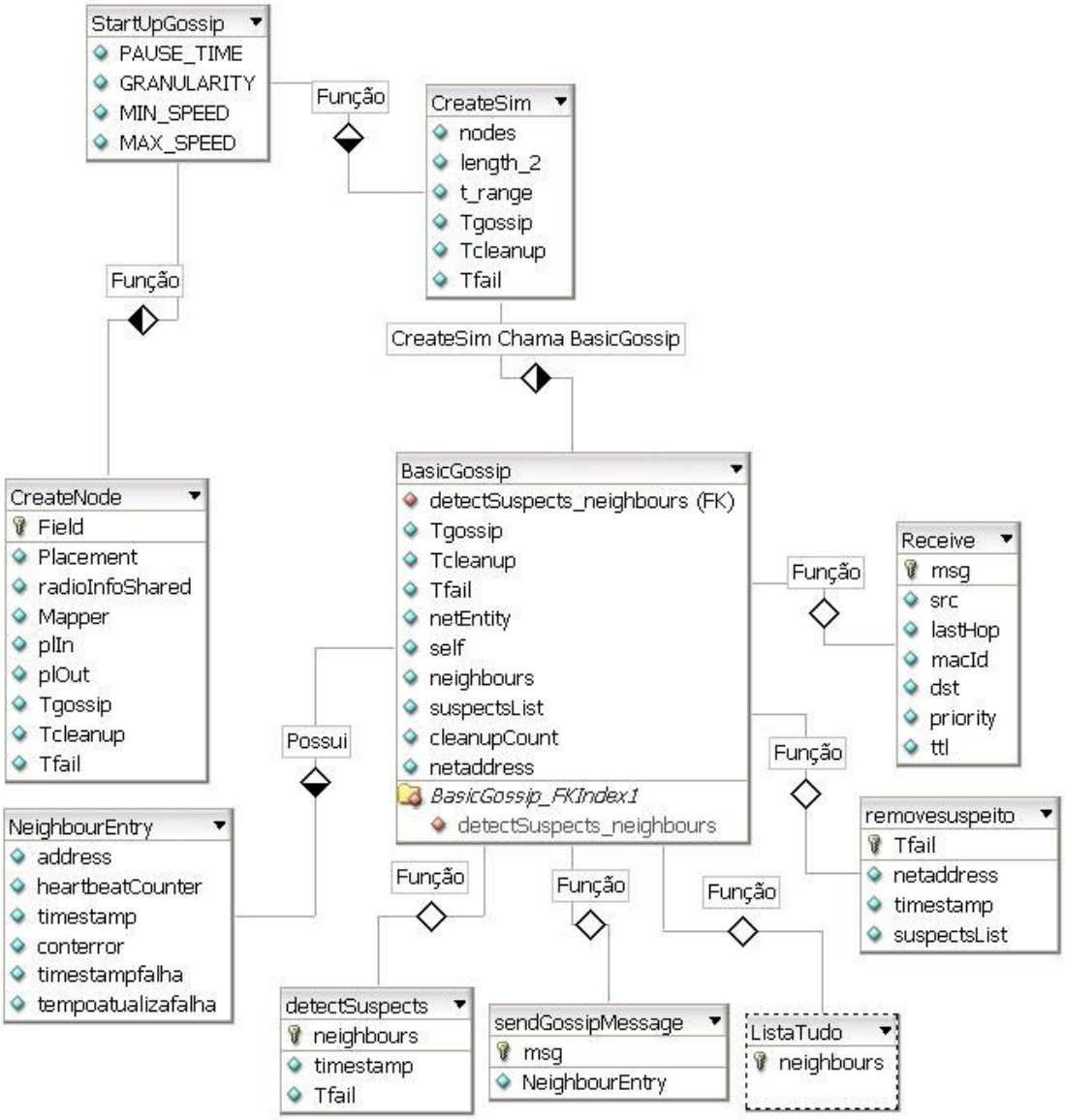

Figura 5.1 – Diagrama de Classes

6 VALIDAÇÃO EXPERIMENTAL

Os testes do algoritmo foram realizados utilizando o JiST/SWANS. Para rodar os algoritmos, é necessário baixar os arquivos .jar do site do JiST/SWANS, ter uma máquina virtual Java instalada no computador e adicionar os caminhos corretos do JiST/SWANS nas variáveis de ambiente. Todas essas configurações podem ser retiradas do site do simulador JiST/SWANS.

A seguir, a figura 6.1 mostra uma tela de configuração que foi utilizada para realizar uma simulação:

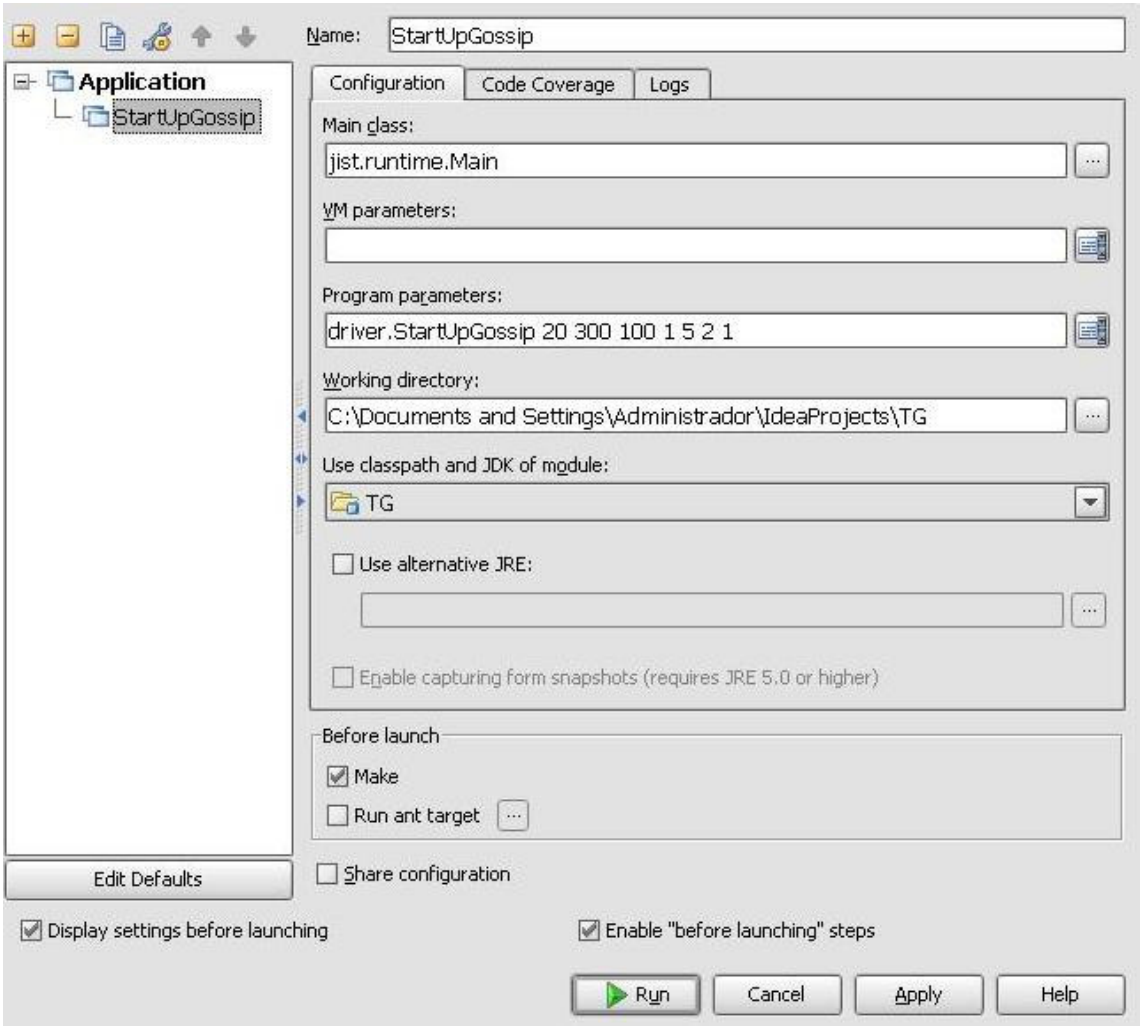

Figura 6.1 – Tela de Configuração

Na figura 6.2, é utilizado o seguinte comando: “jist.runtime.Main driver.StartUpGossip 20 300 100 1 5 2 1”. Para que uma simulação funcione corretamente, deve estar configurado os seguintes endereços das bibliotecas, como listados na figura 6.2.

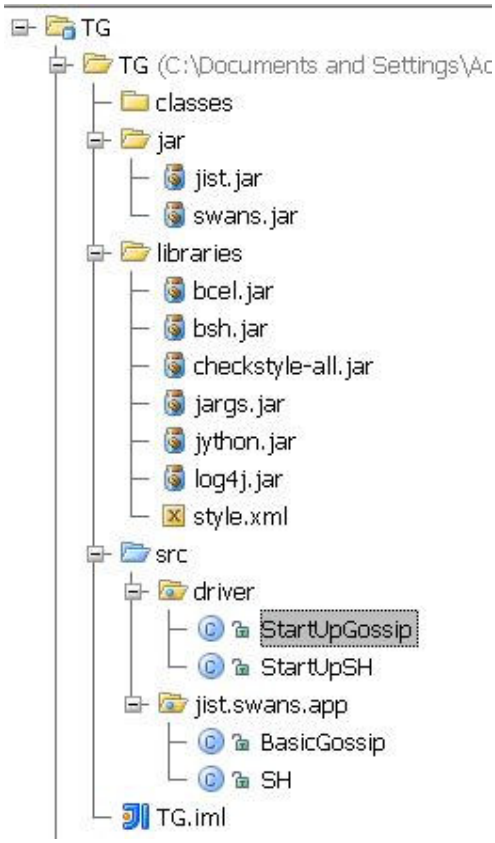

Figura 6.2 – Configuração das bibliotecas

Podemos notar que na figura 6.2 as bibliotecas estão configuradas conforme o padrão que é proposto no site do simulador JiST/SWANS. É importante que seja seguido este padrão para que a execução funcione corretamente.

As simulações foram realizadas em apenas um computador, mas elas podem ser realizadas utilizando mais de um computador ou outros dispositivos que suportem a comunicação via troca de mensagens. Existem diversas tecnologias que permitam que um sistema possa trocar informações entre diversos computadores, como *Sockets*, JGroups, RPC e RMI.

Os testes foram executados como se fossem realizados dentro de uma rede ad hoc que utiliza a quantidade de nodos definida pelo usuário na hora de executar a simulação.

Podemos confirmar este fato analisando a tela de execução do algoritmo no momento que um nodo imprime a sua lista de vizinhos. Cada vizinho possui um endereço de rede que modifica apenas o último valor do seu endereço em relação aos seus vizinhos, e não existe um valor repetido nesta lista de valores. Os últimos valores variam de 0 até o número de nodos utilizados na simulação.

6.1 Plataforma de Hardware

Para a execução da simulação, foi utilizado um computador desktop com um processador AMD Athlon 2800++, com 1 GB de memória DDR 333

6.2 Plataforma de Software

Foi utilizado o software Windows XP Professional com *Service Pack* 2. Também foi utilizada a máquina virtual Java padrão na versão 1.6.0_03. O simulador JiST/SWANS foi utilizado em sua versão v1.0.6. A linguagem de programação utilizada foi Java.

6.3 Modelo de Falha

A simulação baseia-se em falhas de comunicação, esta falha também conhecida como falha de omissão, onde não existe uma resposta sobre um determinado nodo. O processo baseia-se em cada nodo enviar uma notificação do seu estado com sua lista de vizinhos em um determinado tempo, se não obter uma resposta de outro nodo vizinho num tempo finito, ele deve perceber esta falha. Os nodos da simulação têm um tempo de vida determinado no inicio da simulação. Todos os nodos possuem o mesmo tempo de vida, e todos devem responder as mensagens desde o inicio da simulação. Se um nodo não responder, é porque está fora do alcance de transmissão e ocorreu uma falha de omissão.

6.4 Testes

Em cada simulação, foi escolhido um total de 30 nodos para serem simulados durante a execução da simulação. A movimentação dos nodos é feita de forma que tenha um caminho aleatório, com o modelo *RandomWaypoint* que é configurado em uma das classes do algoritmo. O *RandomWaypoint* tem uma velocidade de movimentação mínima e máxima e um tempo que fica parado até começar a movimentar-se novamente, que foram definidas como zero (nodo parado) até 2ms.

Este modelo é um dos modelos que estão disponíveis para ser utilizados pelo JiST/SWANS, e já vem compilado nas bibliotecas do JiST/SWANS, que é responsável por fazer a mobilidade da simulação.

Foram realizados diferentes testes para as simulações. Os valores escolhidos como parâmetros para a execução foram escolhidos de acordo com parâmetros proposto por Friedman et al (2003) e por GRACIOLI e NUNES, que fizeram um estudo e análise dos melhores valores para serem utilizados nas simulações.

Os valores escolhidos para as simulações tem as seguintes variações:

- número de nodos de 30,
- alcance de transmissão de 50m,
- timeouts que variam de 10s a 30s,
- tempo de varredura da lista (*tcleanup*) igual ao tempo de *timeout* até duas vezes o tempo do *timeout*.

Todos os resultados são apresentados com base nos vizinhos do nodo 29, que foi escolhido para ser utilizado como base de resultados dos testes. O tempo total de execução dos testes utilizado em todas as execuções foi de 1800 segundos (30 minutos).

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 10s e tempo de detecção da falha igual a 12s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.3.

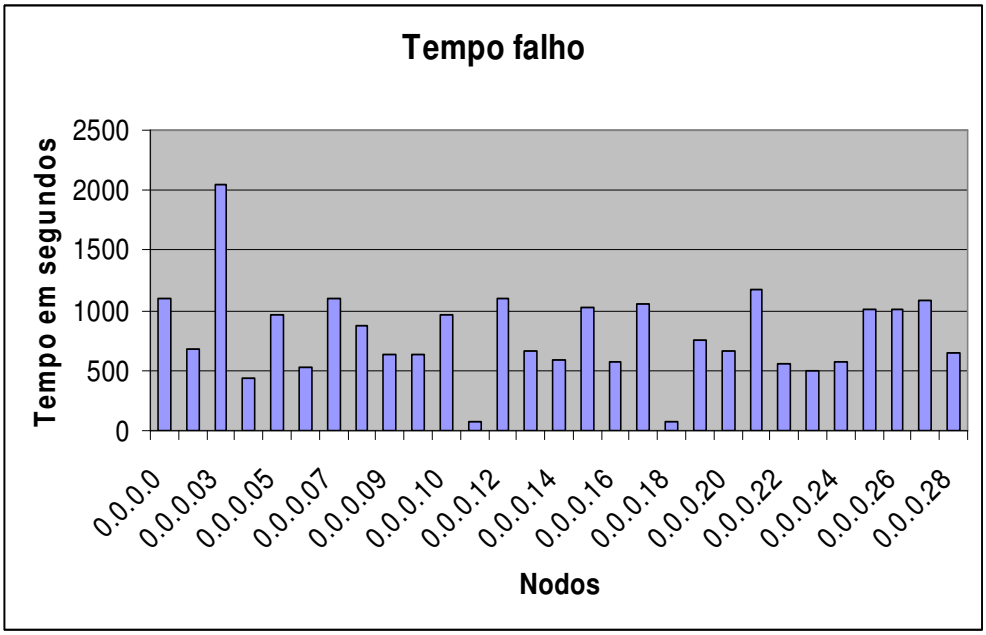

Figura 6.3 – Resultado 1

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 10s e tempo de detecção da falha igual a 15s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.4.

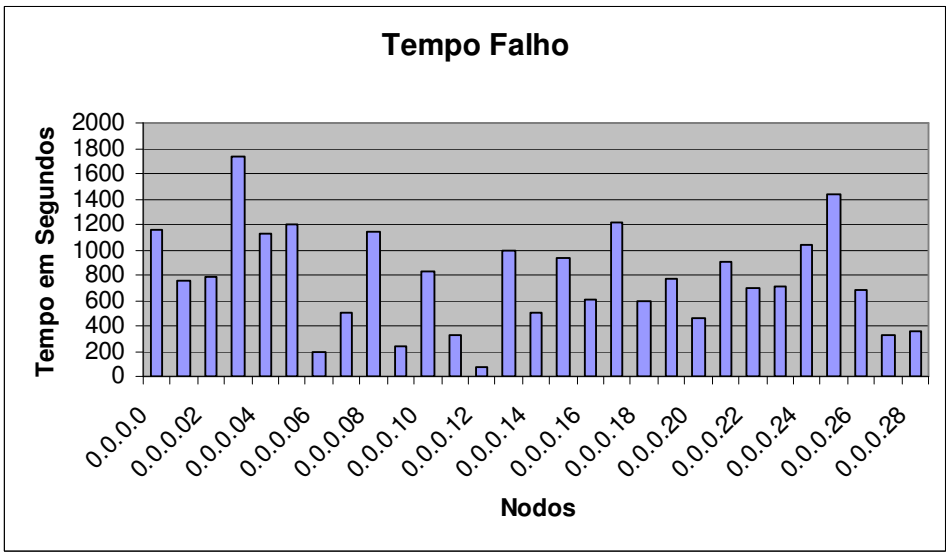

Figura 6.4 – Resultado 2

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 10s e tempo de detecção da falha igual a 18s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.5.

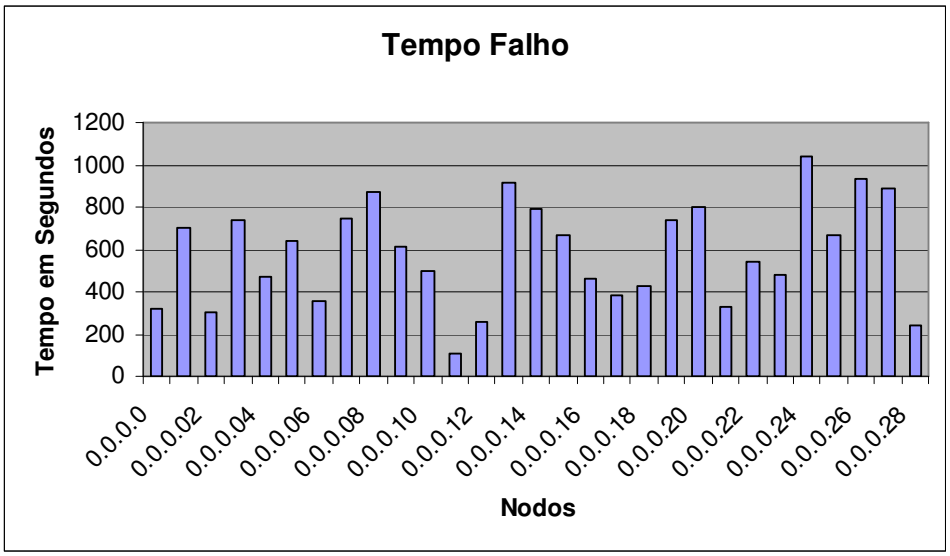

Figura 6.5 – Resultado 3

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 10s e tempo de detecção da falha igual a 20s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.6.

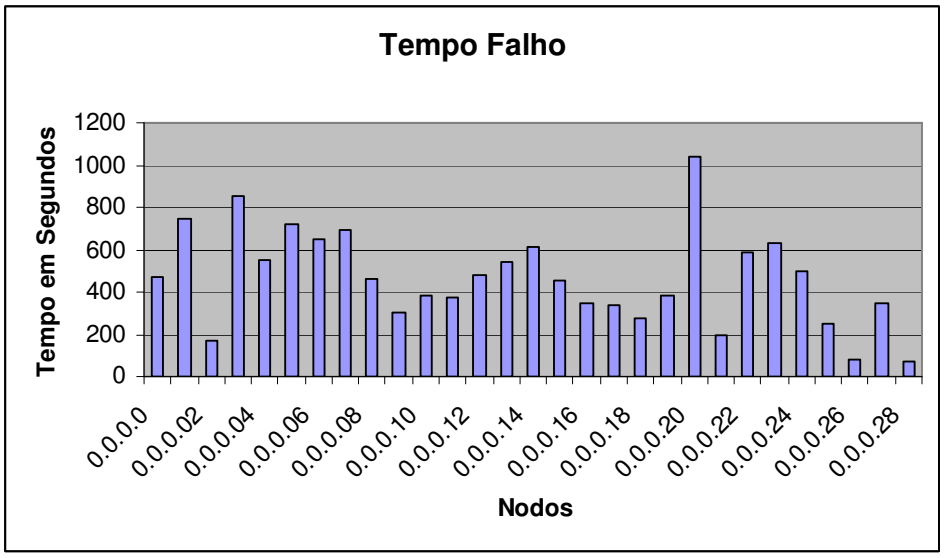

Figura 6.6 – Resultado 4

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 10s e tempo de detecção da falha igual a 20s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.7.

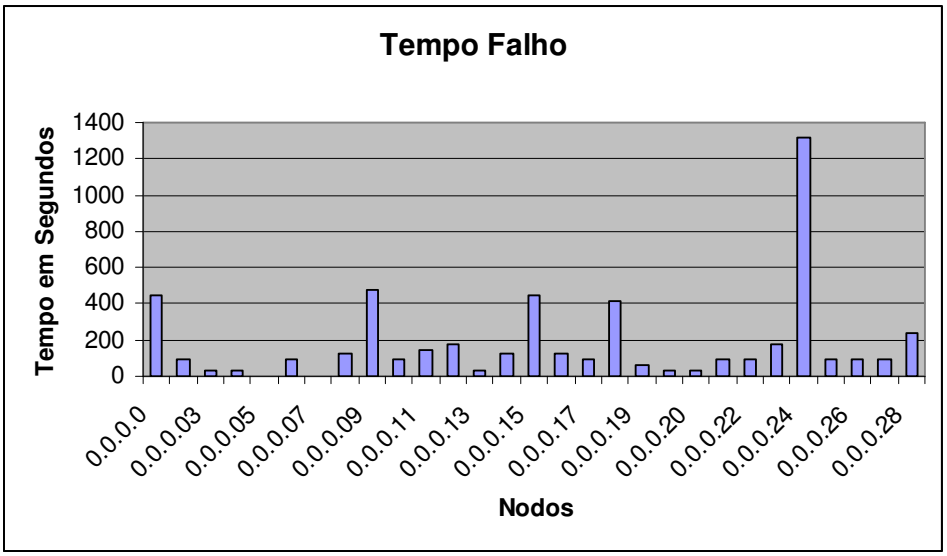

Figura 6.7 – Resultado 5

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 30s e tempo de detecção da falha igual a 35s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.8.

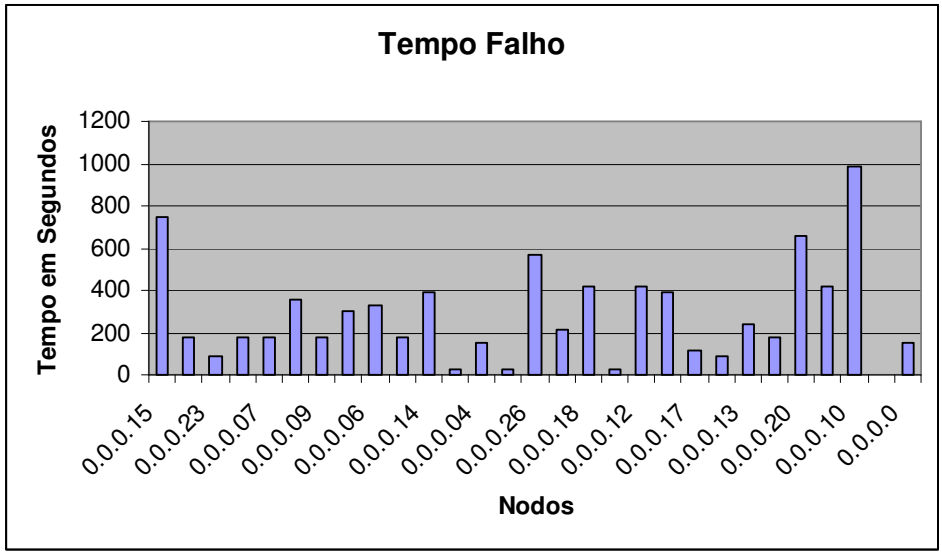

Figura 6.8 – Resultado 6

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 30s e tempo de detecção da falha igual a 50s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.9.

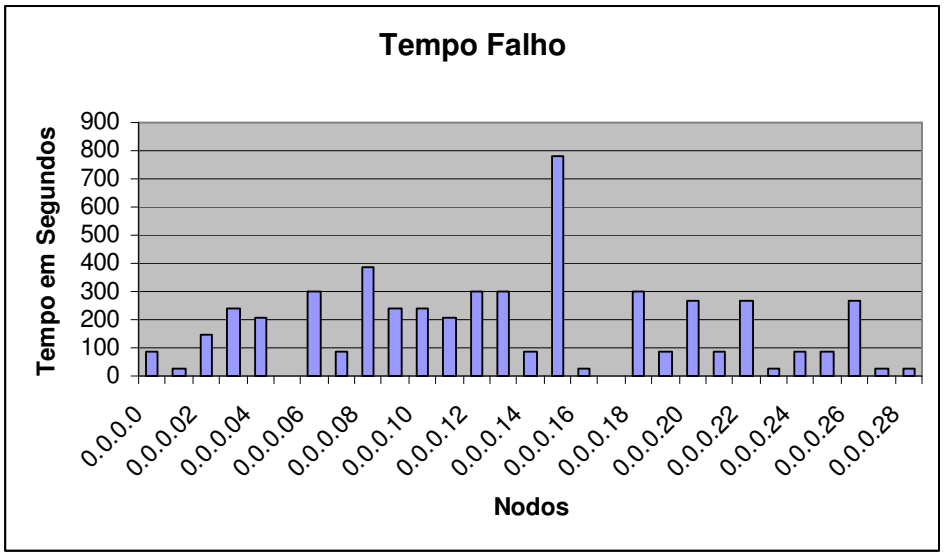

Figura 6.9 – Resultado 7

Com os seguintes parâmetros de entrada, com o tempo de *timeout* igual a 30s e tempo de detecção da falha igual a 60s, o algoritmo apresentou o seguinte resultado que pode ser visto na figura 6.10.

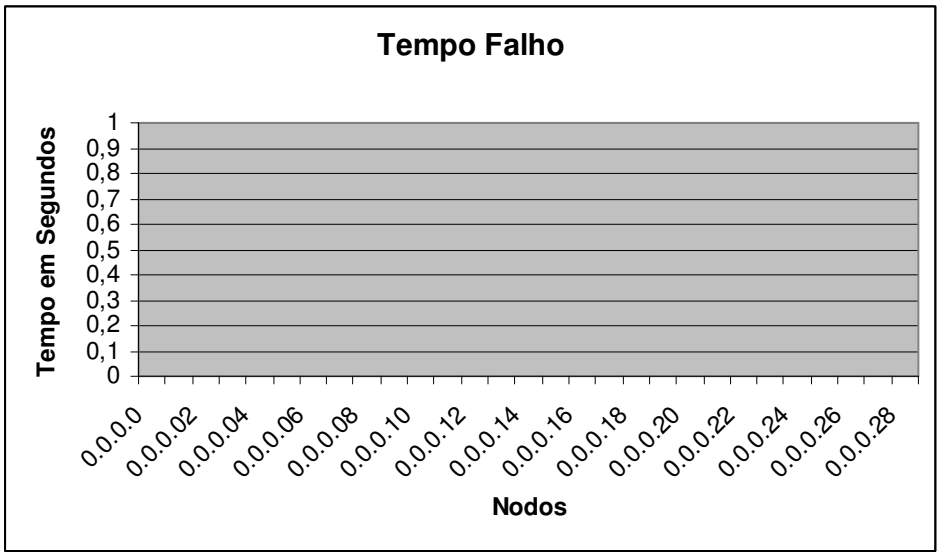

Figura 6.10 – Resultado 8

7 CONCLUSÃO

Levando em consideração o uso dos detectores de defeitos por aplicações que devem determinar suas ações a partir de resultados obtidos através de um determinado tempo, ou na última vez que este nodo falhou, esta nova implementação é uma boa opção, pois lida com mais situações do que apenas com o estado atual de um nodo. As aplicações podem ter mais regras de avaliação, pois o detector de defeitos fornece informações sobre o estado dos nodos e também sobre o tempo que eles permanecem suspeitos.

Esta nova proposta de algoritmo pode ser utilizada para a camada de detecção de defeitos local proposto por Sridhar (2006) e pode ser utilizada também para a camada de detecção de mobilidade. Com isto, as duas camadas podem ter mais informações sobre os nodos vizinhos. Se um nodo vizinho está ativo ou se está móvel, e quanto tempo um nodo vizinho está se movendo e está ativo e não está ao alcance de transmissão.

Os gráficos apresentados no capítulo 6 apresentaram dados que possibilitam chegar as seguintes conclusões:

- se usarmos um tempo muito grande para cada rodada, e utilizarmos o dobro do tempo para o tempo da detecção da falha, vemos que o detector de defeitos não detecta nenhuma falha, pois o nodo pode ficar falho e voltar a ativa dentro do intervalo da rodada, o que não é uma boa medida, pois pode estar omitindo falhas, como mostra a figura 6.10;

- quanto maior o intervalo entre as rodadas, menor o número de falhas detectadas;

- utilizando um tempo de detecção de falha não muito distante do tempo de cada rodada, com o tempo de detecção de falha estimado em variações como: o tempo de cada rodada mais a metade do tempo de cada rodada até o dobro de cada rodada (em casos onde o tempo de cada rodada não é um valor muito alto), o algoritmo consegue detectar mais falhas; e

- quanto mais próximo o tempo de detecção do tempo de cada rodada, o algoritmo consegue obter mais falhas dos nodos, caso elas existam.

7.1 Trabalhos Futuros

Este trabalho foi executado em apenas um dos algoritmos, o *basic gossip*, e para trabalhos futuros, a sugestão é a implementação do tempo de falha para os outros algoritmos e a execução de testes com a mesma entrada, gerando assim gráficos onde a comparação pode ser feita com os outros algoritmos.

Também pode ser utilizado o algoritmo *basic gossip* modificado (com o tempo de falha de cada nodo) na execução do algoritmo de Sridhar, podendo utilizar o tempo de falha como opção para a determinação de tempo em que o nodo esteve como falho, ou esteve apenas como fora do alcance de transmissão.

REFERÊNCIAS BIBLIOGRÁFICAS

AGUILERA, M. K.; CHEN, W.; TOUEG, S.; In: MAVRONICOLAS, M.; TSIGAS, Ph.: Lecture Notes in Computer Science, Vol. 1320: **Distributed Algorithms**, Proc. of 11th International Workshop, WDAG'97, Saarbrücken, Germany, pages 126-140. Springer, September 1997.

BARR, R.; HAAS, Z. J.; RENESSE, R. van. **Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator (JiST/SWANS)**. Disponível em: <http://jist.ece.cornell.edu/>.

FELBER, P.; DÉFAGO, X.; GUERRAOUI, R.; OSER, P. **Failure Detectors as First Class Objects**. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS (DOA'99), 1999, Edinburgh, Scotland. Proceedings**. . .** [S.l.: s.n.], 1999. p.132–141.

FRIEDMAN, R.; TCHARNY, G.; **Evaluating Failure Detection in Mobile Ad-Hoc Networks**. Outubro (2003)

GRACIOLI, G.; NUNES, R. C. **Detecção de Defeitos em Redes Móveis Sem Fio: Uma Avaliação entre as Estratégias e seus Algoritmos**. In: Workshop de Testes e Tolerância a Falhas (SBRC/WTF), Belém/PA, 2007.

HAYASHIBARA, N.; DÉFAGO, X.; URBÁN, P.; KATAYAMA, T. **Definition and specification of accrual failure detectors**. In Proc. IEEE/IFIP Intl. Conf. on Dependable Systems and Networks (DSN'05), pp. 206-215, Yokohama, Japan, June 2005.

HUTLE, M. **An efficient failure detector for sparsely connected networks**. Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks. Innsbruck: Austria, 2004

JALOTE, P. **Fault tolerance in distributed systems**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1994.

LORENZI, F.; FERREIRA, G. L.; RIZZETTI, T. A.; NUNES, R. C. ; AUGUSTIN, I.; **Análise Comparativa de Detectores de Falhas para Redes Móveis**. In: VI Simpósio de Informática da Região Centro/RS, 2007, Santa Maria - RS. Anais SIRC. Santa Maria : UNIFRA, 2007. v. 1. p. 1-10.

MACEDO, D.; CORREIA, L.; SANTOS, A.; LOUREIRO, A.; NOGUEIRA, J. **Avaliando aspectos de tolerância a falhas em protocolos de roteamento para redes de sensores sem fio**. - VI Workshop de Testes e Tolerância a Falhas. Fortaleza, CE, 2005.

RENESSE, R. V.; MINSKY, Y.; HAYDEN, M. **A Gossip-Style Failure Detection Service**. [S.l.: s.n.], 1998. (TR98-1687).

SILVA, A. J. S. **As Tecnologias de Redes Wireless**. Boletim bimestral sobre tecnologia de redes produzida e publicada pela RNP – Rede Nacional de Ensino e Pesquisa, 15 de maio de 1998, volume 2, número 5 - Disponível em: <http://www.rnp.br/newsgen/9805/wireless.html - Acesso em: 19/09/2007

SRIDHAR, N. **Decentralized Local Failure Detection in Dynamic Distributed Systems**. *srds*, pp. 143-154, 25th IEEE Symposium on Reliable Distributed Systems (SRDS'06), 2006

SUN-MICROSYSTEMS. **Java Technology**. Disponível em: <http://java.sun.com/>.

TANENBAUM, A. S.; STEEN, M.V. **Distributed systems principles and paradigms.** Ed. Prentice Hall, Upper Saddle River, New Jersey : 2002.

WEBER, T.S.; Tolerância a Falhas: conceitos e exemplos. Programa de Pós-Graduação em Computação - Instituto de Informática - UFRGS. 2001

APÊNDICE A MODIFICAÇÂO DO ALGORÍTMO

O objetivo deste apêndice é apresentar os detalhes que foram modificados no algoritmo, para mostrar onde é feito o cálculo do tempo de falha do algoritmo e mostrar como atualiza o tempo da última falha do nodo.

O algoritmo aqui apresentado é o algoritmo *Gossip* Básico com as modificações feitas para ter as informações adicionais: tempo de falha e *timestamp* da última falha.

Para rodar o algoritmo, deve-se configurar o local das bibliotecas do *Jist/Swans*. O algoritmo utiliza recursos já implementadas.

Esses recursos foram:

```
import jist.swans.mac.MacAddress;
import jist.swans.net.NetInterface;
import jist.swans.net.NetAddress;
import jist.swans.misc.Message;
import jist.swans.Constants;
import jist.runtime.JistAPI;
import java.util.*;
```

A classe *NeighbourEntry* é responsável pelas informações dos nodos vizinhos. Nesta classe foi adicionado as informações conterror (responsável por contar o tempo que o nodo está falho) e a informação timestampfalha (responsável por guardar o último tempo que o nodo foi dado como falho)

```
private static class NeighbourEntry
 {
  /** endereço do vizinho */
  public NetAddress address;
  /** contador de heartbeats */
  public int heartbeatCounter;
  /** tempo do ultimo incremento do contador de heartbeat*/
  public long timestamp;
  /** quantidade de tempo que esta como falho*/
  public long conterror;
 // ultima vez que atualizou como falho.
 public long timestampfalha;
```

Para atualizar esses campos, foi necessário modificar o algoritmo nas seguintes fases:

- quando um nodo adiciona como falho outro nodo vizinho é atualizado o campo última falha.

- quando um nodo remove o vizinho da lista de suspeitos é atualizado o seu tempo de falha.

Atualização do tempo de última falha:

```
 // verifica os contadores de heartbeats locais e detecta os suspeitos
   private void detectSuspects()
      {
      long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;

      Iterator it = new HashMap(neighbours).values().iterator();
      while(it.hasNext()) {
         NeighbourEntry n = (NeighbourEntry) it.next();

         //compara com o endereço do proprio nodo
         if(n.address.equals(netaddress)) {
            continue;
         }

         if((time - n.timestamp) >= Tfail) {
            System.out.println("ADICIONA FALHA: O nodo |" + netaddress + "| adiciona como falho o nodo |" +
            n.address + "| no tempo de execução " + jist.runtime.JistAPI.getTime()/Constants.SECOND);
            suspectsList.put(n.address, n);
** na linha abaixo segue onde é feita a alteração do tempo de última falha.
             n.timestampfalha=jist.runtime.JistAPI.getTime()/Constants.SECOND;
            neighbours.remove(n.address);
         }
      }
   }
```

Alteração do tempo falho:

```
public void receive(Message msg, NetAddress src, MacAddress lastHop,
    byte macId, NetAddress dst, byte priority, byte ttl)
 {
   HashMap receivedList = ((MessageGossip)msg).getData();
   //recebe os dados da mensagem gossip

   //se a lista de suspeitos contem o endereco.
   if(suspectsList.containsKey(src))
      {
      long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;
      System.out.println("        REMOVE FALHA: O nodo |" + netaddress + "| remove o nodo |" + src + "| da lista
de suspeitos no tempo " + time);
      Iterator it2 = receivedList.values().iterator();
      while(it2.hasNext())
         {
         NeighbourEntry nrecebida = (NeighbourEntry) it2.next();   //lista recebida
```

```
NeighbourEntry ndele = (NeighbourEntry) neighbours.get(nrecebida.address); //lista local
// o heartbeat recebido nao esta na lista local, adiciona ele
if(ndele == null)
{
neighbours.put(nrecebida.address, nrecebida);
}
else if (ndele.address.equals(src))
      {
```

** Na lista abaixo é feita a comparação do tempo de última atualização , se não for igual, atualiza o campo de tempo falho.

```
      if (ndele.tempoatualizafalha!=ndele.timestamp)
      {
         ndele.conterror=ndele.conterror+(ndele.timestamp-ndele.timestampfalha);
         ndele.tempoatualizafalha=ndele.timestamp;
      }
```

APÊNDICE B ALGORITMO COMPLETO

Classe BasicGossip

```
//////////////////////////////////////////////////
// Trabalho de Graduação
// Miguel Baggio
// baggio@inf.ufsm.br
// SH.java
//

package jist.swans.app;

import jist.swans.mac.MacAddress;
import jist.swans.net.NetInterface;
import jist.swans.net.NetAddress;
import jist.swans.misc.Message;
import jist.swans.Constants;

import jist.runtime.JistAPI;

import java.util.*;

import org.python.modules.time;

/**
 * version 1.0
 * @since SWANS1.0
 * Modificado por Miguel Angelo Baggio - baggio@inf.ufsm.br
 *
 */
public class SH implements AppInterface, NetInterface.NetHandler
{
   private int valor;

   /**
  * informação sobre um vizinho
  */
 private static class NeighbourEntry
 {
   /** endereço do vizinho */
   public NetAddress address;
   /** contador de heartbeats */
   public int heartbeatCounter;
   /** tempo do ultimo incremento do contador de heartbeat*/
   public long timestamp;
   /** quantidade de tempo que esta como falho*/
```

```
 public long conterror;
// ultima vez que atualizou como falho.
 public long timestampfalha;

 public long tempoatualizafalha;

  public NeighbourEntry(NetAddress n, int count, long t, long conter,long tempof, long tempofalha2)
     {
     address = n;
     heartbeatCounter = count;
     timestamp = t;
     conterror = conter;
     timestampfalha = tempof;
     tempoatualizafalha = tempofalha2;
     //adicionado aqui o contador
     }
}

//////////////////////////////////////////////////
// constants
//

/** periodo para enviar uma mensagem gossip */
public int Tgossip; //em segundos

/** periodo para varrer a lista de vizinhos, qtas vezes após o Tgossip  */
public int Tcleanup;

/** diferença para inserir um nodo na lista de suspeitos */
public int Tfail;


//////////////////////////////////////////////////
// messages
//

/**
 * Heartbeat packet.
 */
private static class MessageGossip implements Message
{
  protected HashMap neighbourList;

    /** {@inheritDoc} */
  public int getSize()
  {
     return neighbourList.size() * 100;
  }
  /** {@inheritDoc} */
  public void getBytes(byte[] b, int offset)
  {
```

```
    System.out.println("Nao implementado");
  }

  public void setNeighbourList(HashMap n) {
    neighbourList = n;
  }

  public HashMap getData() {
    return neighbourList;
  }

 } // class: MessageGossip

////////////////////////////////////////////////
// locals
//

  /** network entity. */
  private NetInterface netEntity;

  /** self-referencing proxy entity. */
  private Object self;

  /** lista de vizinhos */
  private HashMap neighbours;

  /** lista de suspeitos */
  private HashMap suspectsList;

  /**  contador de cleanup */
  private int cleanupCount = 0;

  /** endereço do nodo */
  NetAddress netaddress;

  int cont;

  long contaerro;



////////////////////////////////////////////////
// initialize
//

/**
 * Create new heartbeat application instance.
 *
 * @param address node identifier

 */
```

```
public SH(NetAddress address, int Tgossip, int Tcleanup, int Tfail)
{
  this.self = JistAPI.proxyMany(
        this, new Class[] { AppInterface.class, NetInterface.NetHandler.class });
  this.netaddress = address;
  this.Tgossip =  Tgossip;
  this.Tcleanup = Tcleanup;
  this.Tfail = Tfail;
  suspectsList = new HashMap();
  neighbours = new HashMap();
  long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;
  NeighbourEntry n = new NeighbourEntry(netaddress, 0, time, 0, 0,0);
  neighbours.put(netaddress, n);
}

//////////////////////////////////////////////////
// entity
//

/**
 * Set network entity.
 *
 * @param netEntity network entity
 */
public void setNetEntity(NetInterface netEntity)
{
  this.netEntity = netEntity;
}

/**
 * Return self-referencing NETWORK proxy entity.
 *
 * @return self-referencing NETWORK proxy entity
 */
public NetInterface.NetHandler getNetProxy()
{
  return (NetInterface.NetHandler)self;
}

/**
 * Return self-referencing APPLICATION proxy entity.
 *
 * @return self-referencing APPLICATION proxy entity
 */
public AppInterface getAppProxy()
{
  return (AppInterface)self;
}


//////////////////////////////////////////////////
```

```
        // NetHandler methods
        //

        /** {@inheritDoc}
         *    Recebimento da mensagem gossip do vizinho com ip "src"
         * */
        public void receive(Message msg, NetAddress src, MacAddress lastHop,
           byte macId, NetAddress dst, byte priority, byte ttl)
        {

          HashMap receivedList = ((MessageGossip)msg).getData();
          //recebe os dados da mensagem gossip

          //se a lista de suspeitos contem o endereco.
          if(suspectsList.containsKey(src))
             {
             long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;
             System.out.println("        REMOVE FALHA: O nodo |" + netaddress + "| remove o nodo |" + src + "| da lista de suspeitos no
tempo " + time);
             //miguel
             Iterator it2 = receivedList.values().iterator();
             while(it2.hasNext())
                {
                NeighbourEntry nrecebida = (NeighbourEntry) it2.next();   //lista recebida
                NeighbourEntry ndele = (NeighbourEntry) neighbours.get(nrecebida.address); //lista local
                // o heartbeat recebido nao esta na lista local, adiciona ele
                if(ndele == null)
                {
                neighbours.put(nrecebida.address, nrecebida);
                }
                else if (ndele.address.equals(src))
                      {
                      if (ndele.tempoatualizafalha!=ndele.timestamp)
                      {
                         ndele.conterror=ndele.conterror+(ndele.timestamp-ndele.timestampfalha);
                         ndele.tempoatualizafalha=ndele.timestamp;
                      }
                      // System.out.println("Tempo atual:"+time);
                     // System.out.println("Nodo: "+ndele.address+"| Nodo Recebido: |"+nrecebida.address+ "|");
                     //System.out.println("Timestamp:                "+ndele.timestamp+                "                Timestamp:
"+nrecebida.timestamp);//nAtual.conterror=nAtual.conterror+(nAtual.timestamp-nAtual.timestampfalha);
                     // System.out.println("TimestampF:"+ndele.timestampfalha +" TimestampF: "+nrecebida.timestampfalha);
                      if (ndele.conterror>0)
                         System.out.println("Segundo o nodo |"+netaddress+"| o nodo |"+ndele.address+"| tem o tempo de falha igual a
:"+ndele.conterror);
                      }
                }
              //miguel
             suspectsList.remove(src);
             }
          // compara os heartbeats da lista local com a lista recebida
```

```
  Iterator it = receivedList.values().iterator();
  while(it.hasNext()) {
     NeighbourEntry nReiceived = (NeighbourEntry) it.next();   //lista recebida
     NeighbourEntry nAtual = (NeighbourEntry) neighbours.get(nReiceived.address); //lista local
     // o heartbeat recebido nao esta na lista local, adiciona ele
     if(nAtual == null)
        {
        neighbours.put(nReiceived.address, nReiceived);
        }
           else if(nAtual.heartbeatCounter < nReiceived.heartbeatCounter)
            {
            // se o heartbeat recebido eh maior doq o atual
            nAtual.heartbeatCounter = nReiceived.heartbeatCounter;
            nAtual.timestamp = jist.runtime.JistAPI.getTime()/Constants.SECOND;
            }

     }
}

////////////////////////////////////////////////
// AppInterface methods
//

/**
 * Compute random heartbeat delay.
 *
 * @return delay to next heartbeat
 */
private long calcDelay()
{
   return (long) ((Tgossip * Constants.SECOND) + (Math.abs(Constants.random.nextLong()) % 999999)) ;
}

/** {@inheritDoc} */
public void run(String[] args)
{
  int contamsg;
  cleanupCount++;

  if((cleanupCount % Tcleanup) == 0)
     {
     //verifica se existe algum suspeito
     detectSuspects();
     }

  // envia a lista de vizinhos para algum vizinho

  sendGossipMessage();
  // schedule next - executa novamente
  JistAPI.sleepBlock(calcDelay());
  ((AppInterface)self).run();
```

```
        //ListaErros(90);
        }

        // envia a mensagem gossip
        private void sendGossipMessage() {

            Message msg = new MessageGossip();
            cont++; //contador de heartbeats local

            // incrementa seu heartbeat e atualiza o timestamp antes de enviar a mensagem
            NeighbourEntry n = (NeighbourEntry) neighbours.get(netaddress);
            if(n == null) {
                long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;
               // System.out.println("end: " + netaddress.getIP());
                n = new NeighbourEntry(netaddress, cont, time,0,0,0);
                neighbours.put(netaddress, n);
            } else {
                n.timestamp = jist.runtime.JistAPI.getTime()/Constants.SECOND;
                n.heartbeatCounter++;

                //System.out.println("["+ netaddress.getIP() + "] aqui: " + n.timestamp + " e " + n.heartbeatCounter);
            }

            //BROADCAST THE LIST
            //System.out.println("Broadcast realizado por " + netaddress);
            ((MessageGossip)msg).setNeighbourList(neighbours);
                netEntity.send(msg, NetAddress.ANY, Constants.NET_PROTOCOL_HEARTBEAT,
                        Constants.NET_PRIORITY_NORMAL, (byte) 1);
        }

        // verifica os contadores de heartbeats locais e detecta os suspeitos
        private void detectSuspects()
            {
            long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;

            Iterator it = new HashMap(neighbours).values().iterator();
            while(it.hasNext()) {
                NeighbourEntry n = (NeighbourEntry) it.next();

                //compara com o endereço do proprio nodo
                if(n.address.equals(netaddress)) {
                    continue;
                }

                if((time - n.timestamp) >= Tfail) {
                    System.out.println("ADICIONA FALHA: O nodo |" + netaddress + "| adiciona como falho o nodo |" + n.address + "| no
tempo de execução " + jist.runtime.JistAPI.getTime()/Constants.SECOND);
                    suspectsList.put(n.address, n);
                    n.timestampfalha=jist.runtime.JistAPI.getTime()/Constants.SECOND;
                    neighbours.remove(n.address);
```

```
            }
          }
        }

        private void removesuspeito()
        {
          long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;
          Iterator it = new HashMap(neighbours).values().iterator();
          while(it.hasNext())
            {
            NeighbourEntry n = (NeighbourEntry) it.next();
            //compara com o endereço do proprio nodo
            if(n.address.equals(netaddress))
              {
              continue;
              }
            if((time - n.timestamp) >= Tfail) {
              System.out.println("O nodo |" + netaddress + "| adiciona como falho o nodo |" + n.address + "| no tempo de execução "
+ jist.runtime.JistAPI.getTime()/Constants.SECOND);
              suspectsList.put(n.address, n);
              neighbours.remove(n.address);
            }
        }
        }


        private void listatudo()
        {
        Iterator it = new HashMap(neighbours).values().iterator();
        System.out.println("Sou o nodo "+netaddress+" e tenho os seguintes vizinhos:");
        while(it.hasNext()) {
        NeighbourEntry n = (NeighbourEntry) it.next();
        //compara com o endereço do proprio nodo
         if(n.address.equals(netaddress))
            {
            continue;
            }
          else
            {
            System.out.println("- |"+n.address+" e tenho o tempo de falha igual a : "+n.conterror);
             }
        }
          /*   int imprime=this.valor;
         // for(int i=1;i<=10;i++){        //O loop é executado 10 vezes
           System.out.println(imprime+" "); //Será impressa na tela uma contagem de 1 até 10.
        // }*/
        }


        /** {@inheritDoc} */
```

```
 public void run()
 {
  run(null);
  listatudo();
 }

}
```

Classe StartUpGossip

```
package driver;

import jist.swans.Constants;
import jist.swans.misc.Util;
import jist.swans.misc.Mapper;
import jist.swans.misc.Location;
import jist.swans.field.Field;
import jist.swans.field.Placement;
import jist.swans.field.Mobility;
import jist.swans.field.Spatial;
import jist.swans.field.Fading;
import jist.swans.field.PathLoss;
import jist.swans.radio.RadioNoiseIndep;
import jist.swans.radio.RadioInfo;
import jist.swans.mac.MacAddress;
import jist.swans.mac.Mac802_11;
import jist.swans.net.NetAddress;
import jist.swans.net.NetIp;
import jist.swans.net.PacketLoss;
import jist.swans.app.SH;

import jist.runtime.JistAPI;

  /**
  * Trabalho de Graduacao de Miguel Angelo Baggio
  * Algoritmo Gossip Básico Modificado
   * Este algoritmo foi alterado para ter mais informações para a camada de detecção de mobilidade poder ter um
   * melhor desempenho, através de informações que ela possa tomar decisões.
  **/


public class StartUpGossip {

 /** random waypoint pause time. */
 public static final int PAUSE_TIME = 15;
 /** random waypoint granularity. */
 public static final int GRANULARITY = 2;
 /** random waypoint minimum speed. */
 public static final int MIN_SPEED = 0;
```

```
/** random waypoint maximum speed. */
public static final int MAX_SPEED = 2;

/**
 * Initialize simulation node.
 *
 * @param i node number - número do nodo
 * @param field simulation field - campo de simulacao
 * @param placement node placement model - modelo de colocação
 * @param radioInfoShared shared radio information  - informacao compartilhada
 * @param protMap shared protocol map - protocolo compartilhado
 * @param plIn incoming packet loss model - modelo de perda de pacote de chegada
 * @param plOut outgoing packet loss model - modelo de perda de pacote de saida
 */
public static void createNode(int i,
    Field field, Placement placement,
    RadioInfo.RadioInfoShared radioInfoShared, Mapper protMap,
    PacketLoss plIn, PacketLoss plOut, int Tgossip, int Tcleanup, int Tfail)
{
  // create entities
  RadioNoiseIndep radio = new RadioNoiseIndep(i, radioInfoShared);
  Mac802_11 mac = new Mac802_11(new MacAddress(i), radio.getRadioInfo());
  NetAddress netAddress = new NetAddress(i);
  NetIp net = new NetIp(netAddress, protMap, plIn, plOut);
  SH app = new SH(netAddress, Tgossip, Tcleanup, Tfail);

  // hookup entities
  field.addRadio(radio.getRadioInfo(), radio.getProxy(), placement.getNextLocation());
  field.startMobility(radio.getRadioInfo().getUnique().getID());
  radio.setFieldEntity(field.getProxy());
  radio.setMacEntity(mac.getProxy());
  mac.setRadioEntity(radio.getProxy());
  byte intId = net.addInterface(mac.getProxy());
  mac.setNetEntity(net.getProxy(), intId);
  net.setProtocolHandler(Constants.NET_PROTOCOL_HEARTBEAT, app.getNetProxy());
  app.setNetEntity(net.getProxy());
  app.getAppProxy().run(null);
}

/**
 * Initialize simulation field.
 *
 * @param nodes number of nodes
 * @param length length of field
 * @param t_range nodes' transmission range
 * @return simulation field
 */
public static Field createSim(int nodes, int length, int t_range, int Tgossip, int Tcleanup, int Tfail)
{
  Location.Location2D bounds = new Location.Location2D(length, length);
  Placement placement = new Placement.Random(bounds);
```

```
        Mobility mobility = new Mobility.RandomWaypoint(bounds, PAUSE_TIME, GRANULARITY, MAX_SPEED, MIN_SPEED);
        Spatial spatial = new Spatial.HierGrid(bounds, 5); //responsavel pela propagacao do sinal, hierarchical binning
        Fading fading = new Fading.Rayleigh(); //atenuacoes que podem ocorrer seguem a distribuicao de propabilidade de Rayleigh
        PathLoss pathloss = new PathLoss.TwoRay(); //perda da propagacao depende da distancia

        Field field = new Field(spatial, fading, pathloss, mobility, Constants.PROPAGATION_LIMIT_DEFAULT);
         /* aqui se passa o limite de alcance */
        RadioInfo.RadioInfoShared radioInfoShared = RadioInfo.createShared(
          Constants.FREQUENCY_DEFAULT, Constants.BANDWIDTH_DEFAULT,
          t_range, Constants.GAIN_DEFAULT,
          Util.fromDB(Constants.SENSITIVITY_DEFAULT), Util.fromDB(Constants.THRESHOLD_DEFAULT),
          Constants.TEMPERATURE_DEFAULT, Constants.TEMPERATURE_FACTOR_DEFAULT, Constants.AMBIENT_NOISE_DEFAULT);

        Mapper protMap = new Mapper(Constants.NET_PROTOCOL_MAX);
        protMap.mapToNext(Constants.NET_PROTOCOL_HEARTBEAT);
        PacketLoss pl = new PacketLoss.Uniform(0.20);

        for(int i=0; i<nodes; i++)
        {
          createNode(i, field, placement, radioInfoShared, protMap, pl, pl, Tgossip, Tcleanup, Tfail);
        }

        return field;
      }

      /**
       * Benchmark entry point: heartbeat test.
       *
       * @param args command-line parameters
       */
      public static void main(String[] args)
      {
        if(args.length<4)
        {
          System.out.println("syntax: swans driver.sh <Número de Nós> <Tamanho do campo> <Tempo de execução> <Potencia de Transmissao em dbm> <Tgossip> <Tcleanup> <Tfail>");
          System.out.println("    eg: swans driver.sh 5 100 5 15 10 2 10");
          return;
        }

        int nodes = Integer.parseInt(args[0]);
        int length = Integer.parseInt(args[1]);
        int time = Integer.parseInt(args[2]);
        int t_power = Integer.parseInt(args[3]);
        int Tgossip = Integer.parseInt(args[4]);
        int Tcleanup = Integer.parseInt(args[5]);
        int Tfail = Integer.parseInt(args[6]);
        float density = nodes / (float)(length/1000.0 * length/1000.0);
```

```
    System.out.println("nodos                  = "+nodes);
    System.out.println("tamanho do campo       = "+length+" x "+length);
    System.out.println("tempo                  = "+time+" segundos");
    System.out.println("alcance de transmissao = "+t_power+" dbm");
    System.out.println("Tgossip    = "+Tgossip+" Tcleanup: " +Tcleanup+ " Tfail: " +Tfail);
    System.out.print("Criando a simulação dos nodos... ");
    Field f = createSim(nodes, length, t_power, Tgossip, Tcleanup, Tfail);
    System.out.println("done.");

    System.out.println("Average density = "+f.computeDensity()*1000*1000+"/km^2");
    System.out.println("Average sensing = "+f.computeAvgConnectivity(true));
    System.out.println("Average receive = "+f.computeAvgConnectivity(false));
    JistAPI.endAt(time*Constants.SECOND);
  }

}
```