UFSM

TRABALHO DE GRADUAÇÃO

AFDService: Implementação do algoritmo *Gossip* e de uma ferramenta gráfica para configuração

Aluno:
Ricardo Laurini Silva

Orientador:
Raul Ceretta Nunes

Santa Maria, RS, Brasil

2005

AFDService: Implementação do algoritmo *Gossip* e de uma ferramenta gráfica para configuração

Por

Ricardo Laurini Silva

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

Bacharel em Ciência da Computação

Curso de Ciência da Computação

Trabalho de Graduação nº 197
Santa Maria, RS, Brasil
2005

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

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

AFDService: Implementação do algoritmo *Gossip* e de uma ferramenta gráfica para configuração

elaborado por

Ricardo Laurini Silva

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

COMISSÃO EXAMINADORA

_____
Raul Ceretta Nunes - UFSM
(Orientador)

_____
Márcia Pasin - UFSM

_____
Rogério Corrêa Turchetti - UNIFRA

Santa Maria 2005

Agradecimentos

O aprendizado ao longo deste curso foi muito significativo, não somente pelo conhecimento científico adquirido, mas também, pela experiência de vida acumulada.

De fato, esta é uma grande conquista para mim, porém, reconheço com muita alegria, que não haveria de consegui-la sem a colaboração de várias pessoas.

Obrigado a Deus, que está presente de forma incansável e renovadora em todos os momentos de minha vida, abençoando cada um dos meus passos.

A minha família, por saber que minhas conquistas serão sempre comemoradas como suas próprias conquistas. Especialmente, aos meus pais (Olívo Joicemar e Emeri) e a minha irmã (Flávia).

Ao professor Raul, por ter me orientado na realização deste trabalho de forma eficaz e sempre com muita responsabilidade.

Aos meus amigos, pela torcida e carinho sempre dispensados a minha pessoa. Todas as palavras e gestos foram muito importantes.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AWT	Abstract Window Toolkit.
FD	Módulo de detecção de defeitos do AFDService.
IP	*Internet Protocol.*
$P_{mistake}$	Probabilidade de ocorrer uma detecção de falha errônea.
TCP	*Transmission Control Protocol.*
$T_{cleanup}$	Intervalo de tempo quando um membro é removido da lista.
T_{fail}	Intervalo de tempo em que o valor do *heartbeat* não foi incrementado.
T_{gossip}	Intervalo entre duas mensagens *gossip.*
T_M	Tempo médio de duração ao erro.
T_{MR}	Tempo médio de recorrência ao erro.
TS	Módulo de predição de *timeout* do AFDService.
UDP	*User Datagram Protocol.*

]

LISTA DE FIGURAS

RESUMO

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

AFDService: Implementação do Algoritmo *Gossip* e de uma ferramenta gráfica para configuração

Aluno:
Ricardo Laurini Silva

Orientador:
Raul Ceretta Nunes

Um detector de defeitos é uma importante abstração para viabilizar a implementação de protocolos tolerante a falhas em sistemas distribuídos assíncronos.

Este trabalho apresenta uma implementação do algoritmo de detecção de defeitos *estilo-gossip* proposto por Renesse, Minsky e Hayden. A implementação do detector *gossip* tem a finalidade de ser inserido no serviço *AFDService*, um Serviço de Detecção de Defeitos Adaptativo implementado em Java.

O algoritmo *gossip* é chamado de "fofoqueiro". O objetivo deste protocolo é distribuir as informações para um grupo. O mecanismo do *gossip* consiste em que cada membro do grupo passa a informação periodicamente a um membro escolhido aleatoriamente. Uma de suas principais características é a escalabilidade e resilência a falhas de um processo ou mensagens perdidas.

O algoritmo *gossip*, embora diferente dos outros detectores de defeitos incluído no *AFDService* (*pull*, *push* e *dual*), pode ser implantado ao repositório de detectores de defeitos do serviço *AFDService*, utilizando as interfaces definidas pelo serviço.

Também foi implementada uma interface gráfica para relacionar o usuário ou aluno a um aplicativo do serviço AFDService. A principal função deste software é mostrar um histórico de transições de estados dos componentes monitorados pelo serviço. Uma outra motivação é ajudar o aluno ter uma definição mais prática de um sistema detecção de defeitos distribuídos.

1. INTRODUÇÃO

É crescente a necessidade de se construir sistemas distribuídos[1] que requerem tolerância a falhas, haja visto o uso destes sistemas em aplicações que exigem um bom nível de confiança no funcionamento. O motivo é o custo de falhas nos sistemas, que pode ser alto, trazendo conseqüências indesejáveis, tais como: perda de clientes, perda de produção, perda de confiabilidade.

A implementação de mecanismos de tolerância a falhas em sistemas distribuídos fornecem confiança e serviço contínuo a despeito de falhas de algum de seus componentes [JALOTE, 1994].

Como característica inerente, os sistemas distribuídos possuem redundância, o que é explorado por mecanismos de tolerância a falhas para aumentar a confiabilidade e disponibilidade do sistema nos casos de falhas, possibilitando que a operação continue sendo realizada mesmo durante a falha de alguns componentes.

Além da redundância, os sistemas distribuídos costumam ser assíncronos, o que na prática fazem os mecanismos de tolerância a falhas usarem limites de tempo de comunicação (*timeouts*) para alcançar terminação dos algoritmos.

Entretanto, mesmo usando *timeouts*, em ambientes assíncronos, sujeitos a falhas, é impossível determinar precisamente se um processo remoto encontra-se falho ou apenas mais lento que os demais. Este indeterminismo na resposta de um processo dificulta a especificação e validação de algoritmos de acordo, necessários na manutenção da consistência de um sistema com replicação.

Uma alternativa para minimizar o problema do indeterminismo é acrescentar um detector de defeitos a cada entidade de um sistema distribuído assíncrono [GARTNER, 1999].

Um detector de defeitos é um oráculo distribuído que fornece sugestões sobre quais processos podem ter deixado de funcionar.

Cada processo tem o acesso a um módulo detector de defeitos local, o módulo monitora outros processos no sistema e mantém uma lista dos processos

[1] Um sistema distribuído consiste num conjunto finito de processos que se comunicam por troca de mensagens através de um subsistema de comunicação [GARTNER, 1999].

atualmente suspeitos.

Cada processo consulta periodicamente seu módulo do detector de defeitos e usa a lista dos suspeitos para tomar decisões. Embora os detectores de defeitos não possam determinar com exatidão o estado dos processos do sistema (falho ou não falho), sua utilização facilita a especificação e a validação de protocolos de acordo.

Existem diversos algoritmos de detecção de defeitos, dentre eles: o *push*, o *pull* e o *dual*. Modelos estes já implementados no serviço de detecção de defeitos adaptativo (AFDService) [NUNES 2003]. O AFDService é um serviço de detecção de defeitos que monitora processos de um sistema distribuído. Ele foi projetado com uma arquitetura configurável e flexível que permite a inclusão facilitada de novos algoritmos de detecção.

Em sua especificação, o AFDService define um conjunto genérico de interfaces e modela três componentes básicos: os objetos **monitoráveis**, os quais são componentes da aplicação (processos, objetos, *threads*) que devem ser observados pelos monitores; os objetos **notificáveis**, os quais são componentes da aplicação que desejam ser notificados de maneira assíncrona sobre as alterações de estado dos objetos monitoráveis; e os objetos **monitores**, os quais implementam o algoritmo de detecção de defeitos. Logo colecionam informações sobre o estado (suspeito ou não suspeito) dos componentes monitoráveis da aplicação. Um componente monitor interage com a aplicação cliente, respondendo às suas solicitações de estado ou notificando-a quando ocorrerem mudanças de estado.

Como o AFDService foi projetado para ser um repositório de algoritmos de detecção, novos algoritmos, tal como o algoritmo de estilo *gossip* [RENESSE; MINSKY; HAYDEN, 1998], podem ser adicionados ao serviço.

O algoritmo é chamado “fofoqueiro” (gossip) porque cada membro envia periodicamente informações para outros membros escolhidos aleatoriamente. Eventualmente, pelas fofocas aleatórias, todos devem conhecer o estado dos vizinhos e conseqüentemente devem detectar quais estão falhos.

O detector *gossip* possui algumas vantagens interessantes; 1) Ele é imune a mensagens perdidas e processos falhos; 2) a probabilidade de que um membro esteja erroneamente relatando uma falha é independente do número de processos; e 3) é escalável, a demanda da largura de banda cresce linearmente com o número de processos [RENESSE; MINSKY; HAYDEN, 1998].

Neste contexto, este trabalho possui dois objetivos: implementar o algoritmo

de detecção de defeitos *gossip* usando as interfaces do AFDService; e implementar uma ferramenta gráfica para o estudo de detectores de defeitos.

Do ponto de vista didático, através da ferramenta gráfica o usuário poderá ter uma visão dinâmica sobre os estados dos objetos monitorados, bem como compreender e analisar o comportamento dos diversos algoritmos, pois o usuário poderá informar a configuração desejada como, por exemplo, o tamanho do limite de tempo de espera (*timeout)*, o estilo do detector (push, pull, dual ou gossip).

As divisões deste trabalho são especificadas da seguinte forma. O capítulo 2 apresenta a definição de detectores de defeitos, sendo apresentadas propriedades que definem o funcionamento dos detectores de defeitos. Também são apresentados os algoritmos básicos de detectores de defeitos *push*, *pull* e dual. O capítulo 3 explica o algoritmo *gossip*, suas propriedades e sua estrutura. No capítulo 4 é realizado um estudo sobre o AFDService, suas vantagens, estrutura e interfaces. No capítulo 5 e 6 são apresentados a implementação do algoritmo gossip e a ferramenta gráfica. E finalmente, no capítulo 7 são apresentadas as conclusões deste trabalho.

2. DETECTORES DE DEFEITOS

Este capítulo apresenta o modelo de sistema distribuído adotado neste trabalho e, também, uma revisão sobre algoritmos de detectores de defeitos dos estilos *pull*, *push*, *dual* e *gossip*.

2.1 MODELO DE SISTEMA COM DETECTOR

Considera-se o um sistema distribuído onde cada processo do sistema tem acesso a um módulo de detecção de defeitos local. Tal módulo monitora um subgrupo dos processos do sistema e mantém uma lista onde são relacionados os processos suspeitos de falha.

Como os módulos detectores são considerados não confiáveis, ou seja, podem suspeitar erroneamente de um processo, um módulo detector pode adicionar um processo *p* à sua lista de suspeitos mesmo que esse processo esteja funcionando. Se, em outro instante, este módulo concluir que a suspeita sobre *p* foi um engano, o processo *p* pode ser retirado da lista de suspeitos. Dessa forma, um processo do sistema tem como referência apenas a percepção momentânea do seu módulo de detecção de defeitos local.

Como os módulos são distribuídos e cada processo tem acesso apenas ao módulo local, em um mesmo instante de tempo dois módulos podem apresentar listas de suspeitos diferentes, de acordo com a visão (percepção) de cada um. Os erros de detecção cometidos não influenciem no comportamento dos módulos suspeitos. Um detector de defeitos corresponde ao conjunto de módulos de detecção distribuídos.

Para determinar as características dos detectores de defeitos, estes foram classificados de acordo com as propriedades de *completeness* (abrangência, completeza) e *accuracy* (precisão) [CHANDRA; TOUEG, 1996].

A propriedade *completeness* refere-se à capacidade do detector identificar

todos os processos que estão realmente falhos, enquanto *accuracy* determina a precisão desta suspeita, ou seja, diz respeito à inclusão de processos corretos nas listas de suspeitos. Ao respeitar essas duas propriedades, um detector de defeitos garante que os algoritmos que o utilizem não perderão a consistência das decisões, nem ficarão indefinidamente bloqueados (preservando as propriedades *safety* e *liveness*, respectivamente).

Segundo essas propriedades, um detector que continuamente inclua e remova os processos da sua lista de suspeitos não é capaz de garantir as condições mínimas de estabilidade ao seu cliente, violando a propriedade de *accuracy* dos detectores.

2.2 ALGORITMOS PARA DETECÇÃO DE DEFEITOS

Nesta seção serão apresentados quatro algoritmos para detecção de defeitos: *push*, *pull*, *dual* e *gossip*. Todos parte do repositório do AFDService. O *gossip* foi incluído neste trabalho.

Inicialmente serão revisadas as definições dos modelos básicos de detectores de defeitos (*push* e *pull*), considerando a classificação de acordo com o fluxo de mensagens de controle, isto é, de acordo com a maneira com que as informações são propagadas entre os componentes do sistema.

Em seguida, é apresentado o modelo *dual*, que é realizado pela composição de dois modelos básicos, e o *gossip*, que se baseia em outro princípio de propagação de mensagens, o da fofoca.

Uma breve descrição de detectores de defeitos adaptativos usando *timeouts* variáveis também é realizada.

2.2.1 Detector estilo *push*

No estilo *push* [FELBER; DÉFAGO; GUERRAOUI 1999], o fluxo de controle segue a mesma direção do fluxo das informações, ou seja, dos processos monitorados em direção ao monitor/detector.

Para isso, em um detector *push*, os processos monitorados precisam estar ativos. Eles enviam periodicamente mensagens identificadas (conhecidas como *"I´m alive!"*), que têm como função demonstrar que se o processo está enviando mensagens ele ainda está operacional (informação suficiente se for considerado um modelo de falhas por colapso).

Se um monitor não receber tais mensagens dentro de um intervalo de tempo, ele passa a suspeitar do processo monitorado.

A figura 2.1 demonstra como o estilo *push* pode ser usado para monitorar os processos de um sistema.

As mensagens trocadas entre o detector e o cliente são diferentes das mensagens trocadas entre os processos monitorados e o monitor.

O detector notifica o cliente quando um processo monitorado é considerado suspeito, enquanto as mensagens de "*I'm Alive!*" são enviadas continuamente do monitorado para o monitor.

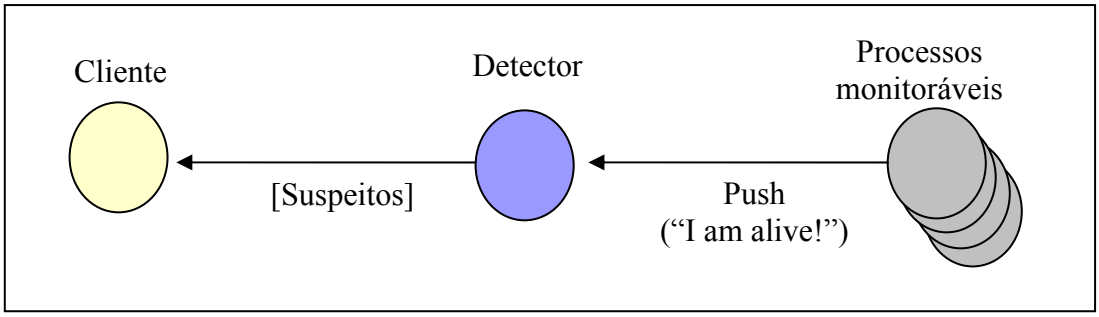

FIGURA 2.1 – Estilo push de monitoramento.

A troca de mensagens entre o processo monitor e os processos monitorados no modelo push é mostrada na Figura 2.2.

Nela podem-se identificar os processos monitoráveis, que periodicamente enviam mensagens "*I'm Alive!*", e o processo monitor que utiliza *timeouts* para detectar defeitos. Quando o monitor recebe uma mensagem de um processo monitorado, ele reinicializa o *timeout* relativo aquele processo. Se o *timeout* expirar e

o monitor não tiver recebido nenhuma mensagem nova do processo monitorado, então o monitor indica a suspeita incluindo o monitorado na sua lista de suspeitos.

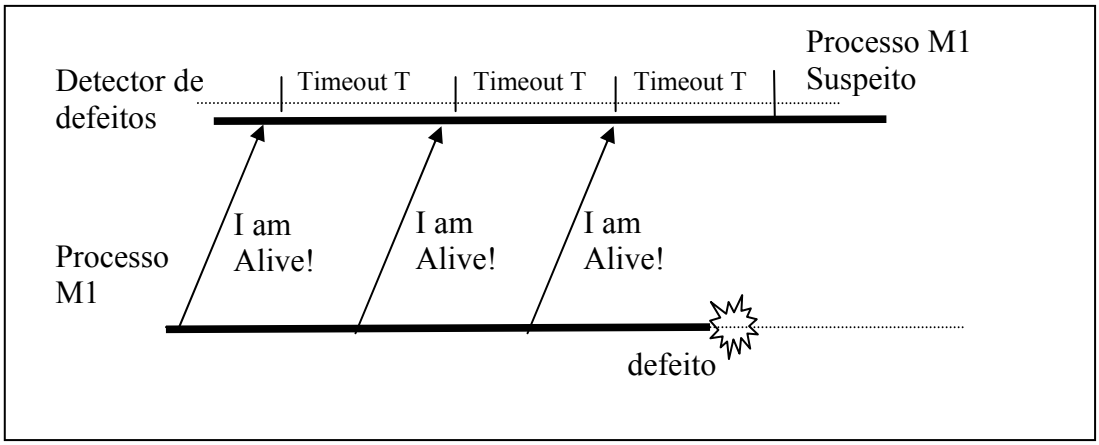

FIGURA 2.2 – Monitorando mensagens no estilo *Push*.

2.2.2 Detector estilo pull

O estilo *pull* [FELBER; DÉFAGO; GUERRAOUI, 1999] apresenta o fluxo de informações em sentido oposto ao fluxo de controle, ou seja, as informações do detector de defeitos são obtidas através de requisição aos processos monitorados.

Dessa forma, os processos monitorados podem ser passivos e, somente quando questionados pelo monitor, enviam uma mensagem de resposta.

O processo monitor do detector de defeitos periodicamente envia mensagens de *liveness request* aos processos monitorados, ou seja, mensagens que questionam se os processos estão ainda operacionais. Se um processo monitorado responde à requisição do monitor, significa que ele está operando e que não é suspeito.

Devido à troca de mensagens entre os processos monitorados e o processo monitor ser bidirecional (*two-ways*), esse estilo tende a ser menos eficiente do que o estilo *push*, mas apresenta vantagens quanto ao desenvolvimento da aplicação, uma vez que os processos monitorados não precisam estar ativos nem ter conhecimento sobre a temporização do sistema (por exemplo, a freqüência com que o monitor

espera receber mensagens).

A Figura 2.3 exibe o fluxo das informações deste estilo de detecção.

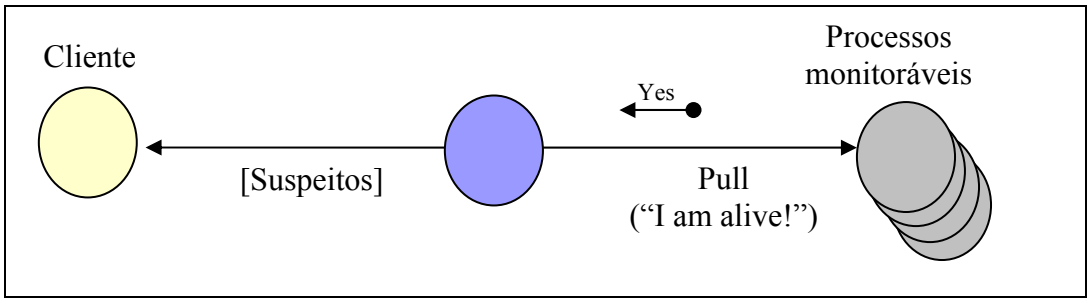

FIGURA 2.3 – O fluxo do estilo *Pull*

A forma com que as mensagens são trocadas entre o processo monitor e os processos monitorados são detalhados na Figura 2.4.

Periodicamente, o monitor envia uma mensagem de questionamento (*liveness-request*) para o(s) processo(s) monitorado(s), e fica esperando a resposta. Se um processo não enviar a resposta (ou a mensagem for perdida), então a expiração de um *timeout* levará a suspeita do monitorado.

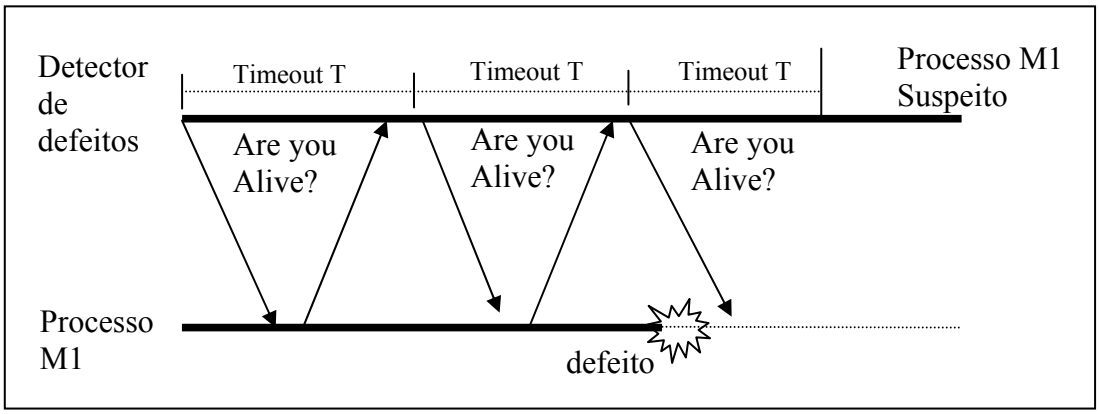

FIGURA 2.4 – Mensagens de monitoramento no estilo Pull.

2.2.3 Detector estilo *dual*

O estilo *pull* tem a vantagem de que a configuração do detector é centralizada no processo monitor e não é distribuída por todos os processos monitoráveis.

Conseqüentemente, foi introduzido um estilo combinado com os dois modelos, chamado de *dual* [FELBER; DÉFAGO; GUERRAOUI 1999], o qual é fusão

dos estilos *push* e *pull*.

Para este estilo não tem a necessidade dos monitores terem um conhecimento prévio dos estilos de detecção suportados pelos processos monitorados.

O protocolo do estilo dual é dividido em duas fases distintas. Durante a primeira fase, todos os monitorados assumem o uso do estilo *push* para minimizar a troca de mensagens. Quando o *timeout* expirar, o detector muda para a segunda fase. Nesta ele assume que todos os monitorados que não enviaram as mensagens "*I am Alive!*" durante a primeira fase devem ser consultados segundo o estilo *pull*. Nesta fase, o detector envia uma mensagem "*Are you alive?*" para cada processo monitorado e espera uma mensagem *"I am Alive!"* como resposta. Se o monitorado não enviar a mensagem dentro de algum limite de tempo (timeout da segunda fase), o detector suspeitará do monitorado.

A Figura 2.5 ilustra o monitoramento com o protocolo *dual*. Na primeira fase, o processo *M1* está ativo e periodicamente envia mensagens de "*I am Alive*". O detector usa dois timeouts, uma para cada fase. Enquanto as mensagens forem recebidas dentro de um intervalo de tempo T1, o detector continua operando de acordo com o estilo *push*.

Quando T1 expira, o detector passa a operar no estilo *pull* e o monitor envia uma mensagem de requisição de estado ao monitorável *M1.* Somente quando T2 expira o monitorável é colocado na lista de suspeitos do monitor.

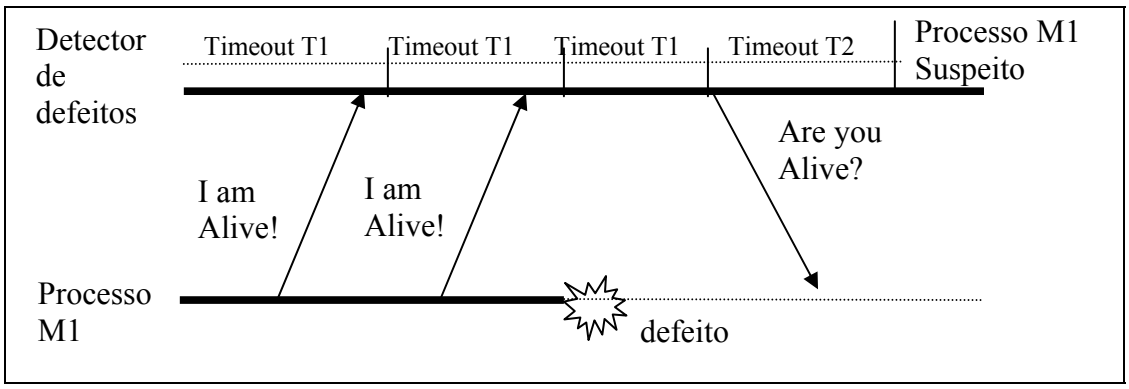

FIGURA 2.5 – Mensagens de monitoramento no estilo dual.

2.2.4 Versões adaptativas de *push*, *pull* e *dual*

Os detectores adaptativos, de modo geral, não apresentam um modelo diferente para realizar detecção de defeitos. Estes detectores na realidade são providos de um mecanismo que procura melhorar a precisão das suspeitas dos detectores realizando adaptações dinâmicas sobre o *timeout* dos processos.

No modo tradicional de implementar os detectores, onde se utilizam *timeouts* com valor fixo, é comum verificar que a detecção sofre distorção devido às diferentes velocidades de processamento e de comunicação dos processos monitorados. As comunicações dos processos lentos tendem a serem incorretamente considerados suspeitos, pois estes têm mais possibilidades de exceder o tempo limite do *timeout* [ESTEFANEL, 2001]. Simplesmente aumentar o valor do *timeout,* a fim de englobar processos mais lentos, pode trazer prejuízos à detecção, pois esta tem a sua freqüência de atualização diminuída, aumentando a latência da detecção.

Dessa forma, a escolha de um valor fixo que seja eficaz para ambas situações torna-se uma operação delicada e que está sujeita a todo tipo de influência do ambiente. Se, ao invés de usar um valor fixo, o detector puder corrigir o *timeout* de forma dinâmica, esta detecção possivelmente será mais precisa e cada processo monitorado, independente de sua velocidade, será tratado de forma mais adequada. Esta é a idéia que norteia o projeto de detectores de defeitos adaptativos baseados em *timeouts*.

Os detectores adaptativos foi proposta por Chandra e Toueg [1996]. A idéia busca atingir, de forma incremental, um valor de *timeout* no qual todos os processos corretos, mesmo os mais lentos, possam responder à detecção ainda em tempo hábil.

É claro que, enquanto o *timeout* ideal não é alcançado, os processos que excederem o tempo limite são considerados suspeitos, mas após o transcorrer de um certo número de ciclos de detecção, aumenta a probabilidade de que os processos que ainda não responderam estejam definitivamente falhos.

Além do modelo adaptativo proposto por Chandra e Toueg, onde o próprio detector decide quando aumentar os *timeouts.* Existe um trabalho [NUNES, 2003], que propõe o uso de um módulo especializado para o monitoramento da rede e dos recursos do sistema, de forma que a detecção seja adaptada com base em diversos fatores.

Neste presente trabalho questões de adaptação do *timeout* não foram consideradas, pois o AFDService já provê diferentes mecanismos para adaptar dinamicamente o *timeout* utilizado pelos detectores do seu repositório, bastando para isto que o usuário configure adequadamente o serviço.

2.2.5 Detector estilo *gossip*

O detector de defeitos do estilo *gossip* [RENESSE; MINSKY; HAYDEN, 1998] possui duas variações. Uma é nomeada de *basic-gossiping* (*gossip* básico) e a outra é nomeada de *multi-level gossiping* (*gossip* multi-nivel).

No algoritmo *gossip* básico, um membro mantém uma lista com o valor de um contador, chamado de *heartbeat*, para cada um dos seus vizinhos. O membro ocasionalmente envia sua lista para um membro escolhido randomicamente. Quando recebe a lista, realiza a união com sua lista e adota o maior *heartbeat* para cada membro.

Se o contador *heartbeat* de um dado membro A, contido na sua lista local de um membro B, não for incrementado antes de um dado *timeout*, o *host* B suspeitará do *host* A.

Para possibilitar uma maior escalabilidade, foi proposto uma variação do *gossip* básico chamado de *gossip* multi-nível. O multi-nível usa a estrutura de domínios da Internet e seu mapeamento se baseia em endereço IP, o que possibilita identificar domínios e sub-redes e mapear diferentes níveis. Como a maioria das mensagens é enviada pelo protocolo básico na sub-rede e poucas mensagens são trocadas entre os domínios, esta versão é melhor escalável.

A principal vantagem do *gossip* em relação aos modelos tradicionais (*push*, *pull* e *dual*) é a tolerância às mensagens perdidas. Entretanto, salienta-se que o tempo de detecção de defeitos piora gradualmente com o aumento da probabilidade de mensagens perdidas [RENESSE; MINSKY; HAYDEN, 1998].

Como a implementação do *gossip* é um dos objetivos deste trabalho, o protocolo de funcionamento do detector de defeitos *gossip* é detalhado no próximo capítulo.

3. DETECTOR DE DEFEITOS *GOSSIP*

Este capítulo apresenta em detalhes o detector de defeitos *gossip*. Para tal, inicialmente é apresentado o modelo de sistema distribuído considerado (seção 3.1), em seguida as propriedades do algoritmo (seção 3.2) e finalmente a descrição do protocolo básico utilizado (seção 3.3) e do algoritmo implementado (seção 3.4).

3.1 MODELO DE SISTEMA

O ambiente de execução é a Internet, logo: não existe limite conhecido de tempo para entrega das mensagens nem para o processamento num dado nó; mensagens podem ser perdidas e os nós podem deixar de funcionar (parada total). Em síntese, o sistema distribuído é assíncrono e sujeito a defeitos (links e nós).

Para tratar tal sistema, o projeto do protocolo *gossip* assume que: a maioria das mensagens são entregues dentro de algum tempo predeterminado (sistema parcialmente síncrono); que os relógios não são sincronizados, mas que a taxa do relógio em cada nó deriva muito pouco (*low drift*); e que a detecção é realizada por nó do sistema e não por processo.

3.2 PROPRIEDADES DO DETECTOR DE DEFEITOS GOSSIP

Conforme a seção 2.2.5, no *gossip* uma instância do detector mantém uma lista de seus vizinhos, cada um com um contador associado, e ocasionalmente esta lista é enviada para um membro escolhido randomicamente. Uma suspeita é levantada quando algum membro percebe que o contador parou de incrementar, sinal de que o membro está inacessível. Além disto, o *gossip* combina eficiência em disseminação hierárquica (estilo multi-nível) com a robustez do protocolo *flooding*

(membros periodicamente disseminam sua lista). Como resultado, o algoritmo *gossip* apresenta as seguintes propriedades [RENESSE; MINSKY; HAYDEN, 1998]:

1. A probabilidade de que um membro seja erroneamente relatado como falho é independente do número de processos;

2. O algoritmo é “resiliente” à perda de mensagens e aos processos falhos, pois uma pequena porcentagem de mensagens perdidas ou membros falhos não afeta a detecção de defeitos;

3. Se a derivação do relógio local é insignificante, o algoritmo detecta defeitos precisamente, ou seja, com uma probabilidade de erros conhecida;

4. O tempo de detecção aumenta pouco, numa ordem O(n log n), com o número de processos;

5. O algoritmo é escalável, a demanda da largura de banda cresce linearmente com o número de processos. Em uma rede hierárquica ela é quase constante

3.3 PROTOCOLO GOSSIP BÁSICO

Em 1972, Shostak e Baker descrevam o protocolo *gossip* usando uma analogia com “senhoras e telefones” [SHOSTAK; BAKER, 1972]. Posteriormente esse protocolo serviu de base do projeto *Clearinghouse* para resolver inconsistências de dados [YOGEN; OPPEN, 1983]. O protocolo foi também utilizado em replicação de base de dados na forma de algoritmo epidêmico [TAYLOR; GOLDING, 1992].

Van Renesse, Minsk e Haden foram os primeiros a estudar a aplicação do protocolo *gossip* em detecção de defeitos [RENESSE; MINSKY; HAYDEN, 1998], os quais apresentaram três protocolos: a versão básica (base para este trabalho), a versão hierárquica, e o protocolo que recuperação de processos falhos em particionamento de rede.

Na versão básica, cada membro *gossip* mantém uma lista de membros conhecidos contendo: os endereços dos membros e um inteiro para cada membro, que serve como contador de *heartbeat*.

A cada T_{gossip} instantes de tempo cada membro incrementa o seu contador e seleciona aleatoriamente um outro membro (vizinho) para enviar a sua lista. Quando um membro recebe uma mensagem *gossip* (contendo a lista), o membro realiza uma união da lista da mensagem com a sua lista local e mantém o valor máximo de cada contador *heartbeat* na lista final.

Cada membro também possui, para cada item da lista, o tempo da última vez que o seu contador correspondente foi incrementado. Se o seu contador não foi incrementado por mais que T_{fail} segundos, então o membro é considerado suspeito de falha. O T_{fail} é escolhido de acordo com a probabilidade de ocorrer uma detecção de falha errônea ($P_{mistake}$).

Quando um membro é considerado suspeito, ele não pode ser imediatamente eliminado da lista, pois há uma grande probabilidade de que alguns membros ainda não tenham detectado a falha. Por exemplo, considere que o membro A inicia a suspeita de um membro B, e em seguida recebe uma mensagem de outro membro, com boas informações sobre o B. Se o membro A retirou B de sua lista, ele irá inserir novamente o B na sua lista, pois pensará que B não é suspeito. Então A passará mensagens para outros membros informando que B não é suspeito e, conseqüentemente, o membro B que falhou nunca sairá da lista dos membros não suspeitos.

Conseqüentemente o detector de defeitos *gossip* não remove membros de sua lista de membros até que passe $T_{cleanup}$ instantes de tempo ($T_{cleanup} >= T_{fail}$). O $T_{cleanup}$ é escolhido de modo a fazer com que a probabilidade de receber uma mensagem *gossip* que contradize uma suspeita recente seja menor que $P_{cleanup}$. $P_{cleanup}$ pode ser igual a P_{fail} ou ser ajustado para $2 \times T_{fail}$.

Para verificar que isto funciona, considere que um membro B tenha falhado e considere que outro membro A tenha escutado o último *heartbeat* do *B* num tempo *t*. Com probabilidade P_{fail} todos os outros membros irão escutar o último *heartbeat* do *B* em $t + T_{fail}$, e assim todos os membros irão suspeitar de *B* no tempo $t + 2 \times T_{fail}$. Então, se ajustarmos $T_{cleanup}$ para $2 \times T_{fail}$, então com probabilidade P_{fail}, nenhum membro que tenha falhado, tal como B, irá reaparecer na lista de membros de um membro A, uma vez que o membro foi removido.

Note que, este protocolo somente detecta os nós que se tornam não-confiáveis. Ele não detecta falha de canal entre os nós. Em particular, se os nós *A* e *B* não podem falar um com outro, mas eles podem se comunicar com outro nó *C*, o nó *A* não suspeitará do nó *B*, nem o B suspeitará de A.

Burns, George, e Wallace [1999] dirigiram um estudo focado em uma simulação de análise de desempenho de alguns protocolos *gossips*, operando em um modelo de sistema distribuído. Eles empregaram e modificaram a versão do protocolo *gossip* no qual membros são removidos da lista depois de $T_{cleanup}$ sem a necessidade de verificação do T_{fail}. Sua modificação simplifica o protocolo básico requerendo somente a configuração de dois parâmetros, T_{gossip} e $T_{cleanup}$. Segundo [SUBRAMANIYAN; RAMAN, GEORGE, RADLINSKI, 2005], o tempo do *cleanup* é definido como múltiplo do tempo *gossip*, o qual é o tempo necessário para as informações alcançarem outros nós dentro do tempo limite $T_{cleanup}$[2]. Logo o valor do tempo *cleanup* é dado por $T_{cleanup} \geq n \times T_{gossip}$, sendo n o número de membros do sistema.

O resultado destes estudos ($T_{cleanup} \geq n \times T_{gossip}$) foi utilizado no presente trabalho para simplificar as escolhas dos parâmetros do algoritmo.

A seção seguinte descreverá o algoritmo do protocolo básico implementado neste trabalho.

3.4 DESCRIÇÃO DO ALGORITMO IMPLEMENTADO

Como foi dito na sessão anterior cada membro do protocolo *gossip* possui uma estrutura de dados, que a partir de agora será chamada de *membership* (grupo de membros). Então, cada membro mantém na sua lista local o endereço, ou identificador, um contador *heartbeat* e o *timestamp* de cada um dos membros do grupo. No início, cada membro ajusta os contadores e os *timestamps* em zero.

No exemplo da figura 3.1, a primeira coluna da tabela de *membership*, representa a identificação do membro, a segunda coluna é o contador *heartbeat*, e a coluna final é o *timestamp*.

[2] Este protocolo nomeado de *round-robin* (RR), garante que todos os membros irão receber de um nó arbitrário o valor do *heatbeat* atualizado num limite de tempo.

Id	Cont	Ts
A	0	0
B	0	0
C	0	0

FIGURA 3.1 – Tabela de *membership* (identificador, Contador, *Timestamp*).

Em cada intervalo de tempo (T_{gossip}) um membro incrementa o seu contador e randomicamente seleciona um outro membro para enviar uma mensagem. A mensagem *gossip* possui o identificador e o contador dos membros da lista *membership*. O algoritmo utilizado para enviar uma mensagem é mostrado na figura 3.2.

```
Inicialização:
        membershipList ← vazio;
        membershipList[0] ← membro local;

Repete a cada Tgossip:
        // Task
        m ← membershipList[0];
        m.heartbeat ← m.heartbeat + 1;
        target ← choiseTarget(); // escolhe um membro aleatório
        send( target,  membershipList ); // envia a lista membershipList
```

FIGURA 3.2 - Algoritmo de envio da mensagem *gossip*.

Quando um membro recebe a mensagem *gossip* ele compara a mensagem com a sua lista. Se o valor do contador de um dado membro é maior que o valor do contador correspondente na lista local, o novo valor é alterado para o maior valor e o *timestamp* é atualizado no tempo corrente. Como os contadores poderão eventualmente ter *overflow*, a implementação do protocolo pode ajustar o valor do

contador para zero[3]. A figura 3.3 mostra o algoritmo para comparação da mensagem gossip com a lista local.

```
/* incoming ← lista que veio na mensagem;
* timestamps ← lista que marca os tempos;
* horaLocal ← tempo do relógio local;
*/

 receive(incoming);
// busca  tupla <id, heartbeat> em incoming
for all <id, heartbeat> Є incoming do
        if ( process_i.heartbeat  <  incoming_i.heartbeat ) then
                process_i.heartbeat ← incoming_i.heartbeat;
                process_i.timestamps ← horaLocal;
        end if
end for
```

FIGURA 3.3 – Algoritmo de comparação da lista local com a lista recebida

Para detectar membros defeituosos, cada membro periodicamente varre sua lista procurando por um contador "velho" ou desatualizado. Um membro é adicionado na lista de suspeitos se o seu contador não foi atualizado dentro do intervalo $T_{cleanup}$.

Um exemplo da execução do código pelos membros é mostrado na figura 3.4.

```
Inicialização
        listaLocal ← lista local
        Tcleanup ←  2 * Tfail

Repete cada Tcleanup:
for all <id, heartbeat > Є  listaLocal do
        if ( tempo_corrente – listaLocal[i].timestamp > Tcleanup ) then
                Adiciona listaLocal[i].id na lista_de_suspeitos;
        end if
end for
```

FIGURA 3.4 – Algoritmo que verifica intervalo de tempo em que o *heartbeat* foi incrementado

[3] Também pode ser utilizado o protocolo de janela deslizante (*sliding window*).

Na figura 3.5 descreve o envio do *gossip* do membro A e do membro C. Neste caso, o A seleciona randomicamente B para enviar a mensagem *gossip* que contem o identificador do processo (id) e o contador *heartbeat* (cont).

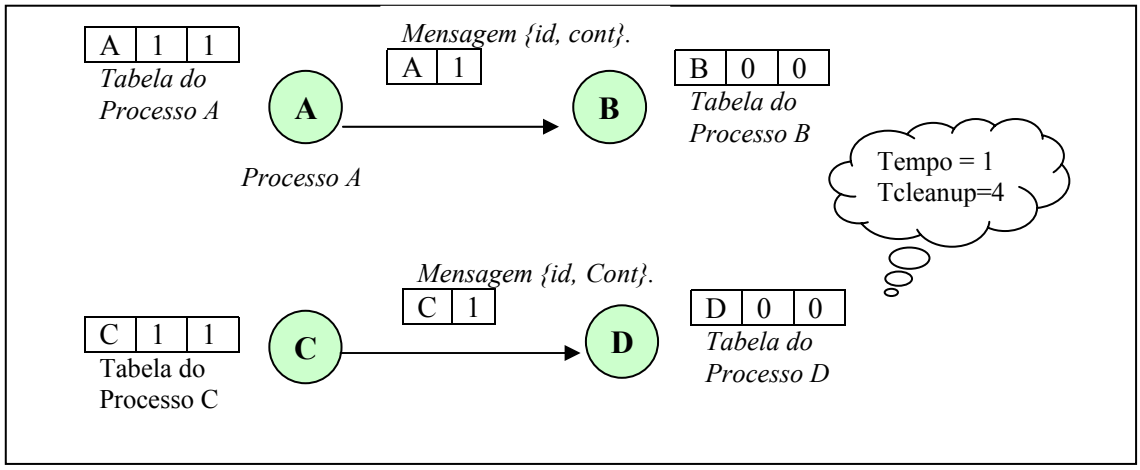

FIGURA 3.5 - Situação durante execução. Os processos A e C enviam mensagens

A Figura 3.6 mostra a situação do grupo depois de B ter processado a mensagem de A e atualizado sua lista. Note que o contador do A na mensagem foi inserido na tabela do B. E atualizado o *timestamp*. Neste caso B usa o tempo 1 para a atualização do *timestamp.*

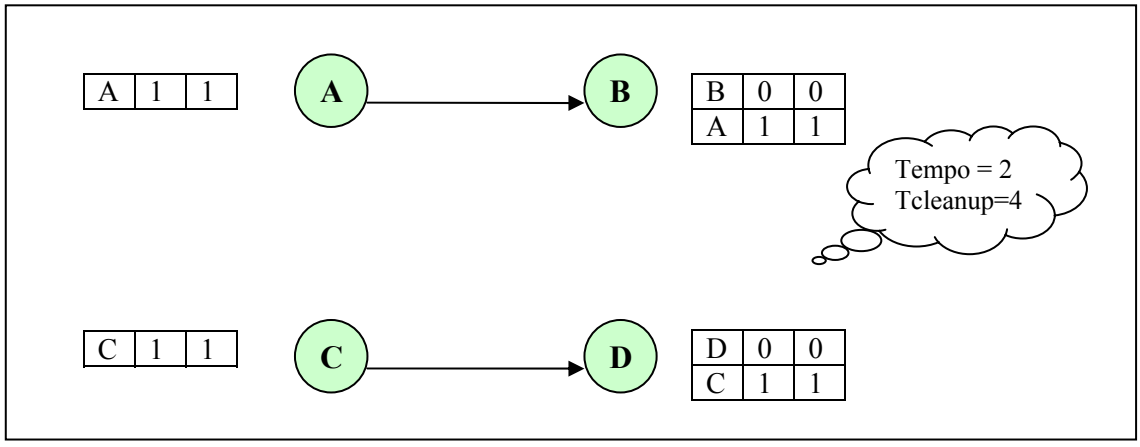

FIGURA 3.6 - Situação do gossip depois da troca de mensagens.

A Figura 3.7 descreve a situação onde os membros do grupo detectaram uma suspeita. A situação do grupo é a seguinte: o tempo atual é 8 (esta unidade de

tempo vem do *timestamp* da máquina local); o membro C parou de atualizar o seu contador no tempo 3, enquanto os membros A, B, e D e seus contadores foram atualizados e enviaram as mensagens *gossip* informando seu estado a seus destinatários.

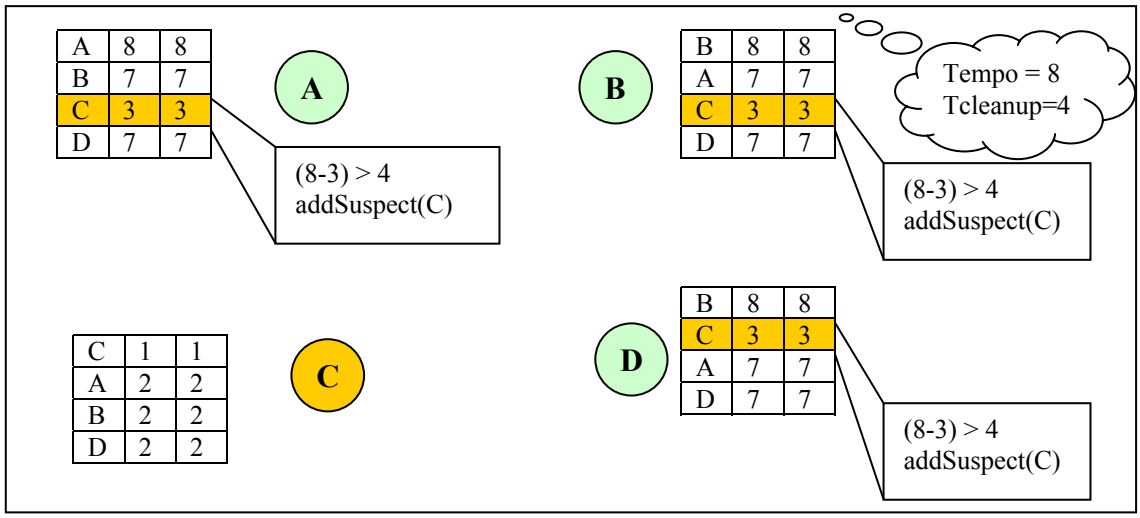

FIGURA 3.7 - Os membros A, B, e D detectam como falho.

A cada período de $T_{cleanup}$ = 4 todos os membros do grupo varrem suas listas para detectar o suspeito, como foi mencionado anteriormente. Quando inicia um novo período $T_{cleanup}$ (tempo atual é 8), os nós A, B e D varrem a suas listas e observam que o *timestamp* do nó C é 3. Além disto cada nó calcula o estado do C subtraindo o *timestamp* do C do tempo local (8-3=5), o que indica que C é suspeito, pois o $T_{cleanup}$ < 5. Então A, B, D, adicionam o membro C na sua lista de suspeito.

4. AFDSERVICE

Neste capítulo é apresentado o AFDService [NUNES, 2003], o serviço de Detecção de Defeitos no qual o algoritmo *gossip* implementado neste trabalho foi incorporado.

As seções a seguir explicam: as vantagens de se ter um detector de defeitos como um serviço (seção 4.1); a arquitetura do AFDService (seção 4.2); e as interfaces do AFDService (seção 4.3);

4.1 DETECTOR DE DEFEITOS COMO UM SERVIÇO

Os detectores de defeitos, úteis em vários domínios de aplicações, como sistemas de controle e supervisão e na resolução de acordo em sistemas distribuídos, podem ser especificados de várias maneiras, sendo algumas das quais são citadas a seguir:

- quanto ao fluxo de informações – *push*, *pull* [FELBER; DÉFAGO; GUERRAOUI, 1999];
- quanto à arquitetura lógica de comunicação – por difusão (*broadcast*)[CHANDRA; TOUEG, 1996], de maneira hierárquica [BRASILEIRO; FIGUEREDO; SAMPAIO, 2002][FELBER; DÉFAGO; GUERRAOUI, 1999] ou em anel [LARREA; ARÉVALO; FERNANDEZ, 1999];
- quanto ao ajuste do *timeout* – que pode ser fixo ou ajustado de acordo com o comportamento dos atrasos de comunicação na rede.

Além dessas características, os detectores podem ter diversas implementações. Uma abordagem comum é o detector estar fundido com a aplicação cliente, o que aumenta o seu desempenho [SERGENT; DÉFAGO;

SCHIPER,1999], no entanto contribui para aumentar a complexidade do projeto, elevando o custo da aplicação.

Para reduzir a complexidade do projeto, o detector de defeitos pode ser encapsulado em uma camada de serviço *middleware*. Com uma modelagem orientada a objetos, detectores de defeitos podem ser objetos primitivos [FELBER; DÉFAGO; GUERRAOUI, 1999], o que facilita uma arquitetura genérica para o serviço. Desse modo, o detector pode ter uma fácil manutenção de código, e as classes podem ser reutilizadas para várias aplicações sem necessidade de alteração de código. Também pode ser adaptado e configurado a variados ambientes e aplicações.

4.2 ARQUITETURA DO AFDSERVICE

O AFDService apresenta uma arquitetura configurável e flexível para um serviço de detecção de defeitos. Ele é configurável através de um arquivo de configuração padrão. É flexível no sentido que permite a troca, estática ou dinâmica, de algoritmo de detecção, predição ou margem de segurança, e permite a rápida inclusão de novos algoritmos.

A arquitetura básica é composta por dois módulos: um módulo de detecção (FD) e um módulo de previsão (TS), como mostra a figura 4.1. Considera-se que cada *host* possui uma referência do serviço de detecção.

A aplicação cliente interage com o módulo FD, recebendo informações de estado sobre os componentes monitorados. As informações podem ser prestadas na forma síncrona (*requisição/ resposta*) ou assíncrona (*callback*).

O módulo FD mantém um repositório de diferentes algoritmos de detecção de defeitos e cada módulo de previsão mantém um repositório de distintos algoritmos de previsão. Os repositórios são controlados por um configurador gerente, sua função é coordenar a interação com a aplicação cliente e configurar a escolha de algoritmos de detecção e de previsão.

A monitoração da acessibilidade dos *hosts* remotos é tarefa do módulo de detecção, mas a estimativa do próximo *timeout* a ser utilizado depende da previsão realizada pelo módulo de predição.

Ambos os módulos (FD e TS) adotam o padrão de projeto *Strategy* [GAMMA; HELM; JOHNSON; VLISSIDES, 1994] [PETER, 2004], o que permite uma fácil agregação de novos algoritmos de detecção de defeitos ou de novos mecanismos de predição de valores.

Como o objetivo deste trabalho é incorporar o algoritmo básico *gossip* ao AFDService, o módulo FD foi o objeto de estudo para a implementação. Para este trabalho não foi usada adaptação dinâmica do *timeout* e conseqüentemente o modulo de previsão.

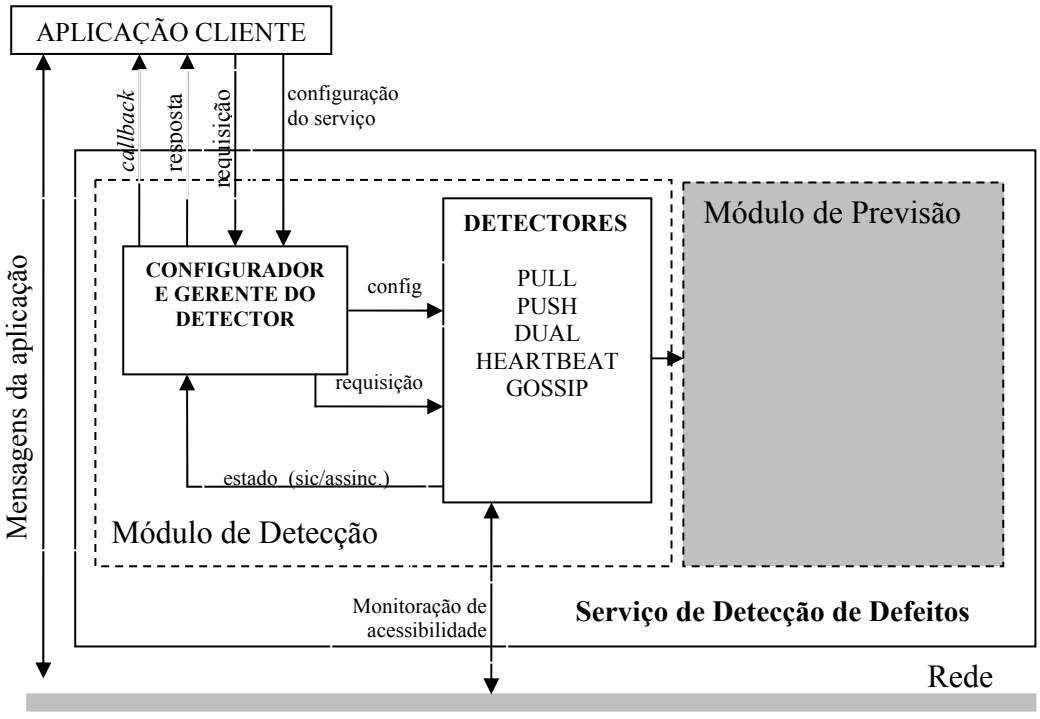

FIGURA 4.1 - Arquitetura dos módulos de detecção e previsão.

4.3 INTERFACES

O AFDService possui um conjunto de interfaces baseado na solução proposta por [FELBER;FAYAD;GUERRAOUI,1999], portanto possibilita a aplicabilidade de um serviço de detecção em qualquer contexto. O diagrama de classes das interfaces do serviço de detecção configurável é ilustrado na figura 4.2 onde a linha pontilhada

salienta as que são orientadas as aplicações.

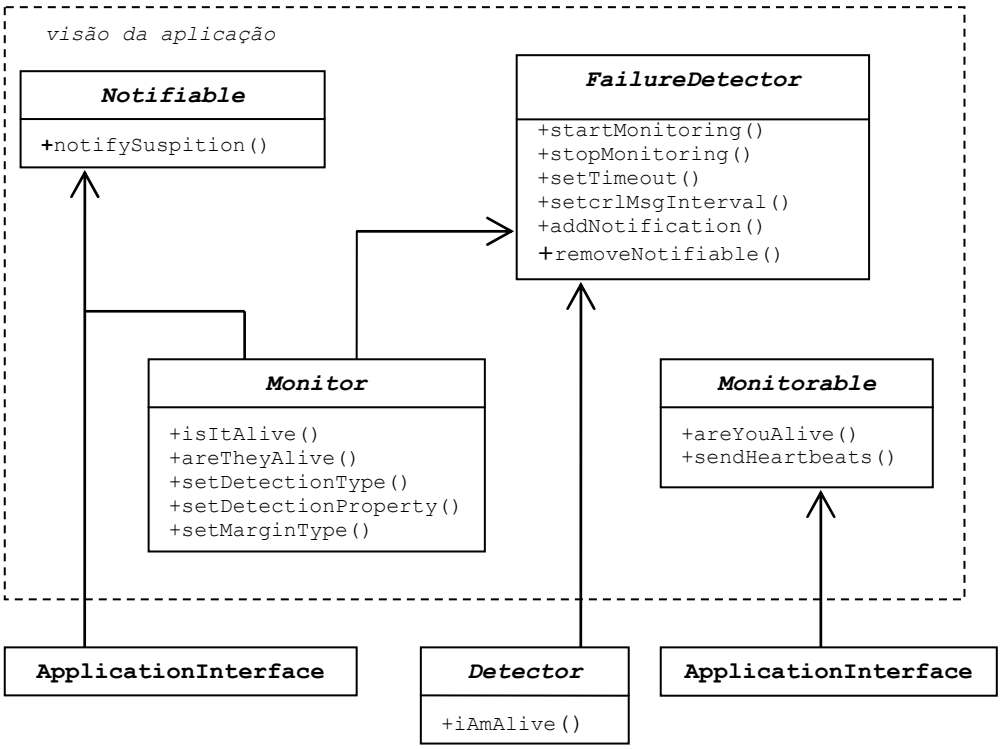

FIGURA 4.2 – Conjunto de interfaces do AFDService

São definidos três componentes básicos para um detector de defeitos: um monitor (***Monitor***), um monitorável (***Monitorable***) e um notificável (***Notificable***). O componente monitor irá observar o componente monitorável e irá avisar o componente notificável de maneira assíncrona as alterações de estado.

A vantagem de usar três componentes orientados à aplicação é que o serviço de detecção pode ser adaptado a uma hierarquia de rede, sem necessidade de modificar a aplicação. Nota-se que o usuário do serviço pode definir quais aplicações serão monitores, quais serão monitoradas e quais devem ser notificadas sobre as mudanças de estados.

Ambos os módulos TS e FD adotam o padrão de projeto *Strategy*, o qual permite que uma família de algoritmos seja utilizada de modo independente e seletiva [PETER 2004]. No padrão *Strategy*, uma classe abstrata que deve ser implementada por toda estratégia concreta (um algoritmo específico).

A figura 4.3 mostra a estrutura do padrão de projeto *Strategy*. Esta estrutura permite que novos algoritmos de detecção ou predição possam ser facilmente

agregados ao serviço.

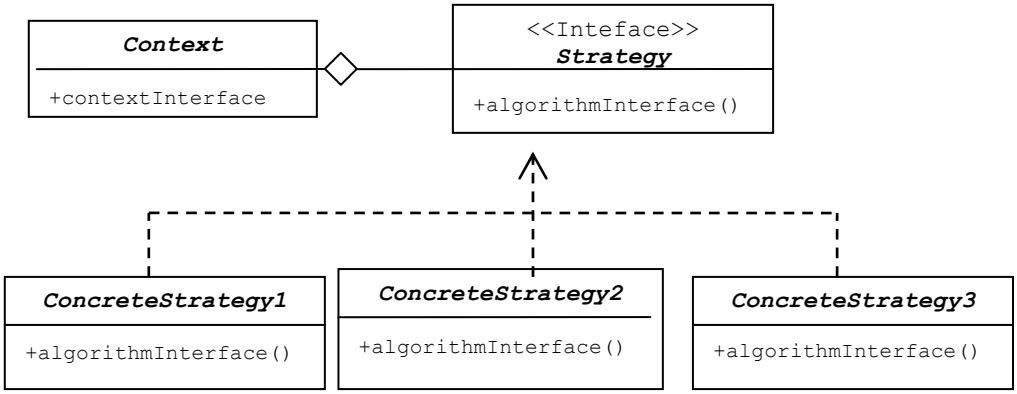

FIGURA 4.3 – Estrutura padrão Strategy

Para isso o padrão utiliza uma classe de contexto (***Context***), que representa o gerente dos algoritmos. Os algoritmos correspondem às estratégias concretas (***ConcreteStrategy***), que implementam uma mesma interface (***Strategy***), de modo que o gerente (FDManager, na figura 4.4) possa chamar métodos das estratégias concretas sem conhecer sua implementação.

Substituindo as classes do modulo FD no padrão *Strategy*, chegamos à arquitetura, ilustrada na figura 4.4. Novos algoritmos de detecção podem ser adicionados facilmente, desde que implementem a interface ***Detector***.

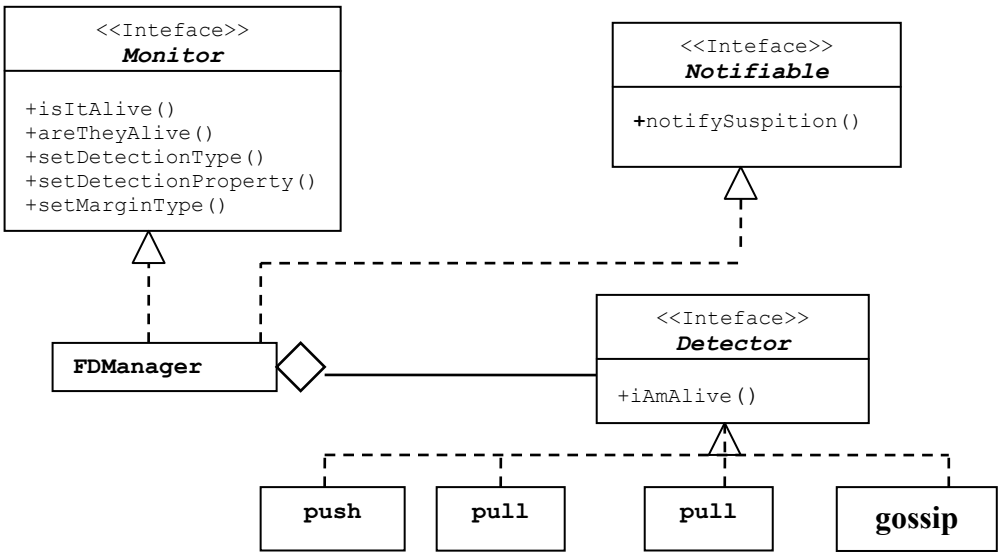

FIGURA 4.4 – Arquitetura do módulo FD.

Em síntese, o padrão *Strategy* permite as inserções facilitadas de novos algoritmos, devendo apenas acrescentar a nova classe e referenciá-la no gerente de estratégias.

5. IMPLEMENTAÇÃO DO ALGORITMO GOSSIP

Neste capítulo são apresentados os detalhes do detector *gossip* implementado, incluindo a descrição das classes usadas. Como comentado na introdução, a primeira parte deste trabalho é o desenvolvimento da implementação do algoritmo de detecção de defeitos do estilo *gossip* junto ao AFDService.

Observa-se que no AFDService o algoritmo *gossip* implementa um modelo de detecção que difere dos outros já implementados no AFDService (*pull*, *push* e *dual)*, pois estes são baseados em *timeouts* para a detecção de defeitos, enquanto que o *gossip* se baseia em probabilidade de erros.

5.1 O GOSSIP NO AFDSERVICE

A implementação do algoritmo *gossip* foi feita em Java, pois o AFDService é implementado em grande parte nesta linguagem. Entretanto, salienta-se que outras linguagens de programação poderiam ter sido utilizadas.

Para a comunicação entre objetos remotos foi utilizado *sockets* UDP. Logicamente será necessário ter uma máquina virtual Java executando em todas as máquinas onde estiver rodando o detector.

As principais interfaces utilizadas nesse trabalho são o *Detector* e o *Notifiable*, sendo o módulo de detecção de defeitos implementado no pacote FD. Foi adicionado um novo algoritmo de detecção, o *gossip* básico.

5.1.1 Interface Detector.

A interface *Detector* especifica um detector de defeitos para a aplicação. Ela provê um amplo conjunto de métodos que possibilita tanto a consulta de estados

quanto a configuração de parâmetros.

A seguir são apresentados os principais métodos da interface e a explicação do seu comportamento.

O serviço precisa prover um modo de ser gerenciado. Desta forma, dois métodos foram definidos para permitir que os módulos distribuídos do serviço possam ser iniciados e interrompidos, de acordo com as necessidades da aplicação.

- startMonitoring (Member member) Este método é utilizado para inscrever um objeto em um domínio de detecção. Ele recebe o identificador do membro (uma referência a um objeto do tipo *Member*).

- stopMonitoring (Member member) Remove o membro, suspendendo o monitoramento do membro.

Uma forma de estar ciente do estado dos objetos monitorados é consultar o estado. Os métodos a seguir podem ser utilizados para recuperar esta informação.

- isItAlive (Member member), areTheyAlive (Vector members) consulta os estados dos objetos monitorados.

Porém, existe outra forma de saber sobre a ocorrência de suspeitas. É preciso configurar o serviço para notificar um processo sobre suspeita de falhas em objetos monitoráveis. A notificação é realizada por *callback* ao objeto notificável. Isto é disponibilizado pelo seguinte método.

- addNotifiable (Notifiable object). Configura um processo para ser notificado de ocorrência de suspeitas.

- removeNotifiable (Notifiable object). Remove o membro notificável da lista de notificáveis.

Para configurar os objetos que precisam ser monitorados pode-se usar os métodos abaixo.

- addMonitorable (Member member) Esse método é utilizado para adicionar um objeto novo para ser monitorado pelo serviço.

- removeMonitorable (Member member) Remove um objeto da lista corrente de monitoráveis.

Qualquer objeto de uma classe que implementa a interface Detector atua como detector de defeitos.

5.1.2 Interface Notifiable

A interface Notifiable especifica um objeto que precisa receber notificações de suspeita e de recuperação.

- notifySuspection (Member suspected) Este método será invocado pelo detector quando da suspeita de um membro.

- notifyRecovery (Member recovered) Este método será invocado pelo detector de efeitos quando ocorrer a recuperação (um membro suspeito volta a ser confiável).

5.1.3 Classe FDManager

Como foi citado anteriormente o *FDManager* é o gerente das estratégias de detecção, a classe *FDManager* implementa a interface *Detector*. Esta classe corresponde à classe *Context* no padrão *Strategy*.

Nesta estrutura um novo algoritmo (Gossip) deve ter a sua identificação incluída na classe *FDManager*. Para que o usuário ou aplicação do serviço possa escolher uma das estratégias de detecção do repositório, o *FDManager* deve conhecer quais são as estratégias de detecção são implementadas.

As identificações dos objetos monitores e monitoráveis é realizada por um objeto da classe *Member*, o qual contém os seguintes atributos: nome, endereço IP

e duas portas (entrada e saída). Estes objetos são adicionados no arquivo de configuração, ou passados diretamente pela aplicação cliente, pelo método *startMonitoring()*. Os objetos monitoráveis são mantidos em uma lista (*monitorableList*) e os membros suspeitos noutra lista (*suspectList*), ambas mantidos pelo *FDManager*.

5.1.4 Classe GossipDetector

A partir destas interfaces e classe apresentadas acima se desenvolveu o detector de defeitos *gossip* integrado ao AFDService.

A implementação do algoritmo *gossip* foi realizada na classe *GossipDetector*, que implementa a interface Detector do AFDService, como mostra a figura 5.1. A princípio, todos os membros são monitores e monitoráveis e o vetor *monitorableList* serve para escolher o destino das mensagens.

Para as mensagens entre os detectores criou-se uma classe de objeto chamado de *Membership* que são definidos pela superclasse *Member*, nele contém o atributo *heartbeat* (contador).

Na classe *GossipDetector* aparecem as seguintes estruturas:

- *membershipList*: vetor que mantém informações dos membros conhecidos pelo detector. Contém objetos do tipo *Membership*.
- *timestamp*: tabela que contém os tempos do último incremento do *heartbeat* de cada membro.
- *Tgossip*: intervalo entre duas mensagens gossip.
- *Tcleanup*: intervalo de verificação da lista *membership*.

A classe *GossipDetector* usa instâncias da classe *Membership*, que são armazenadas no vetor *membershipList*. A classe *Membership* possui como atributos o *heartbeat* e os atributos herdados da classe *Member* do AFDService. A classe *Member* contém os seguintes atributos: nome, endereço IP e uma porta de recepção das mensagens.

Para fazer o envio de mensagens a cada *Tgossip* unidade de tempo, o

detector escalona um objeto do tipo *RequestControler*. Nele é usado o método *gossip()* que incrementa o *heartbeat* pertencente ao próprio detector e marca a hora local na tabela *Timestamps* correspondente ao seu atributo nome. A hora local é registrada pelo método *System.currentTimeMillis()*, que retorna o tempo em milessegundos.

Em seguida, o método *gossip()* envia o seu vetor *membershipList* (inicialmente com um elemento *Membership*, referente a ele mesmo) para um membro escolhido aleatoriamente da lista de monitoráveis.

Como o objeto *Vector* do Java é serializável, pode-se encapsular este objeto e enviá-lo como mensagem. O tamanho das mensagens enviadas é escolhido pelo pior caso, o que corresponde ao tamanho máximo do vetor *membershipList*, ou seja, o numero total de monitoráveis (número de monitoráveis vezes o tamanho do objeto *Member*).

Para a recepção das mensagens, usa-se um objeto servidor (*ReceiveServer*) o qual é responsável por receber todas as mensagens de outros detectores. Para resolver o problema de concorrência de acesso ao servidor de mensagens. Foi criada uma *thread* chamada *DeliveryGossip*.

Quando o servidor recebe uma mensagem, ele cria um novo objeto *DeliveryGossip*, no qual trata cada mensagem recebida. Ele recebe mensagens *gossips*, desserializa a mensagem e invoca o método *mergeLists()* para fazer a junção com a lista do detector.

O método *mergeLists()* compara duas listas (*membership* local com a da mensagem recebida). Para cada identificador igual nas duas listas, verifica-se qual o maior *heartbeat*, atualizando-o para esse valor. Marca-se também a hora local na tabela *timestamps* correspondente.

Se a lista recebida conter um membro que não pertence à lista local, então este membro será acrescentado na lista local. Isto significa que este membro pode ser um novo membro, visto pela primeira vez, ou um membro que se recuperou. Como todos os membros são inicializados como suspeitos é necessário realizar um *callback* ao *notifyRecovery()* do *FDManager* para esse novo membro.

O método *mergeLists()* é acessado por várias *threads DeliveryGossip*. Para evitar problemas de concorrência o *mergeLists()* é sincronizado com o modificador *synchronized*, que garante o acesso exclusivo ao método.

A cada *Tclearup* unidade de tempo, o *GossipDetector* escalona uma *thread*

do tipo *CheckTclearup*. Ele cria uma *thread* que varre o vetor de membership, e avalia se um dos membros registrados será inserido na lista de suspeitos. Se o último incremento do *heartbeat* de um membro contido na lista membership não foi incrementado dentro deste intervalo (*timestamp* – hora corrente < *Tclearup*). Então esse membro é detectado como um suspeito e o *CheckTclearup* adiciona o membro na lista de suspeito realizando um *callback* ao método *addSuspection()* da classe *FDManager*.

O *FDManager*, quando instanciado, pode receber uma lista de monitoráveis, com a invocação do método *startMonitoring()*. Caso a lista seja vazia, ou contenha apenas o membro local, o AFDService faz dele um objeto monitorado e instancia somente o *ReceiveGossip*. Caso contrário, se a lista conter mais elementos, o AFDService opera localmente o detector de defeitos sobre os membros monitoráveis.

Para informar uma suspeita, existe duas maneiras: A assíncrona implica que o cliente deve implementar a interface *Notifiable*; e a síncrona, invoca os métodos *isItAlive (um monitor)* ou *areTheyAlive (vetor monitoráveis)* do objeto *FDManager*.

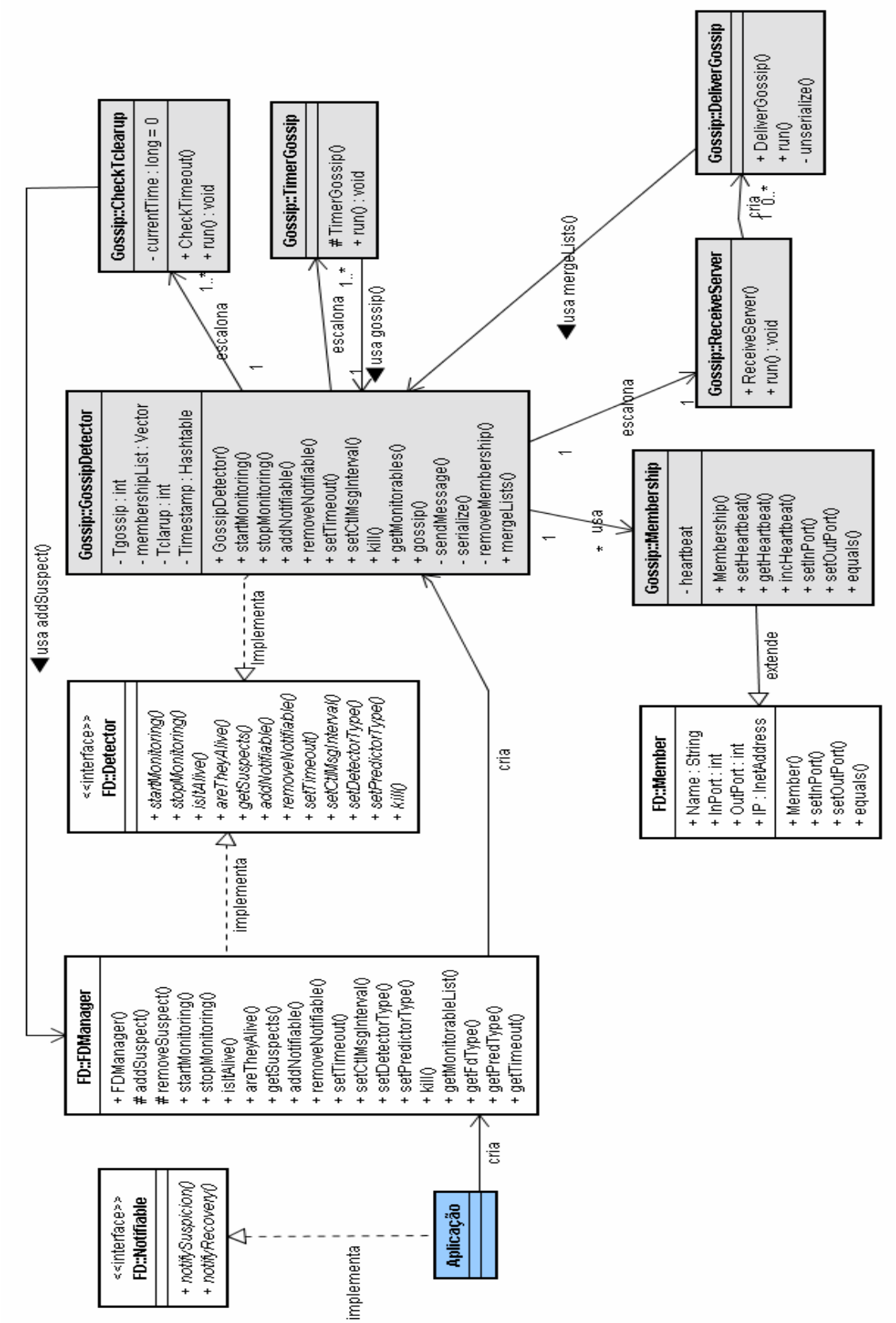

FIGURA 5.1 - Diagrama de classes modificada acrescentada o algoritmo gossip (no pacote "Gossip::").

5.1.5 Exemplo de utilização do serviço para o algoritmo gossip.

Um programa de aplicação foi implementado como cliente e ajustado para ser um objeto notificável. Ele implementa a interface *Notifiable* para receber as notificações feitas pelo detector.

As configurações iniciais do detector são realizadas via leitura de um arquivo padrão local, contendo endereços dos membros monitorados (alvos das mensagens *gossip*), o tipo de detector (GOSSIP) e o tipo de predição (OFF).

Em tempo de execução, a aplicação cliente interage com o objeto *FDManager* recebendo informações de estado sobre os componentes monitorados.

As informações podem ser prestadas de forma síncrona (requisição/resposta) ou assíncrona (callback).

A figura 5.2 mostra os principais trechos de código.

```
public class ClienteAplicacao implements Notifiable {
   ...
   public ClienteAplicacao() {
   ...
            fd = new FDManager(arquivo, timeout);
            fd.addNotifiable( this );
   ...
   }
   ...
 /* Este método será invocado pelo detector de defeitos quando o detector de
  * defeito começa a suspeitar de algum membro sendo monitorado.
  */
   public void notifySuspicion( Member suspected ) {
          System.out.println( suspected.IP +" e um suspeito." );
   }

  /* Este método será invocado pelo detector de efeitos quando ocorrer a
   * recuperação de algum membro erroneamente considerado suspeito.
   */
   public void notifyRecovery( Member recovered ) {
          System.out.println( recovered.IP +" se recuperou." );
   }
   ...
}
```

Figura 5.2 – Trechos de código Java comentado, extraído da aplicação cliente.

5.2 TESTES DO ALGORITMO GOSSIP

Esta seção propõe mostrar a metodologia para os teste do algoritmo posteriormente os resultados obtidos.

Para testar o *gossip* e avaliar a sua qualidade de serviço, foram escolhidas duas métricas propostas por Chen, Toueg e Aguilera. Estas métricas independem da implementação do detector [CHEN; TOUEG; AGUILERA, 2002].

As métricas escolhidas são:

1. **Tempo para recorrência ao erro** *(Mistake recurrence time T_{MR})* – mede o tempo entre dois erros consecutivos cometidos pelos detectores (suspeitas incorretas).

2. **Duração de um erro** (*Mistake duration- T_M*) – mede o tempo que o detector de defeitos leva para corrigir um erro.

5.2.1 Computação das métricas no teste.

Nesta seção, explica-se como foram computados os instantes de transição de estado e as métricas de qualidade de serviço.

Conforme as métricas definidas acima, devem ser computadas com base num histórico de transições de estados de um detector de defeitos, onde o histórico indica precisamente todos os instantes de transição de estado suspeito ($t_{Stransição}$) para confiável ($t_{Ttransição}$).

A computação dos instantes de transição corresponde à leitura do relógio local sempre que uma troca de estado for percebida.

O histórico ordenado de transição é representado por $H_{transição.}$ Cada vez que uma notificação acontece, duas transições são computadas: $H_{transição}[t] \leftarrow t_{Stransição}$ e $H_{transição}[t+1] \leftarrow t_{Ttransição}$, onde t indica a ordem das transições no histórico.

As métricas de precisão são definidas em livres de falhas e com função dos instantes de transição de estados $t_{Stransição}$ e $t_{Ttransição}$.

Seguindo a definição das métricas de precisão (seção 5.2), o tempo de um erro (duração de uma falsa suspeita) é computado por

$$t_M = t_{Ttransição} - t_{Stransição}$$

onde $t_{Stransição}$ e $t_{Ttransição}$ são adjacentes e $t_{Stransição}$ precede $t_{Ttransição}$. Ciente que as transições foram inseridas no histórico $H_{transição}$ de maneira ordenada, tem

$$t_M = H_{transição}[t+1] - H_{transição}[t] \text{ para todo } t < n*2 \mid t = 1, 3, 5, \ldots$$

onde $H_{transição}[t+1] = t_{Ttransição}$ e $H_{transição}[t] = t_{Stransição}$.

Da mesma maneira, o tempo de recorrência ao erro é computado por

$$t_{MR} = H_{transição}[t+2] - H_{transição}[t] \text{ para todo } t < (n-1)* 2 \mid t = 2, 4, 6,$$

onde $H_{transição}[t+2]$ e $H_{transição}[t]$ são dois instantes $t_{Stransição}$ adjacentes

5.2.2 Resultados e análise dos testes

Os testes práticos foram realizados em um ambiente de interação com uma rede interna. Assim foram utilizadas 9 máquinas em sistema (Linux).

Nos testes foram usados os seguintes parâmetros; para o período de envio das mensagens (T_{gossip}) foi usado em 100 ms (um intervalo menor sobrecarrega os recursos da rede), e para parâmetro *Tfail* foram escolhidos valores empiricamente até encontrar um ponto de estabilidade. Ou seja, quando não tiver detecções erradas, ocasionando transições entre suspeitos e recuperados consecutivamente.

Foram realizados testes no qual foram computados a média de duração de um erro (T_{MR}) e a média de recorrência ao erro(T_M).

O detector de defeitos foi avaliado em relação a sua precisão, no qual o mais preciso é o que oferece:

- o menor tempo médio de duração ao erro (T_M); e
- o maior tempo médio de recorrência ao erro (T_{MR})

Supondo que as execuções são livres de falhas, como as utilizadas nos testes, sabe-se que uma medida de T_M só ocorre quando uma falsa suspeita é

descoberta, ou seja, o intervalo (*Tcleanup* = 2 * *Tfail*) deve ser expirado e posteriormente uma mensagem válida do componente monitorável deve ser recebida. Logo, tal métrica auxilia na verificação do quanto o valor do *Tfail* é menor do que deveria ter sido adotado para evitar uma falsa suspeita.

A figura 5.3 mostra o resultado do teste da medida de T_M em relação a *Tfail*.

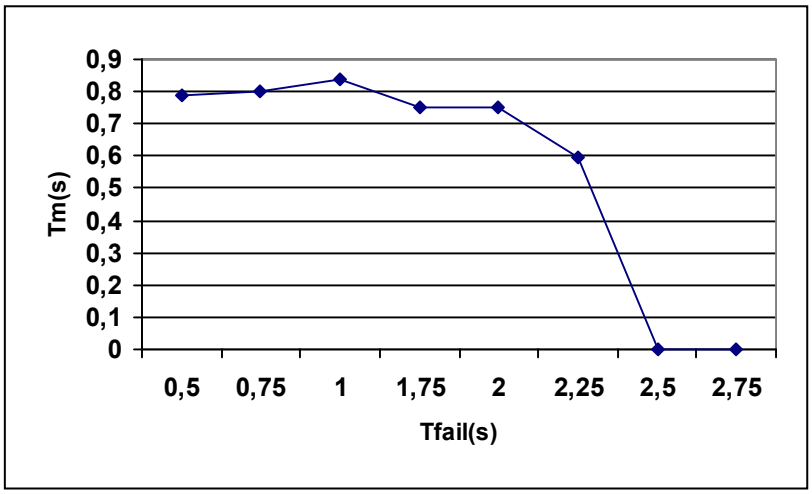

FIGURA 5.3 – Tempo médio de duração ao erro (T_M).

O gráfico mostra que o *Tfail* entre 0,5 e 2,25 segundos é menor do que deveria ter sido adotado. Logo o número de falsas suspeita decresce com o aumento do *Tfail.*

Quando o *Tfail* for maior que 2,25 segundos, não ocorre mais transições de estados, logo não existe mais erro de detecção (o valor do T_M é zero).

Já a métrica de precisão T_{MR}, que descreve o tempo decorrido entre dois erros consecutivos do detector, permite verificar o quão está próximo do que seria o *Tfail* ideal.

A figura 5.4 mostra o tempo médio de recorrência ao erro em relação *a Tfail*.

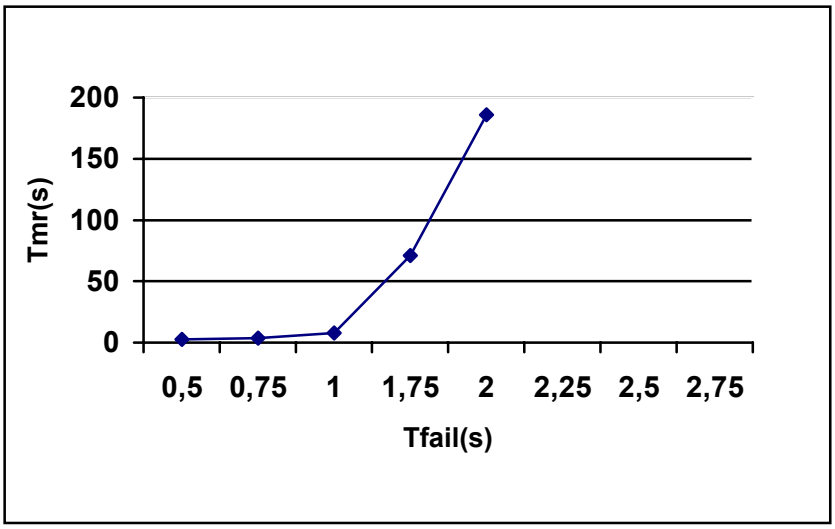

FIGURA 5.4 – Tempo médio de recorrência ao erro (T_{MR}).

Note no gráfico acima, que a média de recorrência ao erro é crescente em relação ao *Tfail*. Conforme o *Tfail* aumenta, diminui o número de ocorrência de erros. Logo, a diferença entre dois erros consecutivos aumenta. Pois existem poucos erros em um intervalo muito grande, tendo pouca detecções falsas, chega-se próximo ao *Tfail* ideal.

Em síntese as métricas auxiliam a interpretar e estabelecer uma melhor qualidade ao serviço. No caso do experimento realizado, o algoritmo *gossip* com 9 processos é preciso a partir de *Tfail* maior que 2,5 segundos.

6. DESENVOLVIMENTO DA INTERFACE GRÁFICA

Este capítulo expõe uma seção de desenvolvimento a interface gráfica, nomeada de FDView, a sua estrutura e como interage com o AFDService.

A seção 6.2 mostra o layout e as funções de cada interface gráfica da ferramenta de visualização.

6.1 OBJETIVO E ESTRUTURA DA INTERFACE GRÁFICA

A interface gráfica tem como objetivo ajudar o usuário ou o aluno a compreender as características de um determinado detector de defeitos.

A interface gráfica acrescenta ao aluno uma visão prática do sistema de detecção de defeitos. O aluno define a configuração do detector como, por exemplo, *timeout* ou *Tfail*, intervalo de envio de mensagem e números de monitores ou monitoráveis, podendo ter diferentes comportamentos ou qualidade de serviço no seu funcionamento.

A interface gráfica tem a vantagem de ser mais amigável ao usuário, aumentado a atratividade para conhecer o serviço, pois possibilita a visualização gráfica das ações efetuada pelo detector do AFDservice.

Com a ferramenta, o usuário tem uma noção mais prática dos conceitos de detecção de defeitos em sistemas distribuídos.

Para implementação do layout da interface foram usados componentes dos Pacotes Java Swing e AWT (*Abstract Window Toolkit.*).

A ferramenta funciona através de uma conexão *socket TCP* (mais segura, pois mensagens não são perdidas e chegam em ordem), que faz conexão com o aplicativo cliente do AFDService. Com isso o usuário pode ter acesso a qualquer monitor da rede que se encontra num host que não tenha restrições ao uso de porta que o usuário escolheu, ou da porta padrão do serviço.

A figura 6.1 mostra como a interface gráfica interage com o AFDService.

A implementação do cliente (aplicação notificável) foi modificada com o acréscimo de uma *thread* servidor *(1)*. Nela cria-se um *socket* TCP que aguarda uma solicitação de conexão por uma porta padrão. A execução do cliente e o monitoramento na rede são iniciados, e o cliente recebe ocasionalmente notificações *(2)*.

Na ferramenta gráfica, um *socket* solicita uma conexão à aplicação cliente *(3)*. Quando a aplicação aceita a conexão, ela envia as configurações para que sejam atualizadas as informações do monitor na ferramenta FDView *(4)*. Após este protocolo inicial, os dois têm permissão para interagir entre si *(5)*, enviado ou recebendo informações de estados dos *hosts* (nós suspeitos/recuperados) ou parâmetros de configuração (timeout, intervalo de mensagem).

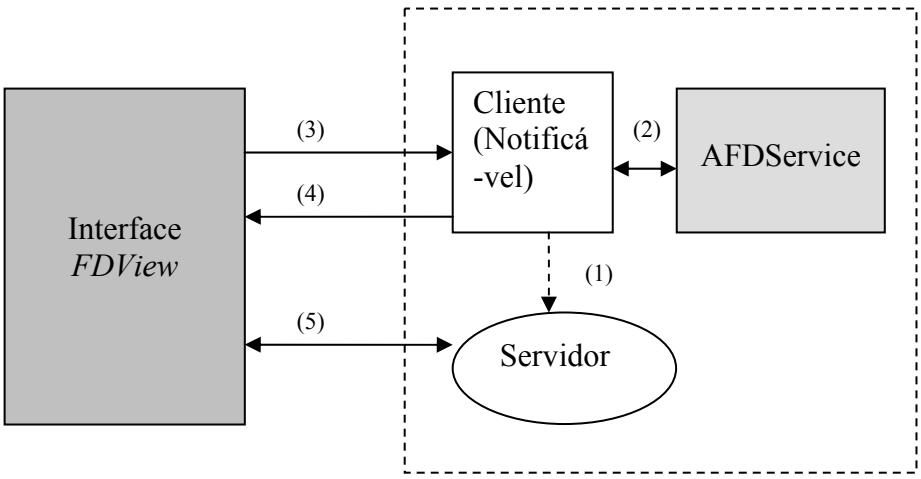

FIGURA 6.1 – Esqueleto da Interface com o FD.

Quando o cliente recebe uma notificação do FDManager *(2)*, é enviada uma mensagem á interface sobre o ocorrido (código da figura 6.2). Dessa forma a Interface atualiza o histórico.

Na mensagem é contido o tipo da mensagem (SUSPEITO/ RECUPERADO). O tipo indica para a ferramenta se o membro é suspeito ou recuperado. E somente envia a mensagem se o cliente estiver conectado (*isConectado()*) com a ferramenta gráfica.

Quando a ferramenta gráfica recebe a mensagem, ela processa de acordo

com o seu tipo e desenha no seu histórico a notificação ocorrida.

```
/*Este método será invocado pelo detector de defeitos quando o detector
 * de defeitos começa a suspeitar de algum membro sendo monitorado.
 */
public void notifySuspicion( Member suspected ) {
        if ( isConectado() ){
                sendMembro( SUSPEITO, suspected );
        }
}

/*Este método será invocado pelo detector de defeitos quando o detector
 * de defeitos começa a suspeitar de algum membro sendo monitorado.
 */
public void notifyRecovery( Member recovered ) {
        if ( isConectado() ){
                sendMembro( RECUPERADO, recovered );
        }
}
```

FIGURA 6.2 – Trecho de código do cliente.

6.2 AS INTERFACES DO FDVIEW

A interface tem a função de relacionar o aluno (usuário) com o detector do AFDService. Ela mostra ao usuário os estados dos nós monitorados, dizendo quais estão vivos ou suspeitos naquele momento. Possui também, entradas para o usuário configurar o detector como, por exemplo, *timeout* e intervalo das mensagens. As seções a seguir mostram as telas e suas ações da interface FDView.

6.2.1 Tela Principal

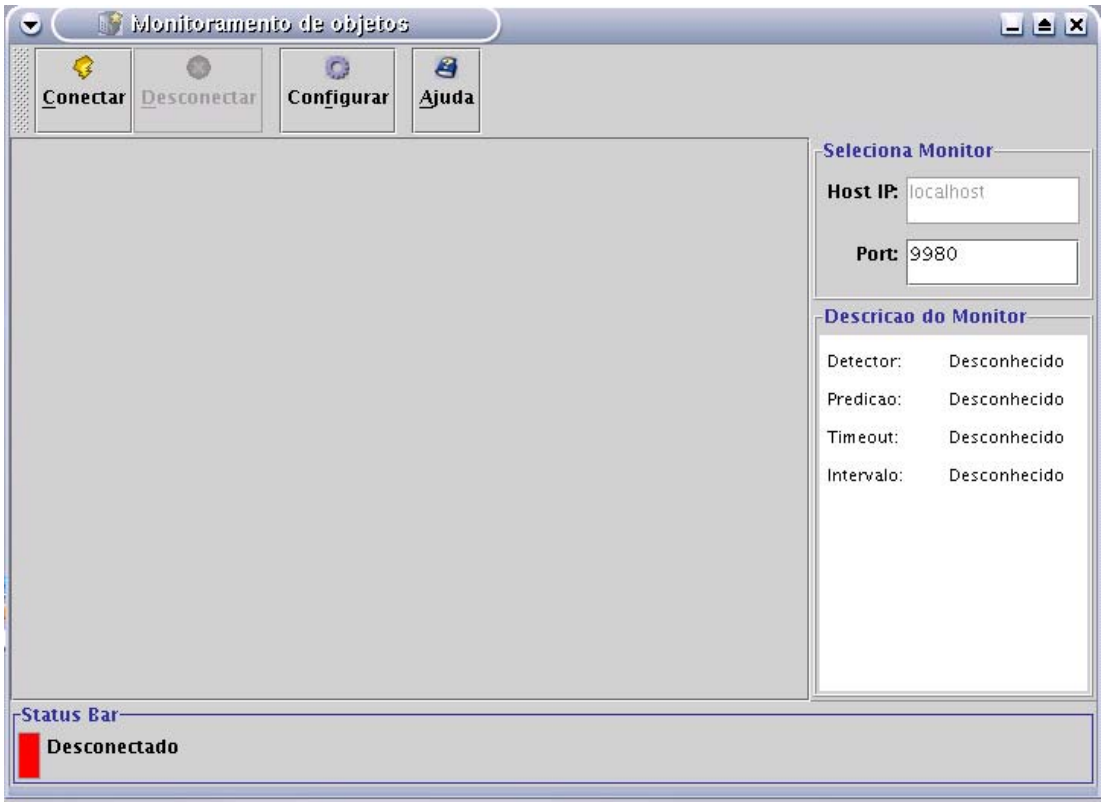

FIGURA 6.3 - Layout do FDView.

O FDView (figura 6.3) tem as seguintes partes:

- *Barra de tarefas*: apresenta as funções do programa, possui uma lista de botões no topo do layout: *Conectar*, *Desconectar*, *Configurar* e *Ajuda*.

- *Seleciona Monitor*: Serve como definição do endereço IP do cliente (aplicação cliente notificável) a ser gerenciado ou visualizado. O usuário deve informar o IP e a porta (o padrão é o IP local na *port* 9980) e clicar no botão *"Conectar"*. A ação relacionada ao botão conecta ao cliente selecionado e atualiza os campos de informação e o histórico.

- *Descrição do Monitor:* apresenta a descrição do monitor, o tipo de detector, se é adaptativo, o *timeout* ou *Tfail*, e o intervalo de mensagem.

- *Histórico dos Hosts:* quando conectado mostra uma lista de *host* com o estado atual (histórico de estados vivo, suspeito ou parado).

- *Status Bar*: mostra se o aplicativo está conectado com o cliente (verde), desconectado (vermelho) e também emite uma mensagem de erro se a conexão entre o monitor falhar.

6.2.2. Ações da Barra de Tarefas

A "*barra de tarefas*" apresenta as seguintes ações:

- *Conectar:* conecta o aplicativo com o cliente;
- *Desconectar: des*conecta o aplicativo do cliente;
- *Configurar:* abre uma janela para o usuário configurar o detector;
- Sobre*:* apresenta uma janela com informações sobre o software: título, autor, data, versão etc.

6.2.3 Tela de Configuração

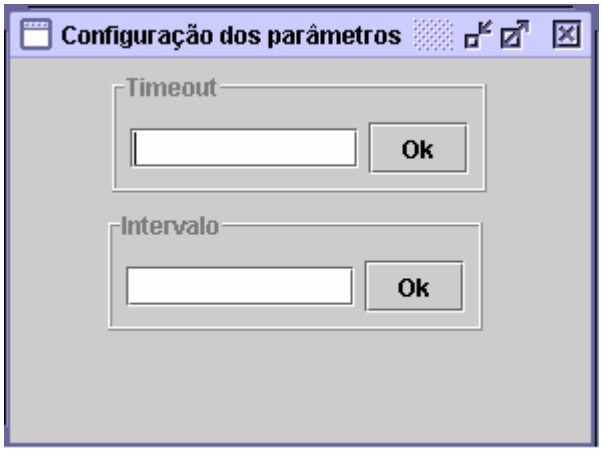

FIGURA 6.4 - Entrada para configurar parâmetros.

A tela de configuração (figura 6.4) permite que o usuário administre o detector

de defeitos. Emite um aviso se o usuário fizer uma ação ilegal.

- *Timeout*: Entrada para o usuário configurar o *timeout* do detector. Se o tipo de detector for o *gossip*, então a entrada serve para configurar o *Tfail*.

- *Intervalo*: Entrada para o usuário configurar o intervalo de envio das mensagens do detector. Do mesmo modo acima, se no caso o tipo de detector for o *gossip*, então a entrada serve para configurar o *Tgossip*.

6.2.4 Histórico de Detecção

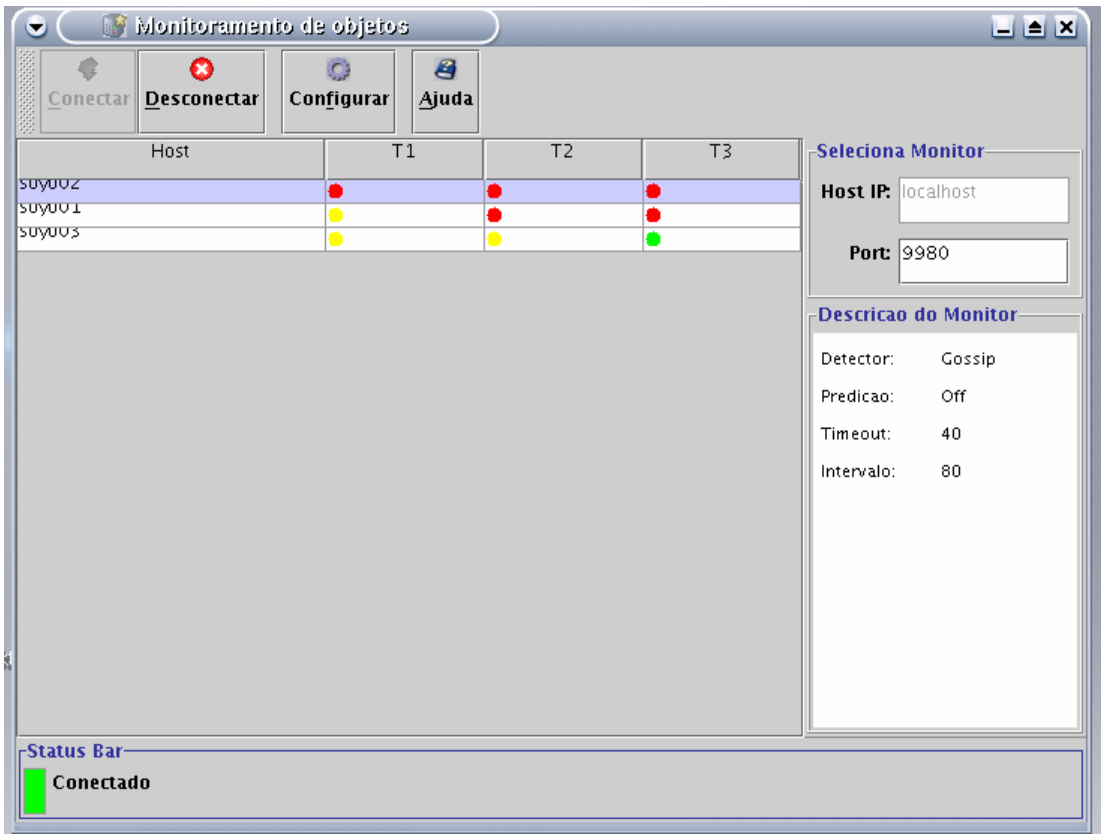

FIGURA 6.5 - Mostra o FDView conectado com o AFDService e o histórico

O principal componente do layout da interface, o histórico, mostra as mudanças dos estados de cada *host* monitorado pelo detector local.

Os históricos dos nós monitorados são mostrados em uma tabela na parte central do layout (figura 6.5). Cada linha da tabela é um *host*. As colunas são os

estados do *host*.

Quando acontece uma notificação do módulo detector, a aplicação avisa a ocorrência ao FDView, e este então adiciona uma nova coluna na tabela, com os estados atualizados de cada *host*.

São três estados diferentes, parado, vivo e suspeito. Cada um é representado pelas respectivas cores amarelo, verde e vermelho. O estado “parado” significa que o nó não está sendo monitorado pelo detector. No estado “vivo”, o nó é considerado vivo pelo detector. E o estado “suspeito”, o nó é suspeito. Inicialmente todos os hosts são suspeitos.

7. CONCLUSÃO

Este trabalho foi dividido em duas partes; o desenvolvimento do algoritmo gossip e da interface gráfica. Na primeira parte o algoritmo de detecção estilo gossip foi adicionado no módulo de detecção AFDService. Na segunda parte a interface gráfica, teve o objetivo ajudar o aluno a entender na prática os protocolos de detecção utilizado no AFDService.

O estilo *gossip* possui paradigmas diferentes de outros estilos de detectores aplicados no AFDService. Enquanto o algoritmo push, pull e dual são baseados em timeouts, o gossip é baseado em probabilidades, pois para escolher do T_{fail} é necessário calcular a probabilidade de uma falsa detecção [RENESSE; MINSKY; HAYDEN ,1998].

Embora o estilo gossip tenha outro paradigma de funcionamento, ele pôde ser facilmente integrado ao AFDService, demonstrando a flexibilidade da arquitetura do serviço.

A arquitetura do AFDService facilita a aplicação do gossip, pois possui interfaces de objetos hierárquicos. Estas interfaces fornecem uma visão genérica de um detector. Por exemplo, a interface Detector oferece um amplo conjunto de métodos necessários para consulta de estados e configuração de parâmetros do detector, independente do tipo de detector. Qualquer objeto de uma classe que implementa o Detector atua como detector de defeitos.

O protocolo *gossip* sofre algumas limitações. Uma limitação é que as mensagens *gossip* podem resultar em uma falsa detecção. Um nó falsamente detecta uma falha em um segundo nó, simplesmente porque não recebeu nenhuma mensagem *gossip* que contém um contador heartbeat recente para o segundo nó.

Uma segunda limitação do gossip é que a banda da rede pode ser desperdiçada. Os nós podem emitir mais de uma mensagem *gossip* ao mesmo destino em um intervalo *Tgossip* enquanto outros nós não receberem nenhuma mensagem.

Por outro lado, o protocolo gossip básico é resiliente às falhas de rede e de *hosts*. Como diferentes mensagens podem ter as mesmas informações, alguma perda de mensagem não exerce influência sobre a detecção de defeitos. O algoritmo gossip tem como propriedade inerente a escalabilidade permitindo monitorar um

grande número de objetos.

O número de mensagens que um algoritmo troca reflete diretamente no tráfico da rede que é uma das principais restrições para a escalabilidade em sistemas distribuídos. Por isto, o mecanismo de disseminação do algoritmo implementado objetiva reduzir o número de mensagens enviadas em um único instante de um processo, de forma a reduzir o número de mensagens para todos processos.

Ao invés de enviar para muitos, o detector envia apenas para o seu vizinho escolhido aleatoriamente. Assim, a largura de banda é pequena em relação ao número de processos, possibilitando a escalabilidade do algoritmo.

A ferramenta gráfica é um acréscimo ao AFDService, porque ela fornece uma visão gráfica das transições dos estados dos objetos monitorados. Também possibilita ao usuário alterar a configuração em tempo de execução. Logo, o usuário pode verificar as conseqüências de sua ação, podendo compreender os conceitos relacionados à detecção de defeitos.

Para o futuro testar o gossip com o protocolo de consenso. A ferramenta pode acrescentar melhorias e fornecer mais recursos ao usuário como, configuração para todos os módulos FD e TS. Além disto, um teste de usabilidade (verificação se a ferramenta é mais intuitiva) e adicionar uma ferramenta para gerência SNMP (*Simple Network Management Protocol*), pode ser realizado para tornar a ferramenta mais didática.

8 BIBLIOGRAFIA

[CHANDRA: TOUEG, 1996] CHANDRA, T. D.; TOUEG, Sam. **Unreliable failure detectors for reliable distributed systems**, ACM Press New York, NY, USA 1996.

[CHEN; TOUEG; AGUILERA, 2002] CHEN, W.; TOUEG, S.; AGUILERA, M. K. **On the quality of Service of Failure Detectors**. IEEE Transactions on Computers, Los Alamitos, v.51, n.5, p.561-580, May 2002.

[BRASILEIRO; FIGUEREDO; SAMPAIO, 2002] BRASILEIRO, F. V.; FIGUEIREDO, J. C. A. De; SAMPAIO, L. M. R. **A Hierarquia Failure Detection Service with Perfect Semantics**. Workshop de Testes e Tolerância a Falhas, WTF, 3.,2002 Buzios. – Brasil. Anais. Rio de Janeiro: UFRJ, 2002. p.25-32.

[BURNS; GEORGE; WALLACE, 1999] BURNS, M., GEORGE, A. AND WALLACE, B. **Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems**. *Cluster Computing*. Vol 2, No 3, 1999.

[FELBER; DÉFAGO; GUERRAOUI 1999] FELBER, Pascal; DÉFAGO, Xavier; GUERRAOUI, Rachid. **Failure Detectors as First Class Objects**. Swiss Federal Institute of Technology, Switzerland 1999.

[ESTEFANEL, 2001] ESTEFANEL, Luiz A. B.; **Avaliação dos Detectores de Defeitos e sua Influência nas operações de Consenso**. Porto Alegre: Instituto de Informática – UFRGS, 2001 (Trabalho Individual de Mestrado).

[GAMMA; HELM; JOHNSON; VLISSIDES, 1994] GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design Patterns, Elements of Reusable Object-Oriented Software**. Massachusetts: Addison Wesley, 1994. 395 p.

[GARTNER, 1999] GARTNER, Felix C. **Fundamentals of fault-tolerant distributed computing in assynchronous environments**. ACM Computing Surveys, 1999.

[JALOTE, 1994] JALOTE, P. **Fault tolerance in distributed systems**. Prentice Hall, Englewood Cliffs, New Jersey, 1994. 432p.

[LARREA; AREVALO; FERNANDEZ, 1999] LARREA, M.; AREVALO, S.; FERNÁNDEZ, A. **Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous System.** International Symposium on Distributed Computing, DISC, 13., 1999, Bratislava, Slovak Rep. Proceedings... Berlin: Springer-Verlag, 1999. p34-48. (Lecture Notes in Computer Science, v. 1693).

[NUNES, 2003] NUNES, R. C; **Adaptação Dinâmica do Timeout de Detectores de Defeitos Através do uso de Séries Temporais**. Porto Alegre: Instituto de Informática – UFRGS, 2003 (Tese de doutorado).

[PETER, 2004] PETER, J. J. **Padrões de Projeto em Java, Reutilizando o Projeto de Software**. Revista Mundo Java, n. 6, ano I, Curitiba - PR, 2004.

[RENESSE; MINSKY; HAYDEN ,1998] RENESSE, Robbert Van; MINSKY, Yaron; HAYDEN, Mark. **A Gossip-Style Failure Detection Service**. Cornell University, Ithaca, 1998.

[SERGENT; DÉFAGO; SCHIPER,1999] SERGENT, N.; DÉFAGO, X; SCHIPER, **A. Failure Detectors: Implementation Issues and Impact on Consensus Performance**. Lausanne: Éscole Polytechnique Fédérale de Lausanne – EPFL, 1999. (Technical Report SSC/1999/019).

[SHOSTAK; BAKER, 1972] SHOSTAK, S. AND BAKER, B. **Gossips and Telephones**. Discrete Mathematics. Vol 2, No 3, 191-193, 1972.

[SUBRAMANIYAN; RAMAN, GEORGE, RADLINSKI, 2005] R. SUBRAMANIYAN, P. RAMAN, A. D. GEORGE, e M RADLINSKI. *GEMS:* **Gossip-Enabled Monitoring**

Service for Scalable Heterogeneous Distributed Systems. High-performance Computing and Simulation (HCS) Research Laboratory Department of Electrical and Computer Engineering, University of Florida P.O. Box 116200, Gainesville, FL 32611-6200, 2005.

[SUN, 2005] SUN Microsystems. **The Source for Java Technology**. Disponível em: <http:\\www.java.sun.com>. Acesso em: out. 2005.

[TAYLOR; GOLDING, 1992] TAYLOR, K. AND GOLDING, R. **Group Membership in the Epidemic Style**. Dept. of Computer Science Rep. UCSC-CRL-92-1. University of California at Santa Cruz, 1992.

[YOGEN; OPPEN, 1983] YOGEN, D. AND OPPEN, D. **The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment**. ACM Transactions on Office Information Systems. Vol 1, No 3, 230-253, 1983.