UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

UMA COMPARAÇÃO ENTRE ALGORITMOS DE DETECÇÃO DE DEFEITOS PARA REDES MÓVEIS SEM FIO

TRABALHO DE GRADUAÇÃO

Giovani Gracioli

Santa Maria, RS, Brasil

2007

UMA COMPARAÇÃO ENTRE ALGORITMOS DE DETECÇÃO DE DEFEITOS PARA REDES MÓVEIS SEM FIO

por

Giovani Gracioli

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

Bacharel em Ciência da Computação

Orientador: Prof. Dr. Raul Ceretta Nunes

Trabalho de Graduação Nº 223
Santa Maria, RS, Brasil

2007

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

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

UMA COMPARAÇÃO ENTRE ALGORITMOS DE DETECÇÃO DE DEFEITOS PARA REDES MÓVEIS SEM FIO

elaborado por
Giovani Gracioli

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

COMISSÃO EXAMINADORA:

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

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

Prof. Msc. Rogério Corrêa Turchetti (UNIFRA)

Santa Maria, 01 de março de 2007.

O que as vitórias têm de mau é que não são definitivas. O que as derrotas têm de bom é que também não são definitivas. — JOSÉ SARAMAGO

Costumo voltar atrás, sim; não tenho compromisso com o erro. — JUSCELINO KUBITSCHECK

AGRADECIMENTOS

Ao fim de mais essa etapa, gostaria de deixar algumas considerações para umas pessoas que foram importantes durante essa caminhada. Primeiramente quero agradecer as duas pessoas mais importantes da minha vida, a minha mãe (Rosilene Marisa Bertochi) e ao meu pai (Leonildo Gracioli), sem eles nada disso seria possível, obrigado por tudo. Ao resto da minha família, tios, tias, avós e primos pelo apoio dado em horas complicadas.

Gostaria de agradecer também o professor Raul, pela orientação na realização deste trabalho sempre com muita responsabilidade e de forma eficaz. Com certeza uma amizade foi construída ao longo desse período.

Por fim, aos meus amigos e colegas pelo apoio e torcida, especialmente ao Guilherme Piêgas Koslovski, Eduardo Maikel Muller, João Vicente Lima (a máquina) e Carlos Hugo (Alemão).

RESUMO

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

UMA COMPARAÇÃO ENTRE ALGORITMOS DE DETECÇÃO DE DEFEITOS PARA REDES MÓVEIS SEM FIO
Autor: Giovani Gracioli
Orientador: Prof. Dr. Raul Ceretta Nunes
Local e data da defesa: Santa Maria, 01 de março de 2007.

Uma rede móvel sem fio é caracterizada pela escassez de recursos, tais como alcance de transmissão e consumo de energia. Esse tipo de rede tem a vantagem de utilizar todo o potencial de dispositivos móveis, como PDAs e computadores portáteis, por exemplo. Por outro lado, essas redes têm uma maior vulnerabilidade a erros e defeitos, porém, técnicas de tolerância a falhas podem ser usadas para contorná-los. O detector de defeitos, que nada mais é que um algoritmo distribuído que fornece informações sobre suspeitas de defeitos em componentes monitoráveis, é um bloco de construção importante para a implementação dessas técnicas de tolerância a falhas. Neste contexto, este trabalho apresenta uma revisão entre as características dos principais algoritmos de detecção de defeitos para redes móveis sem fio e os compara através de testes realizados em um simulador.

Palavras-chave: Detectores de Defeitos, Tolerância a Falhas, Redes Móveis Sem fio, Sistemas Distribuídos.

ABSTRACT

Graduation Work
Graduation in Computer Science
Federal University of Santa Maria

A COMPARATION BETWEEN FAILURE DETECTOR ALGORITHMS FOR WIRELESS MOBILE NETWORKS

Author: Giovani Gracioli
Advisor: Prof. Dr. Raul Ceretta Nunes

A wireless mobile network is characterized by shortage of resources, such as transmission range and energy consumption. This network has the advantage to use all the potencial of mobile devices, like PDAs and portable computers, for example. On the other hand, these networks are more vulnerable to errors and failures. However, techniques of fault tolerance can be used to skirt them. The failure detector is a distributed algorithm that supplies informations about failures suspicion in monitored components. The failure detector is an important building block for the implementation of fault tolerance techniques. In this context, this work reviews the characteristics of the main failure detection algorithms for wireless mobile networks and compare them through tests carried out in a simulator.

Keywords: Failure Detectors, Fault Tolerance, Wireless Mobile Networks, Distributed Systems.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AODV	*Ad hoc On-demand Distance Vector*
CSMA	*Carrier Sense Multiple Access*
DSDV	*Destination-Sequenced Distance Vector*
DSR	*Dynamic Source Routing*
GloMoSim	*Global Mobile Information Systems Simulation Library*
IP	*Internet Protocol*
JiST	*Java in Simulation Time*
LAN	*Local Area Network*
LAR	*Location Aided Routing*
MAC	*Media Access Control*
MACA	*Multiple Access with Collision Avoidance*
NS2	*Network Simulator 2*
ODMRP	*On-Demand Multicast Routing Protocol*
PARSEC	*Parallel Simulation Environment for Complex Systems*
RMSF	Redes Móveis Sem Fio
SWANS	*Scalable Wireless Ad hoc Network Simulation*
T_D	Tempo de Detecção (*Detection Time*)
T_M	Tempo Médio de Duração de Falsas Suspeitas (*Mistake Duration*)
T_{MR}	Tempo de Recorrência ao Erro (*Mistake Recurrence Time*)
TCP	*Transmission Control Protocol*
TDMA	*Time Division Multiple Access*
TORA	*Temporally-Ordered Routing Algorithm*
UDP	*User Datagram Protocol*
ZRP	*Zone Routing Protocol*
WAN	*Wide Area Network*
WRP	*Wireless Routing Protocol*

SUMÁRIO

1 INTRODUÇÃO

As facilidades encontradas com computação distribuída atrelada com o crescente uso de equipamentos móveis e não confiáveis, leva a uma necessidade de construção de *softwares* que sejam capazes de suportarem falhas. Falhas são inevitáveis, porém as conseqüências indesejadas podem ser evitadas pelo uso adequado de técnicas de tolerância a falhas (JALOTE, 1994; WEBER, 2002). Como qualquer técnica de tolerância a falhas implica em um custo associado, o domínio da área ajuda na escolha da melhor técnica levando-se em consideração a relação "Custo/Benefício" para o sistema.

A utilização das vantagens da mobilidade dos computadores de mão (PDAs) dotados com um poder de conectividade *wireless* podem ter todo o seu potencial explorado em áreas como a saúde. Um exemplo do seu uso é o recebimento de informações pelo médico sobre seu paciente sem a necessidade de locomoção até o leito aonde o paciente se encontra. Redes sem fio e equipamentos portáteis dão capilaridade à infra-estrutura, que possibilita o envio de sinais cardíacos coletados em instrumentos ligados diretamente ao paciente (marca-passos ou eletrocardiógrafos), à sistemas de visualização e monitoramento, bem como emitir alarmes e relatórios quando alguma anomalia cardíaca for detectada. Em síntese, a detecção de defeitos em uma rede dessa importância, tratando-se de sinais vitais, torna-se fundamental e indispensável.

O detector de defeitos é um bloco básico para a construção de mecanismos de tolerância a falhas em sistemas distribuídos (JALOTE, 1994). Um detector de defeitos nada mais é que um algoritmo que fornece informações sobre suspeitas de defeitos em nodos ou processos monitorados (FELBER et al., 1999). Essas informações são recolhidas através de trocas de mensagens realizadas pelos nodos que contém um detector de defeitos acoplado. Duas propriedades caracterizam um detector (CHANDRA; TOUEG, 1996): abrangência (*completeness*) e exatidão (*accuracy*). A abrangência se refere a capacidade do detector

descobrir todos os processos que estão com defeito enquanto que a exatidão, se refere ao fato de não suspeitar de um processo que está em seu estado normal, ou seja, se refere a precisão na suspeita de um defeito.

Vários algoritmos para a detecção de defeitos em redes com computadores fixos (LANs e WANs) vem sendo propostos ao longo dos anos (AGUILERA; CHEN; TOUEG, 1997; RENESSE; MINSKY; HAYDEN, 1998; FELBER et al., 1999). Entretanto, o número de redes móveis (redes de sensores, redes ad hoc) compostas por computadores móveis com a capacidade de comunicação sem fio (*wireless*) cresce rapidamente e as soluções que antes eram adequadas para LANs ou WANs nem sempre podem ser reutilizadas, pois redes móveis sem fio são caracterizadas pela escassez de recursos tais como alcance de transmissão e consumo de energia e pela mudança freqüente na topologia da rede, devido a mobilidade (MATEUS; LOUREIRO, 2005).

O objetivo deste trabalho é implementar e comparar alguns algoritmos de detecção de defeitos para redes móveis sem fio (HUTLE, 2004; TAI; TSO; SANDERS, 2004; FRIEDMAN; TCHARNY, 2005) e também o algoritmo *gossip* (RENESSE; MINSKY; HAYDEN, 1998) no simulador JiST/SWANS (BARR; HAAS; RENESSE, 2004a). Mais especificamente, esses objetivos podem ser decompostos em:

- Comparar os algoritmos quanto ao tempo médio de falsas detecções, tempo médio de recorrência ao erro, tempo de detecção e número de *broadcasts*[1];
- Comparar o desempenho dos algoritmos em termos de quantidade de nodos presente na rede;
- Avaliar o comportamento dos algoritmos com mudanças no alcance de transmissão dos nodos;
- Avaliar o custo da formação de um *cluster* através da comparação do serviço de detecção de defeitos proposto por (TAI; TSO; SANDERS, 2004) com o algoritmo *Gossip*.

O restante deste trabalho está organizado da seguinte maneira. O capítulo 2 apresenta uma breve introdução sobre alguns conceitos em sistemas distribuídos, tolerância a falhas e detecção de defeitos que são importantes para o acompanhamento do resto do texto. No

[1]O número de *broadcasts* é referente ao número de mensagens transmitidas, com isso pode-se ter uma estimativa do consumo de energia através da utilização do rádio.

capítulo 3, redes ad hoc e redes de sensores são definidas e também alguns dos problemas que são encontrados nesses dois tipos de rede são identificados. Os algoritmos de detecção de defeitos para redes móveis sem fio são descritos no capítulo 4. No capítulo 5, uma revisão dos simuladores existentes para redes móveis sem fio são apresentados. Neste capítulo ainda, o simulador JiST/SWANS utilizado nos testes deste trabalho é descrito em maiores detalhes. O capítulo 6 apresenta as avaliações e os resultados dos testes dos algoritmos realizados no simulador. Uma proposta de um estudo de caso aonde um detector de defeitos pode ser utilizado é apresentado no capítulo 7. Por fim, conclusões são feitas no capítulo 8.

2 CONCEITOS SOBRE SISTEMAS DISTRIBUÍDOS E DETECÇÃO DE DEFEITOS

Um sistema distribuído é caracterizado pelo fraco acoplamento existente entre os nodos que o constroem, isto é, cada nodo tem sua própria memória e unidade de processamento. Não existe um relógio global ao sistema, e toda troca de informações entre os nodos deve ser feita através de troca de mensagens. Uma rede de computador não é sinônimo para sistema distribuído. Uma rede porém, fornece toda a infra-estrutura para se criar um sistema distribuído.

Um sistema distribuído deve suportar falhas, pois o custo de uma falha no sistema pode ser alto, trazendo conseqüências como indisponibilidade de serviços aos usuários. As falhas em um sistema distribuído podem ser classificadas segundo (CRISTIAN; AGHILI; STRONG, 1994) como:

- **Crash:** parada ou perda no estado interno.
- **Omissão:** sem resposta para alguns pedidos.
- **Temporização:** resposta atrasada ou adiantada.
- **Resposta:** respostas incorretas para alguns pedidos.
- **Arbitrária:** comportamento arbitrário e imprevisível.

Para evitar falhas, técnicas de tolerância a falhas podem ser usadas. Uma das características dos sistemas distribuídos é a redundância, o que é explorado por componentes de tolerância a falhas, possibilitando assim que o sistema não pare de funcionar mesmo na presença de falhas. Os sistemas distribuídos também são assíncronos, o que faz com que os mecanismos de tolerância a falhas tenham que usar um limite de tempo entre o envio e recebimento de mensagens de controle em seus algoritmos. Em um sistema assíncrono,

apesar do limite de tempo, é impossível determinar se um processo está em estado errôneo ou se está apenas com problemas de comunicação ou processamento, causando assim um problema de indeterminismo. Para atenuar o problema de indeterminismo, detectores de defeitos podem ser usados (GARTNER, 1999). Um detector de defeito nada mais é do que um algoritmo distribuído (um monitor) que fornece informações sobre suspeitas de defeitos em componentes monitoráveis (FELBER et al., 1999).

Para *Felber* (FELBER et al., 1999) um detector de defeitos deve ser a primeira classe de serviço distribuído a ser construída. O detector deve fornecer uma interface de acesso para as aplicações que o queiram utilizar, criando assim uma hierarquia onde o detector de defeitos executa em cima do sistema operacional mas abaixo da camada de aplicação. Essa abstração faz com que as aplicações clientes tenham uma visão limitada sobre alguns serviços disponibilizados pelo detector de defeitos.

2.1 Falha, Erro e Defeito

Entre os conceitos em tolerância a falhas, está o de determinar o que é falha, erro ou defeito. O conceito aqui apresentado é utilizado neste trabalho. Em um sistema, um defeito (*failure*) não deve ser tolerado, mas sim evitado. Basicamente um defeito é definido como um desvio da especificação. Um erro é definido quando após o seu processamento leva-se a um defeito. Por fim, uma falha ou falta é definido como a causa física do erro.

A figura 2.1 mostra um modelo de falha, erro e defeito proposto por *Barry W. Johnson* (PRADHAN, 1996). Uma falha acontece no universo físico, um erro no universo da informação e o defeito no universo do usuário. Como exemplo desse cenário, imagine uma falha em um componente de *hardware*. Esta falha causará um estado errôneo na aplicação que utilizará os dados deste *hardware*, transformando-se em um defeito para o usuário da aplicação.

2.2 Propriedades e Classes dos Detectores de Defeitos

O comportamento dos detectores de defeitos é especificado por duas propriedades (CHANDRA; TOUEG, 1996): a abrangência (*completeness*) e a exatidão (*accuracy*). Abrangência se refere a capacidade de detectar defeitos de nodos que realmente falharam, enquanto que a exatidão se refere a capacidade de não cometer falsas suspeitas. De acordo

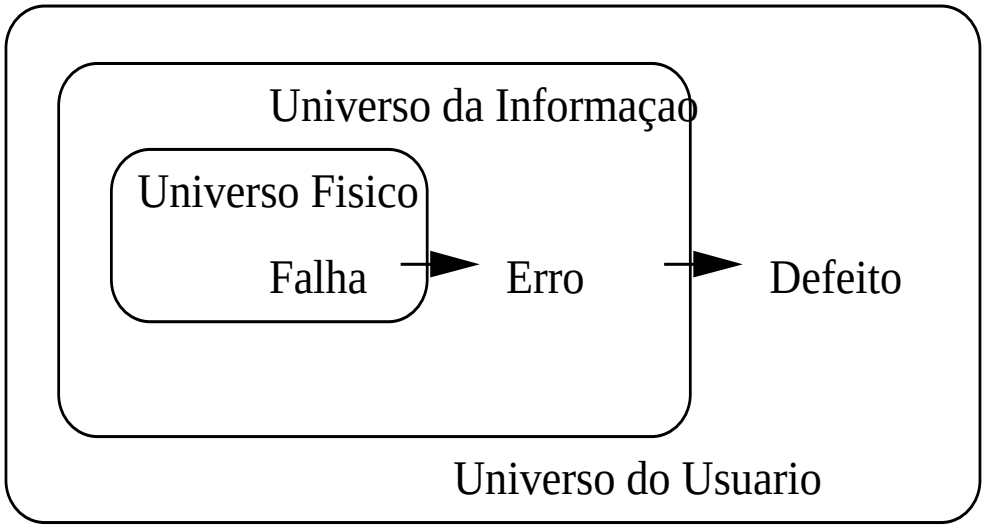

Figura 2.1: Modelo dos universos de falha, erro e defeito.

com essas propriedades, os detectores de defeitos podem ser classificados nas seguintes classes:

- **Abrangência Forte:** *todos* os processos corretos suspeitarão permanentemente de *todos* processos falhos.
- **Abrangência Fraca:** *algum* processo correto suspeitará permanentemente de *todos* processos falhos.
- **Precisão Forte:** *nenhum* processo é suspeito antes de ter realmente falhado.
- **Precisão Fraca:** *pelo menos um* processo correto jamais será suspeito pelos outros processos.
- **Precisão Eventualmente Forte:** *todos* os processos somente são considerados suspeitos após realmente falharem.
- **Precisão Eventualmente Fraca:** *algum* processo correto nunca é suspeito antes de ter realmente falhado.

A tabela 2.1 apresenta as oito possíveis combinações entre as propriedades dos detectores de defeitos. Um detector de defeitos é dito perfeito se este apresenta as características de abrangência e precisão fortes e é denotado pelo símbolo $\mathcal{P}$.

Tabela 2.1: Classes de detectores de defeitos.

	Precisão			
Abrangência	Forte	Fraca	Eventual Forte	Eventual Fraca
Forte	Perfeito $\mathcal{P}$	Forte $\mathcal{S}$	Eventualmente Perfeito $\Diamond\mathcal{P}$	Eventualmente Forte $\Diamond\mathcal{S}$
Fraca	$\mathcal{Q}$	Fraco $\mathcal{W}$	$\Diamond\mathcal{Q}$	Eventualmente Fraco $\Diamond\mathcal{W}$

2.3 Detectores de Defeitos para Redes com Nodos Fixos

Nesta seção são revisados alguns detectores de defeitos inicialmente projetados para redes com nodos fixos (LANs e WANs). Estes detectores podem monitorar processos, objetos ou nodos (um processo/objeto por nodo) e são descritos nas próximas subseções.

2.3.1 Modelo Push

No modelo *Push* (FELBER et al., 1999) os componentes ou processos monitorados devem estar ativos, enquanto o monitor (detector de defeitos) é passivo. Cada objeto monitorado deve periodicamente enviar mensagens do tipo *"I am alive!"* ao monitor (um *heartbeat*[1]). O detector de defeitos suspeita de um componente quando não recebe a mensagem em um certo intervalo de tempo T (*timeout*). O detector *Push* tem a vantagem de que as mensagens são unidirecional, o que diminui o número de mensagens trocadas.

A figura 2.2 mostra um exemplo de funcionamento do algoritmo *Push*. O processo monitorável *M1* envia periodicamente uma mensagem *"I am alive!"* ao monitor, que ao receber a mensagem, reinicia o *Timeout T*. Quando o processo monitor não recebe a mensagem do monitorado devido ao seu defeito, uma suspeita é levantada.

2.3.2 Modelo Pull

No modelo *Pull* (FELBER et al., 1999) os componentes ou processos monitorados são passivos, o monitor neste caso é o ativo. O monitor (detector de defeitos) envia mensagens do tipo *"Are you alive?"* periodicamente aos objetos monitoráveis. Quando o monitor não recebe a resposta dos componentes monitoráveis em um certo intervalo de tempo T (*timeout*), o monitor começa a suspeitar do defeito no componente. Neste estilo a troca de mensagens é bidirecional, deixando o modelo menos eficiente que o estilo *Push*, porém apresenta a vantagem de que os objetos monitoráveis não precisam estar ativos

[1]Não confundir este *heartbeat* com o detector de defeitos *Heartbeat* apresentado na seção 2.3.4.

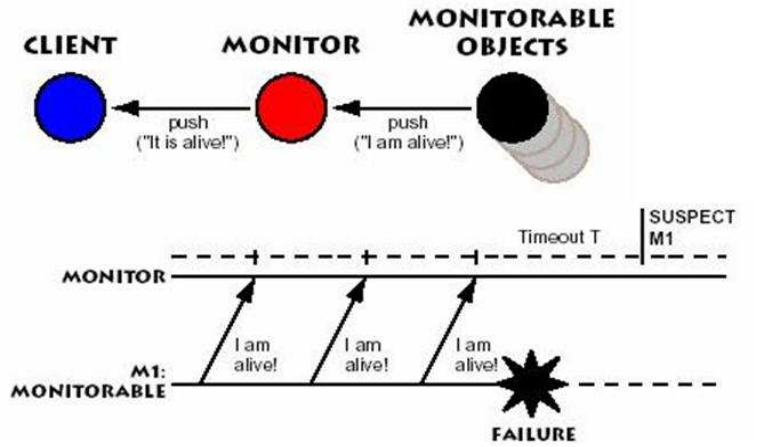

Figura 2.2: Modelo Push.

nem conhecer os *timeouts* do sistema como um todo.

A figura 2.3 exemplifica o modelo *Pull*. O monitor ao enviar uma mensagem *"Are you alive?"* ao monitorado inicia a contagem do *timeout*. O processo monitorado ao receber esta mensagem responde dizendo que está operacional. O monitor ao não receber esta mensagem de confirmação, suspeita do processo monitorado.

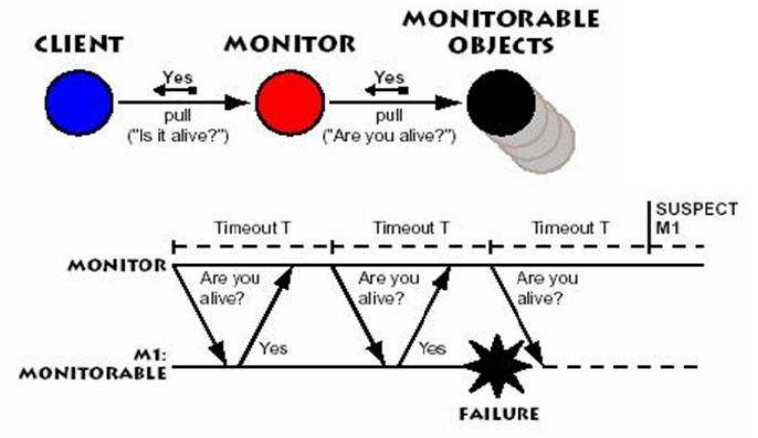

Figura 2.3: Modelo Pull.

2.3.3 Modelo Dual

O estilo *Dual* (FELBER et al., 1999) é um combinação dos estilos *Push* e *Pull*. Neste modelo a detecção se dá em duas fases. Na primeira fase todos os objetos monitoráveis se portam como no estilo *Push*, ao expirar o *timeout* o monitor passa para a segunda fase. Na segunda fase o monitor assume que todos os objetos monitoráveis que não enviaram a mensagem *"I am alive!"* devem ser consultados (*"Are you alive?"*) e estes processos

devem responder *"I am alive!"*. Caso não receba a resposta em determinado intervalo de tempo (*timeout2*) o monitor suspeitará do objeto monitorado.

O modelo *Dual* é descrito na figura 2.4. O processo *M1* é ativo e periodicamente envia mensagem ao monitor (detector de defeitos) dizendo que está operacional (*"I am alive!"*). São utilizados dois *timeouts* (*Timeout T1* e *Timeout T2*), um para cada fase. Durante a primeira fase o detector de defeitos se comporta de acordo com o algoritmo *Push*, esperando receber mensagens do tipo *"I am alive!"*. Após o *Timeout T1* expirar, e o detector de defeitos não receber a mensagem *"I am alive!"* do processo *M1*, o detector passa a operar de acordo com o estilo *Pull* e envia mensagem *"Are you alive?"* para o processo *M1*. Quando o *Timeout T2* expira, o monitor suspeita do processo monitorável, pois a mensagem *"I am alive!"* não foi recebida durante o *timeout T2*.

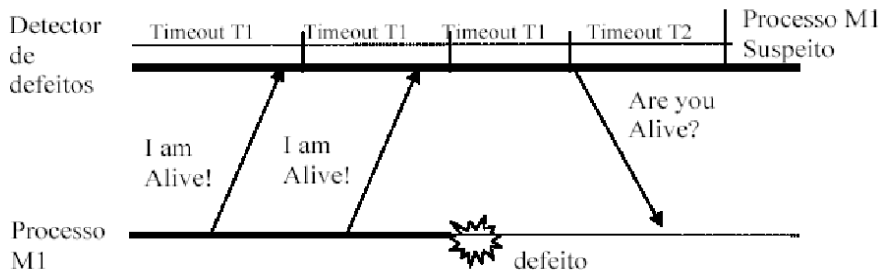

Figura 2.4: Modelo Dual.

2.3.4 Heartbeat

O modelo de detecção de defeitos *Heartbeat* (AGUILERA; CHEN; TOUEG, 1997), diferentemente dos modelos *Push, Pull e Dual*, não utiliza *timeouts*. Neste modelo um nodo monitor *p* mantém para cada nodo monitorado *q* um contador de *heartbeats*. Periodicamente cada nodo monitorado envia uma mensagem *"I am alive"* (*heartbeat*) aos nodos monitores que, ao receber a mensagem, incrementam o contador correspondente àquele monitorado. Caso o contador não seja mais incrementado uma suspeita é levantada. Note que a idéia básica é suspeitar de um processo comparando os valores dos contadores recebidos com os contadores anteriores, esta tarefa de comparação é delegada para a camada de aplicação que está fazendo uso do detector. Essa estratégia de deixar a decisão sobre suspeitas de defeitos para a camada superior faz com que as aplicações tomem as decisões conforme as suas necessidades. Por outro lado, as aplicações se tornam mais complexas.

2.3.5 Gossip

No modelo *Gossip* básico (RENESSE; MINSKY; HAYDEN, 1998) cada nodo da rede é um monitor/monitorado e mantém uma lista contendo o endereço e um inteiro (contador de *heartbeat*) para cada nodo do sistema. A cada intervalo T_{gossip}, um nodo escolhe aleatoriamente um ou mais vizinhos para enviar a sua lista. A lista recebida é unida com a lista que o receptor possui e o maior valor do contador de *heartbeat* presente nas listas é mantido na lista final. Cada nodo mantém o instante do último incremento do contador de *heartbeat*. Se o contador não for incrementado em um intervalo de T_{fail} unidades de tempo então o membro é considerado suspeito. Sua principal vantagem é tolerar a perda de mensagens, mas o tempo de detecção de defeitos aumenta se a probabilidade de perda de mensagens aumenta (RENESSE; MINSKY; HAYDEN, 1998). O protocolo *Gossip* por escolher randomicamente os seus vizinhos que receberão a informação, consegue reduzir o número de mensagens se comparado com a estratégia de disseminação em que a informação é transmitida de todos para todos (*broadcast*).

Uma versão do algoritmo que permite uma maior escalabilidade também foi proposta. O *Gossip* multi-nível, usa a estrutura de domínios e seu mapeamento em endereço IP, o que possibilita identificar domínios e sub-redes e mapear diferentes níveis. Poucas mensagens são trocadas entre as sub-redes e entre os domínios de cada sub-rede o protocolo *Gossip* básico é utilizado. Essa versão consegue reduzir ainda mais o número de mensagens transmitidas entre os processos, o que a deixa mais escalável.

A figura 2.5 apresenta uma descrição em linguagem algoritmica do protocolo *Gossip* básico. Na inicialização do algoritmo, a lista de membros local é vazia (linha 2) e o próprio membro local se adiciona na sua lista (linha 3). Em cada envio de mensagem (a cada intervalo T_{gossip}), o contador de *heartbeats* local é incrementado (linha 6) e então o membro local escolhe aleatoriamente um ou mais vizinhos para enviar a sua lista (linhas 7 e 8). Para detectar um suspeito, o algoritmo varre a lista de membros local procurando por contadores que não foram incrementados a cada $T_{cleanup}$ unidades de tempo (linhas 12, 13 e 14). Por fim, a cada recebimento de mensagem, para cada endereço do nodo e contador de *hearbeats* pertencentes a lista local (linha 17) que também pertencem a lista recebida e contém um valor no contador de *hearbeats* mais recente (linha 18), este valor mais recente é atribuído como sendo o novo valor do contador local (linha 19) e o tempo da última atualização do contador é atualizado (linha 20).

1: **Na inicialização:**
2: $listaLocal = \emptyset$ /* Inicializa a lista de membros local como vazia */
3: Adiciona o membro local na $listaLocal$
4:
5: **Em cada envio de mensagem:**
6: $heartbeatLocal++$ /* incrementa o *heartbeat* local */
7: $vizinhos = escolheVizinhoAleatorio()$ /* escolhe um ou mais vizinhos para enviar a lista local */
8: $send(vizinhos, listaLocal)$ /* envia a lista local de vizinhos */
9:
10: **Para detectar um membro suspeito:**
11: Repete a cada $T_{cleanup}$ unidades de tempo
12: para cada $(endereco, heartbeat) \in listaLocal$ do
13: if($tempoAtual$ - $timestamp[i] > T_{cleanup}$) then
14: adiciona i na lista de suspeitos
15:
16: **Em cada recebimento de mensagem** $receive(listaRecebida)$**:**
17: para cada $(endereco, heartbeat) \in listaLocal$ do
18: if($listaLocal[i]$.heartbeat < $listaRecebida[i]$.heartbeat) then
19: $listaLocal[i]$.heartbeat = $listaRecebida[i]$.heartbeat
20: $listaLocal[i]$.timestamp = $tempoAtual$
21:

Figura 2.5: Algoritmo *Gossip*.

3 REDES MÓVEIS SEM FIO

Na computação móvel sem fio, o principal objetivo é obter acesso as informações de maneira simples e direta através de uma rede de comunicação sem fio. Como exemplo dessa versatilidade, pode ser citado aplicações que permitem celulares se conectarem a computadores portatéis via rede sem fio para troca de dados. Outro exemplo da utilidade de redes sem fio é o *GPS (Global Positioning Systems)*, que são sensores que permitem saber a localização exata de um determinado equipamento na superfície da terra. Neste capítulo as redes ad hoc e redes de sensores são descritas com maiores detalhes. No contexto desse trabalho, uma rede móvel sem fio é caracterizada como sendo uma rede ad hoc ou uma rede de sensores, já que nodos sensores podem ter características semelhantes às redes ad hoc, como comunicação sem fio e mobilidade, por exemplo.

3.1 Redes Ad Hoc

Em uma rede ad hoc os nodos são capazes de se comunicar diretamente com outros sem a necessidade da criação de uma infra-estrutura de rede, tal como *backbones* ou pontos de acesso (MATEUS; LOUREIRO, 2005). Os nodos em uma rede ad hoc se comunicam sem uma conexão física, formando uma rede '*on the fly*', no qual os dispositivos fazem parte da rede apenas durante a comunicação ou, no caso de dispositivos móveis, apenas enquanto estão dentro do alcance de transmissão. Uma rede ad hoc pode ser formada não somente por computadores, mas também por qualquer aparelho que tenha um dispositivo *wireless*, como por exemplo PDAs ou celulares.

Redes ad hoc tem uma preocupação com o roteamento das mensagens, devido a ausência de pontos de acesso ou estações centrais. Com isso, cada nodo da rede passa a ter uma importância maior em comparação a uma rede fixa, pois para que cada mensagem possa chegar a um destino é necessário que haja uma contribuição dos outros nodos que fazem

parte da rede. Devido a mobilidade, a dificuldade de criar rotas se torna ainda maior.

Tabela 3.1: Vantagens e Desvantagens das Redes Ad Hoc.

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

A tabela 3.1 lista algumas vantagens e desvantagens do uso de redes ad hoc.

3.2 Redes de Sensores

Segundo (PEREIRA; AMORIM; CASTRO, 2003) uma rede de sensores pode ser definida como uma rede sem fio formada por um grande número de sensores plantados sob uma base ad hoc para detectar e transmitir algum fenômeno ou característica física do ambiente que se queira medir. As informações coletadas pelos sensores são transmitidas para uma base de dados central.

Já no contexto de sistemas distribuídos, uma rede de sensores é definida como uma classe particular de sistemas distribuídos (HEIDEMANN et al., 2001), possuindo características próprias como o grande número de nodos, topologia de rede dinâmica e consumo de energia restrito devido a alimentação por bateria. Estas características deixam uma rede de sensores próxima ao conceito de uma rede ad hoc e dificultam a reutilização de algoritmos projetados para sistemas distribuídos tradicionais (PEREIRA; AMORIM; CASTRO, 2003).

Outro enfoque para redes de sensores é a de sensores individuais que são usados para coletar informações sobre um determinado fenômeno que podem juntos formarem uma rede sem fio (PEREIRA; AMORIM; CASTRO, 2003). Estes sensores, dependendo do fenômeno a ser monitorado, podem se locomover. Um exemplo dessa situação é o monitoramento comportamental de animais, cujo o sensor se move junto com os movimentos do animal. Portanto, redes de sensores podem ser classificadas como estáticas (onde não existe movimentação dos nodos) e dinâmicas (onde a principal característica é a monitoração de fenômenos móveis). Uma outra classe de redes de sensores são redes de sensores auto-organizáveis (LIM, 2003), que são capazes de espontâneamente e dinamicamente construirem a rede. Os nodos sensores nesse tipo de rede podem se adaptar de forma autônoma a falhas, sendo auto-suficientes e auto-reconfiguráveis.

Um sensor é composto basicamente por 5 componentes (PEREIRA; AMORIM; CASTRO, 2003): o sensor responsável pela detecção do fenômeno a ser monitorado, uma memória para guardar as informações coletadas, um processador, e um transmissor-receptor para o envio e recebimentos de mensagens e dados. Os sensores podem ser usados para monitorar ambientes que sejam de difícil acesso, como por exemplo zonas de guerra e desastres ambientais.

Projeto de protocolos, aplicações e algoritmos para redes de sensores devem levar em consideração algumas características (PEREIRA; AMORIM; CASTRO, 2003):

- **Latência:** refere-se ao tempo de cada medida do fenômeno.
- **Eficiência:** uso correto da energia disponível para alcançar uma maior vida útil ao sensor.
- **Precisão:** os dados de cada medida devem ser precisos.
- **Tolerância a Falhas:** devido as falhas dos sensores a rede deve ser tolerante a falhas.
- **Escalabilidade:** uma rede de sensores pode ter um grande número de sensores que devem ser suportados.

3.3 Problemas nas Redes Móveis Sem Fio

Os algoritmos para detecção de defeitos inicialmente propostos para LANs e/ou WANs vistos no capítulo anterior, devido a alguns problemas próprios das redes móveis sem fio, são difíceis de serem reutilizados em redes móveis sem fio. Tais problemas são apontados por diversos autores (WANG; KUO, June 2003; TAI; TSO; SANDERS, 2004; HUTLE, 2004; FRIEDMAN; TCHARNY, 2005; MATEUS; LOUREIRO, 2005) e encontram-se elencados e detalhados a seguir.

- **Problema da Mobilidade:** em uma rede *wireless* o conceito de mobilidade é muito importante e o algoritmo de detecção de defeitos deve considerar essa propriedade. Os nodos estão sempre em movimento. Hora eles fazem parte da rede, hora eles estão fora do alcance de transmissão. O serviço de detecção de defeitos não deveria suspeitar de um nodo somente porque ele saiu momentaneamente da vizinhança.

- **Problema na Qualidade da Comunicação Wireless:** um grande desafio para detectores de defeitos em redes móveis sem fio é a qualidade na comunicação *wireless*. Esse problema abrange a vulnerabilidade a perda de mensagens e desempenho da rede (largura de banda, atraso na comunicação, latência, etc). Se uma mensagem é perdida no momento em que o algoritmo está em funcionamento, um nodo pode ser identificado erradamente como falho, violando a propriedade de exatidão. Quando é identificado um defeito, e o nodo que a identificou não consegue transmitir esse defeito para todos os outros nodos operacionais existe uma quebra na propriedade de abrangência. O detector de defeitos deve ser capaz de trabalhar sem violar as propriedades de abrangência e exatidão, que são desejáveis para um detector.

- **Problemas de Bateria e Consumo de Energia:** como são alimentados por bateria, os nodos possuem uma séria limitação de energia e o modelo de troca de informações e dados da rede deve ser otimizado para consumo mínimo de energia, maximizando o tempo de operação de um nodo. Achar meios de economizar energia, descobrindo algoritmos cada vez melhores de disseminação de informação pela rede é um grande desafio.

- **Problema de Escalabilidade dos Nodos:** com o aumento da quantidade dos nodos, o que é comum em redes móveis sem fio, os algoritmos de detecção de defeitos podem ter comportamentos distintos, como aumento exponencial no número de mensagens trocadas entre os nodos, o que resultaria em um maior consumo de energia e uma maior probabilidade de perda de mensagens. Os algoritmos de detecção de defeitos devem suportar a escalabilidade das redes móveis sem fio e não deveriam ter um desvio muito grande nos seus comportamentos com o aumento do número de nodos na rede.

3.4 Conclusões Parciais

Este capítulo apresentou as principais características das redes ad hoc e das redes de sensores. Pode-se notar que ambas têm características semelhantes, como por exemplo, mobilidade e consumo de energia limitado. Devido a essas semelhanças, os problemas encontrados nesses tipos de redes que dificultam a migração dos algoritmos de detecção de defeitos projetados inicialmente para redes fixas para as RMSF são os mesmos.

O próximo capítulo deste trabalho apresentará os principais algoritmos de detecção de defeitos para RMSF que tentam resolver os problemas das redes ad hoc e de sensores citados anteriormente neste capítulo e também as suas principais características.

4 DETECTORES DE DEFEITOS PARA REDES MÓVEIS SEM FIO

Em um estudo realizado recentemente (GRACIOLI; NUNES, 2006), os algoritmos de detecção de defeitos para redes móveis sem fio foram classificados em duas estratégias: baseados em formação de *cluster* e baseados no algoritmo *Gossip*. Nas próximas seções são apresentados e comentados alguns dos algoritmos que fazem parte dessas duas estratégias, juntamente com conceitos necessários para que se possa ter um maior entendimento sobre os mesmos.

4.1 Detector baseado em Formação de Cluster

Nesta seção são apresentados alguns conceitos sobre formação de *cluster*, bem como alguns algoritmos de formação de *cluster*. Ainda, um detector de defeitos baseado em formação de *cluster* é descrito.

4.1.1 Algoritmos de Formação de Cluster

Os principais objetivos do algoritmo de formação de *cluster* são permitir que a comunicação em redes móveis sem fio seja robusta, escalável e eficiente e ainda permite resolver o problema da mobilidade dos nodos. Um *cluster* pode ser visto como um círculo com o raio igual ao alcance de transmissão do nodo central. O nodo central é chamado de *clusterhead* (CH), enquanto que um nodo que está a uma distância de um-hop de dois *clusterheads* é chamado de *gateway* (GW). Qualquer nodo dentro do seu *cluster* está a uma distância de um-hop do seu *clusterhead*. Os nodos que não são nem *clusterheads* e nem *gateways* são chamados de membros ordinários (OM). Um *cluster* tem somente um CH e a função do CH, no caso de detectores de defeitos, é suspeitar dos nodos defeituosos que fazem parte do seu *cluster*.

A arquitetura de comunicação pode ser *intra-cluster* ou *inter-cluster* (TAI; TSO; SANDERS, 2004). A comunicação *intra-cluster* é a comunicação somente dentro do cluster. A comunicação *inter-cluster* é a troca de mensagens entre dois ou mais *cluster*, somente *gateways* e *clusterheads* participam dessa comunicação e um algoritmo de roteamento pode ser usado.

Um algoritmo de formação de *cluster* normalmente inicia escolhendo os *clusterheads* e *gateways* que farão parte dos *clusters*. A escolha de um nodo para se tornar um CH ou GW é feita através de algumas regras utilizadas pelos algoritmos de formação de *cluster*. A seguir alguns algoritmos de formação de *cluster* são descritos.

- **Mais Alta Conectividade (*Highest-Connectivity*):** esse algoritmo proposto por (PAREKH, ITS, 1994) calcula o grau de um nodo levando em consideração a sua conectividade. Primeiro cada nodo envia para todos os seus vizinhos (*broadcast*) a lista dos nodos que consegue se comunicar (incluindo ele mesmo). Um nodo é eleito CH se este tem o maior número de nodos conectados, comparando com sua vizinhança. Um nodo é eleito GW se está dentro do alcance de transmissão de dois ou mais CHs.

- **Mais Baixo ID (*Lowest-ID*):** (GERLA; TSAI, 1995) propôs um algoritmo que trabalha no conceito de identificador de nodo (ID). Cada nodo é marcado com um ID único e periodicamente o nodo envia (*broadcast*) a lista com todos os vizinhos que consegue se comunicar (incluindo ele mesmo). O CH é o nodo que possui o menor ID entre sua vizinhança. O GW é aquele que está dentro do alcance de transmissão de dois ou mais CHs. As simulações feitas em (GERLA; TSAI, 1995) mostram que esse algoritmo tem melhor desempenho se comparado com o de mais alta conectividade, pois ele tem uma formação de *cluster* mais estável.

- **Peso do Nodo:** (BASAGNI; CHLAMTAC; FARAGO, 1997) propuseram um algoritmo onde cada nodo recebe um peso, o nodo é escolhido como CH se este possuem o mais alto peso em comparação com os pesos dos seus vizinhos. Simulações feitas em (CHATTERJEE; DAS; TURGUT, 2000) mostram que o algoritmo do peso do nodo tem um melhor desempenho se comparado com os algoritmos de mais alta conectividade e mais baixo ID.

- **Variante do Peso do Nodo:** (CHATTERJEE; DAS; TURGUT, 2000) proporam

uma variante do algoritmo do peso do nodo. O algoritmo usa uma métrica de peso combinada para a escolha dos CHs. Ele leva em consideração o poder de transmissão, a mobilidade e a potência da bateria dos nodos. A idéia básica do algoritmo é ter fatores de pesos flexíveis, adaptando-se as mudanças e necessidades de cada rede. Quanto maior o número de CHs, os pacotes transmitidos passarão por mais nodos, resultando em uma maior latência, por esta razão é buscado um número mínimo de CHs para maximizar os recursos utilizados. Resultados de simulações feitas em (CHATTERJEE; DAS; TURGUT, 2000) mostram que esse algoritmo tem um melhor desempenho em termos de reafiliação [1] comparando com os algoritmos descritos anteriormente.

4.1.2 Algoritmo de Detecção de Defeito

O detector de defeitos proposto em (TAI; TSO; SANDERS, 2004) é baseado em uma arquitetura de comunicação em *cluster*. Para resolver o problema da mobilidade, os *clusters* são controlados e reconfigurados de maneira independente e dinâmica, um em relação a outro. O detector possui dois algoritmos: um responsável pela formação dos *clusters* e outro pelo serviço de detecção de defeitos (SDD).

O algoritmo de formação de *cluster* (AFC) é uma variante do algoritmo *Lowest-ID* proposto em (GERLA; TSAI, 1995) com algumas características próprias:

- **(**i**)** assegura que um GW é afiliado a um e somente um *cluster* e que existirão vários nodos candidatos a GW;
- **(**ii**)** cria CHs substitutos (*Deputy Clusterheads* - DCHs) e *backup gateways* (BGWs) que deixam o detector mais flexível a defeitos dos nodos;
- e **(**iii**)** permite que as informações coletadas no primeiro *round* do AFC sejam utilizadas também no primeiro *round* do algoritmo do SDD[2].

Depois que o CH for identificado pelo AFC, ele identifica cada membro pertencente ao seu *cluster* e transmite a todos (*broadcast*) a organização do conjunto. Cada membro terá uma visão inicial de todo o *cluster* local e o CH saberá de que nodos deverá esperar as mensagens durante a execução do SDD.

[1] Reafiliação: nodos se movendo de um cluster para outro.
[2] Depois do primeiro round, os algoritmos de formação de cluster e do SDD executarão separadamente.

O Algoritmo do SDD proposto em (TAI; TSO; SANDERS, 2004) é baseado em *heartbeats* e consiste em 3 fases. Na primeira, cada membro do *cluster* local *C* envia ao CH e aos seus vizinhos (*broadcast*) uma mensagem *heartbeat* que contenha o seu NID (identificador do nodo). Na segunda, todo nodo envia para o seu CH um relatório de todos os *heartbeats* dos nodos que ele conseguiu ouvir durante a primeira fase. Na terceira, analisando as informações coletadas na primeira e segunda fases, o CH identifica os nodos suspeitos e então transmite uma mensagem de atualização para o *cluster*, indicando sua decisão. Um nodo *v* é determinado falho se e somente se o CH não receber nem o *heartbeat* de *v* na primeira fase nem o relatório de *v* na segunda fase e nenhum dos relatórios recebidos pelo CH na segunda fase reflete o conhecimento do *heartbeat* pertencente ao *v*. Um CH, por sua vez, é julgado suspeito se e somente se o DCH não receber nem o *heartbeat* do CH na primeira fase nem o relatório do CH na segunda fase, e nenhum dos relatórios que o DCH receber contém informações sobre o *heartbeat* do CH e ainda se o DCH não receber a mensagem de atualização dos estados enviada pelo CH na terceira fase.

Um problema na comunicação dentro de um *cluster* é que múltiplos vizinhos podem responder a um pedido de um nodo simultaneamente, resultando em um desperdício de energia. Para tratar desse problema, cada nodo, assim que receber uma mensagem de um vizinho, ajusta um período de espera em função do seu NID (único na rede), permitindo que cada nodo tenha um período de espera diferente e balance seu consumo de energia.

4.2 Detectores baseados no Algoritmo Gossip

Friedman e Tcharny (FRIEDMAN; TCHARNY, 2005) criaram uma adaptação do modelo de detecção de defeitos *Gossip* para redes móveis sem fio. O algoritmo usa um contador de *heartbeat* que é incrementado toda vez que um novo *heartbeat* de algum vizinho é recebido. Na teoria, a cada π unidades de tempo (equivalente a um *timeout*) um *heartbeat* deveria ser recebido, mas na prática isso não acontece devido ao problema da mobilidade ou devido ao aumento do caminho (maior quantidade de nodos presente na rede) e com isso o tempo π pode ser excedido. Para evitar isso, o algoritmo permite a passagem de no máximo γ *heartbeats* para não suspeitar de um nodo entre o recebimento de dois *heartbeats* consecutivos. O algoritmo espera que os nodos estejam em movimento, hora fazendo parte da rede, hora estando fora do alcance de transmissão. O principal

objetivo é encontrar um valor de γ, com o passar dos *rounds*, que mesmo que um nodo vá para fora do raio de ação, este nodo não seja suspeito, pois o valor de γ estará ajustado para o tempo desse movimento. O algoritmo mantém as seguintes propriedades:

- Garante que todos os processos serão eventualmente suspeitos (propriedade de precisão eventualmente forte, como descrita na seção 2.1).
- A cada π unidades de tempo o número total de mensagens enviadas é *O(n)*, sendo *n* o número de nodos da rede.
- Com o aumento linear do número de nodos, o número de mensagens produzidas também aumenta linearmente. Devido a esse aumento, uma sobrecarga na rede pode ocorrer, mas para evitar isso o intervalo entre cada *heartbeat* pode ser escolhido conforme o aumento do número de nodos.
- O algoritmo tem um melhor desempenho quando a conectividade e o número de nodos da rede aumentam. Isso se deve ao fato de que quando a conectividade da rede é ruim, a disseminação de uma detecção de defeito realizada por um nodo alcançará um pequeno número de vizinhos criando um número maior de falsas detecções. Se o número de nodos da rede aumenta, conseqüentemente a conectividade aumenta.

A figura 4.1 descreve o algoritmo para um nodo *i* qualquer. Na inicialização do nodo, o tempo para considerar um nodo suspeito é inicializado com o valor $\beta(\gamma_0)$ (linha 8), onde γ_0 é o tempo máximo esperado para receber o primeiro *heartbeat* do nodo mais distante presente no sistema. O valor de γ_0 é calculado pela fórmula abaixo:

$$\gamma_0 = \left[\frac{D}{D-r}\right]$$

onde *D* é o diâmetro da rede e *r* é a distância esperada que pelo menos um vizinho de *i* esteja de *j*. O diâmetro da rede é calculado pela seguinte fórmula:

$$D = \left[\frac{n-1}{n+1} N\sqrt{2}\right]$$

já o valor *r* é calculado pela equação:

$$r = \left[\frac{N^2}{n-2}\right]$$

sendo n o número de nodos da rede e N o tamanho da área. A cada π unidades de tempo o nodo *i* incrementa seu *heartbeat* (linha 11) e emite um *broadcast* com sua lista local (linha 12). O nodo *i* ao receber a lista do nodo *j*, compara os contadores de *heartbeats* presentes nas duas listas (linha 15) para todo membro *k* e se o contador local for menor do que o contador recebido o membro *k* não é considerado suspeito (linha 16) e um novo valor para o tempo de suspeita do membro é calculado (linha 17) segundo a fórmula:

$$\beta(\gamma) = \gamma * (\pi + \sigma)$$

Onde π é o tempo para enviar um *heartbeat* e σ é o atraso da rede esperado para a distância de um hop. A operação *ArrayMax* (linha 18) simplesmente retorna os maiores valores dos contadores de *heartbeats* comparando os contadores do *array* local com os do *array* recebido. Por fim, o nodo *j* é considerado suspeito se um *heartbeat* não for recebido após a passagem do tempo de suspeita contido na lista local (linhas 20 e 21).

1: **Notação do nodo i:**
2: $alive_i$ vetor dos contadores de *heartbeat* para cada nodo
3: $suspected_i$ vetor de *bitmap* dos nodos suspeitos
4:
5: **Inicialização do nodo i:**
6: $alive_i$ = [0,0...0]
7: $suspected_i$ = [0,0...0]
8: para todo j faça $suspect\_timer_i[j].set(\beta(\gamma_0))$ /* tempo inicial para considerar um nodo suspeito */
9:
10: **A cada π unidades de tempo:**
11: $alive_i[i]$ = $alive_i[i]$ + 1
12: $broadcast(alive, alive_i)$
13:
14: **No recebimento** $receive(alive, alive_j)$ **do nodo** j
15: $\forall$ *k,* if($alive_i[k]$ < $alive_j[k]$)
16: $suspected_i[k]$ = 0
17: $suspect\_timer_i[k].set(\beta(\gamma_f)$
18: $alive_i$ = $ArrayMax(alive_i, alive_j)$
19:
20: **Ao** $suspect\_timer_i[j]$**.timeout**:
21: $suspected_i[j]$ = 1

Figura 4.1: Código do algoritmo proposto em (FRIEDMAN; TCHARNY, 2005) para um nodo *i* qualquer.

Hutle (HUTLE, 2004) propôs um detector de defeitos eventualmente perfeito, denotado por $\diamondsuit\mathcal{P}$, para uma rede que não necessita estar completamente conectada. O número total de nodos do sistema não precisa ser conhecido, porém um nodo apenas necessita

conhecer o *jitter* (variação de atraso) na comunicação entre seus vizinhos. Esse conhecimento na variação de atraso implica em um sincronismo na comunicação direta entre dois nodos vizinhos. O sincronismo é conseguido reservando um limite na largura de banda para o detector de defeitos e limitando o número de mensagens. Devido a falhas de comunicação ou defeitos nos nodos, a rede pode ser particionada. Os nodos podem somente se comunicar com seus vizinhos via *broadcast*. Falhas na comunicação, particionamento da rede e comunicação via *broadcast* são algumas características encontradas em redes ad hoc.

Neste algoritmo (veja a figura 4.2), cada nodo *p* mantém para todo outro nodo *q* que *p* conhece um contador de *heartbeats* que tem o mais recente *heartbeat* recebido de *q*, um contador de distância (número de *hops*) que contém a estimativa sobre a atual distância de *q* e um vetor de *time-stamp* que mantém o último *round* que *p* recebeu um *heartbeat* de *q*. Através do contador de distância, é possível obter uma noção de quantos *hops* de distância está um nodo até outro, com isso o tempo de detecção de defeito de um nodo pode ser ajustado conforme a distância atual presente no contador. Para detectar um defeito é mantido um conjunto de todos os processos que o detector de defeitos não suspeita, chamado de lista de detecção. Conseqüentemente, um processo irá suspeitar de outro se este não estiver na sua lista de detecção.

Inicialmente, *p* coloca o valor zero em seu contador (linha 7) e em sua própria distância (linha 8), e para todo nodo *q* diferente de *p* a distância é inicializada com um valor infinito (linha 8). Também, o próprio nodo *p* se adiciona na lista de detecção (linha 9). A cada intervalo de tempo *T*, *p*:

- incrementa seu contador que também é usado como número do *round* local (linha 13);
- a cada Δ^k *rounds* todos os processos conhecidos com distância *k* são colocados em $unsent_p$ (conjunto de processos que ainda não foram enviados - linha 15). Desta forma, somente mensagens de membros pertencentes ao conjunto $unsent_p$ são enviadas. Com isso existe a garantida que todo *heartbeat* de um processo com distância *k* é enviado no máximo em cada Δ^k *rounds*;
- são enviados e removidos de $unsent_p$, em cada *round*, os processos que tem a menor distância até *p* (linhas 17, 18 e 19);

- se *p* não receber uma nova mensagem de um outro processo previamente detectado, *p* irá suspeitá-lo (linha 22).

1: **Variáveis**
2: $\forall_q \in \prod$: $hbc_p[q], distance_p[q], last_p[q] \in \mathbb{N}$ /* variáveis para controle dos heartbeats */
3: $unsent_p \subseteq \prod$
4: $detected_p \subseteq \prod$ /* lista de detecção */
5:
6: **Inicialização**
7: $\forall_q$: $hbc_p[q]$ = 0
8: $distance_p[p]$ = 0, $\forall_q \neq p$; $distance_p[q] = \infty$
9: $detected_p = \{p\}$
10: $unsent_p = \emptyset$
11:
12: **A cada *T* unidades de tempo:**
13: $hbc_p[p]$ = $last_p[p]$ = $hbc_p[p]$ + 1 /* Isso também equivale ao *round* local*/
14: para cada $k \geqslant 0$, tal que Δ^k divide $hbc_p[p]$:
15: Adiciona todo q com distância $distance_p[q] = k$ ao $unsent_p$
16: para 1 até Δ + 1:
17: q = um ítem em $unsent_p$ na qual a $distance_p[q]$ é mínima
18: remove q de $unsent_p$
19: $broadcast_p(q, hbc_p[q], distance_p[q])$
20: para cada q $\in detected_p$:
21: if($hbc_p[p]$ - $last_p[q])T > \eta\Delta^k + k\varepsilon$ /* $\eta = \frac{2T}{\Delta - 1}$, $k = distance_p[q]$ */
22: remove q de $distance_p$ /* suspeita de q */
23: $distance_p[q] = \infty$
24:
25: **Em cada recebimento** $deliver_p(q, new\_hbc, new\_dist)$:
26: if($distance_p[q] > new\_dist$ + 1)
27: $distance_p[q] = new\_dist$ + 1
28: if($new\_hbc > hbc_p[q]$) /* mais recente *heartbeat* */
29: $hbc_p[q] = new\_hbc$ /* atualiza o *heartbeat* de q*/
30: $last_p[p] = hbc_p[q]$ /* seta a última recebida para o *round* atual */
31: if($q \notin detected_p$)
32: adiciona q a $detected_p$ /* detectou q */

Figura 4.2: Algoritmo proposto em (HUTLE, 2004) para um nodo *p* qualquer.

Neste algoritmo, cada nodo envia *O(n)* mensagens em cada *round*. Ao receber uma mensagem, a distância local do nodo *q* é comparada com a nova distância recebida mais o valor um (linha 26) e se a distância local conter um valor mais alto, a distância recebida mais um é adotada como a nova distância local (linha 27). O contador de *heartbeat* recebido também é testado com o contador local (linha 28). Se o contador recebido conter o valor mais alto, este é considerado como o novo valor do contador local (linha 29) e também o valor do último *round* em que *p* recebeu um *heartbeat* de *q* é atualizado para o valor do *round* local (linha 30). Ainda, se *q* não estiver na lista de detecção local, *p* o adiciona (linha 32).

4.3 Conclusões Parciais

Um detector de defeitos com uma arquitetura de comunicação baseada em *cluster* permite que haja uma visão sobre a hierarquia dos nodos na rede (CH, GW e OM). Permite também, que um algoritmo de detecção de defeitos seja executado paralelamente dentro do *cluster* (TAI; TSO; SANDERS, 2004). Em cada *cluster* existe uma divisão das tarefas, o AFC é responsável por tratar do problema da mobilidade e escalabilidade dos nodos enquanto o algoritmo do SDD é responsável em detectar o defeito. O resultado de uma detecção dentro de um *cluster* é enviado para outros *clusters* através dos GWs, de maneira que seja resistente ao problema da vulnerabilidade a perda de mensagens (TAI; TSO; SANDERS, 2004). Essa estratégia explora a redundância de mensagens que é comum em redes móveis sem fio, o que atenua os efeitos do problema da vulnerabilidade a perda de mensagens, deixando o SDD mais eficiente, robusto e ajuda a contornar os problemas de conectividade causados pela gerência de energia em cada nodo (*sleep/wakeup*).

Os detectores de defeitos baseados no protocolo *Gossip* são flexíveis pois não dependem de uma hierarquia fixa, ou seja, independe de uma topologia (RENESSE; MINSKY; HAYDEN, 1998). A configuração dos nodos é feita dinamicamente através da troca de mensagens, tratando do problema da mobilidade. Em geral, são menos eficientes do que abordagens hierárquicas (TAI; TSO, 2004). Também, possuem a característica de serem robustos, pois cada nodo difunde suas informações para um ou mais de seus vizinhos fazendo com que essas informações possam chegar para outro nodo, que não faz parte da sua vizinhança, tornando a detecção abrangente. As vezes, tendem a sacrificar o alcance do nodo pela velocidade ou eficiência (TAI; TSO, 2004).

5 SIMULAÇÃO

Redes móveis sem fio têm uma grande gama de aplicações, são utilizadas desde a área militar até a área civil. São úteis em ambientes aonde uma infra-estrutura, tais como *backbones* ou pontos de acessos, são impossíveis de serem conseguidos ou se tratando de redes de sensores, aonde é necessário um monitoramento de alguma característica através da medida realizada pelos sensores. Também são caracterizadas principalmente pela grande escalabilidade e mobilidade dos nodos.

Teste de algoritmos, protocolos e pesquisas neste tipo de rede são dependentes das capacidades das ferramentas de simulação, ou seja, na capacidade de escalabilidade do simulador utilizado (SCALABLE WIRELESS AD HOC NETWORK SIMULATION., 2005). Nesta seção são revisados os principais simuladores existentes para RMSF (UCLA, 1998; DARPA, 2006) e é apresentado o simulador JiST/SWANS (BARR; HAAS; RENESSE, 2004b) utilizado nos testes deste trabalho.

5.1 Network Simulator 2

O *Network Simulator 2* ou simplesmente *ns2* (DARPA, 2006) é um simulador amplamente utilizado e conhecido. Tem sido desenvolvido para suportar protocolos e mobilidade em redes sem fios. O núcleo do simulador, que implementa a lógica e as primitivas da máquina de simulação e interage com módulos desenvolvidos em OTcl (uma extensão do Tcl orientada a objeto), foi escrito na linguagem de programação C++. O OTcl é usado para a escrita de *scripts* de simulação e configuração. Ambas as linguagens (C++ e OTcl) podem ser utilizadas para descrever e implementar protocolos ou algoritmos a serem simulados.

O simulador implementa quatro protocolos de roteamento para redes *ad hoc*: DSDV (*Destination-Sequenced Distance Vector*), DSR (*Dynamic Source Routing*), AODV (*Ad*

hoc On-Demand Distance Vector) e TORA (*Temporally Ordered Routing Algorithm*). Suporta o padrão IEEE 802.11 ou TDMA (*Time Division Multiple Access*) para a camada MAC (*Media Access Control*), e para a camada de transporte implementa os protocolos TCP e UDP. Permite que sejam configurados o consumo de energia e o alcance de sinal do rádio em cada nodo. Um nodo está dentro de um plano de coordenadas cartesianas tridimensionais, cujo são usadas em conjunto com as coordenadas de destino e a velocidade do nodo para a sua movimentação durante a simulação. De acordo com pesquisas realizadas (RILEY; AMMAR, 2002) observa-se que o *ns2* pode ser usado para simular poucos milhares de nodos.

O *Network Simulator* ainda conta com uma versão que permite as simulações existentes serem executadas em um ambiente distribuído com um mínimo de mudanças necessárias. O *Parallel/Distributed ns (pdns)*, permite que sejam feitas simulações com um maior número de nodos se comparado com a versão não paralela e distribuída do *ns*.

5.2 Global Mobile Information Systems Simulation Library

O *Global Mobile Information Systems Simulation Library (GloMoSim)* (UCLA, 1998) é um simulador de redes sem fio baseado em biblioteca. Foi desenvolvido utilizando PARSEC (*PARallel Simulation Environment for Complex systems*) (BAGRODIA; MEYER, 1998), que é uma linguagem baseada em C altamente otimizada para simulações seqüenciais e paralelas. O *GloMoSim* tem a facilidade e a vantagem de que novos módulos e protocolos podem ser escritos em PARSEC e adicionados a sua biblioteca, agregando novas funcionalidades ao simulador.

GloMoSim usa arquivos de configurações baseado em texto para descrever os elementos que fazem parte da rede de simulação, a mobilidade e ainda o fluxo de dados entre esses elementos. O simulador implementa uma técnica chamada "agregação de nodo", onde os estados dos múltiplos nodos em simulação são armazenados dentro de uma única entidade PARSEC. Com isso há uma redução no consumo da memória, mas por outro lado, aumenta a complexidade do código (SCALABLE WIRELESS AD HOC NETWORK SIMULATION., 2005). No modelo de programação usado pelo PARSEC, as entidades são usadas para descrever módulos monolíticos, que podem ser protocolos, nodos, *links* ou até mesmo a troca de informações entre as próprias entidades.

O simulador implementa os protocolos AODV, Bellman-Ford, DSR, Fisheye, LAR

(*Location Aided Routing*), scheme 1, ODMRP (*On Demand Multicast Routing Protocol*) e WRP (*Wireless Routing Protocol*) para roteamento. Na camada MAC são suportados os protocolos IEEE 802.11, CSMA (*Carrier Sense Multiple Access*) e MACA (*Multiple Access with Collision Avoidance*). Na camada de transporte implementa UDP e TCP. A versão seqüencial do *GloMoSim* é distribuída livremente. Já a versão paralela é comercializada como *QualNet*. O *GloMoSim* pode ser usado para simulação de rede com milhares de nodos (RILEY; AMMAR, 2002).

5.3 Java in Simulation Time/Scalable Wireless Ad hoc Network Simulation

Java in Simulation Time (JiST) (BARR; HAAS; RENESSE, 2004b) é um simulador baseado em máquina virtual. Ao contrário dos outros simuladores citados acima, os desenvolvedores do *JiST* optaram por não desenvolverem uma nova linguagem de simulação, pois raramente novas linguagens são adotadas pela comunidade científica, também optaram por não criar uma biblioteca específica para a simulação, pois necessitaria de muitos desenvolvedores e também porque bibliotecas não portáveis impõem uma estrutura de programação não-natural para conseguirem desempenho e concorrência (BARR; HAAS; RENESSE, 2004a). Os objetivos do simulador, por outro lado, são executar simulações de eventos discretos usando um padrão (uso de uma linguagem de programação popular), com eficiência (otimização da simulação) e transparência (separar eficiência de precisão[1]). O *JiST* converte uma máquina virtual (a própria máquina virtual java) em um sistema de simulação flexível e eficiente. Nas próximas subseções o simulador é descrito em maiores detalhes.

5.3.1 Arquitetura do JiST

O *JiST* é composto por quatro componentes: um compilador, uma linguagem de tempo de execução junto com sua máquina virtual, o *rewriter* e o *kernel* em tempo de simulação.

O compilador e máquina virtual são componentes padrões da linguagem de programação Java (SUN-MICROSYSTEMS, 1996). O *rewriter* e o *kernel* são responsáveis pela semântica de execução em tempo de simulação. O *rewriter* é um carregador de

[1]Separar eficiência de precisão, neste caso, tem o sentido de computar um valor válido independentemente de como o simulador é construído e executar a simulação sem chamadas a bibliotecas específicas.

classe dinâmico que intercepta todo pedido de carregamento de classe e depois verifica e modifica esses pedidos sem modificar a lógica do programa. O objetivo é transformar as instruções dentro do programa de simulação em código com as semânticas de simulação apropriadas para que o *JiST* possa entendê-las. É implementado usando o *Byte-Code Engineering Library* (DAHM, 2001). O *rewriter* também faz análises que ajudam a otimização no tempo de execução. Durante a execução, as classes modificadas interagem com o *kernel*.

O *JiST* cria a noção de entidades (*entities*), que nada mais são do que os objetos da linguagem Java encapsulados, com isto o simulador consegue ter um controle dos componentes que fazem parte da simulação. O controle das entidades é garantido quando a interface *JistAPI.Entity* é implementada. O *rewriter* é responsável por assegurar que todas as classes que forem rotuladas como entidades estejam de acordo com algumas regras, como por exemplo, garantir que os campos das classes não sejam públicos e nem estáticos, garantir que todo método público retorne *void*, entre outras considerações. O simulador não permite que uma entidade seja referenciada diretamente por outra. Para que isso seja possível, o *rewriter* insere *stub objects* chamados separadores (*separators*) que conseguem isolar uma entidade da outra. O *JiST* ainda cria objetos chamados de *timeless objects*, que são objetos que nunca são modificados em simulações futuras. O programador da simulação pode explicitamente definir um objeto como sendo um *timeless object* através do uso da interface *JiST.Timeless*. As razões pela existência dos *timeless objects* são puramente para melhorar o desempenho, já que um objeto sendo *timeless* pode ser passado para uma entidade por referência e não por valor, evitando cópias de memória desnecessárias.

A figura 5.1 (BARR; HAAS; RENESSE, 2004a) mostra o sistema da arquitetura do *JiST* proposto pelos autores. As simulações são compiladas, analisadas e modificadas pelo *rewriter* e executadas pelo *kernel* de simulação.

5.3.2 Scalable Wireless Ad hoc Network Simulation (SWANS)

O SWANS é um simulador para redes Ad hoc sem fio que executa em cima da estrutura do *JiST*, por isso consegue um alto *throughput* de simulação e um consumo de memória reduzido (veja a próxima subseção). Uma outra grande vantagem desse simulador, comparando com o *ns2* (DARPA, 2006) e o *GloMoSim* (UCLA, 1998), é que

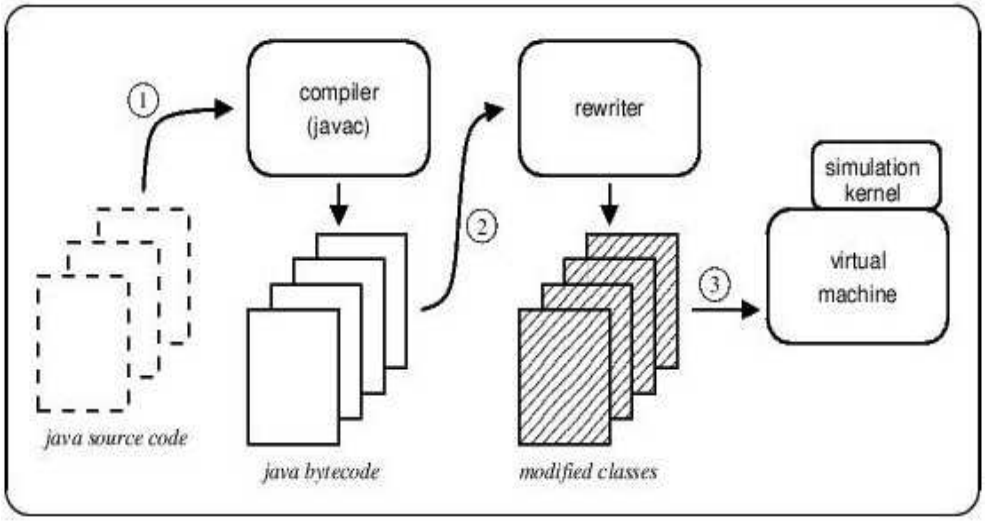

Figura 5.1: Modelo arquitetural do JiST.

códigos escritos na linguagem Java podem ser executados sem nenhuma modificação, já que existe um analisador (*rewriter*) que modifica as chamadas para as classes do padrão Java pelas classes do simulador.

O SWANS pode ser configurado para formar uma rede sem fio completa disponível para a simulação. A figura 5.2 (BARR; HAAS; RENESSE, 2004b) mostra as diferentes classes que podem ser combinadas para caracterizar a formação de nodos da rede. O simulador implementa os algoritmos ZRP (*Zone Routing Protocol*), DSR e AODV para roteamento, os protocolos UDP e TCP para a camada de transporte e MAC IEEE 802.11b.

Tabela 5.1: Exemplos dos componentes existentes no SWANS.

Mobilidade	*RandomWaypoint, RandomWalk e Mobility.Teleport*
Localidade	*Placement.Random*
Aplicação	*AppInterface (AppJava e AppHeartbeat), socket e stream*

A tabela 5.1 apresenta um resumo dos componentes que podem ser usados em simulações realizadas pelo *SWANS*. O modelo de mobilidade pode ser o *RandomWaypoint* ou o *RandomWalk* ou ainda o *Mobility.Teleport*[2]. A localização inicial do nodo é realizada de forma aleatória através da classe *Placement.Random*. O simulador ainda tem exemplos de aplicações: a classe *AppJava* permite que aplicações de rede desenvolvidas em Java sejam executadas no *SWANS* sem modificações, a classe *AppHeartbeat* executa o protocolo de descobrimento de nodos *heartbeat*. Implementa também alguns tipos de *sockets* (*UdpSocket, TcpServerSocket, TcpSocket*) e *streams* (*InputStream, OutputStream, Reader,*

[2]Uma descrição dos modelos de mobilidade pode ser encontrada em (THOMA; NUNES, 2006).

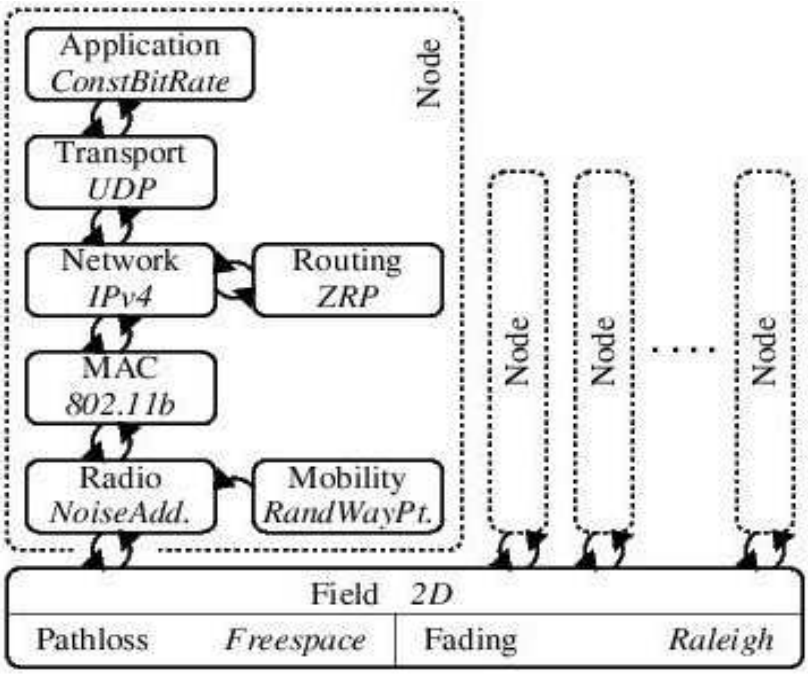

Figura 5.2: SWANS: exemplos de componentes que podem ser configurados para formar uma rede sem fio.

Writer, InputStreamReader, OutputStreamWriter, BufferedReader, BufferedWriter).

Os programas de simulação no *JiST/SWANS* são escritos na linguagem Java. Todos os padrões de classes disponíveis pela linguagem podem ser usados, adicionalmente o desenvolvedor da simulação tem acesso a algumas primitivas presentes no *JiST/SWANS*, tais como o atual tempo de simulação e o avanço no tempo de simulação (*getTime() e sleep()*), por exemplo.

5.4 Avaliação entre os Simuladores

O *JiST/SWANS* por ser desenvolvido na linguagem de programação Java apresenta algumas vantagens (BARR; HAAS; RENESSE, 2004b): portabilidade de código, gerenciamento de memória de objetos e variáveis que não são usadas em um longo período de tempo através do *garbage collection*[3], além da segurança garantida pela isolação dos objetos tratados como entidades. Outros beneficios do simulador são permitir a comunicação inter-processos através do compartilhamento de memória entre as entidades, fazer alterações a nível de *bytecode* não necessitando acesso ao código fonte e o suporte a execução de simulações paralelamente.

[3]O *garbage collection* também é responsável por proteger a memória de possíveis erros, o que deixa o simulador mais robusto.

Nos testes desenvolvidos em (BARR; HAAS; RENESSE, 2004a), os autores compararam o *JiST* com outros simuladores através de *micro-benchmarks* que avaliaram o desempenho de *throughput* e consumo de memória. O *GloMoSim* (UCLA, 1998), o *ns2* (DARPA, 2006) e o *C-based Parsec engine* (BAGRODIA; MEYER, 1998), apresentaram resultados piores em relação ao *JiST*. Os autores também compararam o *SWANS* com os simuladores *ns2* (DARPA, 2006) e *GloMoSim* (UCLA, 1998), que são amplamente utilizados em simulações de redes sem fio. A avaliação se deu através da execução da simulação do protocolo de descobrimento de nodos *heartbeat*, colhendo os resultados em termos de tempo e consumo de memória da simulação. O *SWANS* apresentou cerca de 10 vezes menor o tempo de simulação para uma rede com o mesmo número de nodos em relação ao *ns2*. Com o mesmo consumo de memória o *SWANS* pode simular redes que são uma ou duas vezes maiores do que o *GloMoSim* e o *ns2* conseguem, respectivamente.

Resumidamente, alguns aspectos que contribuem para que o *JiST/SWANS* tenha esse alto desempenho e esse baixo consumo de memória são (SCALABLE WIRELESS AD HOC NETWORK SIMULATION., 2005):

- **Compilação Dinâmica**: sendo a simulação em tempo de execução realizada através da máquina virtual Java, as otimizações de códigos são feitas dinâmicamentes em cada simulação. Além da compilação dinâmica, o *gargabe collection*, a portabilidade e a verificação de segurança em tempo de execução oferecidas pelo Java, também garantem um ganho de desempenho e um baixo consumo de memória.

- **Código *inlining***: o compilador pode eliminar chamadas a funções no *SWANS* e no *kernel* de simulação (*JiST*) adicionando códigos *inline*.

- **Sem troca de contexto**: o *JiST* consegue fazer operações e processamento em entidades sem troca de contextos. Isso é permitido através da isolação das entidades.

- **Sem cópia de memória**: a criação da noção de *timeless objects* permite que objetos sejam passados para entidades por referência, evitando cópias de memória.

- **Modificações nos programas de simulação**: a análise do *bytecode* realizada pelo *rewriter* permite que sejam feitas modificações para que se consiga uma otimização na simulação.

5.5 Conclusões Parciais

Portanto, a escolha do *JiST/SWANS* para a simulação e implementação dos algoritmos se deu devido a algumas características interessantes para este trabalho, tais como:

- o uso da linguagem *Java*, amplamente conhecida e utilizada;
- bom desempenho e um baixo consumo de memória nas simulações;
- e a existência de toda a estrutura necessária para realizar a simulação de uma rede móvel sem fio.

6 AVALIAÇÕES E TESTES

A implementação dos algoritmos e os testes realizados foram executados no simulador *JiST/SWANS* na sua versão 1.0.6. A versão da máquina virtual *Java* utilizada foi a 1.4.2_08.

O modelo de mobilidade utilizado na implementação dos algoritmos foi o *Random Waypoint* (JOHNSON; MALTZ, 1996). Neste modelo, o nodo escolhe uma direção aleatória e uma velocidade entre os valores mínimo e máximo disponíveis e se move até o ponto de destino dentro do campo de simulação. Ao chegar no destino, o nodo permanece parado por um tempo definido (tempo de pausa) e após o término desse tempo repete o processo novamente. Quanto maior é o tempo de pausa mais estática é a rede e quanto mais este valor se aproxima a zero maior é a mobilidade existente na rede. Nos testes, o tempo de pausa estipulado foi de 15 segundos[1], a velocidade varia de 0 (o nodo está parado) até 2 m/s. Os testes foram realizados em um campo de 300m x 300m em uma simulação de 30 minutos e variando o número de nodos e o alcance de transmissão. Três métricas da qualidade de serviços (QoS) em detectores de defeitos propostas por (CHEN; TOUEG; AGUILERA, 2002) e o número de *broadcasts* para cada algoritmo foram avaliadas:

- Tempo de Recorrência ao Erro (*Mistake recurrence time* - T_{MR}): mede o tempo entre dois erros consecutivos cometidos pelo detector (suspeita incorreta).
- Tempo Médio de Duração de Falsas Suspeitas (*Mistake Duration* - T_M): mede o tempo que o detector de defeitos leva para corrigir um erro.
- Tempo de Detecção (*Detection Time* - T_D): mede o tempo para que todos os nodos suspeitem de um nodo defeituoso.

[1] Conforme este tempo de pausa se aproxima a zero, a rede terá uma maior mobilidade.

- Número de *Broadcasts*: mede o número de *broadcasts* em cada algoritmo para poder ter uma estimativa do consumo de energia.

Os primeiros testes realizados tiveram o objetivo de determinar quais parâmetros (T_{gossip} e $T_{cleanup}$ por exemplo) dos algoritmos seriam o ideal para o ambiente de simulação. Após definidos esses valores, uma comparação entre os algoritmos foi possível. Nas próximas seções, a escolha dos parâmetros e os resultados da comparação entre os algoritmos são descritos.

6.1 Escolha dos Parâmetros dos Algoritmos

O objetivo desta primeira bateria de testes é verificar quais são os melhores parâmetros para cada algoritmo. No caso do algoritmo *Gossip*, os testes buscam obter o T_{gossip} e o $T_{cleanup}$ que têm os melhores valores para as métricas desejadas. O valor do T_{gossip} também serve como base para os outros dois algoritmos baseados no *Gossip*. Já no algoritmo baseado em *cluster*, busca-se obter o melhor valor para o tempo (*timeout*) entre a primeira e segunda fases (T_{fases}) e também o tempo entre duas primeiras fases consecutivas (T_{SDD}), como exemplificado na figura 6.1.

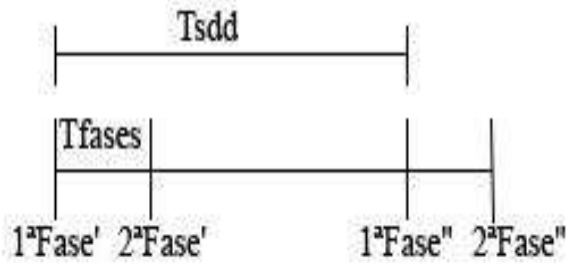

Figura 6.1: Exemplo das fases e *timeouts* do algoritmo baseado em formação de *cluster*.

6.1.1 Valores para o Algoritmo *Gossip*

Como descrito na seção 2.3.5, o algoritmo *Gossip* possui parâmetros que são usados para o seu funcionamento. A cada intervalo de tempo T_{gossip} o nodo escolhe aleatoriamente um ou mais vizinhos para enviar a sua lista e a cada $T_{cleanup}$ unidades de tempo o nodo varre a sua lista procurando por vizinhos que não atualizaram seus contadores de *heartbeats*. Porém, para o ambiente de teste apresentado anteriormente, esses valores

ainda não são conhecidos. Por esta razão, a primeira tarefa é escolher quais os valores para T_{gossip} e $T_{cleanup}$ que melhor se encaixam no ambiente de simulação. Para isso o número de nodos foi 20 com um alcance de transmissão de 25 metros. Os valores para T_{gossip} foram fixados em 5, 8, 10, 12 e 15 segundos. Já os valores para o $T_{cleanup}$ foram atribuídos como sendo igual, 2 e 3 vezes o valor do T_{gossip}.

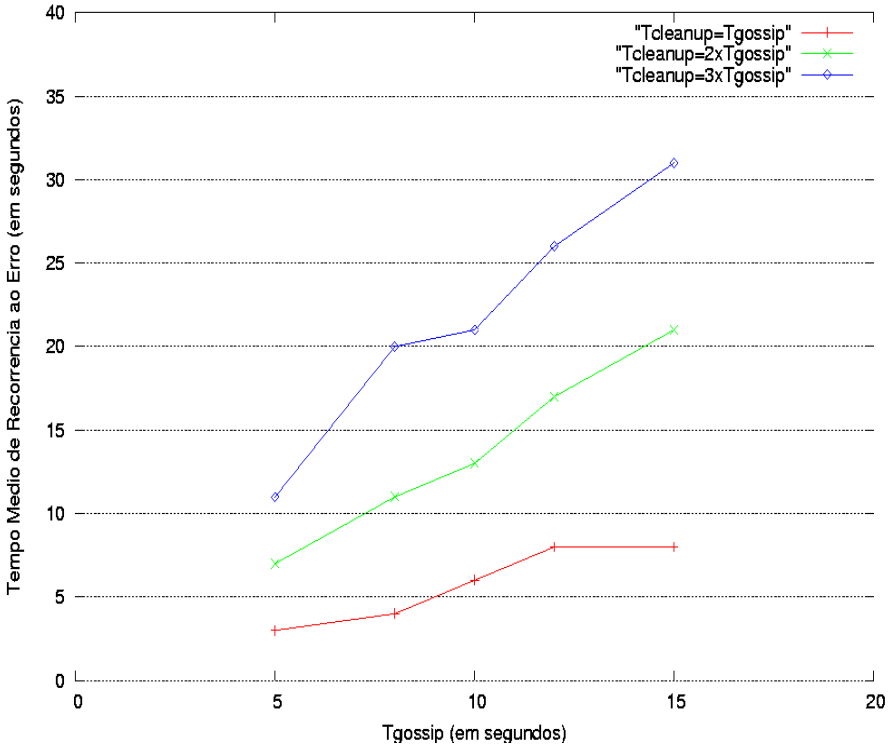

Figura 6.2: Tempo médio de recorrência ao erro (T_{MR}) do algoritmo *Gossip*.

O primeiro teste avaliou o tempo médio de recorrência ao erro e é apresentado na figura 6.2. Pode-se perceber que o T_{MR} é diretamente proporcional ao valor do $T_{cleanup}$. Quanto maior o $T_{cleanup}$, maior o T_{MR}. Isso se deve ao fato que um nodo com um valor para o $T_{cleanup}$ igual a 3 vezes o T_{gossip}, varre a sua lista procurando por nodos que não atualizaram seu contador de *heartbetas* menos vezes do que um nodo que tem o valor igual ao T_{gossip}. Já o T_{gossip} também apresentou uma influência no valor do T_{MR}, conforme o seu incremento o T_{MR} também obteve um aumento. Isso pode ser explicado pois o T_{gossip} influência na freqüência do envio das mensagens.

A seguir o tempo médio de duração de falsas suspeitas foi medido. Conforme a figura 6.3, com o aumento do T_{gossip}, o T_M também sofre um aumento. Já para o $T_{cleanup}$, o valor de T_M também sofre uma variação, mas essa variação é menor se comparada com o T_{gossip}. O T_{gossip} tem uma maior influência neste caso pois aumenta o intervalo entre o

envio da lista de vizinhos de cada nodo, com isso, aumenta também o tempo que um nodo leva para obter ciência que sua suspeita está incorreta.

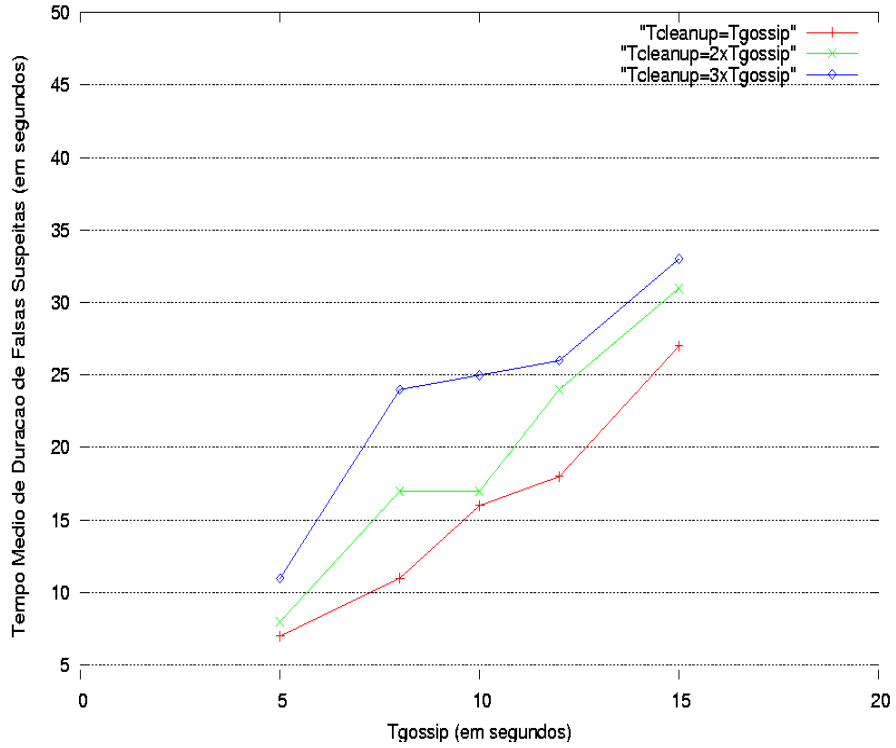

Figura 6.3: Tempo médio de duração de falsas suspeitas (T_M) do algoritmo *Gossip*.

Em seguida, o tempo de detecção foi avaliado. A figura 6.4 mostra um aumento linear no T_D conforme o T_{gossip} e o $T_{cleanup}$ aumentam. O T_D está diretamente ligado nas freqüências de envio das mensagens e na procura por nodos que não incrementaram seus contadores de *heartbeats*. Então com um aumento do T_{gossip} e do $T_{cleanup}$, o T_D também se torna maior.

Com base na análise dos gráficos, o valor do T_{gossip} escolhido foi 12 segundos. E o valor para o $T_{cleanup}$ foi 2 vezes o valor do T_{gossip}, 24 segundos.

6.1.2 Valores para o Algoritmo Baseado em *Cluster*

O detector de defeitos proposto em (TAI; TSO; SANDERS, 2004) é baseado em uma arquitetura de comunicação em *cluster* (como visto na seção 4.1). Neste algoritmo, são realizadas duas fases. Na primeira, o nodo envia um *broadcast* contendo seu ID e na segunda o nodo envia a lista de todos os vizinhos que foi possível receber o ID na primeira fase. O tempo (*timeout*) entre a primeira e segunda fases é chamado de T_{fases} e o tempo entre duas primeiras fases consecutivas (tempo entre o envio de dois *broadcasts*)

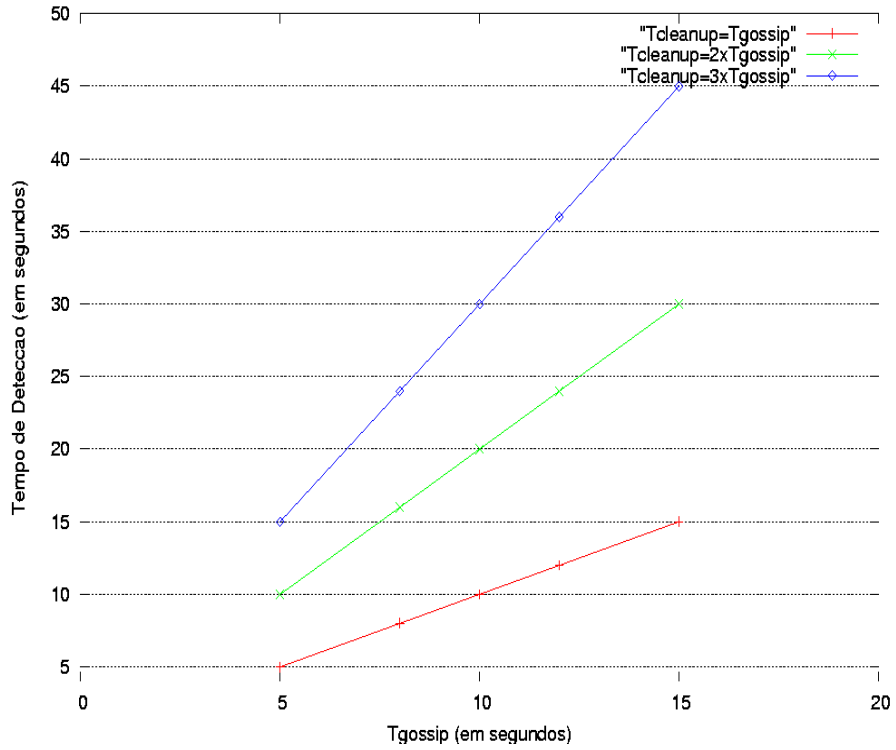

Figura 6.4: Tempo de detecção (T_D) do algoritmo *Gossip*.

é chamado de T_{SDD}. Assim como o algoritmo *Gossip*, esses valores não são conhecidos, portanto, os resultados dos testes apresentados a seguir buscam obter os melhores valores para o T_{fases} e T_{SDD} para o ambiente de simulação.

Em todos os testes realizados neste algoritmo o T_{SDD} tem o valor igual ao T_{gossip}, ou seja, 12 segundos. No entanto, os valores considerados para o T_{fases} são relacionados com o T_{SDD} e são apresentados na tabela 6.1.

Tabela 6.1: Valores considerados para o T_{fases}.

Relação entre o T_{fases} e o T_{SDD}	**Valor em Segundos**
T_{fases} = T_{SDD} / 3	4
T_{fases} = T_{SDD} / 2	6
T_{fases} = T_{SDD} * 3 / 4	9
T_{fases} = T_{SDD}	12

A figura 6.5 apresenta as médias em segundos para o tempo de detecção, tempo médio de recorrência ao erro e tempo médio de duração de falsas suspeitas. O T_D mostrou-se inversamente proporcional ao T_{fases}, ou seja, diminuiu o seu valor com o aumento do T_{fases}. Isso se deve ao fato de que quanto mais o T_{fases} se aproxima do T_{SDD} as duas fases vão se tornando uma só e assim o T_D diminui. Já o T_{MR} começou em um valor muito

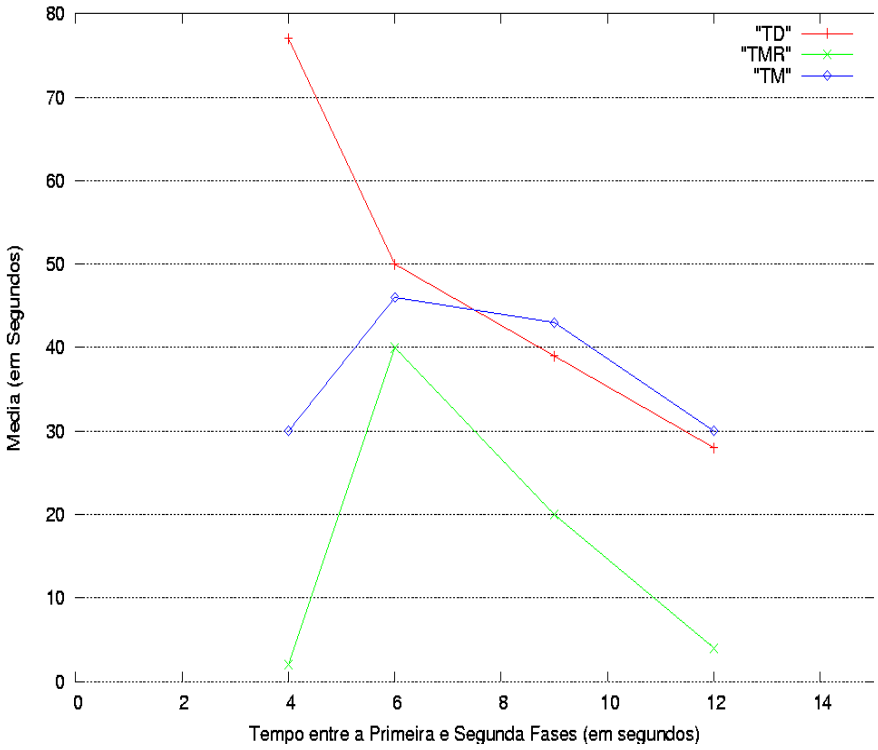

Figura 6.5: Tempo de detecção (T_D), tempo médio de recorrência ao erro (T_{MR}) e tempo médio de duração de falsas suspeitas (T_M) para o algoritmo baseado em *Cluster*.

ruim e atingiu o seu melhor estado com um valor para o T_{fases} de 6 segundos. Vale a pena lembrar que quanto mais alto o valor para o T_{MR}, melhor é para o detector de defeitos, pois menos erros são cometidos. Analisando o T_M, os melhores valores são alcançados com um T_{fases} igual a 4 e 12 segundos. A explicação para isso é que com um T_{fases} igual a 4 segundos mais vezes a mensagem contendo a lista que pôde ser recebida na primeira fase pode ser transmitida na segunda fase. Já para o T_{fases} igual a 12 segundos, as duas fases são unidas em uma só, então, em um mesmo *broadcast*, é possível enviar o ID e a lista recebida.

Em uma análise das métricas do T_D, T_{MR} e T_M o valor para o T_{fases} escolhido foi 6 segundos, pois no gráfico foi o valor que menos variou nas 3 métricas. Portanto, após definidos os valores de T_{gossip} (12 segundos), $T_{cleanup}$ (2 vezes T_{gossip}), T_{SDD} (12 segundos) e T_{fases} (6 segundos) é possível comparar os algoritmos variando o número de nodos (20, 30, 40, 50 e 60) e o alcance de transmissão (25, 50 e 75 metros), bem como o número de *broadcasts*. Os resultados dessa comparação são apresentados na próxima seção.

6.2 Comparação entre os Algoritmos

6.2.1 Tempo Médio de Recorrência ao Erro

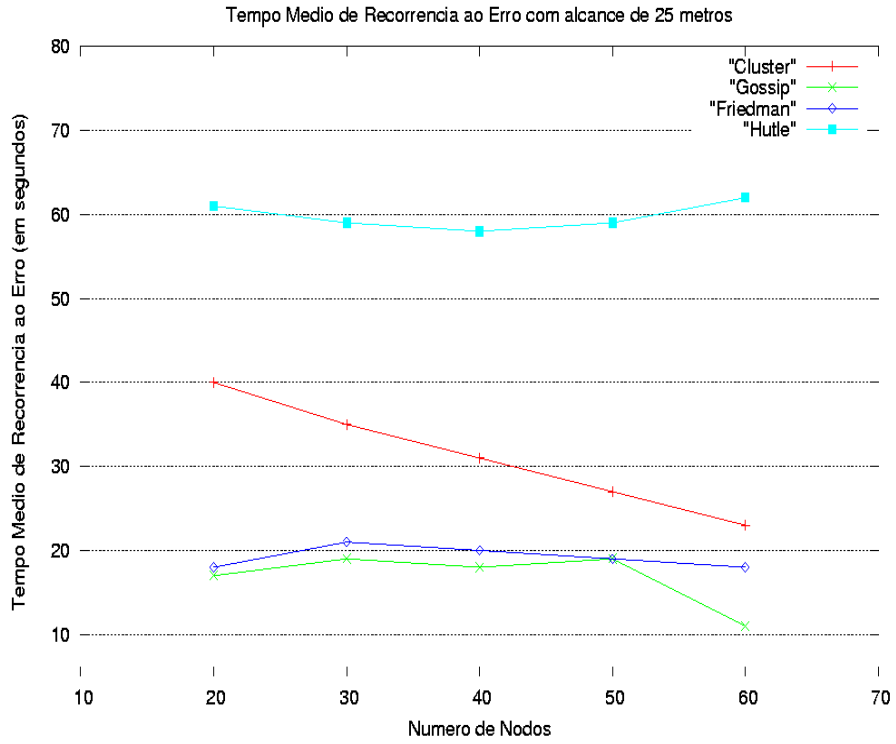

Figura 6.6: Tempo médio de recorrência ao erro com um alcance de transmissão dos nodos de 25 metros.

A figura 6.6 apresenta os valores para o T_{MR} com um alcance de transmissão dos nodos de 25 metros. Nesta métrica, quanto maior for o seu valor, mais preciso é o detector de defeitos, pois menos vezes um nodo será considerado suspeito. Analisando o gráfico, pode-se perceber que o algoritmo proposto pelo *Hutle* tem os melhores valores para a métrica medida. Já o detector baseado em *cluster* apresenta um declínio em seus valores conforme o número de nodos aumenta, o que não acontece com os detectores baseado no algoritmo *Gossip* e com o próprio algoritmo *Gossip*, pois seus valores apresentam uma pequena variação para mais ou para menos e são praticamente constantes.

Na figura 6.7 os valores do T_{MR} com um alcance de transmissão de 50 metros são apresentados. Novamente o detector proposto pelo *Hutle* obteve os melhores resultados. Porém, esses valores são um pouco mais baixos se comparado com os valores do alcance de transmissão de 25 metros, pois com um maior alcance de transmissão os nodos se comunicam com mais vizinhos e assim a probabilidade de considerar suspeitos aumenta. Com o detector baseado em *cluster*, os valores para a métrica se apresentaram

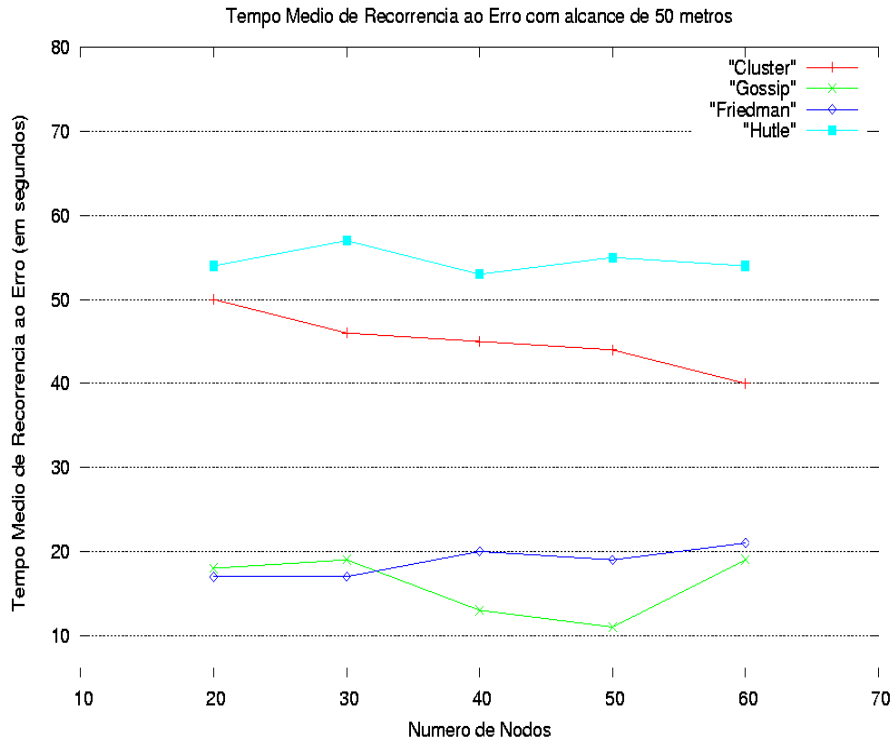

Figura 6.7: Tempo médio de recorrência ao erro com um alcance de transmissão dos nodos de 50 metros.

mais constantes do que os anteriores, porém, continua existindo uma queda com o aumento no número de nodos. Já os valores para o algoritmo *Gossip* e o detector proposto pelo *Friedman* são novamente inferiores aos outros dois detectores e não apresentaram grandes mudanças.

Os valores do T_{MR} para os detectores de defeitos com um alcance de transmissão dos nodos de 75 metros são apresentados na figura 6.8. Neste gráfico, o detector de defeitos baseado em *cluster* apresentou melhores resultados para a métrica com 20, 30 e 50 nodos. Pode ser concluído, que neste detector, conforme o alcance de transmissão aumenta, o T_{MR} também aumenta, porém conforme o número de nodos aumenta o T_{MR} diminui, pois existe uma maior probabilidade de recorrer ao erro. Já com o algoritmo proposto pelo *Hutle*, o T_{MR} se mantém praticamente constante com o aumento no número de nodos, mas diminui com o aumento do alcance de transmissão. Isso é devido a maior disseminação da informação que ocorre quando o alcance de transmissão é maior. Por outro lado, o algoritmo *Gossip* e o detector proposto pelo *Friedman* não obteram uma grande mudança em seu comportamento mesmo com a variação no número de nodos e alcance de transmissão.

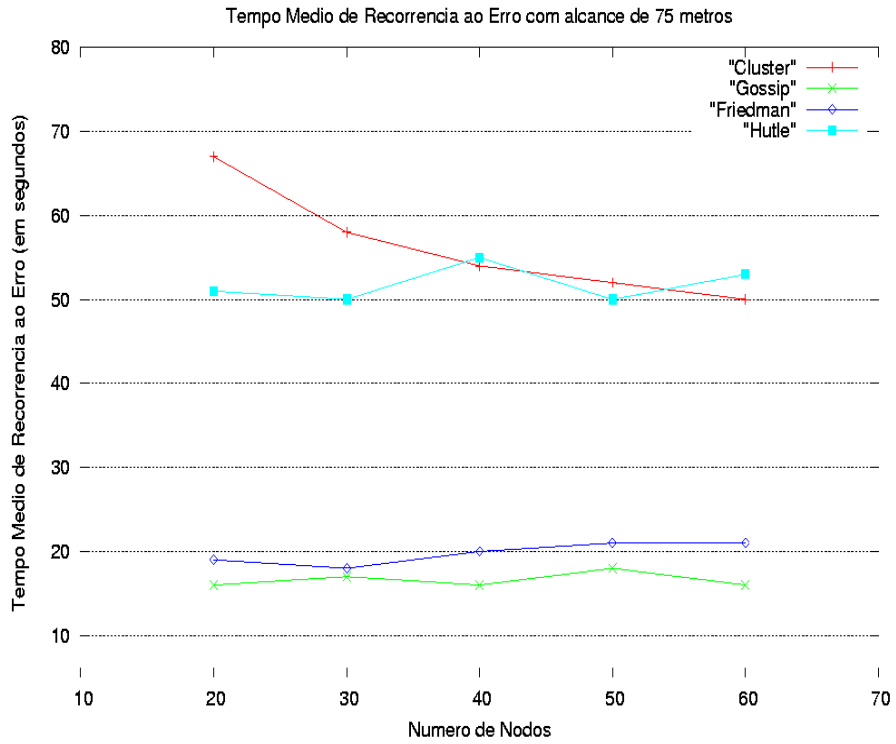

Figura 6.8: Tempo médio de recorrência ao erro com um alcance de transmissão dos nodos de 75 metros.

6.2.2 Tempo Médio de Duração de Falsas Suspeitas

O tempo médio de duração de falsas suspeitas mede o tempo que o detector de defeitos leva para corrigir um erro, ou seja, é o tempo de duração de um erro. Quanto menor for esse tempo, mais confiável e exato é o detector de defeitos. A figura 6.9 apresenta os valores para o T_M com um alcance de transmissão dos nodos de 25 metros. O gráfico mostra que os melhores resultados para a métrica são do algoritmo proposto pelo *Friedman*. Um comportamento semelhante também foi obtido pelo algoritmo *Gossip*. Já o detector de defeitos baseado em *cluster* obteu um aumento no valor da métrica conforme o acrescímo no número de nodos mais significativo do que os algoritmos baseado no *Gossip* e também apresentou o pior resultado para uma rede com 60 nodos.

Na figura 6.10 os valores do T_M para um alcance de transmissão de 50 metros podem ser analisados. Pode-se perceber que todos os algoritmos obteram uma diminuição nos valores para a métrica. Isso é explicado devido ao maior alcance e portanto, a comunicação é feita com mais vizinhos. Mais uma vez os algoritmos proposto pelo *Friedman* e o *Gosisp* obteram os melhores valores. Por outro lado, o detector baseado em *cluster* obteve um melhor resultado do que o detector proposto pelo *Hutle* mesmo em uma rede

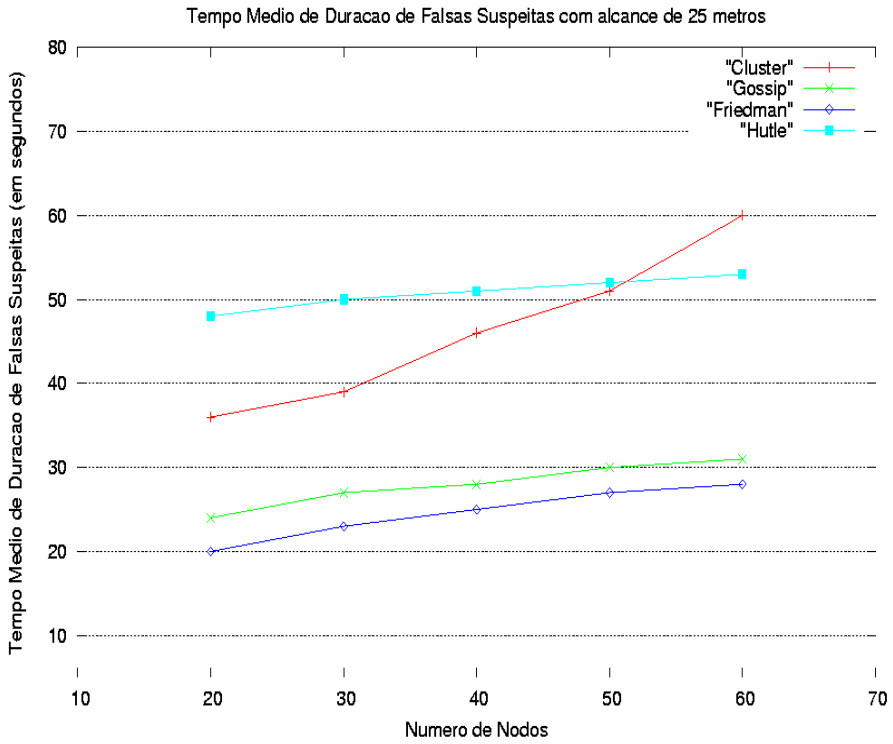

Figura 6.9: Tempo médio de duração de falsas suspeitas com um alcance de transmissão dos nodos de 25 metros.

com 60 nodos.

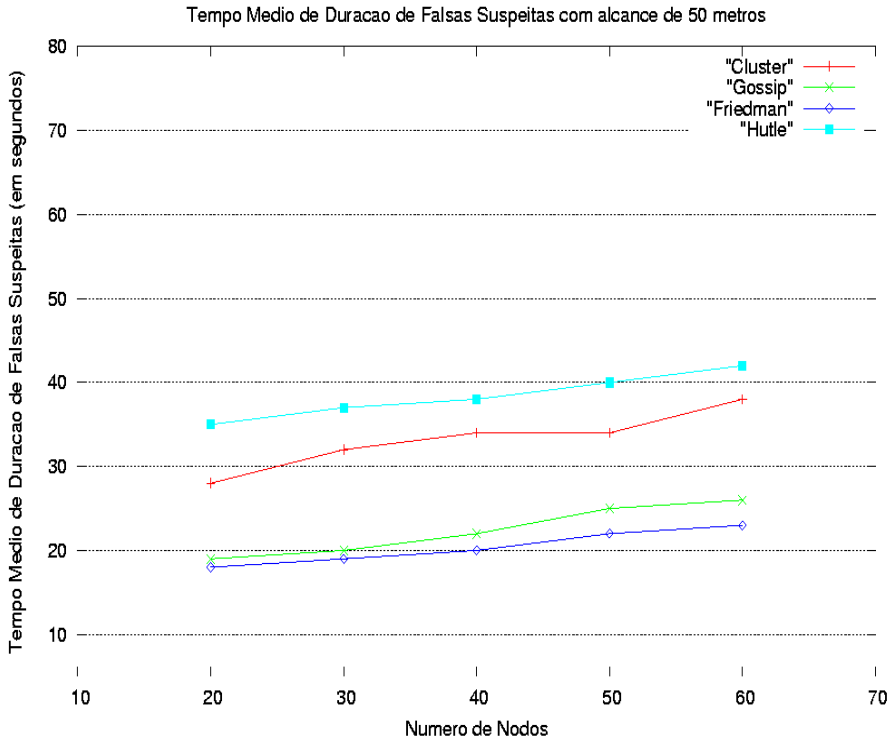

Figura 6.10: Tempo médio de duração de falsas suspeitas com um alcance de transmissão dos nodos de 50 metros.

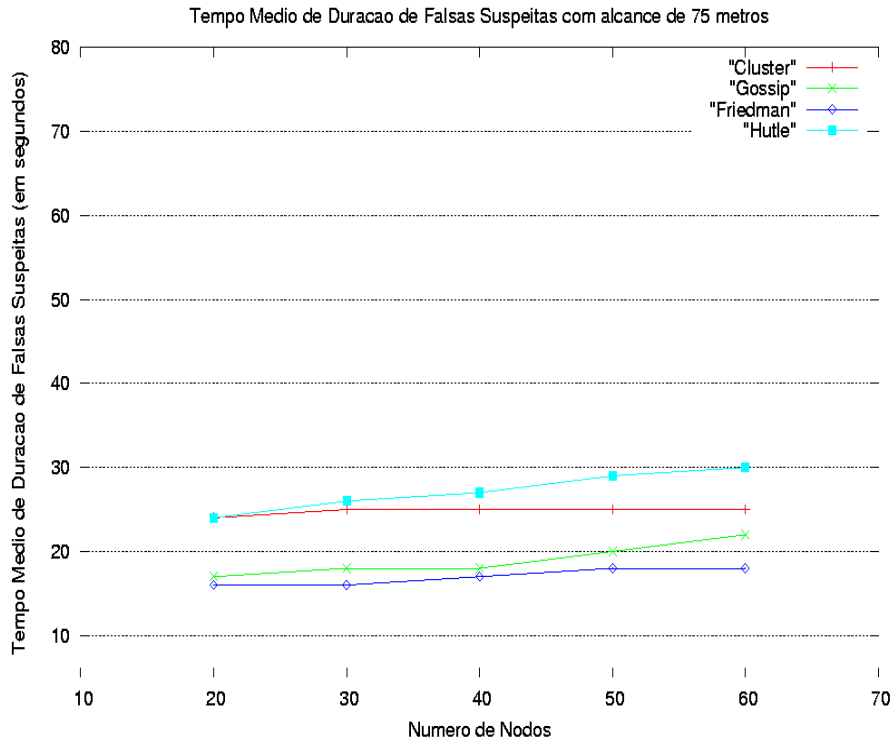

Figura 6.11: Tempo médio de duração de falsas suspeitas com um alcance de transmissão dos nodos de 75 metros.

Os valores do T_M para o alcance de transmissão dos nodos de 75 metros são apresentados na figura 6.11. Mais uma vez, todos os algoritmos diminuíram os seus valores para o T_M se comparado com os valores anteriores, justamente devido ao aumento do alcance de transmissão dos nodos. Novamente o detector proposto pelo *Friedman* e o *Gossip* apresentaram os melhores valores e um comportamento semelhante. Por outro lado, o detector baseado em *cluster* e o detector proposto pelo *Hutle* melhoraram o seu desempenho. Em uma análise, o comportamento desses dois algoritmos tiveram uma melhora mais significativa conforme o aumento do alcance de transmissão se comparado com o algoritmo *Gossip* e o algoritmo do *Friedman* que apresentaram um comportamento semelhante mesmo com a variação do alcance de transmissão.

6.2.3 Tempo de Detecção

Como explicado anteriormente, o tempo de detecção mede o tempo para que todos os nodos suspeitem de um nodo defeituoso. Quanto menor for o valor do T_D, mais rápido o detector tomará consciência dos nodos defeituosos e conseqüentemente será um detector mais veloz. A figura 6.12 mostra os valores para o T_D com um alcance de transmissão de 25 metros. Os detectores de defeitos baseados no algoritmo *Gossip* apresentaram um

tempo de detecção praticamente constante, sem grandes variações. Especificamente, o T_D no algoritmo *Gossip* se mostrou dependente do valor do $T_{cleanup}$. O algoritmo proposto pelo *Hutle* apresentou os melhores resultados. Já o detector baseado em *cluster* sofreu um aumento no seu T_D com 50 e 60 nodos.

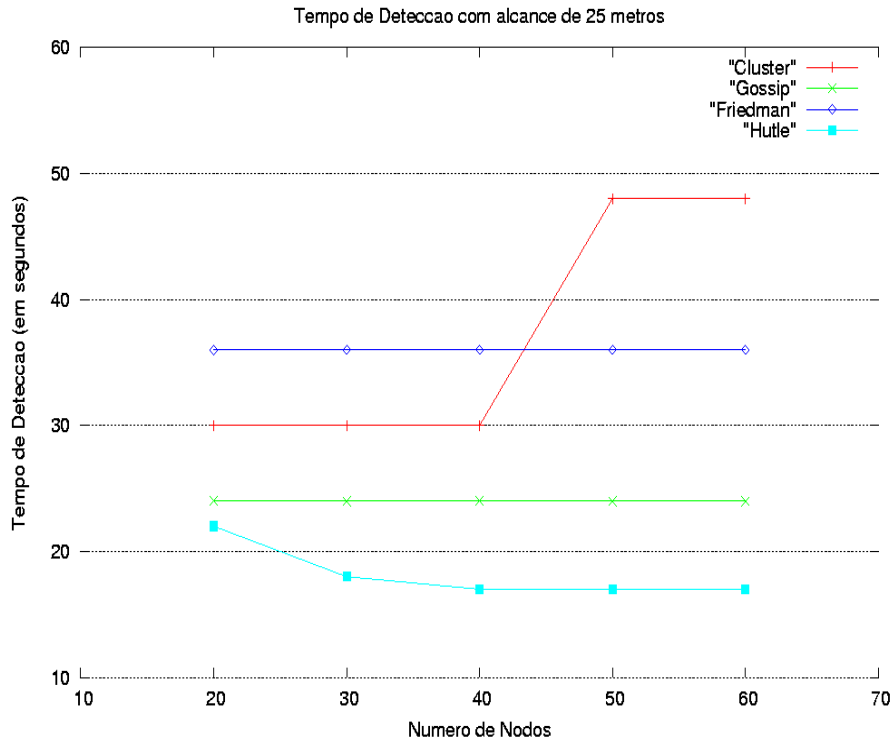

Figura 6.12: Tempo de detecção com um alcance de transmissão dos nodos de 25 metros.

A figura 6.13 mostra o T_D para um alcance de transmissão dos nodos de 50 metros. Analisando o gráfico, o algoritmo *Gossip* e o algoritmo do *Friedman* mantiveram os mesmos valores em comparação com o gráfico do alcance de transmissão de 25 metros. Por outro lado, o detector baseado em *cluster* apresentou uma redução em seus tempos médios. Isso porque com um alcance de transmissão maior, o número de reafiliações é menor, isto é, menos nodos se movem de um *cluster* para outro e portanto menor a probabilidade de erros e menor o tempo de detecção. O algoritmo proposto pelo *Hutle* novamente obteve os melhores resultados e se mostrou constante mesmo com o aumento no número de nodos.

Por fim, os resultados para o T_D com um alcance de transmissão dos nodos de 75 metros são apresentados na figura 6.14. O algoritmo do *Friedman* sofreu uma diminuição em seus valores. Sendo assim, apresentou os mesmos resultados do que o algoritmo *Gossip*. Outro ponto a ser lembrado é que o detector baseado em *cluster* obteve os valores

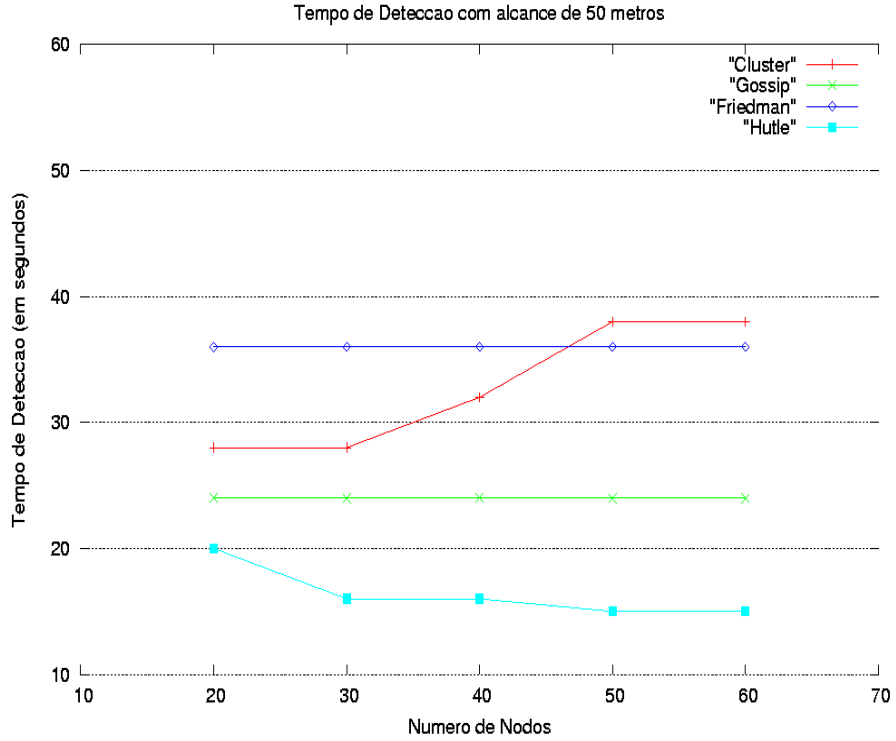

Figura 6.13: Tempo de detecção com um alcance de transmissão dos nodos de 50 metros.

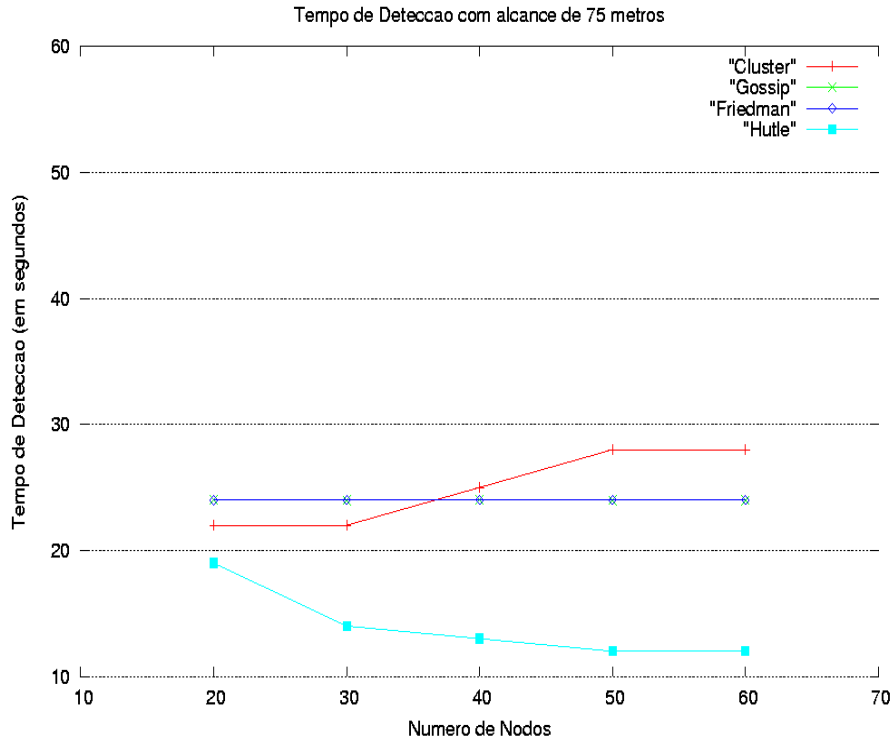

Figura 6.14: Tempo de detecção com um alcance de transmissão dos nodos de 75 metros.

mais baixos conforme o alcance de transmissão aumenta e também aumenta o tempo de detecção com o acréscimo no número de nodos. O algoritmo do *Hutle* apresenta os

melhores valores para o T_D e sofre pouca variação com os aumentos no alcance de transmissão e no número de nodos.

6.2.4 Número de *Broadcasts*

A tabela 6.2 mostra o número total de *broadcasts* para o algoritmo *Gossip*, para os detectores propostos pelo *Hutle* e *Friedman* e para o detector baseado em formação de *cluster*. O algoritmo *Gossip* e o algoritmo proposto pelo *Friedman* possuem o mesmo número de *broadcasts* mesmo com a variação no alcance de transmissão, isso porque o envio de mensagens não depende do alcance e sim do intervalo entre o envio de duas mensagens consecutivas. Já o algoritmo proposto pelo *Hutle*, sofre um aumento no número de *broadcasts* conforme o alcance de transmissão também aumenta. Isso se deve ao fato de que o algoritmo envia, a cada intervalo de tempo, um *broadcast* para cada vizinho que tem uma distância mínima (um *hop*). Dessa forma, com o aumento do alcance de transmissão um nodo passa a ter mais vizinhos com a distância mínima e conseqüentemente envia um maior número de *broadcasts*. Por fim, no detector de defeitos baseado em *cluster*, o número de *broadcasts* sofre uma pequena queda com o aumento do alcance de transmissão. A explicação para isso é que com um alcance de transmissão maior, menos *cluster* são formados e conseqüentemente um menor número de *clusterheads*, portanto menos mensagens de atualizações dos *clusters* são emitidas pelos *clusterheads*.

Tabela 6.2: Número de *broadcasts* para os detectores de defeitos com variação no alcance de transmissão e número de nodos.

Detector de Defeitos	**20 nodos**	**30 nodos**	**40 nodos**	**50 nodos**	**60 nodos**
Cluster 25 metros	5285	7890	10469	13099	16059
Cluster 50 metros	5245	7875	10444	13069	15802
Cluster 75 metros	5225	7828	10424	13019	15703
Gossip	3000	4500	6000	7500	9000
Friedman	3000	4500	6000	7500	9000
Hutle 25 metros	6667	13294	14587	26531	50372
Hutle 50 metros	6797	14110	20016	32456	52404
Hutle 75 metros	7700	15536	29835	36542	54790

6.3 Conclusões Parciais

Após a análise dos resultados, pode-se concluir que o detector de defeitos proposto pelo *Hutle* apresentou o melhor comportamento no T_{MR} e T_D. Isso significa que para um

ambiente semelhante ao ambiente simulado, onde a velocidade para detectar um erro e a recorrência ao erro sejam os fatores mais relevantes, este algoritmo se torna a melhor opção. O detector de defeitos baseado em *cluster* também obteve bons resultados para o T_{MR}, principalmente quando o rádio tem maior alcance, mas não demonstrou competitividade em outras métricas. Por outro lado, em um ambiente onde a energia consumida seja o fator mais importante, o detector proposto pelo *Friedman* e o *Gossip* são os que apresentaram menores números no envio de *broadcasts*. O algoritmo do *Friedman* também apresentou os melhores valores para o T_M e também poderia ser uma opção onde essa métrica fosse a mais relevante.

7 PROPOSTA DE UM ESTUDO DE CASO

Tecnologias de comunicação sem fio têm despertado um interesse especial em profissionais da área da saúde. Essas tecnologias deixam o atendimento ao paciente mais seguro e preciso, além de fornecer um suporte para melhor controlar as suas informações e os seus sinais vitais. Este capítulo apresenta um estudo de caso onde um detector de defeitos pode ser usado em um ambiente hospitalar para ajudar e facilitar os médicos no atendimento aos seus pacientes. A descrição do ambiente e a função do detector de defeitos dentro desse ambiente serão detalhados nas próximas seções.

7.1 Descrição do Ambiente

O ambiente aqui utilizado será um hospital. Nele, os pacientes estarão ligados aos seus marca-passos que se conectarão a computadores de mão (PDAs) através de uma tecnologia sem fio (*Bluetooth*). A tecnologia *Bluetooth* (BLUETOOTH, 2007) permite uma comunicação sem fio que transmite dados e voz entre quaisquer dispositivos eletrônicos, desde que em curtas distâncias e que estejam equipados com um transmissor *Bluetooth*. Assim, esta tecnologia permite que um número elevado de comunicações ocorra dentro de uma mesma área. De modo diferente de outras soluções ad hoc onde todos os dispositivos compartilham o mesmo canal, no *Bluetooth* existe um grande número de canais independentes e não-sincronizados cada qual servindo somente a um número limitado de participantes. Os marca-passos são responsáveis por enviar informações sobre os sinais vitais dos pacientes para os PDAs, dando a possibilidade de um diagnóstico remoto aos médicos conectados ao sistema através dos seus dispositivos portáteis. Um exemplo desse cenário é o projeto *CardioMonitor* - Sistema de Monitoramento Cardíaco com Conectividade *Wireless* (MARTINS et al., 2006), que trata de uma rede de troca de informações e

dados formada por eletrocardiógrafos[1] e/ou marca-passos[2]. O ambiente é exemplificado na figura 7.1.

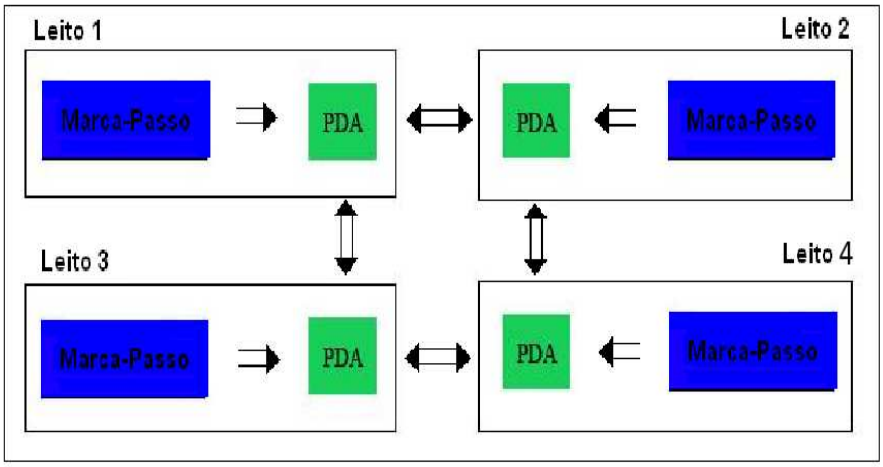

Figura 7.1: Exemplo do ambiente hospitalar. O marca-passo envia as informações do paciente para o PDA e os PDAs formam um rede ad hoc.

Eletrocardiógrafos e/ou marca-passos, quando conectados a uma rede, dão suporte para localizar um médico ou enfermeiro e também possibilitam o armazenamento e monitoramento das informações sobre os pacientes em tempo real, bem como a visualização das curvas dos sinais, independentemente do local. Porém, para garantir que o sistema computacional, ou seja, a rede formada pelos PDAs, funcione corretamente e seja confiável e segura, técnicas de tolerância a falhas auxiliam para que isso seja possível.

A detecção de defeitos é um bloco de construção importante para a implementação de técnicas de tolerância a falhas, permitindo que se crie uma noção de estados (corretos ou incorretos) entre os computadores portáteis, o que é importante para o desenvolvimento e validação de sistemas confiáveis que executam em ambientes assíncronos sujeitos a defeitos, como é o caso de redes hospitalares sem fio. A função do detector de defeitos dentro do sistema é descrita a seguir.

[1]O eletrocardiógrafo é um instrumento que realiza a mensuração de um eletrocardiograma. Um eletrocardiograma é um sinal biológico de natureza analógica, ou seja, é uma variação contínua e diminuta de potencial elétrico em função do tempo, que precisa ser amplificada eletronicamente e visualizada por algum método adequado.

[2]Um marca-passo é um instrumento que, ligado diretamente ao coração de uma pessoa, mantém os batimentos cardíacos numa freqüência pré-determinada.

7.2 A função do Detector de Defeitos

Basicamente a função do detector de defeitos dentro do sistema é controlar os membros (equipamentos) e os canais de comunicação. A redundância dos canais de comunicação pode ser explorada através de *broadcasts* e pode servir para descobrir equipamentos que estejam com dificuldades de comunicação. O controle de membros na verdade serve para suspeitar que um dado processo está com defeito e pode ser realizado com um dos detectores de defeitos apresentados no capítulo 4, conforme a característica desejada para o sistema.

Porém, a idéia de usar o detector de defeitos somente para detectar os suspeitos pode ser expandida. O projeto *Memento* (ROST; BALAKRISHNAN, 2006), por exemplo, explora a idéia de que um sistema de gerenciamento de redes auto-gerenciáveis (como é o caso de redes ad hoc), deve prover aos seus usuários e administradores:

- Detecção de Defeitos: informação sobre defeitos dos nodos.
- Alerta de Sintomas: informar aos usuários sobre sintomas que causam defeitos ou atuam no desempenho do sistema.
- Inspeção: ajudar a deduzir porque os defeitos ou sintomas ocorrem.

Essas três classes de informação permitem o usuário achar erros em seus *softwares* efetivamente, escolher os parâmetros corretos para um melhor desempenho, monitorar o comportamento do *hardware*, saber a carga da rede sem fio oferecida, entender porque esses defeitos ocorrem e prevenir defeitos mesmo antes que eles aconteçam.

O projeto *CardioMonitor* também expande a função do detector de defeitos para um sistema de gerenciamento que além de detectar o defeito, dá a possibilidade ao médico de analisar as informações e os dados do paciente fornecidos pelo marca-passo, visualizar as curvas dos sinais vitais, bem como emitir alertas quando alguma anomalia é detectada. Portanto, a idéia de criar um sistema de gerenciamento para redes auto-gerenciáveis e incorporar neste sistema o detector de defeitos, juntamente com outras funções necessárias para o ambiente e para o seu correto funcionamento deixa o sistema mais confiável e seguro, além de disponibilizar um maior número de serviços para os seus usuários, o que é muito útil em um hospital, por exemplo.

7.3 Conclusões Parciais

Em um sistema de gerenciamento de marca-passos cardíacos a escolha do algoritmo de detecção de defeitos é muito importante para o seu correto funcionamento. A existência de diversos detectores de defeitos dificulta a escolha, mas a comparação dos algoritmos de detecção de defeitos realizada neste trabalho pode ajudar a decidir qual detector tem as características desejadas para o ambiente do sistema desejado.

8 CONCLUSÃO

Este trabalho apresentou as principais características e realizou uma comparação entre alguns algoritmos de detecção de defeitos para redes móveis sem fio. Mais especificamente, o trabalho identificou os problemas que dificultam a migração de detectores de defeitos de redes com nodos fixos (*Push, Pull e Dual*) para redes móveis sem fio, bem como apresentou e discutiu as principais estratégias de detecção (detectores de defeitos baseado no algoritmo *Gossip* e detector de defeitos baseado em formação de *cluster*) que tentam tratar dos problemas identificados. Nesta primeira etapa, pode-se perceber que os algoritmos *Push, Pull* e *Dual* não são usados em redes móveis sem fio, pois o envio de mensagens é periódico, o que aumenta a quantidade de mensagens transmitidas e conseqüentemente aumenta o consumo de energia dos nodos e a probabilidade de perda de mensagens.

Em uma segunda etapa, os detectores de defeitos baseados no algoritmo *Gossip* (detectores propostos pelo *Hutle* e pelo *Friedman*), juntamente com o próprio algoritmo *Gossip* e o detector baseado em *cluster* foram comparados em relação ao número de *broadcasts*, tempo médio de recorrência ao erro (T_{MR}), tempo médio de duração de falsas suspeitas (T_M) e tempo de detecção (T_D). Como resultado dessa comparação, o algoritmo proposto pelo *Hutle* apresentou os melhores desempenhos para o T_{MR} e T_D, porém foi o detector que mais emitiu *broadcasts*. O detector proposto pelo *Friedman* obteve a melhor performance em relação ao T_M. Já o detector baseado em *cluster* teve um rendimento intermediário em praticamente todas as métricas. Com isso, em um ambiente onde a velocidade do detector e o número de recorrência ao erro são os fatores mais relevantes, o detector proposto pelo *Hutle* se mostrou a melhor opção. Por outro lado, em um ambiente onde o consumo de energia é essencial, o algoritmo do *Friedman* ou o detector baseado em *cluster* podem ser boas alternativas.

Na seqüência deste trabalho, alguns assuntos poderão ser explorados em trabalhos futuros, como por exemplo:

- tentar diminuir o número de *broadcasts* no detector de defeitos do *Hutle*;
- verificar a influência dos algoritmos de roteamento em mensagens *inter-cluster* no detector baseado em *cluster*;
- Implementar e avaliar os algoritmos em um ambiente real, como por exemplo, o ambiente descrito no capítulo 7.

REFERÊNCIAS

AGUILERA, M. K.; CHEN, W.; TOUEG, S. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In: WORKSHOP ON DISTRIBUTED ALGORITHMS, 1997. **Anais...** [S.l.: s.n.], 1997. p.126–140.

BAGRODIA, R.; MEYER, R. PARSEC: a parallel simulation environment for complex system. **Computer Magazine**, [S.l.], 1998.

BARR, R.; HAAS, Z. J.; RENESSE, R. van. JiST: embedding simulation time into a virtual machine. **Proceedings of EuroSim Congress on Modelling and Simulation**, [S.l.], September 2004.

BARR, R.; HAAS, Z. J.; RENESSE, R. van. **Java in Simulation Time / Scalable Wireless Ad hoc Network Simulator (JiST/SWANS)**. http://jist.ece.cornell.edu/ - Último acesso em Fevereiro de 2007.

BASAGNI, S.; CHLAMTAC, I.; FARAGO, A. **A generalized clustering algorithm for peer-to-peer networks**. 1997.

BLUETOOTH. **The Bluetooth Technology Web Site**. http://www.bluetooth.com/ - Último acesso em Fevereiro de 2007.

CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM**, [S.l.], v.43, n.2, p.225–267, 1996.

CHATTERJEE, M.; DAS, S.; TURGUT, D. An on-demand weighted clustering algorithm (WCA) for ad hoc networks. In: IEEE GLOBECOM, 2000. **Proceedings...** [S.l.: s.n.], 2000.

CHEN, W.; TOUEG, S.; AGUILERA, M. K. On the quality of service of failure detectores. **IEEE Transactions On Computer**, [S.l.], v.51, n.2, p.561–580, 2002.

CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock Synchronization in the Presence of Omission and Performance Failures, and Processor Joins. In: YANG, Z.; MARSLAND, T. A. (Ed.). **Global States and Time in Distributed Systems, IEEE Computer Society Press**. [S.l.: s.n.], 1994.

DAHM, M. **Byte Code Engineering with the BCEL API**. [S.l.: s.n.], 2001. (B-17-98, Freie Universit at Berlin, Institut f ur Informatik).

DARPA. **The Network Simulator (NS2)**. http://www.isi.edu/nsnam/ns/ - Último acesso em Fevereiro de 2007.

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. **Int. Journal of Wireless and Mobile Computing**, [S.l.], 2005.

GARTNER, F. C. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. **ACM Computing Surveys**, [S.l.], v.31, n.1, p.1–26, 1999.

GERLA, M.; TSAI, J. Multicluster, mobile, multimedia radio network. **Journal of Wireless Networks**, [S.l.], v.1, n.3, p.255–265, 1995.

GRACIOLI, G.; NUNES, R. C. Detectores de Defeitos para Redes Wireless Ad Hoc. **Anais - IX Escola Regional de Redes de Computadores (ERRC)**, [S.l.], 2006.

HEIDEMANN, J. S.; SILVA, F.; INTANAGONWIWAT, C.; GOVINDAN, R.; ESTRIN, D.; GANESAN, D. Building Efficient Wireless Sensor Networks with Low-Level Naming. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 2001. **Anais...** [S.l.: s.n.], 2001. p.146–159.

HUTLE, M. An efficient failure detector for sparsely connected networks. **Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN 2004), Innsbruck, Austria**, [S.l.], Feb. 2004.

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

JOHNSON, D. B.; MALTZ, D. A. Dynamic Source Routing in Ad Hoc Wireless Networks. In: IMIELINSKI; KORTH (Ed.). **Mobile Computing**. [S.l.]: Kluwer Academic Publishers, 1996. v.353.

LIM, A. **Self-configurable sensor networks**. Disponível em http://www.eng.auburn.edu/users/lim/sensit.html - Último acesso em Fevereiro de 2007.

MARTINS, J. B.; PRIOR, A.; RODRIGUES, C. R.; NUNES, R. C.; AITA, A. L.; SALENGUE, D. C.; MAZZUTI, C. Sistema de Marca-Passo-Cardíaco com Monitoramento e Controle Bluetooth. **XX CBEB**, [S.l.], 2006.

MATEUS, G. R.; LOUREIRO, A. A. **Introdução à computação móvel**. 2ª.ed. [S.l.: s.n.], 2005.

PAREKH, A. K. **Selecting routers in ad-hoc wireless networks**. ITS, 1994.

PEREIRA, M. R.; AMORIM, C. l. de; CASTRO, M. C. S. de. **Tutorial sobre Redes de Sensores**. Cadernos do IME UERJ- Série Informática - Vol 14 - Junho 2003 - Disponível em http://www.ime.uerj.br/cadernos/cadinf/vol14/ - Último acesso em Fevereiro de 2007.

PRADHAN, D. K. **Fault-Tolerant System Design**. [S.l.]: Englewood Cliffs: Prentice-Hall, 1996.

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

RILEY, G. F.; AMMAR, M. H. Simulating Large Networks – How Big is Big Enough? **Conference on Grand Challenges for Modeling and Sim**, [S.l.], January 2002.

ROST, S.; BALAKRISHNAN, H. Memento: a health monitoring system for wireless sensor networks. In: IEEE SECON, 2006, Reston, VA. **Anais...** [S.l.: s.n.], 2006.

WU, J. (Ed.). **Scalable Wireless Ad hoc Network Simulation.** [S.l.]: CRC Press, 2005. p.297–311.

SUN-MICROSYSTEMS. **Java Technology**. http://java.sun.com/ - Último acesso em Fevereiro de 2007.

TAI, A. T.; TSO, K. S. **Failure detection service for ad hoc wireless networks applications**: a cluster-based approach. [S.l.: s.n.], 2004. (IAT-302184, IA Tech, Inc., Los Angeles, CA).

TAI, A. T.; TSO, K. S.; SANDERS, W. H. Cluster-Based Failure Detection Service for Large-Scale Ad Hoc Wireless Network Applications. In: DSN '04: PROCEEDINGS OF THE 2004 INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (DSN'04), 2004, Washington, DC, USA. **Anais...** IEEE Computer Society, 2004. p.805.

THOMA, V. M.; NUNES, C. M. Avaliação de Desempenho de Redes Ad Hoc Utilizando Modelos de Mobilidade. **Anais - IX Escola Regional de Redes de Computadores (ERRC)**, [S.l.], 2006.

UCLA. **Global Mobile Information Systems Simulation Library (GloMoSim)**. http://pcl.cs.ucla.edu/projects/glomosim/ - Último acesso em Fevereiro de 2007.

WANG, S.-C.; KUO, S.-Y. Communication Strategies for Heartbeat-Style Failure Detectors in Wireless Ad Hoc Networks. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (DSN 2003), (SAN FRANCISCO, CA), June 2003. **Proceedings...** IEEE Computer Society, June 2003. p.361–370.

WEBER, T. S. **Um roteiro para exploração dos conceitos básicos de tolerância a falhas**. Instituto de Informática - UFRGS - Curso de Especialização em Redes e Sistemas Distribuídos - Disponível em http://www.inf.ufrgs.br/ taisy/ - Último acesso em Fevereiro de 2007.

APÊNDICE A EXEMPLO DE UMA SIMULAÇÃO

O objetivo deste apêndice é apresentar um exemplo de uma simulação, bem como mostrar como é feita a configuração dos nodos da rede no ambiente de simulação no simulador *JiST/SWANS*. O exemplo aqui apresentado é na simulação do algoritmo *Gossip*, visto anteriormente neste trabalho. Este apêndice não trata da instalação do simulador, nem dos maiores detalhes do uso das suas classes, já que isso é detalhado na documentação disponibilizada pelos autores na página da internet[1].

Para executar o algoritmo basta utilizar o comando abaixo no terminal:

```
swans driver.StartUpGossip 20 300 1800 15 12 2 12
```

Os argumentos recebidos são o número de nodos, o tamanho do campo de simulação em metros (neste caso 300 x 300 metros), o tempo total de simulação em segundos, a potência do rádio em dBm (variando a potência do rádio, o alcance de transmissão também varia, por exemplo, com uma potência de 15 dBm o alcance de transmissão é 50 metros, assim como uma potência de transmissão de 30 dBm é equivalente a um alcance de transmissão de 100 metros), o T_{gossip}, o $T_{cleanup}$ que neste caso é 2 vezes o valor do T_{gossip} e por fim o valor do T_{fail}.

```
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);
Fading fading = new Fading.Rayleigh();
PathLoss pathloss = new PathLoss.TwoRay();

Field field = new Field(spatial, fading, pathloss, mobility, Constants.PROPAGATION_LIMIT_DEFAULT);

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);
```

Figura A.1: Configuração dos parâmetros e criação do campo de simulação.

A classe *StartUpGossip* é responsável por criar o campo de simulação configurando o modelo de localização, o modelo de mobilidade, o modelo responsável pela propagação do sinal, o modelo de distribuição de probabilidade das atenuações, a perda da propagação do sinal, o rádio e a probabilidade de perda de pacotes. Isso é exemplificado na figura A.1

Após configurar e criar o campo de simulação a classe *StartUpGossip* também é responsável por criar os nodos, veja a figura A.2.

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

Figura A.2: Criação dos nodos.

Cada nodo por sua vez configura o seu modelo de ruídos do rádio, a camada MAC, um endereço de rede, a implementação do protocolo IP utilizada na rede (IPv4 baseado no RFC 791) e a classe de aplicação que será executada pelo nodo, que neste caso é o algoritmo *Gossip*, conforme a figura A.3.

```
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);
BasicGossip app = new BasicGossip(netAddress, Tgossip, Tcleanup, Tfail);
```

Figura A.3: Configuração do ruído do rádio, camada MAC, endereço, implementação do protocolo IP e do algoritmo *Gossip*.

```
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);                                    /* Executa o algoritmo Gossip */
```

Figura A.4: O nodo inicia a mobilidade, configura os parâmetros e executa o algoritmo *Gossip*.

Após isso, o nodo inicia a mobilidade, configura os últimos parâmetros e executa o algoritmo da classe *BasicGossip*, como mostra a figura A.4.

```
/* Repete a cada T_gossip */
public void run(String[] args)
{
    cleanupCount++;
    if((cleanupCount % Tcleanup) == 0) {                                    /* Tcleanup = 2 */
        /* verifica se existe algum suspeito */
        detectSuspects();
    }
    /* broadcast a lista de vizinhos */
    System.out.println("B " + netaddress);
    sendGossipMessage();
    /* calcula o período para o próximo heartbeat, T_gossip = 12 segundos */
    JistAPI.sleepBlock(calcDelay());
    /* executa novamente */
    ((AppInterface)self).run();
}

/* Verifica os contadores de heartbeats locais e detecta os suspeitos */
private void detectSuspects()
{
    long time = jist.runtime.JistAPI.getTime()/Constants.SECOND;            /* Tempo de simulação atual */
    Iterator it = new HashMap(neighbours).values().iterator();
    while(it.hasNext()) {
        NeighbourEntry n = (NeighbourEntry) it.next();
        /* compara com o endereço do próprio nodo */
        if(n.address.equals(netaddress)) {
            continue;
        }
        if((time - n.timestamp) > Tfail) {
            System.out.println("A " + netaddress + " " + n.address + " " + jist.runtime.JistAPI.getTime()/Constants.SECOND);
            suspectsList.put(n.address, n);                                 /* Adiciona na lista de suspeitos */
            neighbours.remove(n.address);                                   /* Remove da lista de vizinhos local */
        }
    }
}

/* Repete a cada recebimento de mensagem */
public void receive(Message msg, NetAddress src, MacAddress lastHop, byte macId, NetAddress dst, byte priority, byte ttl)
{

    HashMap receivedList = ((MessageGossip)msg).getData();
    if(suspectsList.containsKey(src)) {
        System.out.println("R " + netaddress + " " + src + " " + jist.runtime.JistAPI.getTime()/Constants.SECOND);
        suspectsList.remove(src);                                           /* Remove da lista de suspeitos */
    }
    /* compara os heartbeats da lista local com a lista recebida */
    Iterator it = receivedList.values().iterator();
    while(it.hasNext()) {
        NeighbourEntry nReiceived = (NeighbourEntry) it.next();
        NeighbourEntry nAtual = (NeighbourEntry) neighbours.get(nReiceived.address);
        /* O heartbeat recebido não está na lista local, o adiciona */
        if(nAtual == null) {
            neighbours.put(nReiceived.address, nReiceived);
        } else if(nAtual.heartbeatCounter < nReiceived.heartbeatCounter) {
            nAtual.heartbeatCounter = nReiceived.heartbeatCounter;
```

[1]Acesse *http://jist.ece.cornell.edu/* para maiores detalhes.

```
            nAtual.timestamp = jist.runtime.JistAPI.getTime()/Constants.SECOND;
        }
    }
}
```

Figura A.5: Código do algoritmo *Gossip* para o laço que repete a cada T_{gossip}, para a detecção de suspeitos e para o recebimento de mensagens.

A classe *BasicGossip* é a implementação do algoritmo *Gossip* e é apresentada na figura A.5. A cada intervalo T_{gossip} o nodo envia um *heartbeat* (*sendGossipMessage()*) e incrementa o contador *cleanupCount* que é responsável, juntamente com o *Tcleanup*, pela decisão de varrer a lista de suspeitos. A lista de suspeitos é analisada a cada dois envios de mensagens, ou seja, a cada dois T_{gossip}. Nesta análise, o nodo procura por contadores de *heartbeats* que não foram incrementados, levantando assim uma suspeita. Já no recebimento de mensagens, o nodo primeiramente verifica se o emissor da mensagem está na sua lista de suspeitos e se estiver, o remove. Após compara os *heartbeats* da lista local com os *heartbeats* da lista recebida atualizando os seus valores.

Por fim, a figura A.6 apresenta um trecho de exemplo do arquivo de saída gerado pela execução do algoritmo. Com isso é possível obter o número de *broadcasts*, o tempo médio de recorrência ao erro, o tempo médio de duração de falsas suspeitas e o tempo de detecção através de scripts que analisam o arquivo de saída.

```
A 0.0.0.19 0.0.0.5 1116      /* O nodo 0.0.0.19 adicionou o 0.0.0.5 na sua lista de suspeitos no tempo 1116 */
B 0.0.0.19                                  /* O nodo 0.0.0.19 efetuou um broadcast */
B 0.0.0.11                                  /* O nodo 0.0.0.11 efetuou um broadcast */
B 0.0.0.1                                   /* O nodo 0.0.0.1 efetuou um broadcast */
A 0.0.0.3 0.0.0.15 1116      /* O nodo 0.0.0.3 adicionou o 0.0.0.15 na sua lista de suspeitos no tempo 1116 */
R 0.0.0.12 0.0.0.2 1128      /* O nodo 0.0.0.12 removeu o 0.0.0.2 da sua lista de suspeitos no tempo 1128 */
R 0.0.0.19 0.0.0.2 1128      /* O nodo 0.0.0.19 removeu o 0.0.0.2 da sua lista de suspeitos no tempo 1128 */
```

Figura A.6: Exemplo de um arquivo de saída gerado pela execução do algoritmo *Gossip*.