UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

JEYSONN ISAAC BALBINOT

Projeto de um Serviço Configurável de Detecção de Defeitos

Dissertação apresentada como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação

Profa. Dra. Ingrid Jansch-Pôrto
Orientadora

Porto Alegre, março de 2007.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Balbinot, Jeysonn Isaac

Projeto de um Serviço Configurável de Detecção de Defeitos / Jeysonn Isaac Balbinot – Porto Alegre: Programa de Pós-Graduação em Computação, 2007.

100 f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação. Porto Alegre, BR – RS, 2007. Orientadora: Ingrid Jansch-Pôrto.

1. Introdução. 2. Detectores de Defeitos. 3. Comunicação entre Módulos Detectores de Defeitos. 4. Serviço de Detecção de Defeitos. I. Jansch-Pôrto, Ingrid. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitor: Prof. José Carlos Ferraz Hennemann
Vice-Reitor: Prof. Pedro Cezar Dutra Fonseca
Pró-Reitora de Pós-Graduação: Profa. Valquiria Linck Bassani
Diretor do Instituto de Informática: Prof. Flávio Rech Wagner
Coordenador do PPGC: Profa. Luciana Porcher Nedel
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Aos meus pais,

Edgar e Dirce Balbinot

"*I am not discouraged, because every failure is another step forward.*"
– Thomas Edison

AGRADECIMENTOS

Agradeço, primeiramente, a Deus pela minha existência e por ter me dado condições físicas e espirituais para vencer esta etapa da vida.

Agradeço aos meus pais, Edgar e Dirce Balbinot, que sempre me incentivaram, dando-me apoio moral e financeiro e, principalmente, muito amor. Vocês são exemplos de persistência e força de vontade, sem os quais com certeza, essa etapa não seria possível.

Agradeço à minha professora e orientadora, Dra. Ingrid Jansch-Pôrto, pelo convite para ingressar no mestrado do Instituto de Informática da UFRGS, na área de tolerância a falhas. Agradeço, também, por ter me engajado no Projeto DepGriFE e ao profissionalismo, dedicação e paciência que teve nesse período, principalmente nos momentos em que não atendi às suas expectativas.

Agradeço a Dra. Taisy Weber, também orientadora do projeto DepGriFE, pelo apoio técnico e científico durante o andamento do projeto.

Agradeço à empresa HP/Brasil e ao CNPq, pelo apoio financeiro (bolsas de mestrado) para a execução deste trabalho.

Agradeço ao pessoal do laboratório que, por tantas vezes, juntos trabalhamos, discutimos, nos lamentamos, rimos e tomamos deliciosos cafezinhos.

Agradeço ao Tórgan Flores, Márcio Bystronski, Leonardo Dalpiaz, Augusto Bortolás, Juliano Vacaro, Hélio Miranda, Roberto Drebes, Gabriela Jacques-Silva, Joana Trindade, Luiz Antonio Rodrigues e demais colegas de trabalho, pela convivência, apoio técnico, dicas e sugestões.

Agradeço a todas as amizades cultivadas e àqueles que me incentivaram e acreditaram no meu potencial. Ao pessoal da Turma Mística, incluindo os pais da Edi e da Rô, ao pessoal dos almoços no RU e à Turma do Futebol das Quartas.

Agradeço aos meus colegas da República Paraná, Luciano Soler, Guilherme Galante e ao Luiz Antonio Rodrigues, parceiros de todas as horas.

Em fim, agradeço a todos que, direta ou indiretamente, contribuíram para a realização de mais um sonho.

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS

AFDService	*Adaptable Failure Detector Service*
EP	Erro de previsão
IC	Intervalo de Confiança
IP	*Internet Protocol*
LAN	*Local Area Network*
RTT	*Round Trip Time*
SDD	Serviço de Detecção de Defeitos
WAN	*Wide Area Network*

LISTA DE FIGURAS

LISTA DE TABELAS

RESUMO

A detecção de defeitos pode ser usada como base no projeto de algoritmos e aplicações distribuídas que dependem, de alguma forma, de informações de estado sobre processos distribuídos. O problema de acordo entre processos (consenso), que é um dos problemas fundamentais da computação distribuída, bem como difusão atômica (*atomic broadcast*), eleição de líder (*leader election*) e gerenciamento de grupos (*membership*) necessitam de informações de estado dos processos envolvidos, portanto, do resultado da atividade dos detectores. Esses protocolos, geralmente, são usados como blocos básicos para a construção de outros algoritmos, serviços ou aplicações distribuídas tolerantes a falhas.

Os detectores de defeitos, de forma prática, têm sido desenvolvidos com base em parâmetros funcionais de redes locais e não operam bem no contexto de sistemas distribuídos de larga escala e de redes de longa distância (WANs). Sistemas conectados por WANs, geralmente, oferecem um ambiente mais hostil do que as LANs e *clusters*, devido aos atrasos longos e variáveis e à maior probabilidade de ocorrência de defeitos de temporização (flutuações na latência de comunicação) e omissão (perdas de mensagens), impondo um desafio na concepção de mecanismos que detectem defeitos de forma completa, precisa e que atendam a requisitos de *dependabilidade* exigidos pelas aplicações. A detecção de defeitos, também, pode ser oferecida na forma de um serviço, podendo ser este serviço utilizado por diferentes aplicações, sem que estas necessitem agregar a implementação do detector em seus projetos.

Neste trabalho, foram pesquisadas estratégias aplicáveis à organização e à comunicação entre módulos de detecção de defeitos, focando sistemas de larga escala que operem sobre WANs. Está sendo proposto um modelo de serviço configurável que opera sob demanda das aplicações, e utiliza uma organização hierárquica dos módulos detectores de defeitos. Com base nesse modelo, foi implementado e testado um protótipo, utilizando o *framework* de simulação Neko. Os testes avaliaram a utilização da estratégia hierárquica com base no tipo e número de mensagens trocadas pelo serviço durante sua operação. Os resultados mostraram que adotar a hierarquia em dois níveis (LAN e WAN) resulta em poucas mensagens adicionais de controle e significativa redução do número de mensagens trafegando entre redes locais. O serviço tirou proveito do conhecimento da topologia da rede e escalou bem, quando um número maior de máquinas foi utilizado. Adicionalmente, para ajustar dinamicamente a detecção aos atrasos impostos pelas WANs, foi utilizado o pacote de predição de *timeout* do AFDService.

Palavras-Chave: Tolerância a falhas, serviço de detecção de defeitos, WAN, sistemas distribuídos de larga escala.

Design of a Configurable Failure Detection Service

ABSTRACT

The failure detection may be used as basis for the design of algorithms and distributed applications that need information about the state of distributed processes. The agreement problem among processes (consensus) is one of the fundamental problems in distributed computing as well as other protocols such as atomic broadcast, leader election and membership that also need information about involved processes and consequently need also the results from the failure detector activity. These protocols are generally used as basic blocks to design other algorithms, services or fault-tolerant distributed applications.

The failure detectors, in practice, have been developed based on local network parameters; consequently they are not tuned for the context of large-scale distributed systems nor wide area networks (WANs). Systems interconnected by WANs generally are environments more adverse than LAN and traditional clusters, due to variable and long delays and more prone to timing and omission failures. A natural consequence is that it is challenging to develop mechanisms that can accurately detect failures and give the needed support for dependability requirements of the applications. The failure detection may also be offered as a service for the different applications, which do not need to include their own detectors in their design.

In this work are investigated strategies previously defined and applied on the communication of failure detector modules, focusing the analysis on large scale systems on WANs. From this, we propose a configurable failure detection service model that works on demand of applications and adopts the hierarchical organization of failure detection modules. Based on this model, a prototype implementation has been developed and tested using Neko simulation framework. The tests evaluate the utilization of hierarchical strategy based on the type and number of messages exchanged by the service during its operation. The experiments show that the two-level (LAN and WAN) hierarchical structure adopted results in a few additional control messages and a significant reduction on the message traffic between local networks. The service uses the knowledge of the topology and scales well when many machines are used. Additionally, to dynamically adjust the delay imposed by WANs on time detection, the timeout prediction package of AFDService has been used.

Keywords: Fault tolerance, failure detection service, WAN, large-scale distributed systems.

1 INTRODUÇÃO

Inicialmente, faz-se uma breve introdução ao assunto de detecção de defeitos, contextualizando a importância e a aplicabilidade do mesmo em sistemas distribuídos que necessitem oferecer disponibilidade e confiabilidade. Na seção 1.1, são apresentados o interesse e a importância de sistemas distribuídos que objetivam ser tolerantes a falhas. Na seção 1.2, é apresentado um modelo clássico de defeitos (CRISTIAN, 1991; JALOTE, 1994). Na seção 1.3, são relacionadas algumas questões sobre as características e necessidades de detectores de defeitos. A seção 1.4 apresenta os objetivos e a metodologia empregada neste trabalho para atingi-los. A seção 1.5 lista os principais trabalhos relacionados à detecção de defeitos e, por fim, na seção 1.6, é apresentada a estrutura do texto desta dissertação.

1.1 Sistemas Distribuídos Tolerantes a Falhas

Um sistema distribuído típico consiste de um conjunto de processos que trocam mensagens via um canal de comunicação. Esses processos podem ser corretos ou incorretos. Processos corretos se comportam de acordo com suas especificações, enquanto que processos incorretos também podem entrar em colapso (parar de receber ou enviar mensagens) ou se comportarem de forma maliciosa (enviando mensagens que não seguem a especificação). Um sistema distribuído, tolerante a falhas, possui como principal característica redundância de componentes, sejam estes de *software* ou de *hardware*. No entanto, para que os dados replicados permaneçam consistentes, os protocolos que gerenciam as réplicas necessitam de algum tipo de acordo entre os componentes (GUERRAOUI; SCHIPER, 1997b).

Sistemas distribuídos de larga escala e que operem sob redes de longa distância, conhecidas como WANs, são especialmente interessantes porque podem englobar computadores dos mais diferentes tipos e capacidades, distribuídos em uma grande área geográfica. Esses ambientes possuem características que fazem com que a probabilidade de ocorrência de defeitos[1] se torne mais alta, se comparada com ambientes controlados, como *clusters* e redes locais (LANs). Surge, então, o desafio em projetar sistemas tolerantes a falhas para atender requisitos de *dependabilidade*[2] exigidos pelas aplicações distribuídas que operam sobre estes.

---

[1] No contexto deste trabalho, um defeito corresponde à incapacidade de algum componente (de *software* ou *hardware*) de executar sua função.

[2] *Dependabilidade*, ou confiança no funcionamento, como sugerido por Lemos e Veríssimo (1989), é uma tradução que vem se difundindo para o termo em inglês *dependability*, que foi introduzido por Laprie (1985).

Na prática, uma falha e a conseqüente indisponibilidade de um serviço distribuído podem provocar catástrofes ou gerar grandes prejuízos financeiros. Por isto, o interesse por computação distribuída robusta tem crescido significativamente nos últimos anos, principalmente em aplicações implementadas sobre WANs, das quais a Internet é um exemplo, e na qual os atrasos de comunicação não são previsíveis (BAUER; SICHTIU; PREMARATNE, 2001 *apud* NUNES, 2003)

Corporações, empresas de médio e grande porte e universidades demandam, cada vez, mais sistemas robustos, ou seja, que sejam confiáveis e que ofereçam alta disponibilidade para que suas aplicações sejam corretamente distribuídas e executadas com êxito. Para atender tais exigências, conciliando problemas de distribuição de aplicações e a característica não confiável da comunicação entre os componentes envolvidos, a utilização de técnicas para projetar sistemas distribuídos tolerantes a falhas tem sido bastante explorada.

O fato de alguns componentes (nodos) estarem distribuídos e serem, de certa forma, independentes, pode concorrer para aumentar a tolerância a falhas do sistema como um todo. Dessa forma, os diversos processadores, memórias, canais de comunicação e unidades de disco rígido, encontrados num sistema distribuído, provêem redundância que pode ser aproveitada para fins de tolerância a falhas. Se bem projetado, o sistema pode continuar em funcionamento, mesmo na ocorrência de defeitos em alguns componentes. Entretanto, alcançar tolerância a falhas em sistemas distribuídos não é um problema simples e requer complexas técnicas de tolerância a falhas – detecção e tratamento de defeitos (SERGENT; DÉFAGO; SCHIPER, 1999). O escopo deste trabalho trata especificamente da detecção de defeitos, dada a sua notória importância na implementação de técnicas de tolerância a falhas.

1.2 Modelo de Defeitos

Um defeito de sistema ocorre, quando um serviço, que possui uma especificação acordada de funcionamento, opera fora dessa especificação (LAPRIE, 1985). Desta forma, o termo **defeito** denota o fato de que um sistema não se comporta de acordo com sua especificação (GARTNER, 1999). Um detector de defeitos pode ser visto como um módulo local capaz de oferecer informações sobre suspeita de defeitos em outras máquinas ou processos, e, normalmente, é implementado como um algoritmo distribuído, baseado em troca de mensagens.

A forma como se apresentam, ou como são percebidos, os defeitos de um sistema é conhecida como modelo de defeitos. Os modelos de defeitos de processos foram classificados e descritos por Cristian (1991):

- defeito de **omissão**: ocorre quando um componente não responde a algumas entradas ou requisições, possuindo um comportamento intermitente ou transitório;
- defeito de **temporização**: ocorre quando a resposta do componente é funcionalmente correta, mas é dada fora do intervalo de tempo especificado. Esse tipo de defeito pode tanto se manifestar na forma de uma resposta antecipada quanto através de uma resposta tardia (defeito de desempenho);
- defeito de **resposta**: ocorre quando o componente responde, mas de forma incorreta, através de um valor de saída incorreto (representando um defeito de

valor) ou de uma transição de estado que acontece incorretamente (defeito de transição de estado);

- defeito de **colapso** (*crash*): ocorre quando, após a primeira omissão de resposta, o sistema deixa de produzir respostas para as entradas subseqüentes até que seja reinicializado. Um exemplo desse tipo de defeito ocorre quando a máquina em que um processo está executando fica travada, bloqueando a execução de seus códigos;
- defeito **arbitrário**: ocorre quando qualquer comportamento é permitido para o componente. Essa classe inclui as demais classes descritas anteriormente, e é também conhecida pela denominação de defeitos bizantinos (JALOTE, 1994; BIRMAN, 1996).

Com base na possibilidade de reinicialização de um componente que entre em colapso, Cristian (1991) define quatro categorias para a classe de defeitos deste tipo: colapso por amnésia (*amnesia-crash*), colapso por amnésia parcial (*parcial-amnesia-crash*), colapso por pausa (*pause-crash*) e colapso por parada (*halting-crash*). Porém, neste trabalho, da mesma forma como nos modelos apresentados por Jalote (1994) e Birman (1996), é considerado apenas um tipo de defeito de colapso – colapso por parada, chamado somente colapso (*crash*).

Similar ao modelo de defeitos para processo apresentado por Cristian (1991), Jalote (1994) apresenta um modelo de defeitos para o canal de comunicação. No modelo de Jalote (1994) um defeito de **omissão** é caracterizado pela perda de mensagens; defeito de **temporização** pelo atraso na entrega de mensagens; defeito de **resposta** pelo corrompimento (alteração) de mensagens no canal de comunicação; defeito de **colapso** por nenhuma atividade (transmissão ou recebimento); e defeito **arbitrário** por apresentar qualquer tipo de comportamento anteriormente citado.

Jalote (1994), também, define uma hierarquia sobre as classes de defeitos, como ilustra a Figura 1.1. Nela, uma classe mais geral tem a capacidade de detectar defeitos de suas subclasses. Os defeitos de resposta, contidos na classe de defeitos arbitrários, não mantêm relação com as demais classes, ou seja, ela é ortogonal às demais.

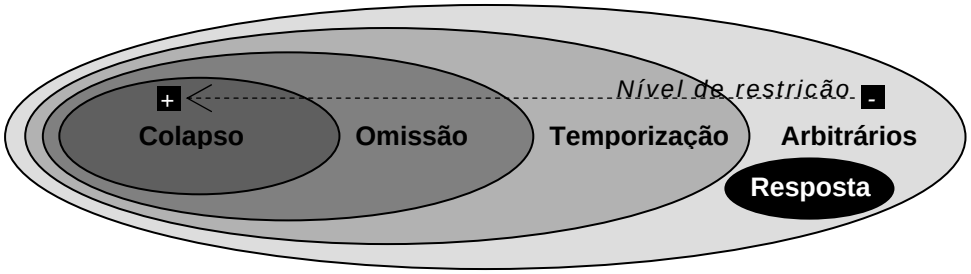

Figura 1.1: Classes de defeitos

A hierarquia dispõe as classes de defeitos de acordo com o nível de restrição das suas semânticas quanto aos impactos que afetam o funcionamento do sistema. Dessa forma, como ilustra a seta tracejada da Figura 1.1, na extremidade mais restritiva, localiza-se a classe de defeitos por colapso e, na extremidade oposta, menos restritiva, a classe de defeitos arbitrários.

1.3 Detectores de Defeitos

Em sistemas distribuídos, muitos problemas, para serem solucionados, necessitam da participação de processos para a coordenação de decisões. O acordo entre processos remotos é um dos problemas fundamentais da computação distribuída (FISCHER; LYNCH; PATERSON, 1985). O consenso pode ser visto como uma forma geral de resolver *problemas de acordo* em sistemas distribuídos. No problema do consenso, um conjunto de processos ou agentes, não necessariamente remotos, deve unanimemente concordar em uma decisão obtida a partir de seus estados iniciais. Tipicamente, essa decisão é feita apenas sobre dois valores, 0 e 1. No entanto, isso não impede que o protocolo seja estendido para decisões sobre valores mais complexos, desde que sejam mantidas as propriedades do consenso (TUREK; SHASHA, 1994 *apud* ESTEFANEL, 2001).

Utilizar consenso é bastante comum em algoritmos de processamento de dados, gerenciamento de arquivos distribuídos e aplicações distribuídas tolerantes a falhas. Todavia, em sistemas puramente assíncronos, e na presença de defeitos, o consenso não tem solução determinística. A conclusão da operação do consenso não pode ser garantida devido à impossibilidade de distinguir processos falhos de processos muito lentos (FISCHER; LYNCH; PATERSON, 1985). Essa restrição é conhecida como Impossibilidade FLP (em homenagem aos seus criadores) e afeta, além do consenso, outros protocolos baseados neste protocolo e que possuem operações tão ou mais complexas, como difusão atômica (GUERRAOUI; SCHIPER, 1997a) e eleição (SABEL; MARZULLO, 1995).

Para resolver o problema da Impossibilidade FLP, Chandra e Toueg (1996) estenderam o modelo de sistema assíncrono apresentado por Fischer, Lynch e Paterson, introduzindo os detectores de defeitos não confiáveis.

Um sistema de detecção de defeitos é constituído por módulos locais em cada máquina. Estes módulos (detectores) podem monitorar um grupo ou subgrupo de processos do sistema, ou mesmo serem monitorados por outros detectores. Cada detector mantém uma lista de suspeitos que está monitorando; o gerenciamento desta lista e a comunicação são feitos via troca de mensagens. Os detectores de defeitos não confiáveis são assim definidos porque podem suspeitar erroneamente de um ou mais processos. Embora esses detectores não possam determinar com exatidão o estado real dos processos, utilizá-los aumenta o conhecimento sobre os demais elementos, auxiliando na finalização das operações de consenso (CHANDRA; TOUEG, 1996).

Chandra e Toueg definiram os detectores através de duas propriedades: abrangência (*completeness*) e precisão (*accuracy*). A abrangência refere-se à capacidade do detector identificar todos os processos que estão falhos, enquanto que a precisão determina a capacidade do detector de evitar erros sobre as suspeitas, a fim de evitar a inclusão de processos corretos nas listas de suspeitos. Variando os níveis de exigência para cada propriedade, obtêm-se versões (classes) de detectores de defeitos.

Felber, Guerraoui e Fayad (1999) oferecem uma modelagem para detectores de defeitos e citam três principais abstrações para os objetos envolvidos em um sistema de monitoração:

- **monitores** – (ou módulos detectores de defeitos) são objetos que coletam informações sobre defeitos dos componentes monitorados;

- **monitoráveis** – são os objetos que podem ser monitorados pelos detectores; estes detectores detectam possíveis defeitos nos objetos monitoráveis;
- **notificáveis** – são objetos que podem ser registrados pelo serviço de monitoração e, posteriormente, notificados, síncrona ou assincronamente, sobre defeitos em objetos monitorados.

A implementação de protocolos de detecção de defeitos, geralmente, segue os modelos básicos *push*, *pull* ou *dual*, que se baseiam no fluxo de mensagens trocadas (FELBER *et al.*, 1999). Esses modelos, ou formas de detecção, são vistos como um serviço independente da aplicação. O serviço serve como suporte para as aplicações ou para outros mecanismos, de modo que estes tomem as devidas providências na presença de defeitos.

Geralmente, o tratamento dos defeitos fica a cargo da própria aplicação (MEDEIROS *et al.*, 2003), que pode responder a um defeito de um componente monitorado de diferentes maneiras (STELLING *et al.*, 1998):

- encerrar a aplicação (*fail-stop*);
- ignorar o defeito e continuar a aplicação;
- alocar um novo recurso e reiniciar o componente da aplicação que apresentou defeito;
- usar replicação e comunicação em grupo para continuar a execução.

Cada um desses comportamentos tem seus custos e benefícios associados, e adotar o mais adequado é dependente da própria aplicação.

Ainda com relação à implementação, é possível encapsular o detector de defeitos na forma de um serviço[3] e este é geralmente oferecido na camada *middleware* (VAN RENESSE; MINSKY; HAYDEN, 1998). Este encapsulamento permite que o detector possa ser utilizado concorrentemente por várias aplicações, sem que elas precisem se preocupar com questões de implementação.

Segundo Hayashibara, Cherif e Katayama (2002), implementações de serviços de detecção de defeitos têm sido propostas para uso em LANs e não atendem eficientemente WANs. As WANs, podem ter muitos componentes, alto nível de assincronismo, longos atrasos de mensagens, alta taxa de perda e topologia dinamicamente mutável, exigindo ajustes ou redefinições dos detectores para melhor atender tais características. Segundo Chen, Toueg e Aguilera (2002), os detectores têm sido utilizados amplamente em ajustes de protocolos de comunicação de redes, comunicação de grupo e gerenciamento de *clusters*.

Quando detectores são projetados para executar em sistemas de larga escala, por exemplo, no caso específico de ambientes do tipo *grid*, eles se deparam com problemas de implementação, como escalabilidade, perda e atraso de mensagens, flexibilidade, dinamismo, explosão do número de mensagens e segurança. Essas questões são motivos de estudo e tratamento para que o sistema de detecção de defeitos possa atingir seus objetivos (HAYASHIBARA; CHERIF; KATAYAMA, 2002).

---

[3] Um serviço é um conjunto de primitivas (operações) e/ou funcionalidades que uma camada oferece para a camada acima. Diz respeito à interface existente entre duas camadas que, por sua vez, tem como provedor a camada inferior e como usuário a camada superior.

1.4 Objetivos e Metodologia

O objetivo deste trabalho foi propor e estender um serviço de detecção de defeitos, tendo por inspiração básica o Serviço de Detecção de Defeitos Adaptativo ou AFDService de Nunes (2003). Porém, além de ser configurável, o serviço proposto teve como enfoque atender aos requisitos necessários para operar tanto em redes locais quanto em redes de longa distância, além de enfatizar soluções para atender a sistemas (aplicações) que possam ser de larga escala.

Considerando os aspectos que haviam sido explorados no AFDService, a ênfase do presente trabalho esteve na investigação, a partir da literatura disponível, do funcionamento de estratégias que podem ser empregadas ou adaptadas para a comunicação entre módulos detectores e como elas poderiam eventualmente beneficiar um serviço de detecção de defeitos que opere em plataformas distribuídas de larga escala.

Os procedimentos de detecção a serem implementados não devem competir por recursos (processamento e memória) de maneira significativa com as aplicações, nem sobrecarregar demasiadamente a rede, ocasionando tráfego de controle demasiado ou desnecessário. Também devem ser flexíveis, de forma que as soluções desenvolvidas tenham condições de serem estendidas e combinadas com novos ambientes e aplicações.

A metodologia consistiu em analisar a difusão de informações entre módulos detectores, variando os mecanismos selecionados (determinísticos e/ou probabilísticos); identificar quais mecanismos e estratégias podem ser usados em redes de longa distância e sistemas de larga escala; com base nesses estudos, modelar e prototipar o serviço configurável de detecção de defeitos; e, por fim, executar testes e avaliar a eficiência do protótipo implementado.

1.5 Trabalhos Relacionados

Na literatura, encontram-se trabalhos relacionados a detectores de defeitos, implementações, estratégias e protocolos. A seguir, em ordem cronológica, alguns desses trabalhos são relacionados.

- O detector *Heartbeat*, proposto por Aguilera, Chen e Toueg, (1997), é um algoritmo de detecção de defeitos aplicável em sistemas distribuídos assíncronos com possibilidade de defeitos de colapso e omissão (perda de mensagens). Foi proposto como uma solução para a impossibilidade de comunicação confiável com algoritmos quiescentes (i.e., algoritmos que em um tempo finito (*eventually*) porém, desconhecido, cessarão o envio de mensagens) nesses ambientes. Apesar de ser um algoritmo de detecção, para fins deste trabalho, o *Heartbeat* tem seu funcionamento básico descrito e funcionamento aplicável como alternativa para comunicação entre módulos detectores.
- O *Globus Heartbeat Monitor* ou HBM (STELLING *et al*., 1998) é um serviço de detecção de defeitos projetado para ser usado em sistemas de computação distribuída, ferramentas ou aplicações. Para monitorar processadores e processos o serviço opera usando uma abordagem hierárquica em dois níveis. O primeiro nível consiste de monitores locais (*local monitors*) responsáveis por monitorar processos do mesmo *host* e enviar periodicamente mensagens para os coletores de dados (*data collectors*). No segundo nível, os coletores de dados identificam defeitos nos componentes e notificam as aplicações. Porém, o serviço é

específico para aplicações desenvolvidas com o Globus Grid *toolkit* e não será abordado neste trabalho.

- Protocolos de detecção de defeitos estilo *gossip*, também conhecidos como protocolos epidêmicos ou mesmo probabilísticos, são desenvolvidos para atender a sistemas distribuídos de larga escala cujo foco é a disseminação de informações sobre detecção de defeitos. Van Renesse, Minsky e Hayden (1998) propuseram o protocolo de detecção de defeitos estilo *gossip* básico e uma variação deste, chamado de multi-nível. O multi-nível usa como estratégia informações da topologia para a disseminação de informações, para diminuir o fluxo de mensagens entre sub-redes e domínios diferentes e para tornar o serviço escalável. Neste trabalho, os dois protocolos são descritos (seção 3.4) como estratégias para a disseminação de informações entre módulos detectores de defeitos.
- Felber (1998) descreveu um serviço na forma de um *framework* para a detecção de defeitos em CORBA (*Common Object Request Broker Architecture*). Para atender a questões de escalabilidade de aplicações com um grande número de participantes e também sobrepor problemas que ocorrem devido à comunicação em longas distâncias, o serviço utiliza uma organização hierárquica. Dessa forma, em cada rede local, dois ou três detectores são responsáveis pela monitoração de todas as entidades pertencentes à rede local. O serviço é acessível aos desenvolvedores através de objetos e interfaces, considerando a monitoração de objetos e não diretamente de processos ou máquinas. Por ser específico ao CORBA e ser orientado à monitoração de objetos, este serviço não é detalhado neste trabalho.
- Larrea, Arévalo e Fernández (1999) propuseram algoritmos de detecção de defeitos para sistemas distribuídos parcialmente síncronos, com base na organização dos processos em anel lógico, a qual define a monitoração e o padrão de propagação de informações sobre defeitos. Em oposição a detectores tradicionais (que utilizam comunicação *todos-para-todos*), onde a troca de mensagens cresce de forma quadrática com o número de detectores, a alternativa de organização em anel de Larrea, Arévalo e Fernandez mostrou que os algoritmos resultaram em uma troca periódica linear de mensagens.
- Burns, George e Wallace (1999) analisaram e compararam três protocolos estilo *gossip* para um serviço de detecção de defeitos em *clusters* heterogêneos de larga escala. Os dois primeiros protocolos são aqueles definidos por Van Renesse, Minsky e Hyden (1998); o terceiro, chamado de *gossip piggyback*, é uma proposta alternativa que adiciona informações de controle e monitoração do protocolo nas mensagens da aplicação, visando à redução do número de mensagens que trafegam pela rede de comunicação.
- Também com a idéia de utilizar as mensagens da aplicação, Fetzer, Raynal e Tronel (2001) propuseram o protocolo de detecção de defeitos *Lazy*. Esse protocolo utiliza as próprias mensagens da aplicação para obter informações de atividade dos componentes monitorados em um sistema distribuído puramente assíncrono.

- Chen, Toueg e Aguilera (2002), além de definirem métricas para a avaliação da Qualidade de Serviço (QoS) de detectores de defeitos, propuseram um algoritmo no qual a aplicação dinamicamente pode configurar os níveis de QoS.
- Bertier, Marin e Sens (2003) avaliaram uma implementação de um detector de defeitos hierárquico, que é uma variação do *Heartbeat*. Este foi adaptado e projetado para tratar questões de escalabilidade e adaptabilidade. Uma camada adapta a qualidade de serviço de acordo com as necessidades da aplicação, e outra oferece uma organização hierárquica para diminuir o fluxo de mensagens e a carga do processador. A implementação do detector faz parte do projeto DARX (*Dynamic Agent Replication eXtension*) que tem por objetivo prover um *framework* para o projeto de sistemas de agentes de larga escala.
- O Serviço de Detecção de Defeitos Adaptativo ou AFDService (NUNES, 2003) é um serviço que monitora nodos de um sistema distribuído. Seu objetivo específico é adaptar *timeouts* dinamicamente de acordo com o comportamento dos atrasos de comunicação. A arquitetura deste serviço é utilizada como base para a proposição da arquitetura do serviço proposto neste trabalho.
- O detector *Accrual* (HAYASHIBARA; DÉFAGO; KATAYAMA, 2003) provê flexibilidade e adaptabilidade, associando uma probabilidade a cada processo monitorado, indicando seu nível de confiança. Difere dos detectores tradicionais que apenas possuem informações binárias sobre estado dos processos monitorados (suspeito ou correto) e delega para a camada de aplicação a ação de interpretação dos dados monitorados.
- Jain e Shyamasundar (2004) propuseram um protocolo de detecção escalável para ambientes de *grids* com o diferencial de ser auxiliado por um serviço de gerenciamento de grupo (*membership*). Esse serviço procura manter balanceado o número de membros que formam a hierarquia e os grupos de detecção.
- Falai e Bondavalli (2005), semelhante ao trabalho de Nunes (2003), descreveram o desempenho de experimentos em WANs para avaliar e comparar a qualidade de serviço (QoS) de um detector de defeitos adaptável estilo *push*, composto de um preditor e uma margem de segurança, onde são variadas as alternativas de configuração para esses parâmetros.

Algumas dessas propostas (e implementações) de protocolos de detecção visam a atender os sistemas distribuídos, alterando estratégias já empregadas em redes locais, mas com algumas modificações que visam a contemplar o dinamismo e a flexibilidade dos sistemas de larga escala que executam em redes de longa distância.

1.6 Organização do Texto

O texto desta dissertação, nos próximos capítulos, tem a estrutura descrita a seguir.

No capítulo 2, são apresentados os conceitos básicos sobre os detectores de defeitos, suas propriedades axiomáticas de funcionamento e a importância destas propriedades para uma correta expressão do mecanismo de suspeitas. Os detectores de defeitos também são descritos segundo os modelos básicos de implementação que os classificam com base no fluxo de mensagens trocadas entre os módulos. Para finalizar o capítulo, são descritas as principais métricas de avaliação quanto à qualidade de serviço oferecido pelos mecanismos de detecção de defeitos.

No capítulo 3, são apresentados o funcionamento e as características peculiares das principais estratégias de comunicação e detecção (determinísticas e probabilísticas) encontradas na literatura, visando à aplicação dessas estratégias na comunicação entre módulos detectores de defeitos, bem como uma análise da complexidade das estratégias que podem ser utilizadas para a construção de uma visão global das entidades monitoradas.

No capítulo 4, é especificada a detecção como um serviço e uma breve análise de sua aplicabilidade. São relacionados os desafios que surgem quando se deve atender a redes de longa distância e a sistemas de larga escala. É apresentada uma tabela com as estratégias estudadas no capítulo 3, analisando como essas estratégias podem, ou não, atender alguns requisitos. Também são descritos os dois principais modos de operação de um serviço de detecção de defeitos. A arquitetura do AFDService (NUNES, 2003) é descrita e, com base nesta, é elaborada uma nova proposta de arquitetura para o serviço de detecção defeitos, com a descrição do funcionamento e interação dos módulos que o compõem.

No capítulo 5, relata-se o processo de implementação do protótipo desenvolvido neste trabalho. É descrito o mapeamento do modelo do serviço de detecção, definido no capítulo 4, para camadas implementadas sobre o simulador de sistemas distribuídos chamado Neko (URBÁN; DÉFAGO; SCHIPER, 2001). Também são descritos alguns detalhes da implementação e do funcionamento dos algoritmos envolvidos no protótipo do serviço.

No capítulo 6, são definidos, executados e avaliados testes com o protótipo desenvolvido. Além disso, as estratégias escolhidas para o modelo e o protótipo são avaliadas frente aos desafios que surgem quando o serviço de detecção opera sobre redes de longa distância e monitora um grande número de máquinas.

Finalmente, é feita a conclusão do trabalho e são apresentadas direções para trabalhos futuros.

2 DETECTORES DE DEFEITOS

Este capítulo apresenta os conceitos básicos relacionados a detectores de defeitos. Inicialmente, na seção 2.1, é apresentado o modelo do sistema distribuído, utilizado neste trabalho. Em seguida, na seção 2.2, é apresentada uma definição para um detector de defeitos distribuído. As propriedades abstratas sobre os detectores de defeitos, abrangência (*completeness*) e precisão (*acurracy*), e suas combinações são descritas na seção 2.3. A seção 2.4 reproduz a classificação dos detectores com base no fluxo de mensagens trocadas entre monitores e monitorados. Na seção 2.5, é mostrado um conjunto de métricas utilizadas para avaliação da qualidade de serviço, específica para detectores de defeitos e, por fim, a seção 2.6 encerra o capítulo com as conclusões parciais.

2.1 Modelo do Sistema

Algoritmos distribuídos podem ser projetados supondo-se diferentes comportamentos e modelos do sistema. Os modelos existentes variam, dependendo de restrições temporais, e são vinculados a determinados aspectos do ambiente, como, tempo de transferência de mensagens ou computação de ações locais.

Quando os limites temporais existem e são conhecidos, pode-se estimar o tempo que levará a execução dos protocolos do sistema e das funções da própria aplicação distribuída. Este modelo de sistema distribuído é denominado **síncrono** e é eficiente para tolerância a falhas por permitir a detecção de defeitos simplesmente usando *timeouts* (GORENDER; MACEDO, 2002).

Segundo Chandra e Toueg (1996), é considerado um sistema distribuído **assíncrono** (ou *time free*) aquele em que não existem restrições temporais. Um sistema dito assíncrono não possui limitações quanto ao atraso de mensagens, variações do relógio interno das máquinas e tempo necessário para executar uma tarefa.

Porém, em sistemas puramente assíncronos e sujeitos a defeitos, o acordo distribuído consistente é impossível de ser alcançado (FISCHER; LYNCH; PATERSON, 1985). Para contornar essa impossibilidade, como alternativa para a resolução do acordo, o sistema considerado neste trabalho, é um **sistema com sincronismo parcial fraco** (CHANDRA; TOUEG, 1996 *apud* NUNES, 2003). Nele, para todas as execuções, existem limites finitos, embora desconhecidos, referentes à velocidade relativa dos processos e ao atraso máximo para que uma mensagem seja entregue. Adicionalmente, esses limites são válidos após um instante de tempo finito, também desconhecido, chamado de instante de estabilização global (*Global Stabilization Time* - GST) (DWORK; LYNCH; STOCKMEYER, 1998 *apud* NUNES, 2003).

Desta forma, o sistema é composto por um conjunto finito de $n > 1$ processos, $\Pi = \{p_1, p_2, ..., p_n\}$, e cada par é conectado por um canal de comunicação não confiável, sujeito à perda e a atraso de mensagens. Esses processos se comunicam por troca de mensagens e possuem um relógio físico local, incrementado monotonicamente. Os relógios locais não necessitam estar sincronizados e não existe conhecimento sobre suas possíveis diferenças (*drift*).

2.2 Definição de Detectores de Defeitos

Um **detector de defeitos** é um mecanismo que utiliza um algoritmo baseado em troca de mensagens para monitorar o estado de um processo. O detector considera o processo suspeito quando não recebe mensagens do mesmo dentro de um período de tempo pré-estabelecido (*timeout*). Sendo assim, um detector de defeitos pode ser visto como um oráculo que permite encapsular o indeterminismo do atraso de comunicação, provendo informações sobre defeitos em componentes.

A definição clássica de **detectores de defeitos distribuídos** é dada por Chandra e Toueg (1996): “Cada processo tem acesso a um módulo detector de defeitos local. Cada módulo monitora um subconjunto de processos do sistema, e mantém uma lista onde são relacionados os processos suspeitos de falha. Os módulos detectores são ditos não confiáveis, por isso, podem suspeitar erroneamente de um processo e, desta forma, um processo *p* pode ser adicionado à lista de suspeitos mesmo funcionando. Se, em outro instante, o módulo concluir que a suspeita sobre *p* foi um engano, ele pode ser removido da lista de suspeitos. Além do mais, em qualquer instante, dois módulos detectores locais podem ter percepções diferentes sobre determinados processos, isso porque o sistema considerado é parcialmente síncrono”.

Uma exigência do modelo é que os enganos cometidos por um detector não impeçam os processos corretos de se comportarem de acordo com suas especificações, mesmo sendo erroneamente considerados suspeitos por todos os demais processos (CHANDRA; TOUEG, 1996). Em termos práticos, para tentar manter a consistência entre as listas de suspeitos dos diversos módulos, as implementações normalmente definem que, periodicamente, a lista de um processo é enviada aos demais, auxiliando na formação de uma lista unificada (ESTEFANEL, 2001).

2.3 Propriedades de um Detector de Defeitos

Como já mencionado na Introdução, Chandra e Toueg definem e classificam os detectores em termos de propriedades abstratas, ao invés de vinculá-los a uma implementação específica em *hardware* ou *software*. Isto permite desenvolver aplicações e provar suas correções com base exclusivamente nestas propriedades, sem precisar referir-se a parâmetros de baixo nível de rede (como a duração exata de *timeouts*, topologia da rede, o atraso de mensagens e a precisão de relógios locais). Esta abordagem permite descrever aplicações e provar suas propriedades através de uma abordagem modular (CHANDRA; TOUEG, 1996).

As duas classes de propriedades axiomáticas definidas para os detectores são: a abrangência (*completeness*) e a precisão (*accuracy*). Informalmente, a abrangência exige que um detector, em um tempo finito, suspeite de cada processo que realmente tenha falhado por colapso, enquanto que a precisão restringe os erros que o detector pode cometer sobre as suspeitas.

A abrangência pode ser facilmente atingida com o uso de *timeouts*. Mas, diferentemente da abrangência, a precisão é difícil de ser implementada, e somente é alcançada em sistemas parcialmente síncronos (NUNES, 2003).

Ao respeitar essas duas propriedades, um detector garante que os algoritmos que o utilizem não perderão a consistência nem ficarão bloqueados indefinidamente. Assim, as propriedades *safety* (segurança, no sentido de manter a consistência das operações e/ou evitar catástrofes) e *liveness* (a capacidade das operações progredirem), também são respeitadas (ESTEFANEL, 2001).

Chandra e Toueg apresentam dois níveis de abrangência e quatro níveis de precisão, obtidos pela variação das exigências de cada propriedade sobre os elementos monitorados. Essas variações têm por objetivo aproximar as características exigidas às possibilidades de um sistema real (CHANDRA; TOUEG, 1996 *apud* ESTEFANEL, 2001). Para serem úteis, as propriedades de abrangência devem ser combinadas com as de precisão, gerando oito tipos de variações conforme mostradas e nomeadas na Tabela 2.1 e descritas na seqüência.

Tabela 2.1: Classes de detectores de defeitos

Completeness	***Accuracy***			
	Strong	*Weak*	*Eventual Strong*	*Eventual Weak*
Strong	*Perfect* $\mathcal{P}$	*Strong* S	*Eventually Perfect* $\Diamond\mathcal{P}$	*Eventually Strong* $\Diamond S$
Weak	Q	*Weak* $\mathcal{W}$	$\Diamond Q$	*Eventually Weak* $\Diamond\mathcal{W}$

Fonte: CHANDRA; TOUEG, 1996. p. 234.

- Abrangência forte (*Strong Completeness*): em um tempo finito (*eventually*) todos os processos que falharam serão permanentemente suspeitos por todos os processos corretos.
- Abrangência fraca (*Weak Completeness*): em um tempo finito (*eventually*) todos os processos que falharam serão permanentemente suspeitos por alguns processos corretos.
- Precisão forte (*Strong Accuracy*): nenhum processo é considerado suspeito antes de ter falhado.
- Precisão forte em um tempo finito (*Eventual Strong Accuracy*): existe um instante de tempo após o qual os processos corretos não são considerados suspeitos por nenhum processo correto.
- Precisão fraca em um tempo finito (*Eventual Weak Accuracy*): existe um instante de tempo após o qual algum processo correto não é considerado suspeito por nenhum processo correto.
- Precisão fraca (*Weak Accuracy*): existe pelo menos um processo correto que jamais se torna suspeito de falha.

A classe *P* deve atender as propriedades de abrangência forte e precisão forte e é a classe com semântica mais forte de todas. Detectores dessa classe devem ser capazes de suspeitar apenas dos processos que realmente falharam e, todos os seus módulos devem suspeitar de um processo falho, dentro de limites de tempo.

Já a classe ◊*W* deve atender as propriedades de abrangência fraca e precisão fraca em um tempo finito, sendo considerada a classe com semântica mais fraca de todas. Detectores dessa classe devem garantir que, após um tempo, pelo menos um processo não é suspeito antes que falhe, além disso, se um processo falhar, ele será suspeito por pelo menos um módulo do detector.

A classe ◊*W* é a de semântica mais fraca que permite a solução do consenso de forma determinística em ambientes assíncronos, conforme demonstrado por Chandra, Hadzilacos e Toueg (1996). Por este motivo, a maioria das implementações de detectores de defeitos existentes é dessa classe.

2.4 Classificação dos Detectores de Defeitos Quanto ao Fluxo de Mensagens

Na prática, para cumprir sua função, os módulos locais de detecção de defeitos devem trocar mensagens de controle, também conhecidas como mensagens de *heartbeat* ou monitoração. Devido à simplicidade, os tempos de espera limitados (*timeouts*) são amplamente utilizados nas implementações de detectores, apesar de não poderem indicar precisamente defeitos em ambientes assíncronos. Outro motivo para utilizar *timeouts* é que em situações práticas não é necessário esperar indefinidamente pela mensagem de um processo para ter certeza de que este processo esteja defeituoso, basta apenas esperar por um tempo suficientemente longo (SAMPAIO; BRITO; OLIVEIRA, 2003).

De acordo com os *timeouts* e com o fluxo das mensagens entre monitor e monitorado, o trabalho de Felber *et al.* (1999) classificou os detectores baseado nas características básicas de dois modelos, *push* e *pull*. Na literatura, pode-se ainda encontrar a nomenclatura de protocolos *heartbeat* e *ping*, para os modelos *push* e *pull*, respectivamente (DAHLGREN, 2004; HAYASHIBARA, 2004). Além destes, é apresentado um modelo misto chamado *dual* que agrega características dos dois modelos unidirecionais. Não existe um modelo que seja mais eficiente e preciso que o outro, pois cada um se adapta melhor a certos tipos de ambientes e aplicações.

2.4.1 Modelo *push*

Neste modelo, a direção do fluxo de controle é a mesma do fluxo de informações. Os processos monitorados são ativos e o monitor é passivo. Cada componente monitorado envia periodicamente mensagens de *heartbeat* do tipo I_AM_ALIVE para o detector que o está monitorando, com a intenção de informar que está “vivo” (funcionando). O monitor recebe as mensagens e, quando dentro do tempo estimado (*timeout*) não recebe uma mensagem esperada, surge então a suspeita de defeito no componente e uma situação de colapso (*crash*) é indicada (o componente é adicionado à lista de suspeitos).

Como vantagem, o modelo *push* possibilita otimizar as mensagens transmitidas devido tanto ao uso de mensagens não-bloqueantes (*one-way*) quanto ao uso de facilidades de *multicast*. Por outro lado, este modelo pode inundar a rede de comunicação com mensagens. Ele é mais indicado para sistemas que necessitem

rapidamente ser informados da suspeita da ocorrência de defeito (FELBER, 1998 *apud* ESTEFANEL, 2001).

Adaptada de Felber *et al.* (1999), a Figura 2.1 ilustra como o modelo *push* é usado para monitorar os objetos. Percebe-se que as mensagens trocadas entre o monitor e o cliente (DOWN ou UP) são diferentes das mensagens de *heartbeat* (I_AM_ALIVE), enviadas pelos objetos monitorados. Geralmente, o monitor apenas notifica o cliente quando um objeto monitorado mudou seu estado, isto é, entrou em estado DOWN ou saiu da lista de suspeitos (UP), enquanto que as mensagens de *heartbeat* são enviadas constantemente.

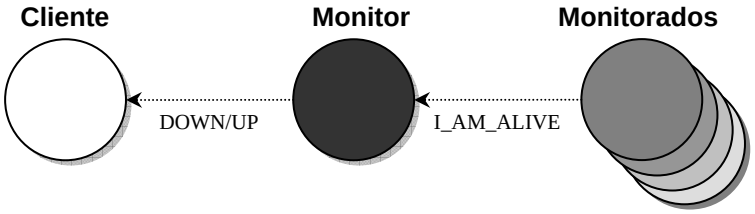

Figura 2.1: Monitoração de objetos no modelo *push*

Na Figura 2.2, também adaptada de Felber *et al.* (1999), é mostrado como as mensagens são trocadas entre os objetos monitorados e o monitor. Para iniciar a monitoração, o monitor envia uma mensagem de inicialização do tipo PUSH_INIT, pela qual informa a taxa em que o monitorável deve enviar as mensagens de *heartbeat*. A partir daí, o objeto monitorado periodicamente envia mensagens de *heartbeat* ao monitor que, ao receber cada mensagem, reinicia o *timeout* daquele objeto, como indicado na Figura 2.2. Caso o *timeout* expire sem que o monitor receba mensagens do objeto monitorado, este objeto monitorado entra para a lista de suspeitos do monitor.

O estabelecimento da relação entre o intervalo de envio e o *timeout* é essencial para a adequação da latência e da precisão da detecção. Isso significa que se a relação for muito próxima, ou seja, se o tempo de envio for muito próximo do *timeout*, a detecção poderá indicar rapidamente a ausência do recebimento da mensagem. Mas, também, poderá induzir falsas suspeitas devido a pequenos atrasos impostos pela rede de comunicação. Uma relação muito distante faz com que aumente a certeza da falha do processo, mas reduz a agilidade da detecção (ESTEFANEL, 2001).

Geralmente, sistemas de detecção usam *timeouts* fixos, impossibilitando adaptar o mecanismo a variações e mudanças que podem ocorrer ao longo do tempo no ambiente real. Contudo, existem algumas estratégias que fazem o ajuste do *timeout* dinamicamente durante o funcionamento.

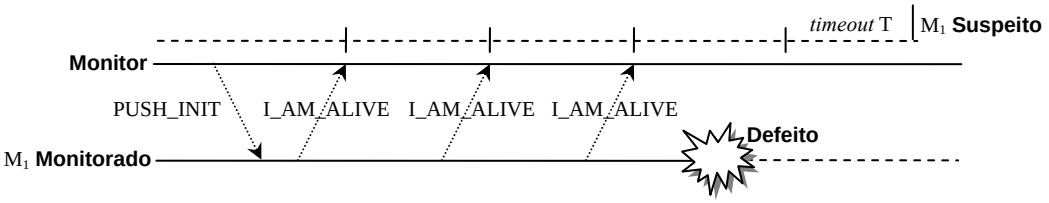

Figura 2.2: Monitoração de mensagens no modelo *push*

2.4.2 Modelo *pull*

Neste modelo, a direção do fluxo de informações é oposta à do fluxo de controle, ou seja, as informações do detector de defeitos só são obtidas através da requisição junto aos processos monitorados (FELBER, 1998). Nesse caso, o processo monitorado é passivo e o monitor é ativo. O monitor envia periodicamente mensagens do tipo requisição ARE_YOU_ALIVE para o componente monitorado. A cada recebimento dessa mensagem, o monitorado responde ao monitor com uma mensagem do tipo YES, para indicar que está "vivo" (funcionando). Quando o monitor não recebe a resposta em um tempo estimado (*timeout*), surge a suspeita de defeito no monitorado e é feita uma indicação de colapso do mesmo (o componente é adicionado à lista de suspeitos).

Como vantagem, o modelo *pull* possibilita aos processos monitorados serem passivos, sem a necessidade de conhecerem *timeouts*, nem quais são os detectores que os monitoram. Porém, obriga o detector a refazer a lista de processos defeituosos toda vez que um cliente solicitar. A Figura 2.3, adaptada de Felber *et al.* (1999), ilustra como o modelo *pull* é usado para monitorar objetos.

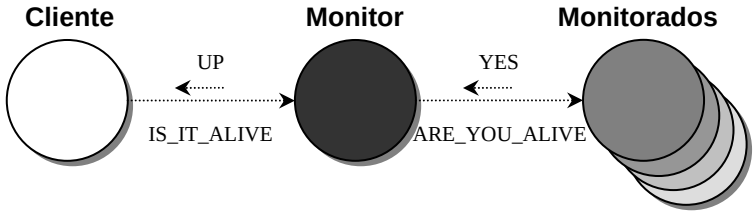

Figura 2.3: Monitoração de objetos no modelo *pull*

Devido à troca de mensagens entre os processos monitorados e o detector ser bidirecional (*two-ways*), esse modelo tende a ser menos eficiente do que o modelo *push*[4], se ambos possuírem o mesmo intervalo de monitoração. Mas o *pull* 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, não há a necessidade de saber qual é a freqüência com que o detector espera receber mensagens) (ESTEFANEL, 2001).

A escolha de qual modelo utilizar depende muito da natureza da aplicação. O protocolo de troca de mensagens é ilustrado na Figura 2.4, também adaptada de Felber *et al.* (1999). Esse modelo é indicado para sistemas que ocasionalmente requerem informações sobre o estado dos processos, podendo ser utilizado para minimizar o número de mensagens.

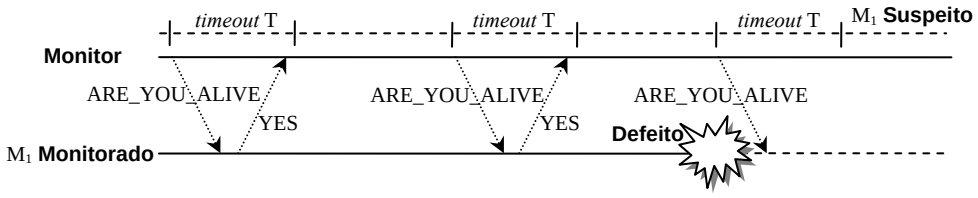

Figura 2.4: Monitoração de mensagens no modelo *pull*

[4] O número de mensagens trafegando tem efeito direto sobre o desempenho dos detectores, pois essas mensagens exigem recursos da rede e de processamento e, por isso, um detector *pull* tende a ser menos eficiente do que um detector *push*. No entanto, essa aparente desvantagem pode ser compensada com uma boa escolha dos parâmetros de operação, bem como pelas necessidades da aplicação distribuída.

2.4.3 Modelo *dual*

Além dos dois modelos anteriormente descritos, *push* e *pull*, o trabalho de Felber *et al*. (1999) propõe a união desses modelos para prover maior flexibilidade, criando um modelo misto (*dual*), no qual ambos podem ser usados simultaneamente com o mesmo conjunto de objetos.

O protocolo do modelo *dual* executa em duas fases. Na primeira, os monitores assumem que os objetos monitorados utilizam o modelo *push* e, por conseguinte, esperam por mensagens do tipo I_AM_ALIVE. Decorrido um tempo (*timeout* T_1), os monitores ingressam na segunda fase e assumem que os objetos monitorados que não enviaram mensagens de *heartbeat* durante a primeira fase podem estar com defeitos e, assim, usam o modelo *pull*. Nessa segunda fase, os monitores enviam mensagens do tipo ARE_YOU_ALIVE para cada objeto monitorado que não enviou mensagem na primeira fase e esperam por uma mensagem do tipo YES (mensagem semelhante às do modelo *pull*). Se na segunda fase os objetos monitorados não responderem em um determinado período de tempo (*timeout* T_2), eles entram na lista de suspeitos dos monitores.

A Figura 2.5, de Felber *et al.* (1999), ilustra o protocolo de troca de mensagens no modelo *dual*. Nesse exemplo, dois objetos são monitorados (M_1 e M_2). M_1 é monitorado no estilo *push*, e M_2 é monitorado no estilo *pull*. M_1 envia mensagens de *heartbeat* para o monitor, enquanto que M_2 apenas responde quando requisitado. O monitor utiliza dois *timeouts*, T_1 e T_2, para as fases 1 e 2. Ele espera por mensagens de *heartbeat* dos objetos monitorados no estilo *push* durante a fase 1. Depois do *timeout* T_1 expirar, o monitor passa para a fase 2 e envia mensagens de requisição para os objetos monitorados dos quais ele não recebeu mensagens de *heartbeat* e espera durante o *timeout* T_2. Depois do intervalo de tempo T_2, o monitor suspeita de todos os objetos que não enviaram mensagens.

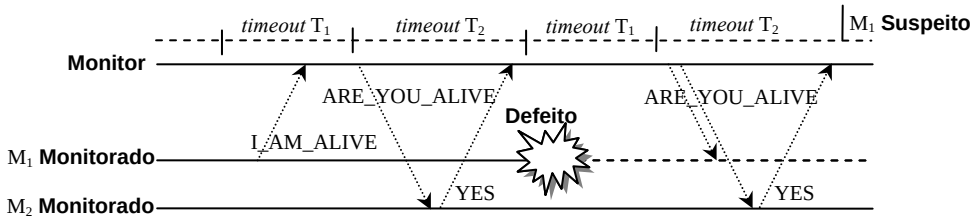

Figura 2.5: Monitoração de mensagens no modelo *dual*

Nesse modelo, podem ser usadas apenas mensagens não-bloqueantes (*one-way*), que são mais eficientes. Um processo monitorado também pode enviar uma mensagem, indicando que está operante sem que esta mensagem tenha sido requisitada. Nesse modelo, o detector pode operar com qualquer tipo de processo, sem a necessidade de conhecer previamente que modelo está associado ao processo a ser monitorado.

2.5 Qualidade de Serviço para Detectores de Defeitos

Um detector de defeitos pode ser visto como uma abstração, usada no desenvolvimento de vários serviços distribuídos tolerantes a falhas. Idealmente, um detector eficiente é aquele que detecta a ocorrência de defeitos com a maior rapidez e precisão possíveis, ou seja, com uma boa qualidade de serviço (QoS). Uma vez detectado o problema, ações de tolerância a falhas podem ser realizadas. Para aplicações que possuam restrições de tempo, é importante que o detector atenda quantitativamente

algumas métricas de QoS relacionadas a aspectos temporais (*timeliness*) (CHEN; TOUEG; AGUILERA, 2002).

Algumas métricas foram propostas por Chen, Toueg e Aguilera (2002) para avaliar a qualidade de serviço de detectores de defeitos que medem a rapidez com que falhas reais são detectadas e o grau de existência de falsas detecções. Para a ilustração da definição das métricas é considerado um sistema simples, composto por somente dois processos, *p* e *q*. Nesse sistema, o processo *q* monitora o processo *p*. O processo *q* nunca entra em colapso e pode detectar defeitos de colapso no processo *p*.

Para avaliar as métricas, são consideradas duas saídas para o detector *q* no tempo *t*: *S* indica que *q* suspeita (*suspect*) de *p*, e *T* indica que *q* confia (*trust*) em *p*. Uma transição (*transition*) ocorre quando a saída do detector *q* muda: uma *S-transition* ocorre quando a saída do detector muda de *T* para *S*; uma *T-transition* ocorre quando a saída do detector muda de *S* para *T*. É assumido que existe somente um número finito de transições durante qualquer intervalo de tempo finito.

Para o sistema considerado, Chen, Toueg e Aguilera (2002) definem dois conjuntos de métricas: primárias e derivadas. A primeira métrica, **Tempo de Detecção** (T_D), relacionada à velocidade do detector é obtida conforme ilustra a Figura 2.6.

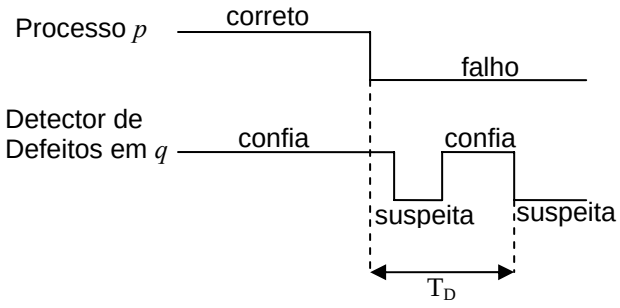

Figura 2.6: Tempo de Detecção (T_D)

As métricas **Tempo de Retorno ao Erro** (T_{MR}), **Duração do Erro** (T_M), **Taxa Média de Erro** (λ_M), **Probabilidade da Precisão da Consulta** (P_A), **Duração do Período Bom** (T_G) e **Duração do Período Bom a Frente** (T_{FG}) são relacionadas à precisão da detecção e são obtidas conforme ilustra a Figura 2.7. A métrica **Taxa Média de Erro** (λ_M) e a **Probabilidade da Precisão da Consulta** (P_A) não possuem ilustração de suas obtenções por se tratarem de métricas estatísticas.

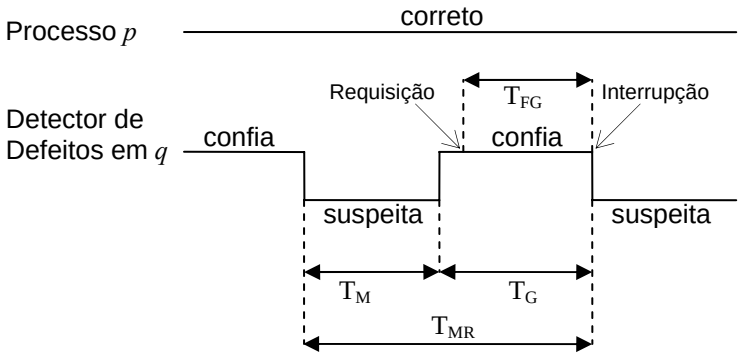

Figura 2.7: Métricas de QoS (T_M, T_G, T_{MR} e T_{FG})

2.5.1 Métricas primárias

Tempo de Detecção (*Detection Time* - T_D): é o tempo decorrido desde quando o processo *p* entra em colapso até o instante em que *q* suspeita de *p* permanentemente. Mais precisamente, é o tempo entre o momento em que *p* entrou em colapso e o momento final da *S-transition*, sem mais transições posteriores. A maneira de obtenção dessa métrica pode ser vista na Figura 2.6, adaptada de Chen, Toueg e Aguilera (2002).

Tempo de Retorno do Erro (*Mistake Recurrence Time* - T_{MR}): é o tempo decorrido entre dois erros de suspeita consecutivos. Mais precisamente, representa o tempo entre uma *S-transition* e a próxima *S-transition*.

Duração do Erro (*Mistake Duration* - T_M): é o tempo decorrido até que o detector corrija um erro de suspeita. Mais precisamente, representa o tempo entre uma *S-transition* e a próxima *T-transition*.

2.5.2 Métricas derivadas

Taxa Média de Erro (*Average Mistake Rate* - λ_M): é a taxa de erros cometidos por um detector de defeitos, isto é, é a média de *S-transitions* por unidade de tempo. É importante para aplicações onde cada erro do detector (cada *S-transition*) resulta num custo de interrupção.

Probabilidade da Precisão da Consulta (*Query Accuracy Probability* - P_A): é a probabilidade de que a saída de um detector esteja correta em um tempo aleatório. Essa métrica é importante para aplicações que interagem com o detector de defeitos, fazendo requisições em tempos aleatórios. Algumas aplicações podem progredir apenas durante *períodos bons* – nos quais o detector não comete erros. Essa observação conduz a outras duas métricas descritas na sequência.

Duração do Período Bom (*Good Period Duration* - T_G): é o intervalo do período bom, aquele em que o detector não comete erros. Representa, mais precisamente, o tempo entre uma *T-transition* e a próxima *S-transition*.

Duração do Período Bom à Frente (*Forward Good Period Duration* - T_{FG}): é o tempo decorrido a partir de um tempo aleatório no qual *q* confia em *p* até a próxima *S-transition* (é a parte restante do período bom).

2.6 Conclusões Parciais

Neste capítulo, foram apresentados os principais aspectos associados aos detectores de defeitos e do modelo do sistema distribuído adotado. Também foram apresentadas as duas principais formas de classificação dos detectores de defeitos: uma voltada para a parte teórica, segundo propriedades axiomáticas (CHANDRA; TOUEG, 1996); e outra voltada para parte prática, segundo o fluxo de mensagens (FELBER *et al.,* 1999). Adicionalmente, foram citadas as principais métricas de avaliação de qualidade de serviço específicas para detectores de defeitos.

Após apresentados esses conceitos básicos, o próximo capítulo concentra-se no estudo de estratégias especializadas para auxiliar os mecanismos de detecção a operarem em diferentes tipos de ambientes. As diferentes estratégias oferecem variações quanto ao funcionamento e alternativas voltadas para sobrepor algumas das dificuldades impostas pelos ambientes, tais como as preocupações que surgem em sistemas de larga escala ou em redes de longa distância.

3 COMUNICAÇÃO ENTRE MÓDULOS DETECTORES

Para adaptar o serviço de detecção de defeitos às características dos sistemas de larga escala, é possível usar diferentes estratégias de comunicação entre detectores, que resultam em especificações diferenciadas para os mecanismos. As principais especificações e estratégias utilizadas são brevemente descritas neste capítulo.

O objetivo principal é utilizar, ou adaptar, uma boa estratégia, dentre as citadas, para tratar da disseminação de informações na rede de comunicação (comunicação entre os módulos detectores), visando, com isso, a reduzir o número de mensagens entre módulos. A forma mais comum de comunicação, já descrita na seção 2.4, especifica o serviço em relação ao fluxo de mensagens (modelos *push*, *pull* e *dual*). Na arquitetura lógica da comunicação, a comunicação pode ser organizada de maneira hierárquica (FELBER *et al.*, 1999; BERTIER; MARIN; SENS, 2003) ou em anel lógico (LARREA; ARÉVALO; FERNÁNDEZ, 1999). Protocolos estilo *gossip* (VAN RENESSE; MINSKY; HAYDEN, 1998; BURNS; GEORGE; WALLACE, 1999) e *Heartbeat* (AGUILERA; CHEN; TOUEG, 1997), também procuram tratar questões relativas à comunicação.

Cada máquina (*host*) do sistema executa um processo de detecção de defeitos que, por sua vez, oferece informações sobre os processos que esteja monitorando para os clientes interessados (aplicações). O número de mensagens trocado entre os detectores influencia diretamente o tráfego da rede, aumentando a probabilidade das mensagens chegarem atrasadas aos seus destinos, aumentando, também a sobrecarga dos sistemas com o processamento dessas mensagens e, conseqüentemente, influenciando na escalabilidade do serviço de detecção. Desta forma, são analisados os mecanismos que podem difundir mensagens entre módulos detectores, considerando, para isso, o número de mensagens trocadas, a fim de garantir a difusão das informações por todos os módulos.

Como os detectores podem ter como característica o funcionamento ininterrupto, a análise feita, neste capítulo, dar-se-á, além do número de mensagens trocadas, com base no número de etapas para que um ciclo de detecção seja completado. Quando possível, será contabilizado o número mínimo de passos (etapas) de comunicação para a disseminação das informações entre os módulos detectores, de modo que todos tenham uma visão global do sistema – neste trabalho, tal métrica pode ser chamada de grau de latência, que também foi utilizada no trabalho de Estefanel (2001). O uso de difusão de mensagens (*broadcast*) (CHANDRA; TOUEG, 1996) pode reduzir o custo de implementação dos algoritmos, mas depende da disponibilidade da rede de comunicação e das ferramentas utilizadas.

Alguns algoritmos e estratégias analisados nas seções subseqüentes, quando possível, foram modificados para atender ao mesmo propósito, que é possibilitar aos

módulos construírem uma visão global do sistema, possuindo informações sobre todas as entidades que estejam sendo monitoradas pelo serviço.

3.1 Comunicação *Todos-para-Todos*

A escalabilidade é uma das maiores preocupações para um serviço que atue em sistemas de larga escala, pois, o elevado número de entidades pode comprometer a eficiência do serviço ou mesmo afetar as demais aplicações que estejam usando a mesma rede de comunicação.

Uma abordagem tradicional de detecção de defeitos é dotar cada entidade participante de um monitor local que provê informações de suspeitas. Porém, essa arquitetura possui problemas de eficiência e escalabilidade, quando utilizada em complexas aplicações distribuídas, nas quais um grande número de participantes esteja envolvido.

Se cada participante monitorar os demais utilizando comunicação ponto-a-ponto, a complexidade do número de mensagens trocadas será $O(n^2)$ para n participantes (comunicação *todos-para-todos*). A comunicação pode ser feita, usando o modelo *push* ou *pull*. Na adoção de um desses modelos é especialmente interessante que o sistema possua suporte à comunicação por difusão (*broadcast*) para tirar melhor vantagem dos modelos.

3.1.1 Utilização do modelo *push*

Neste modelo, cada módulo envia uma mensagem do tipo I_AM_ALIVE para todos os demais módulos, podendo carregar consigo as informações sobre os processos que o módulo está monitorando, o que corresponde a $(n - 1)$ mensagens por módulo. Entretanto, cada módulo deve enviar essas mensagens, de forma que, ao final de um ciclo de detecção, deverão ter sido transmitidas $n(n - 1)$ mensagens no total. Isto representa um custo de $O(n^2)$ mensagens trocadas. A formação da visão global leva apenas um passo.

3.1.2 Utilização do modelo *pull*

No modelo *pull*, o ciclo de detecção envolve a troca de duas mensagens: uma do tipo ARE_YOU_ALIVE, solicitando informações, e outra do tipo YES, representando as informações sobre os processos que o módulo está monitorando. Assim, cada módulo gera $2(n - 1)$ mensagens, mas o número total de mensagens trocadas é de $2n(n - 1)$, que é levemente superior ao do modelo *push.* O custo do algoritmo continua sendo de complexidade $O(n^2)$. A formação da visão global leva dois passos: um para todos os módulos fazerem as requisições e outro para receber as respostas.

3.2 Detector de Defeitos com Configuração Hierárquica

Em contraposição à comunicação *todos-para-todos*, o número de mensagens de detecção e controle que trafega na rede pode ser reduzido, organizando hierarquicamente a arquitetura do serviço (FELBER *et al.*, 1999; BERTIER; MARIN; SENS, 2003). A idéia é confinar o maior tráfego de mensagens localmente.

A Figura 3.1, adaptada de Hayashibara (2004), ilustra uma organização hierárquica em três níveis. As setas finas representam o tráfego de mensagens locais e as setas espessas representam o fluxo de informações entre níveis. Um módulo detector pode ser

responsável por monitorar todos os processos (objetos) monitoráveis do mesmo nível. Já os módulos, organizados de forma hierárquica, mapeiam a propagação das informações.

Comparado com uma implementação tradicional de *todos-para-todos*, que gera um grafo totalmente conectado, a configuração hierárquica mapeia uma estrutura do tipo árvore, o que gera menos mensagens de controle e propagação. Além do mais, essa organização pode tirar vantagem da topologia física do sistema.

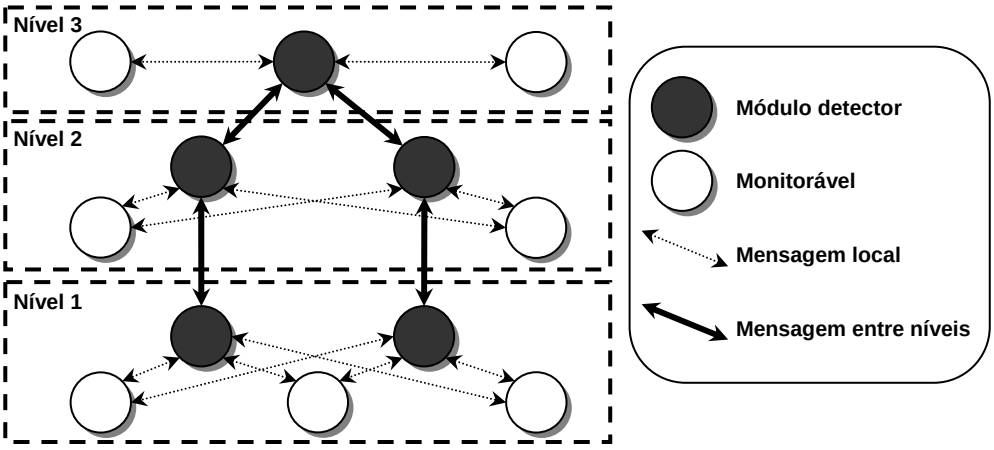

Figura 3.1: Organização hierárquica em três níveis

No trabalho de Felber *et al.* (1999), por exemplo, um ou mais módulos detectores pertencentes a uma rede local possuem informações sobre todos os processos monitorados nessa rede e, transmitem informações sobre esses processos para outros módulos remotos em outras LANs de duas maneiras possíveis:

- Um módulo detector transmite informações sobre processos monitorados específicos, com base nas requisições de outros módulos detectores das outras redes locais;
- Um módulo detector transmite uma tabela completa sobre as informações de todos os processos monitorados em sua rede local.

Desta maneira, módulos detectores de um nível, que tenham interesse em informações de estado sobre entidades de outro nível, necessitam interagir com os módulos detectores daquele nível. Adicionalmente, módulos detectores podem ser monitorados por outros módulos detectores, pois, também, estão sujeitos a defeitos. Ainda é possível reduzir o tráfego de mensagens, combinando informações sobre vários processos em uma única mensagem e, temporariamente, armazenar (*caching*) algumas informações em vários lugares do sistema.

A configuração hierárquica independe do modelo usado para monitorar as entidades de interesse (modelo *push*, *pull* ou *dual*). Ela permite adaptar mais adequadamente os parâmetros do monitor (como *timeouts*) para a topologia da rede ou para a localização de entidades monitoradas. Ainda reduz o número de mensagens trocadas entre máquinas distantes. Um monitor, localizado em uma LAN, pode adaptar-se às características dessa LAN e prover uma qualidade de serviço específica. A redução de tráfego na rede, especialmente quando uma grande quantidade de entidades monitoradas e clientes está envolvida, é a principal razão para a boa escalabilidade da abordagem hierárquica (FELBER *et al.,* 1999).

No trabalho de Bertier, Marin e Sens (2003), foi assumido que cada rede local possui um único módulo responsável pela troca de informações com outras redes locais, chamado de líder. Nesse caso, o sistema é composto por grupos locais, mapeados em cada LAN, e um grupo global, formado pelos líderes (que formam a WAN), como ilustra a Figura 3.2. Cada grupo é um espaço de detecção, e cada membro tem acesso a todos os outros membros pertencentes ao grupo.

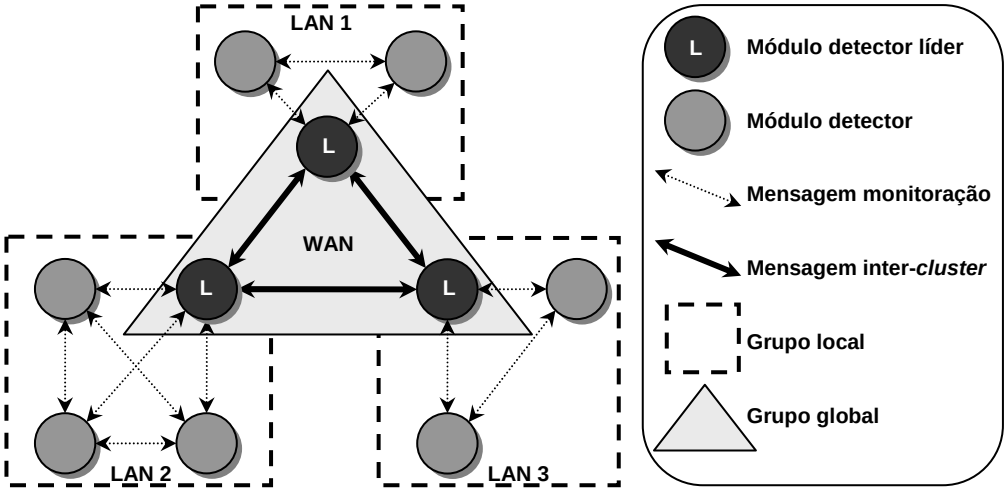

Figura 3.2: Organização hierárquica em dois níveis

Com o objetivo de construir uma visão global do sistema em todos os módulos, cada rede local deve construir uma visão local das entidades monitoradas; essa visão deve ser distribuída para as demais redes, por intermédio dos líderes. A operação pode ser concluída em três etapas:

- **1ª Etapa:** Cada rede local efetua um ciclo de detecção de modo que todos os módulos de cada uma criem visão local unificada.
- **2ª Etapa:** Procede-se a um ciclo de detecção (disseminação) entre líderes, no nível superior da hierarquia, de modo que todos eles construam a visão global do sistema.
- **3ª Etapa:** Cada líder dissemina para os demais módulos de sua rede local a visão global do sistema.

Assim, supondo um sistema hierárquico, composto por n módulos divididos, de forma balanceada, em c redes locais (*clusters*), o número de mensagens trocadas em cada etapa é dado como segue:

- **1ª Etapa:** n/c módulos enviam $(n/c - 1)$ mensagens. Como isto é feito em cada rede local, o total de mensagens resulta em $c(n/c(n/c - 1))$ mensagens.
- **2ª Etapa:** c módulos enviam $(c - 1)$ mensagens, resultando em $c(c - 1)$ mensagens.
- **3ª Etapa:** cada líder deve disseminar a visão global entre os demais módulos de sua rede local. Assim, cada líder envia $(n/c - 1)$ mensagens, resultando em $c(n/c - 1)$ mensagens.

Para atingir a visão global do sistema em todos os módulos detectores pertencentes ao sistema, o total de mensagens trocado nas três etapas resulta em $n^2/c + c^2 - 2n$. Assumindo que o sistema ofereça comunicação por difusão (*broadcast*), pode-se

considerar que o grau de latência deste modelo equivale a 3, um passo para cada etapa, formando a visão global em três passos.

É possível otimizar o número de mensagens na **1ª Etapa**, construindo a visão local apenas para o módulo que se comunicará com as outras redes (líder). Porém, tal otimização centraliza as informações no líder, tornando-o ponto único de falha. A presença de um líder pode comprometer a disponibilidade do serviço devido a sua susceptibilidade a defeitos. Para sobrepor tal deficiência ou vulnerabilidade, é possível efetuar a monitoração e escolha de líder (BERTIER; MARIN; SENS, 2003), e ainda utilizar um mecanismo de redundância, como o conceito de líder *backup* (JAIN; SHYAMASUNDAR, 2004).

Uma outra abordagem, ou alternativa de otimização, seria suprimir a 3ª etapa, deixando a visão global construída apenas nos líderes. Quando um cliente, aplicação ou detector necessitar informações sobre entidades de outras redes locais, faz a consulta no líder correspondente, segundo suas necessidades.

3.3 Algoritmo de Detecção de Defeitos *Heartbeat*

O algoritmo de detecção *Heartbeat* (AGUILERA; CHEN; TOUEG, 1997) procura resolver problemas de comunicação quiescente[5] em ambientes com possibilidade de defeitos de colapso e/ou omissão. Cada módulo que executa o algoritmo, periodicamente envia mensagens de *heartbeat* (I_AM_ALIVE) aos seus vizinhos que recebem as mensagens e incrementam os contadores correspondentes.

Diferente dos modelos tradicionais, que geram uma lista de processos suspeitos (suspeito/correto), este algoritmo não usa *timeouts* nem processa ou interpreta a suspeita sobre os processos, apenas mantém uma lista do número total de mensagens recebidas de cada processo (D_p). A aplicação, ou outro protocolo que o utilize, compara o vetor de valores obtidos em uma chamada anterior, de modo a identificar quais são os processos ou módulos com defeito. O pseudocódigo do algoritmo é dado na Figura 3.3.

```
1   For every process p:
2
3       Initialization:
4           for all q ∈ neighbor(p) do D_p[q] ← 0
5
6       Cobegin
7           || Task 1:
8               repeat periodically
9                   for all q ∈ neighbor(p) do send_{p,q}(HEARTBEAT, p)
10
11          || Task 2:
12              upon receive_{p,q}(HEARTBEAT, path) do
13                  for all q such that q ∈ neighbor(p) and q appears in path do
14                      D_p[q] ← D_p[q] + 1
15                  path ← path . p
16                  for all q such that q ∈ neighbor(p) and q does not appear in path do
17                      send_{p,q}(HEARTBEAT, path)
18      Coend
```

Figura 3.3: Pseudocódigo do algoritmo *Heartbeat*

[5] Comunicação quiescente considera que em um determinado momento o envio de mensagens irá cessar, embora esse instante seja desconhecido.

Na primeira tarefa, linhas 7 a 9 da Figura 3.3, um processo *p* periodicamente envia mensagens do tipo I_AM_ALIVE para todos os seus vizinhos *q*. Essa mensagem é composta de um campo identificador da operação (HEARTBEAT) e do caminho (*path*) inicial, que contém apenas o próprio processo.

A segunda tarefa, linhas 12 a 17, da Figura 3.3, trata as mensagens recebidas que sejam do formato {HEARTBEAT, *path*}. Ao receber uma mensagem, o processo incrementa o contador de todos os seus vizinhos que estejam relacionados no caminho. Em seguida, o processo adiciona seu próprio identificador ao fim do caminho e retransmite a mensagem para os vizinhos que ainda não a receberam.

Para ser comparável às demais formas de disseminação, esse protocolo deve ser modificado da mesma forma que no modelo *pull*, em que a mensagem I_AM_ALIVE pode carregar em seu conteúdo a lista de entidades que o módulo está monitorando. É desta forma, que um módulo, ao receber uma mensagem de outro, atualiza o estado das entidades correspondentes e acrescenta aquelas que ainda não estavam na lista.

No algoritmo *Heartbeat*, os processos (módulos detectores) não precisam conhecer a topologia da rede ou o número de processos do sistema, apenas necessitam saber a identificação de seus vizinhos, que é um conjunto ou subconjunto fixo do total de processos.

A probabilidade de que todos recebam informações sobre um processo cresce à medida que os processos disseminam suas tabelas. De fato, o número mínimo de retransmissões necessárias para atingir todos os processos corresponde ao grau de latência da disseminação, e depende diretamente da quantidade de processos monitorados e do número de vizinhos com que cada processo se comunica por vez. Considerando para este algoritmo um conjunto de vizinhos fixo, o número de passos é (*processos* – *vizinhos*) + 1, considerando a melhor probabilidade de difusão (ESTEFANEL, 2001). O número mínimo de passos para construir a visão global em todos os módulos é igual à maior distância entre dois módulos.

3.4 Protocolo Estilo *Gossip*

Os protocolos estilo *gossip* procuram reduzir o número de mensagens na rede em comparação à comunicação *todos-para-todos*. Esses protocolos possuem algumas variações para tratar questões relativas à comunicação. Van Renesse, Minsky e Hayden (1998) definiram o protocolo *gossip* básico e multi-nível para uso em detecção de defeitos, e é com base nesse trabalho que são descritos os dois protocolos, nas seções que seguem. Burns, George e Wallace (1999) propuseram um terceiro protocolo, o *gossip piggyback*, que é uma variação do multi-nível.

O protocolo *gossip* procura combinar a eficiência da disseminação hierárquica com a robustez de protocolos de inundação (*flooding*). Essencialmente, o protocolo baseia-se na estratégia de cada membro (módulo detector) manter uma lista de membros conhecidos (vizinhos) e periodicamente enviar informações para outros membros, escolhidos aleatoriamente. Assim, em um tempo finito todos podem detectar processos falhos no sistema, alcançando uma visão unificada do mesmo (VAN RENESSE; MINSKY; HAYDEN, 1998). Protocolos baseados nessa abordagem são, algumas vezes, chamados de protocolos epidêmicos.

Os módulos que executam o protocolo devem enviar, além das informações originais desse protocolo, a tabela sobre o estado das entidades que estão monitorando. Desta

forma, quando um módulo recebe a tabela de outro, ele atualiza sua visão e no próximo passo de tempo (*step interval*) a reenvia.

3.4.1 Protocolo *gossip* básico

Neste protocolo, um membro envia informações a outros membros escolhidos aleatoriamente. Cada membro mantém uma lista com o endereço de outros conhecidos e um inteiro associado, chamado *heartbeat counter*. A cada passo de tempo denotado por T_{gossip} segundos, cada membro incrementa os seus contadores e escolhe aleatoriamente outro para enviar sua lista. Quando um membro recebe uma lista, ele atualiza a sua lista com os valores máximos dos contadores para cada membro.

Os membros também mantêm, para cada outro membro, uma lista com o último tempo de incremento do *heartbeat counter*. Se esse contador não for incrementado em um tempo denotado por T_{fail} segundos, então o membro é considerado defeituoso. T_{fail} deve ser escolhido de modo que a probabilidade de ocorrer uma falsa detecção seja menor que um limiar (limite) denotado por $T_{mistake}$.

Após um membro ser considerado defeituoso, ele não pode ser imediatamente esquecido e retirado da lista. O problema é que nem todos os membros detectam o defeito simultaneamente, e um membro A pode receber uma informação de B, sendo que A anteriormente tinha detectado que B estava com defeito. Não retirá-lo imediatamente evita que B seja reinstalado nas listas de outros membros que anteriormente o tenham detectado com defeito.

Por conseguinte, o detector não remove um membro suspeito da sua lista antes de um tempo denominado $T_{cleanup}$ segundos ($T_{cleanup} \geq T_{fail}$). $T_{cleanup}$ deve ser escolhido de modo que a probabilidade de receber informação sobre um membro, após este ter sido detectado com defeito, seja menor que um limiar denominado $P_{cleanup}$. O parâmetro $P_{cleanup}$ pode ser escolhido como sendo igual a uma probabilidade denominada P_{fail} configurando o $T_{cleanup}$ para $2*T_{fail}$. A discussão e análise desses parâmetros são feitas no trabalho de Van Renesse, Minsky e Hayden (1998).

Guo (1998) descreve um algoritmo para um detector estilo *gossip*, como ilustra o pseudocódigo da Figura 3.4. O princípio é que cada membro mantém uma lista L (*"Live"*) de *n* elementos preenchidos com 0 inicialmente. O algoritmo é executado por duas tarefas concorrentes. Na primeira, linhas 7 a 10 da Figura 3.4, cada membro *i* incrementa todos os elementos da lista L de 1 unidade e mantém L[*i*] = 0 e, após este incremento, envia a lista L para um subconjunto de membros escolhidos aleatoriamente. Na segunda, um membro *i* ao receber uma mensagem de dados de um membro *j*, altera o contador de L[*j*] para 0 (linhas 13 e 14 da Figura 3.4); e, ao receber uma lista L' do membro *j*, atualiza a sua com os valores mínimos dos contadores (linhas 15 e 16 da Figura 3.4). Desta forma, valores baixos indicam membros ativos e valores altos correspondem aos que não têm estado ativos recentemente.

Um exemplo de execução do protocolo com 4 membros (A, B, C e D) em um grupo, extraído de Guo (1998), mostra a visão do membro A durante a execução, conforme pode ser visto na Figura 3.5. Neste exemplo, inicialmente a lista L de A é [0, 0, 0, 0]. Após um passo de tempo ou intervalo (*step interval*), o membro A incrementa em uma unidade os contadores de cada membro de sua lista L, com exceção dele mesmo, resultando em [0, 1, 1, 1]. Após o incremento, A envia L para B.

No próximo passo de tempo, a lista é incrementada novamente ([0, 2, 2, 2]) e A a envia para C. Quando A recebe uma mensagem de dados de C, ele atualiza o índice de C para 0, resultando em [0, 2, 0, 2]. Após alguns passos, L é [0, 4, 3, 6]; nesse momento A recebe a lista L' de D ([2, 1, 2, 0]), calcula o mínimo das duas (*ArrayMin*([0, 4, 3, 6], [2, 1, 2, 0]) = [0, 1, 2, 0])) e atualiza sua lista.

```
1   For every process i:
2
3       Initialization:
4           L_i ← [0 0 0 ... 0]
5
6       Cobegin
7           || Task 1:
8               repeat periodically
9                   L_i[j] ← L_i[j] + 1 for all j ≠ i
10                  send(L_i) to a random subset
11
12          || Task 2:
13              upon receive (data) do
14                  L_i[j] ← 0
15              upon receive (L') do
16                  L_i[j] ← ArrayMin(L_i, L')
17      Coend
```

Figura 3.4: Pseudocódigo do algoritmo *gossip*

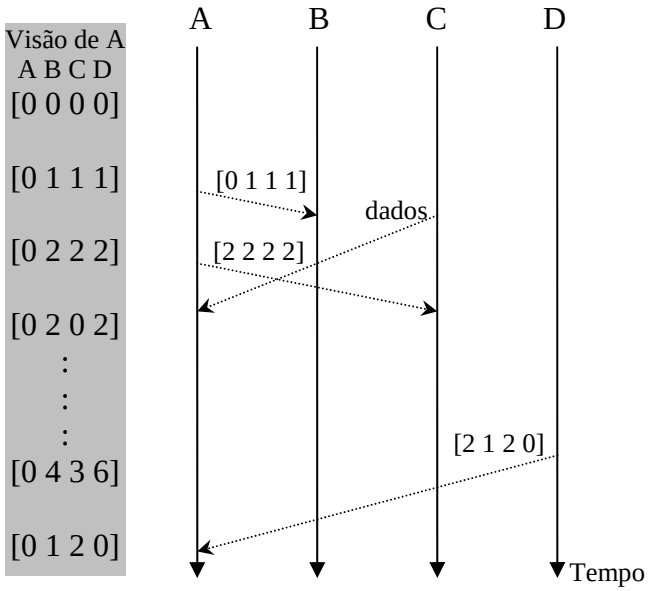

Figura 3.5: Atualização dos contadores em um detector *gossip*

A detecção é feita como no trabalho de Van Renesse, Minsky e Hayden (1998), mas o valor de T_{fail} depende, além do tamanho do intervalo de tempo de envio, do tamanho do subconjunto de cada passo.

Van Renesse, Minsky e Hayden (1998) conduziram uma análise estatística com uma versão melhorada do protocolo descrito por Guo (1998). Essa versão diferencia-se daquela porque em cada passo de tempo somente um membro é escolhido como destinatário da lista. Conseqüentemente, o número de passos necessários aumenta logaritmicamente com o número *n* de membros. O tamanho da lista L aumenta com o tamanho do grupo *n*; então, para manter requisitos de largura de banda constante por

membro, o passo de tempo é escolhido como sendo proporcional a *n*. Com esses requisitos, T_{fail} cresce na ordem de (*n log n*). Assumindo que cada elemento da lista ocupe 1 *byte*, o tamanho da mensagem do protocolo é igual a *n bytes* onde *n* é o tamanho do grupo. Uma estrutura hierárquica pode ser empregada para reduzir o tamanho das mensagens e melhorar a escalabilidade (GUO, 1998).

O protocolo somente detecta defeitos em máquinas completamente inalcançáveis e não os detecta no canal de comunicação entre máquinas. Se A e B não conseguem se comunicar diretamente, mas somente através de outra máquina C, A não suspeitará de B nem B de A. Quando ocorre esse tipo de particionamento, o protocolo pode ciclicamente adicionar e remover membros da lista. Porém, a dinâmica dos protocolos de roteamento da Internet dão conta desse problema (VAN RENESSE; MINSKY; HAYDEN, 1998).

No algoritmo de Guo (1998), os processos podem identificar mensagens repetidas ou velhas através dos contadores relacionados com cada processo e a suspeita ocorre somente após terem sido retiradas de circulação todas as mensagens que carregam o identificador do processo falho. Isso significa que a detecção de um defeito pode ser realizada com um grau de latência idêntico ao da disseminação das informações (ESTEFANEL, 2001).

3.4.2 Protocolo *gossip* multi-nível

Uma vez que membros podem escolher outros sem se preocuparem com a topologia da rede de comunicação, pontes entre redes físicas podem ser sobrecarregadas com muita informação redundante. O protocolo *gossip* multi-nível é uma variação do protocolo *gossip* básico e diferencia-se por possuir conhecimento adicional sobre a topologia da rede e, um parâmetro que especifica a porcentagem para o envio das informações intra e inter-*cluster*. A Figura 3.6 ilustra um exemplo de configuração de componentes com protocolo *gossip* multi-nível (hierárquico).

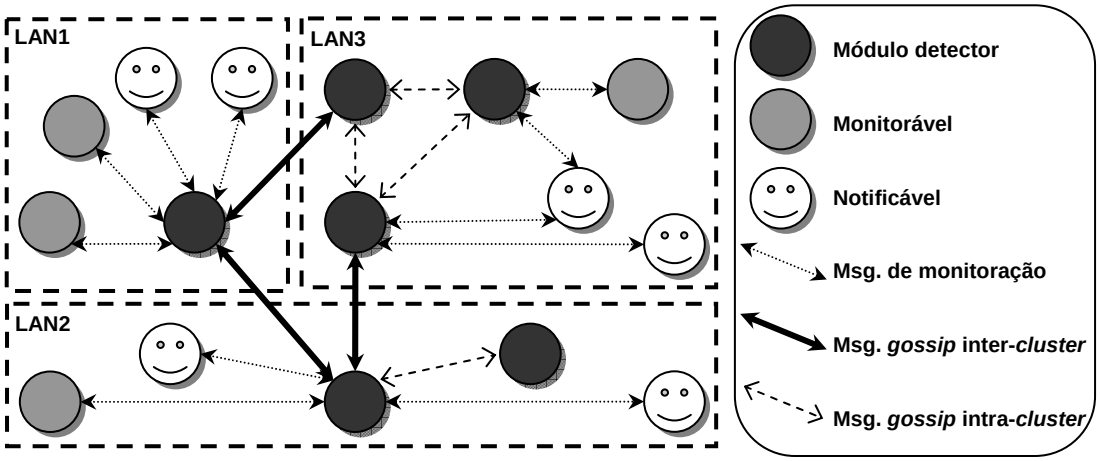

Figura 3.6: Configuração hierárquica com *gossip*

O protocolo detecta fronteiras entre domínios e sub-redes da Internet, reduzindo, assim, o tráfego entre eles. Além da lista normal de identificação dos membros vizinhos utilizada no protocolo *gossip* básico, o protocolo *gossip* multi-nível possui outra lista, vinculada à lista normal, que contém os números do domínio e a máscara de sub-rede de cada membro vizinho, utilizados para determinar o tamanho da sub-rede e os números das máquinas. Essas informações adicionais são enviadas junto com os contadores dos

vizinhos. O protocolo utiliza as informações adicionais para controlar o envio de mensagem entre membros de sub-redes diferentes e de domínios diferentes.

Esse protocolo reduz significativamente a largura de banda consumida entre roteadores da Internet. Concentrando-se no baixo nível da hierarquia e enviando a maioria das mensagens para os membros que pertencem à mesma rede local (mesmo *cluster*), o protocolo *gossip* multi-nível melhora o desempenho e a escalabilidade em relação ao protocolo *gossip* básico. O tempo de detecção de defeitos em sub-redes também é reduzido, mas o número de rodadas necessário para a informação ser disseminada através do sistema, incluindo sub-redes e domínios, é elevado.

3.4.3 Protocolo *gossip piggyback*

O protocolo *gossip piggyback* usa uma abordagem que procura reduzir a largura de banda, utilizada para a disseminação das informações, em comparação aos protocolos *gossip* anteriormente descritos, o básico e o multi-nível. A abordagem refere-se à adição das mensagens do protocolo nas mensagens geradas pela aplicação (*piggyback*), sempre que isto for possível.

O chaveamento entre envios independentes e os associados com as mensagens da aplicação é controlado por um parâmetro, que varia a taxa em que as mensagens do protocolo são adicionadas às da aplicação. Se T_{gossip} é o tempo necessário para cada membro atualizar suas informações, ou seja, receber e enviar informações de um membro de sua lista e, se P é o fator multiplicativo, entre 0 e 1, que determina a parcela de tempo, dentro de T_{gossip} em que as mensagens do protocolo não podem ser adicionadas às da aplicação. Então, as mensagens são enviadas no intervalo $T_{message}$, compreendido entre $T_{gossip}*P$ e T_{gossip} segundos. T_{gossip} e $T_{message}$ são medidos relativamente à transmissão de informações mais recente do protocolo. A Figura 3.7, adaptada do trabalho de Burns, George e Wallace (1999), mostra o intervalo em que as informações do protocolo podem ou não ser adicionadas às mensagens da aplicação.

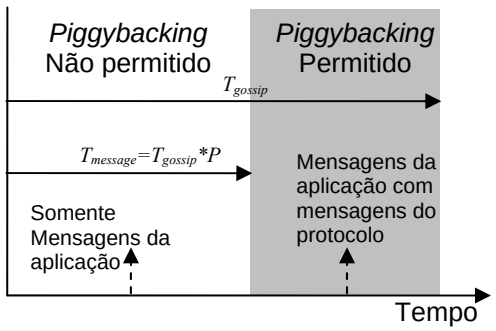

Figura 3.7: Intervalo em que o *piggyback* é, ou não permitido

Na simulação das três variantes do protocolo *gossip*, Ranganathan *et. al* (2001) observaram que o modelo hierárquico (multi-nível) e o *piggyback* foram significativamente mais escaláveis do que o protocolo básico. Porém, eles também concluíram que a implementação do protocolo *piggyback* requer o gerenciamento de muitas *threads* de controle; além disso, a eficiência do protocolo depende da taxa pela qual a aplicação envia mensagens. Se a aplicação não gerar mensagens freqüentemente, o protocolo *piggyback* tende a se comportar como o hierárquico, enviando mensagens por conta própria.

3.5 Detecção em Anel Lógico

O padrão de monitoração e propagação de informações sobre defeitos pode ser baseado em uma organização em anel lógico (LARREA; ARÉVALO; FERNÁNDEZ, 1999). Inicialmente, cada processo monitora o seu sucessor no anel. Se um processo *p* não recebe a resposta de uma requisição feita ao processo *q*, que está monitorando, então *p* suspeita que *q* esteja em colapso e começa a monitorar o sucessor de *q* no anel (*r*), como ilustra o exemplo da Figura 3.8. Se o processo *p* futuramente receber a mensagem de *q*, ele volta a monitorar *q* e refaz o anel.

Usando essa organização em anel, os módulos podem trocar informações sobre os processos monitorados. Para construir uma lista global de estado dos processos monitorados, os módulos necessitam propagar a lista local de processos que estão monitorando. Porém, para construir a lista global em todos os módulos, esses módulos necessitam propagar as suas listas locais.

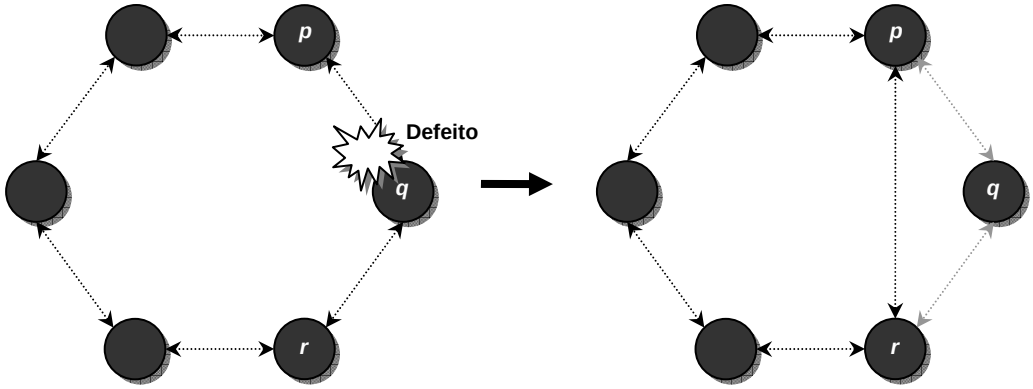

Figura 3.8: Exemplo de monitoração entre módulos em anel lógico

Um módulo inicia a comunicação, enviando sua lista de processos monitorados ao seu sucessor. Ao receber a lista do antecessor, atualiza a sua própria e a propaga novamente ao seu sucessor. Cada módulo, em cada passo do algoritmo, envia e recebe no máximo uma mensagem. Para garantir a visão global em todos os módulos, são necessários dois passos. O primeiro forma a visão global no módulo que iniciou a comunicação e o segundo propaga essa visão para os demais. O total do número de mensagens trocadas tem complexidade de O(*n*). A organização em anel é bastante vulnerável à ocorrência de particionamento no caso de defeitos em um ou mais módulos ou simples quebra do canal de comunicação.

3.6 Protocolo de Detecção de Defeitos *Lazy*

Os protocolos de detecção de defeitos tradicionais, a exemplo dos anteriormente citados, geralmente, atuam em sistemas distribuídos parcialmente síncronos, e não consideram o comportamento das aplicações do nível superior que os utilizam. Esses detectores, continuamente, enviam e processam mensagens de controle, consumindo recursos mesmo quando não usados pela camada superior. Como alternativa, uma nova abordagem pode ser empregada, na qual o custo associado à implementação do detector

é pago somente quando o detector é usado. Esse protocolo é chamado de protocolo de detecção de defeitos *Lazy*[6] (FETZER; RAYNAL; TRONEL, 2001).

Nesse protocolo, são utilizadas as informações da aplicação para obter informações sobre a detecção. Se dois ou mais processos estão se comunicando, as mensagens de aplicação que eles trocam são vistas também como um indício de atividade e assim utilizadas para inferir sobre a detecção de defeitos. Somente quando os processos não estão se comunicando, as mensagens do detector são enviadas. Percebe-se, então, que o protocolo está intimamente relacionado ao padrão de comunicação das aplicações que o utilizam.

O princípio de funcionamento do protocolo exige que as mensagens sejam reconhecidas (*acknowledged*). Se um processo p_i requisitar informação referente a um processo p_j (usando a primitiva QUERY(*j*)), a resposta depende dos atrasos dos tempos de ida e volta das mensagens (RTT - *Round Trip Time*), das mensagens já confirmadas. Diferentemente, se um processo p_i requisitar informação de um processo p_j, enquanto ele não está se comunicando com p_j, mensagens de controle serão enviadas para compensar a ausência de mensagens da aplicação. Como ilustrado na Figura 3.9, os processos da aplicação devem usar três primitivas para envio e recebimento de mensagens e para obter informações sobre processos monitorados. As primitivas para cada processo p_i da aplicação da camada superior são:

- **SEND(**M**)**: usada por p_i para enviar uma mensagem M da aplicação para o processo p_j;
- **RECEIVE(**M**)**: usada por p_i para receber uma mensagem M da aplicação;
- **QUERY(*j*)**: usada para saber se p_j é suspeito de ter entrado em colapso. Esta primitiva retorna um valor de resposta *suspected* ou *no_suspected*.

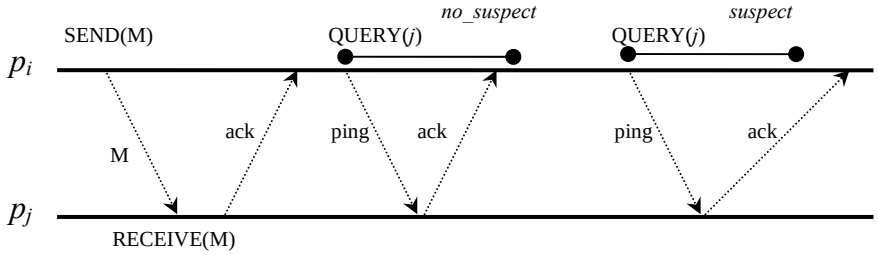

Figura 3.9: Protocolo de detecção de defeitos *Lazy*

3.7 Complexidade das Estratégias para Disseminação de Informações

Quando o serviço de detecção de defeitos dissemina informações de suspeitas sobre as entidades monitoradas independente do interesse das aplicações, tal disseminação é feita com a intenção de construir uma visão global do sistema. Segundo as estratégias anteriormente citadas, a escolha de uma delas pode ser feita visando a dois objetivos diferentes:

- i) minimizar o número de mensagens por rodada[7];

[6] Detalhes do protocolo *Lazy*, pseudocódigo, propriedades e custos das mensagens, podem ser encontrados em Fetzer, Raynal e Tronel (2001).

- ii) minimizar o número de rodadas (passos) para construção de uma visão global em todas as entidades.

Melhorar um desses objetivos implica degradar o outro; então, pode-se escolher uma estratégia que priorize atender um ou outro, ou ainda, obter a melhor relação entre eles, caso os dois objetivos devam ser atendidos simultaneamente. Essa escolha deve ser feita no projeto e no desenvolvimento do serviço, tendo em vista o tipo de aplicações que serão executadas e quais os principais objetivos dessas aplicações em relação à construção de uma visão global.

Para atingir, de forma isolada, o primeiro objetivo (i), uma forma é organizar a troca de mensagens entre os monitores de maneira a formar um anel lógico, como descrito na seção 3.5. Essa configuração considera que cada monitor envia mensagens a um monitor vizinho sucessor e recebe mensagens de um monitor vizinho antecessor, de modo a formar um ciclo fechado. Assim, a formação de uma visão global em todos os módulos pode ser feita com um número mínimo de mensagens por rodada. Porém, essa estratégia compromete o tempo e o número de rodadas para efetuar tal construção. Se na mesma rodada cada um dos módulos envia informações para o seu vizinho sucessor e recebe do seu vizinho antecessor, n mensagens trafegarão na rede e o número total de rodadas para construir a visão global será $(n - 1)$. Como discutido anteriormente, a organização em anel é vulnerável à ocorrência de particionamento no caso de defeitos em um ou mais módulos ou simples quebra do canal de comunicação.

Para atingir, de forma isolada, o segundo objetivo (ii), a melhor forma é utilizar a comunicação *todos-para-todos,* descrita na seção 3.1, pois a visão global é alcançada em apenas uma rodada. Porém, o número de mensagens trocadas cresce exponencialmente com o número de módulos, o que é impraticável para sistemas de larga escala.

Para tentar equilibrar os dois objetivos, a idéia é disseminar informações, usando a estratégia *Heartbeat* ou a *gossip*. Para utilizar essas estratégias, os módulos passam a ter, além da lista de todos os demais módulos, uma lista de vizinhos que pode ser um subconjunto do total de módulos; essas listas formarão a topologia de comunicação e disseminação de informações no sistema.

Na estratégia *Heartbeat*, cada monitor envia periodicamente para todos os seus vizinhos a sua lista de entidades que está monitorando. Se n é o número de monitores do sistema e cada um deles possui v vizinhos, a complexidade é dada por nv mensagens na primeira rodada, e as demais rodadas são dependentes da configuração dos vizinhos. Modelando a topologia em um grafo, segundo os vizinhos de cada monitor, o número mínimo de rodadas para construir a visão global é igual ao diâmetro do grafo.

Na estratégia *gossip*, a diferença é que, de tempos em tempos, cada monitor envia a lista das máquinas que está monitorando apenas para um único módulo da sua lista de vizinhos. A escolha do módulo destinatário é feita aleatoriamente. O número de mensagens que trafegam na rede de comunicação por rodada é fixo em n, independente do número de vizinhos que cada módulo possui. Porém, como a escolha entre os vizinhos é aleatória, a obteção de uma visão global se torna um fator probabilístico e, geralmente, o número de rodadas para atingi-la é superior à estratégia *Heartbeat*.

---

[7] Uma rodada é um passo de execução do algoritmo, no qual os módulos que devem fazê-lo enviam e recebem as mensagens, de acordo com a estratégia escolhida.

Tanto na estratégia *Heartbeat* quanto na *gossip* o número de mensagens trocadas e o número de rodadas para construir a visão global são dependentes do número e do conjunto que formam os vizinhos de cada módulo. A vantagem dessas abordagens em relação à organização em anel é uma maior susceptibilidade a defeitos de módulos, ocasionando menos particionamento, além do que, podem ser independentes do conhecimento de topologia.

Em Van Renesse, Minsky e Hayden (1998), é feita uma análise também baseada em rodadas, porém, em cada rodada, apenas um dos monitores do sistema, também, escolhido aleatoriamente, dissemina suas informações para um de seus vizinhos. Um monitor que recebe uma nova informação é chamado de *infectado*. Por análise, inicialmente um monitor está infectado, e ele é a única origem da infecção do sistema, embora na realidade novas informações possam ser introduzidas em múltiplos monitores de uma só vez. A análise é feita sob duas frentes: uma calculando a probabilidade de crescimento do número de monitores infectados, por rodada; e outra calculando o tempo de detecção de defeitos em monitores conforme o incremento do número desses monitores no sistema.

A Tabela 3.1 foi construída com base na análise do número de mensagens envolvidas na disseminação de informações sobre detecção entre todas as entidades participantes, objetivando construir uma visão global em cada uma delas. Na tabela, *n* representa o número total de entidades envolvidas. Para a estratégia Hierárquica, foi analisada a complexidade com base em dois níveis, onde *c* (*cluster*) representa o número de grupos (LANs) do segundo nível. Além disso, nos níveis da estratégia Hierárquica é utilizada a comuicação *todos-para-todos* para fazer o ciclo de detecção e disseminação. Na estratégia *Heartbeat*, *v* representa o número médio de vizinhos por entidade. A escolha e o número de vizinhos para as estratégias *Heartbeat* e *gossip* influenciam no número de passos para atingir a visão global. Alguns dados não aparecem na Tabela 3.1 pois em algumas estratégias não é possível saber *a priori* o número de passos necessários e a complexidade final para atingir a visão global.

Tabela 3.1: Complexidade da troca de mensagens para formar uma visão global

Estratégias	**N° passos**	**Mensagens por passo**		**Total de mensagens**	**Complexidade**
Todos-para-todos	1	*n(n –1)*		$n^2 - n$	$O(n^2)$
Hierárquico em dois níveis (LAN e WAN)	3	1°	*c(n/c(n/c – 1))*	$n^2/c + c^2 - 2n$	-
		2°	*c(c – 1)*		
		3°	*c(n/c – 1)*		
Anél lógico	*(n – 1)*	*n*		$n^2 - n$	$O(n^2)$
Heartbeat	-	1°	*n(v)*	Dependente do conjunto de vizinhos	-
		Nos demais passos	Dependente do conjunto de vizinhos		
Gossip	-	*n(1)*		Dependente do conjunto de vizinhos	-

A escolha da estratégia de disseminação deve considerar o tipo de aplicações atendidas, o uso que se fará da visão global construída e sua taxa de atualização.

3.8 Conclusões Parciais

Este capítulo descreveu brevemente as principais estratégias ou protocolos, aplicáveis à detecção de defeitos e/ou à disseminação de informações sobre detecção entre as entidades envolvidas (módulos detectores). Quando possível, analisou-se o número de mensagens trocadas e o de rodadas necessárias para construir uma visão global. Foram vistos os principais artifícios empregados para sobrepor algumas dificuldades em aplicar a detecção em diferentes ambientes. A análise das estratégias de detecção e disseminação serve de apoio à proposta de um serviço de detecção de defeitos (arquitetura e funcionamento) encapsulado, independentemente da aplicação, a ser visto no próximo capítulo.

4 SERVIÇO DE DETECÇÃO DE DEFEITOS

Este capítulo apresenta o serviço de detecção de defeitos e algumas de suas peculiaridades de funcionamento, arquitetura e utilização. Na seção 4.1, são descritas algumas vantagens em oferecer a detecção de defeitos como um serviço. Na seção 4.2, são relacionadas algumas aplicações em potencial que podem fazer uso do serviço. Na seção 4.3, são elencadas algumas questões que surgem quando um serviço de detecção deve atender sistemas de larga escala e operar em redes de longa distância. Na seção 4.4, são apresentados dois modos de operação de um serviço de detecção e as particularidades de cada um. Na seção 4.5, é descrita a arquitetura do AFDService (NUNES, 2003) e, baseado nessa arquitetura, na seção 4.6, é proposta uma nova arquitetura para um serviço de detecção de defeitos. Finalmente, na seção 4.7, são apresentadas as conclusões parciais deste capítulo.

4.1 Detecção de Defeitos como um Serviço

Diversos mecanismos vêm sendo desenvolvidos para garantir tolerância a falhas sob uma variedade de ambientes. Porém, na grande maioria dos casos, a implementação desses mecanismos fica a cargo da aplicação, com pouco ou nenhum suporte do sistema operacional, o que aumenta consideravelmente a complexidade do desenvolvimento de tais aplicações. De fato, um dos principais problemas encontrados no desenvolvimento de aplicações distribuídas robustas reside justamente na dificuldade introduzida pela necessidade de se tolerar e tratar defeitos (CRISTIAN, 1991 *apud* VASCONCELOS, 1997).

Diferentes tipos de especificações podem ser priorizadas no desenvolvimento de detectores de defeitos, fazendo com que comumente o detector seja fundido à aplicação cliente, potencialmente aumentando o seu desempenho (SERGENT; DÉFAGO; SCHIPER, 1999). Em contrapartida, essa fusão pode contribuir para aumentar a complexidade de projeto, elevando o custo de desenvolvimento da aplicação que, agora, deve agregar o protocolo de detecção na sua implementação.

Uma alternativa que visa a diminuir tal complexidade é encapsular o detector de defeitos em um serviço, geralmente oferecido na camada *middleware* (VAN RENESSE; MINSKY; HAYDEN, 1998). Isto permite que o detector possa ser reutilizado por várias aplicações sem necessidade de reimplementá-lo, oferecendo às aplicações, de forma transparente, informações sobre detecção. Caso o detector ofereça possibilidades de configuração e seja flexível, pode ser adaptado aos mais variados ambientes e aplicações sem a necessidade de alteração de código.

Com o objetivo de oferecer suporte aos desenvolvedores de aplicações, uma camada de *middleware* é inserida entre o sistema operacional e a aplicação distribuída. Essa

camada provê uma abstração das funcionalidades do serviço detecção, e seu principal objetivo é possibilitar a comunicação entre componentes distribuídos, escondendo das aplicações a complexidade do ambiente de rede subjacente e livrando-as da manipulação explícita de protocolos e serviços de infra-estrutura. O *middleware* oferece aos desenvolvedores um alto nível de abstração dos serviços relacionados à distribuição, acelera o desenvolvimento e a implantação de aplicações e permite que eles se concentrem nos requisitos funcionais de seus projetos, ou seja, na lógica de suas aplicações (DELICATO, 2005).

Oferecer a detecção de defeitos, sob forma de um serviço, corresponde a definir um conjunto de primitivas (operações) que o serviço (localizado em uma camada) vai oferecer para as aplicações (camada acima). Dessa forma, é necessário definir a interface que as aplicações irão utilizar para fazer uso das funcionalidades do serviço. Isso permite também que mais de uma aplicação utilize o serviço simultaneamente.

Oferecer a detecção como um serviço não é uma idéia totalmente inovadora. Além do trabalho de Van Renesse, Minsky e Hayden (1998), outros, como, Stelling *et al*. (1998), Felber *et al*. (1999), Nunes (2003) e Hayashibara *et al*. (2004), também o fizeram. Hayashibara (2004) sugere que, mesmo a longo prazo, seria ideal definir e implementar a detecção de defeitos na forma de um serviço genérico para sistemas distribuídos de larga escala, a exemplo do que ocorre com serviços de rede genéricos, tais como o DNS (*Domain Name Service*), o NIS (*Network Information System*), o NFS (*Network File System*), o Sendmail, entre outros.

4.2 Aplicação de um Serviço de Detecção de Defeitos

A detecção de defeitos, mesmo na forma de um serviço, pode ser usada como base no projeto de algoritmos distribuídos que dependem de alguma forma de acordo (consenso). Atingir o consenso é um dos problemas fundamentais da computação distribuída. Além desse problema, outros protocolos como, o de difusão atômica (*atomic broadcast*), o de eleição de líder (*leader election*) e o de *membership*, necessitam informações de estado dos processos envolvidos.

Esses protocolos de acordo, geralmente, são usados como blocos básicos para a construção de outros algoritmos, serviços ou aplicações distribuídas. Utilizar o consenso é bastante comum em algoritmos de processamento de dados, gerenciamento de arquivos distribuídos e aplicações distribuídas tolerantes a falhas. Uma aplicação usual de consenso se dá nos protocolos de confirmação atômica (*atomic commit*) para a manipulação de banco de dados distribuídos. Em um nível mais alto, podem ser citadas aplicações que envolvem redes de sensores distribuídos, aplicações que gerenciam recursos distribuídos e/ou compartilhados, aplicações específicas para banco de dados que mantêm informações redundantes e consistência entre bases distribuídas, dentre outras.

Dois exemplos de cenários onde a detecção mostra a sua importância são citados em Dahlgren (2004). Um deles é uma aplicação de conferência privada (*multi-user media conference application*), e o outro é a monitoração de um grupo de recursos compartilhados em vários computadores. Como os computadores podem ser organizados tanto em LANs quanto em WANs, a detecção de defeitos é de extrema importância para ambos os exemplos. As aplicações necessitam de algum tipo de componente de detecção de defeitos (serviço). Esse componente é responsável por detectar defeitos rapidamente e com boa precisão, bem como oferecer informações de

disponibilidade para a camada superior da aplicação. A aplicação pode, então, reagir às ocorrências, tomando as providências para as quais esteja programada.

Outro exemplo de aplicação, citada em Sampaio, Brito e Oliveira (2003), é a de um serviço WWW distribuído de comércio eletrônico de uma grande empresa. A necessidade de tolerar defeitos no servidor WEB pode ser abordada, usando redundância, como a replicação primário-cópia (ou replicação passiva), por exemplo, que, conseqüentemente, gera a necessidade de um serviço de detecção que monitore o servidor primário e ative o servidor cópia.

Os exemplos de cenários e de utilização de serviços de detecção de defeitos encontrados são, em sua maioria, voltados a sistemas de pequena escala. Os detectores de defeitos têm sido utilizados amplamente em ajustes de protocolos de comunicação de redes, comunicação de grupo e gerenciamento de *cluster* (CHEN; TOUEG; AGUILERA, 2002). Como o interesse maior deste trabalho é voltado a sistemas distribuídos de larga escala, com muitos participantes interligados por redes de longas distâncias, as aplicações com tais características são aquelas que precisam fazer o gerenciamento de muitos recursos distribuídos e/ou compartilhados, como aplicações em *grids*, *Web Services*, *e-Business* (HAYASHIBARA, 2004) e aplicações relacionadas às redes de sensores distribuídos.

Na área de sensores distribuídos figuram as redes de sensores sem fio (RSSFs) e as redes móveis *ad hoc* (MANET – *Mobile Ad hoc Network*). As RSSFs podem ser vistas como um tipo especial de MANET. Em uma RSSF tradicional, a comunicação entre os elementos computacionais é feita através de estações-base de rádio, que constituem uma infra-estrutura de comunicação. Esse é o caso da Internet. Por sua vez, em uma MANET os elementos computacionais trocam dados diretamente entre si (LOUREIRO *et al.*, 2003).

Do ponto de vista organizacional, RSSFs e MANETs são idênticas, já que possuem elementos que se comunicam diretamente entre si através de enlaces de comunicação sem fio. No entanto, as MANETs têm como função básica prover um suporte à comunicação entre esses elementos computacionais que, individualmente, podem estar executando tarefas distintas. Por outro lado, as RSSFs tendem a executar uma função colaborativa pela qual os elementos (sensores) provêem dados, que são coletados e processados (ou consumidos) para o interesse de uma determinada aplicação (LOUREIRO *et al.*, 2003).

As RSSFs e as MANETs diferem ainda de redes de computadores tradicionais em outros aspectos. Normalmente possuem um grande número de nodos distribuídos, têm restrições quanto ao consumo de energia e devem possuir mecanismos de auto-configuração e adaptação, devido a problemas como falhas de comunicação e perda de nodos. O elevado número de nodos e a elevada taxa de defeitos são os principais fatores que as tornam potenciais estruturas para a utilização do serviço de detecção de defeitos, principalmente quando informações sobre os estados dos nodos têm importância para as aplicações que as estejam utilizando (LOUREIRO *et al.*, 2003).

As aplicações potenciais para esses tipos de redes podem ser categorizadas nas áreas militar, ambiental, de saúde, doméstica e comercial. É possível expandir essa classificação com categorias adicionais, como a exploração espacial, o processamento químico e o socorro a desastres (DELICATO, 2005). Na literatura da área, são encontrados diversos exemplos de aplicações e projetos relacionados (AKYILDIZ *et al.*, 2002; XU, 2002).

4.3 Desafios de um Serviço de Detecção de Defeitos

As redes de longa distância e os sistemas de larga escala possuem alto nível de assincronismo, atrasos longos e variáveis, alta taxa de perda de mensagens e topologia dinamicamente mutável. Tais características exigem ajustes ou redefinições nos detectores para atenderem mais adequadamente os requisitos do sistema e a demanda das aplicações.

Quando detectores de defeitos são implementados para operarem em redes de longa distância, possivelmente oferecendo serviço a sistemas de larga escala, como, por exemplo, para *grids*, deve-se observar alguns aspectos (STELLING *et al*., 1998; HAYASHIBARA; CHERIF; KATAYAMA, 2002), tais como:

- **Explosão do número de mensagens:** apesar de os componentes, a serem monitorados, estarem distribuídos no sistema, o serviço deve evitar a sobrecarrega da rede de comunicação com mensagens de detecção e controle;
- **Perda e atraso de mensagens:** em redes de longa distância, a taxa de perda de mensagens é elevada devido a ruídos e atenuações; além disso, as mensagens podem sofrer atrasos arbitrários e longos. O serviço deve prever tais comportamentos e tratá-los da melhor forma possível;
- **Escalabilidade:** como os sistemas de larga escala possuem uma grande quantidade de componentes, o serviço deve ser capaz de monitorá-los sem, no entanto, sofrer degradação significativa no seu desempenho;
- **Flexibilidade**: o serviço deve ser capaz de adaptar parâmetros para atender a diferentes tipos ou classes de aplicações;
- **Dinamismo:** o serviço deve permitir a inserção e a remoção de componentes no sistema a qualquer tempo (dinamicamente);
- **Segurança:** deve-se evitar que o serviço venha a se comportar de maneira inadequada, comprometendo a segurança do sistema.
- **Abrangência e precisão:** o serviço deve ser capaz de atender requisitos de abrangência e precisão, para evitar tanto os falsos positivos quanto os falsos negativos no que diz respeito à detecção de defeitos nos componentes.
- **Baixa sobrecarga:** o serviço não deve causar impacto significativo no desempenho da aplicação, na máquina e na rede de comunicação. Os processos devem ser leves em termos de uso de memória, processamento e tráfego de mensagens.
- **Aspectos temporais:** o serviço deve identificar problemas em tempo hábil para que atitudes corretivas possam ser tomadas assim que possível.

Com base nos aspectos anteriormente citados, a Tabela 4.1 mostra os resultados de uma análise realizada neste trabalho acerca da efetividade ou deficiências das principais alternativas de comunicação e disseminação de informações (mais bem detalhadas no capítulo 3), que poderiam ser empregadas no serviço de detecção de defeitos. Os requisitos de segurança, abrangência e precisão, sobrecarga e os de aspectos temporais não são relacionados na tabela porque dependem de particularidades da implementação.

Tabela 4.1: Comparação entre estratégias

Estratégia	Descrição	Explosão do n° de mensagens	Atraso e perda de mensagens	Escalabilidade	Flexibilidade	Dinamismo
Todos-para-todos	Cada entidade possui um monitor local e este troca informações diretamente com todos os demais.	O modelo *push* permite o controle da periodicidade dos envios de mensagens dos monitoráveis para o monitor. O modelo *pull* permite o controle das requisições dos monitores aos seus monitoráveis. Considerando um mesmo intervalo de monitoração, no *push* são necessárias *n* mensagens, e no *pull* 2*n* mensagens.	Usa *timeouts*. Geralmente os *timeouts* são fixos, mas podem ser dinâmicos ou adaptativos, dependendo da implementação. A estratégia básica não faz o tratamento da perda de mensagens, podendo gerar falsas suspeitas.	Não é escalável devido à complexidade do número de mensagens trocadas crescer de forma exponencial à medida que o número de entidades no sistema aumenta. Complexidade teórica de troca de mensagens é de $O(n^2)$.	O modelo *push* é indicado para aplicações que necessitem notificação imediata. O modelo *pull* é indicado para aplicações que necessitem das informações sob demanda. O modelo *dual* é um misto das duas.	Se o sistema possuir suporte a *broadcast*, basta cada entidade conhecer tal endereço. Caso contrário, todos devem conhecer a todos sendo necessário tratar de entradas e saídas de entidades (*membership*).
Anel (LARREA; ARÉVALO; FERNÁNDEZ, 1999)	A monitoração e troca de informações entre monitores são organizadas em forma de anel lógico.	Número reduzido de mensagens trocadas em comparação com a estratégia *todos-para-todos*.	A perda e o atraso de mensagens podem causar particionamento e falsas suspeitas.	Possui complexidade *O(n)*, pois transmite no máximo *n* mensagens por rodada.	Baixa flexibilidade, pois a topologia é estática e vulnerável a defeitos ou simples quebra do canal de comunicação (*links*).	Restringe a monitoração e a topologia.
Heartbeat (AGUILERA; CHEN; TOUEG, 1997)	Cada monitor envia mensagens para todos os seus vizinhos.	O número de vizinhos influencia no número de mensagens trocadas.	Utiliza lista com variações de tempo de recebimento de mensagens, a qual é usada para inferir sobre suspeitas.	A complexidade dependente do número de vizinhos de cada monitor.	Cada monitor necessita apenas conhecer seus vizinhos.	A topologia pode ser dinâmica.
Configuração hierárquica (FELBER *et al.,* 1999)	Monitores de uma mesma rede local conhecem todos os componentes monitorados dessa. Um ou mais monitores podem transmitir informações sobre todos ou algum componente que esteja sendo monitorado para outros monitores de outras redes locais.	Monitores somente monitoram componentes da mesma rede local. A configuração hierárquica diminui o número de mensagens trocadas entre máquinas distantes e entre máquinas pertencentes a diferentes níveis da hierarquia. A comunicação entre monitores de diferentes níveis é feita sob solicitação.	Pode usar *timeouts* distintos para cada nível da hierarquia. Possui maior atraso de detecção entre máquinas distantes e entre hierarquias devido ao atraso das mensagens. Pode gerar falsas suspeitas com a perda de mensagens.	Menor troca de mensagens que abordagem tradicional *todos-para-todos*, por isso é mais escalável. Dentro de um nível, trafega a maioria das mensagens de detecção das máquinas pertencentes a este nível. A escalabilidade depende do método usado em cada nível da hierarquia.	Pode usar distintos modelos de detecção nos níveis da hierarquia (como por exemplo, o *push, pull* ou *dual*).	Topologia fixa adotada no trabalho de Felber *et al*. (1999) compromete o dinamismo.
Configuração hierárquica (BERTIER; MARIN; SENS, 2003)	Análogo ao hierárquico de Felber *et al*. (1999), porém possui uma única máquina (líder) responsável pela comunicação entre redes locais.	As mensagens que trafegam entre redes são canalizadas e gerenciadas pelos líderes das redes locais.	A figura de um único monitor líder por rede local concentra a troca de mensagens entre redes. Isso torna os líderes pontos únicos de falhas. As mensagem por necessitarem passar por eles podem sofrer maiores atrasos.	O ponto de saída (comunicação) entre níveis é fixo.	Na implementação de Bertier, Marin e Sens (2003) existe uma camada de adaptação de QoS. É utilizado o modelo de detecção *push* e a margem de segurança é adaptada com uma estimativa de Jacobson.	A topologia pode ser dinâmica, mas os níveis de hierarquia são fixos.
Protocolo *gossip* básico (VAN RENESSE; MINSKY; HAYDEN, 1998)	Cada monitor envia, periodicamente, mensagens para outro, escolhido aleatoriamente dentre os monitores que conhece (vizinhos diretamente conectados).	Reduz o número de envio de mensagens por rodada em relação à comunicação *todos-para-todos*. E o número de mensagens também é reduzido em relação ao *Heartbeat*, pois os monitores enviam mensagens somente para um vizinho por rodada.	Quando ocorre perda de mensagens, o protocolo mostra-se ajustável, causando pouco impacto no tempo de detecção de defeitos. O protocolo utiliza variáveis e parâmetros para tratar atrasos de mensagens.	Mais escalável que a estratégia *todos-para-todos*, pois o número de mensagens trocadas é reduzido, sendo que um monitor troca somente com um vizinho por vez, de forma aleatória. Possui complexidade teórica de *O(n)* mensagens por rodada. A complexidade também é relativa à quantidade de vizinhos que cada membro possui.	Cada monitor que venha a fazer parte do sistema deve possuir a identificação de pelo menos um (vizinho) membro participante. Isso para poder enviar e receber mensagens.	A topologia pode ser dinâmica.
Protocolo *gossip* multi-nível (VAN RENESSE; MINSKY; HAYDEN, 1998)	É uma extensão do protocolo *gossip* básico. Possui conhecimento adicional sobre a topologia e prioriza comunicação entre monitores localizados na mesma rede local.	Menos mensagens trafegando entre redes locais que o protocolo *gossip* básico devido ao uso de uma configuração hierárquica, mais conhecida como multi-nível.	A disseminação de informações entre sub-redes e domínios diferentes é penalizada devido à técnica priorizar a troca entre redes locais.	Mais escalável que o protocolo *gossip* básico. Devido à priorização de troca de mensagens ser feita internamente a cada nível, reduz o número de mensagens em relação ao protocolo *gossip* básico, tornando-se mais escalável.	O número de rodadas necessário para a informação ser disseminada pelo sistema através das sub-redes e domínios é mais elevado que o protocolo *gossip* básico.	Para as análises feitas no trabalho de Van Renesse, Minsky e Hayden (1998) a topologia testada foi estática, mas pode ser dinâmica.
Protocolo *gossip piggyback* (BURNS; GEORGE; WALLACE, 1999)	Procura reduzir a largura de banda utilizada para a disseminação das informações, adicionando estas às mensagens geradas pela aplicação, sempre que isto for possível.	A "carona" que as mensagens de detecção pegam com as mensagens da aplicação visa a reduzir o tráfego na rede de comunicação. Apesar de utilizada no protocolo estilo *gossip*, essa estratégia pode ser utilizada em complemento a outros protocolos.	Beneficia-se de aplicações que fazem retransmissão de mensagens no caso de perda dessas mensagens.	A técnica *piggyback* pode ser associada a qualquer tipo de estratégia, a escalabilidade fica então a cargo da estratégia associada e ao padrão de comunicação da aplicação.	Depende da técnica de detecção à qual é vinculada, porém possuiu complexidade de desenvolvimento mais elevada.	Depende da técnica de detecção à qual é vinculada.
Lazy (FETZER; RAYNAL; TRONEL, 2001)	Utiliza as mensagens que a aplicação troca entre os processos, para fazer a detecção, quando a aplicação não faz comunicação o protocolo envia mensagens próprias de controle e requisição.	A abordagem de utilizar mensagens da aplicação como indicativo de atividade visa a reduzir número de mensagens de detecção e consequentemente o tráfego na rede de comunicação.	Igualmente ao *piggyback,* pode beneficiar-se de aplicações que utilizem retransmissão no caso de perda de mensagens.	Igualmente ao *piggyback* pode ser associado a um tipo de detecção específico, ficando a escalabilidade a cargo desta e também ao padrão de comunicação da aplicação.	Bastante útil para aplicações que efetuem alta troca de mensagens. A eficiência dependente do padrão de comunicação das aplicações.	Depende à qual técnica de detecção é vinculada.

Salvo detalhes de implementações, com a observação das estratégias levantadas, nota-se que, geralmente, essas estratégias utilizam como alternativa uma forma de organização hierárquica e/ou vinculam o detector à comunicação das aplicações. Essas alternativas buscam reduzir o tráfego de mensagens de controle e detecção na rede de comunicação. Quando vinculadas à aplicação, as informações de monitoração e controle podem ser adicionadas às mensagens da aplicação, ou as próprias mensagens da aplicação são utilizadas como indicativo de atividade dos processos ou máquinas.

Utilizar estratégias hierárquicas, dependendo do cenário e dos requisitos da aplicação, pode ser uma alternativa favorável de organização e controle da comunicação entre detectores. Como o serviço proposto neste trabalho visa atender cenários de larga escala sobre redes de longa distância, optou-se por adotar como alternativa a estratégia hierárquica, especificamente aquela baseada em líderes locais, semelhante a do trabalho de Bertier, Marin e Sens (2003). Outra motivação é que, em diferentes níveis da hierarquia, é possível utilizar diferentes algoritmos de detecção com diferentes parâmetros que melhor adaptem o funcionamento do detector de defeitos àquele ambiente e aos requisitos das aplicações.

4.4 Modos de Operação de um Serviço de Detecção de Defeitos

Um serviço de detecção de defeitos pode operar independente das aplicações e continuamente, de modo a propagar as informações sobre as entidades monitoradas para todo o sistema, ou pode ter seu funcionamento (monitoração e disseminação) voltado exclusivamente para as necessidades das aplicações que o estejam usando.

4.4.1 Modo de operação independente das aplicações

Neste modo de operação, o serviço pode monitorar continuamente todas as entidades configuradas que fazem parte do sistema distribuído, independentemente das necessidades e requisições das aplicações. Tradicionalmente, os sistemas de detecção de defeitos provêem informações de suspeitas para todas as entidades participantes e essa disseminação de informações é feita, utilizando uma determinada estratégia, como, *todos-para-todos*, *Heartbeat* ou *gossip*. Isto possibilita que aplicações consultem a qualquer tempo o estado de qualquer entidade monitorada. A desvantagem desse modo de operação é que, mesmo que nenhuma aplicação esteja utilizando o serviço, ele estará continuamente trocando informações entre os módulos, consumindo desnecessariamente recursos de rede, processamento e memória. Neste modelo de disseminação, as estratégias são escolhidas com base no ambiente e no número de entidades envolvidas e, geralmente, são utilizadas para sobrepor algumas limitações, como a escalabilidade, por exemplo, quando o serviço é usado em sistemas de larga escala.

4.4.2 Modo de operação sob demanda das aplicações

A monitoração e propagação de informações sobre as entidades podem ser realizadas de acordo com as necessidades das aplicações. Assim, o serviço monitora diretamente apenas entidades configuradas pela aplicação, e a disseminação de informações é feita somente para as entidades que necessitem de tais informações. Isto evita a transmissão de informações inúteis, salvo as mensagens de controle específicas do serviço. Para o controle e a filtragem de mensagens, pode ser usado um mecanismo de registro, como o *publish/subscribe*, cujas aplicações registram seus interesses por determinadas informações. No trabalho de Hayashibara *et al*., (2005), são discutidas algumas questões

em relação à propagação de informações, fornecidas pelo detector de defeitos *Accrual* em sistemas de larga escala.

4.5 Serviço de Detecção de Defeitos Adaptativo

O Serviço de Detecção de Defeitos Adaptativo, ou AFDService, é um serviço de detecção de defeitos que monitora nodos de um sistema distribuído, e que pode ser configurado para adaptar o seu *timeout* automaticamente, de acordo com observações do comportamento do atraso de comunicação. O serviço foi projetado para oferecer suporte a defeitos do tipo colapso, omissão e temporização.

A arquitetura é considerada configurável, no sentido que permite a configuração de parâmetros de grão fino no detector e, flexível, no sentido que permite a troca, estática ou dinâmica, de algoritmos de detecção, de predição ou de margem de segurança, bem como permite a rápida inclusão de novos algoritmos. Nunes (2003) fez uma implementação, em Java, para o serviço, disponibilizando um algoritmo de detecção estilo *pull*, sete opções de modelos preditivos (LAST, MEAN, WINMEAN, DMA, BROWN e LPF) e três opções de margem de segurança (fixa, variável em função do erro (EP) e variável em função do intervalo de confiança da previsão (IC)).

Segundo Felber *et al*. (1999), utilizar componentes orientados à aplicação permite ao serviço adaptar-se a topologias de redes (hierárquicas ou não), sem modificar a aplicação ou os protocolos distribuídos envolvidos e oferecer um conjunto de métodos para a interação com as aplicações, tendo por finalidade isolar alguns detalhes funcionais.

Nesse contexto, os monitores encapsulam os algoritmos de detecção de defeitos e, por conseguinte, colecionam informações sobre o estado dos componentes monitoráveis da aplicação.

O monitor interage com a aplicação cliente, respondendo às suas solicitações de estado ou notificando a aplicação quando ocorrerem mudanças de estado nos componentes monitoráveis. Conforme ilustra a Figura 4.1, extraída do trabalho de Felber *et al*. (1999), o autor definiu um diagrama de classes que representa um conjunto genérico de interfaces de objetos para possibilitar a aplicabilidade de um serviço de detecção em qualquer contexto. A linha pontilhada enfatiza as interfaces oferecidas à aplicação para interagir com o serviço de detecção de defeitos.

Porém, Nunes, em 2003, observou que tais interfaces restringiam as possibilidades de configuração do serviço de detecção, pois definiam apenas um conjunto mínimo de métodos disponíveis às aplicações. A intenção de Felber *et al.* foi oferecer uma *visão limitada* do serviço ao cliente de modo a esconder da aplicação detalhes funcionais dos detectores. Para fornecer uma *visão mais ampla*, Nunes definiu um novo conjunto de interfaces para tornar o serviço configurável, oferecendo, por exemplo, possibilidade de configuração de estilos de detecção e de algoritmos de predição.

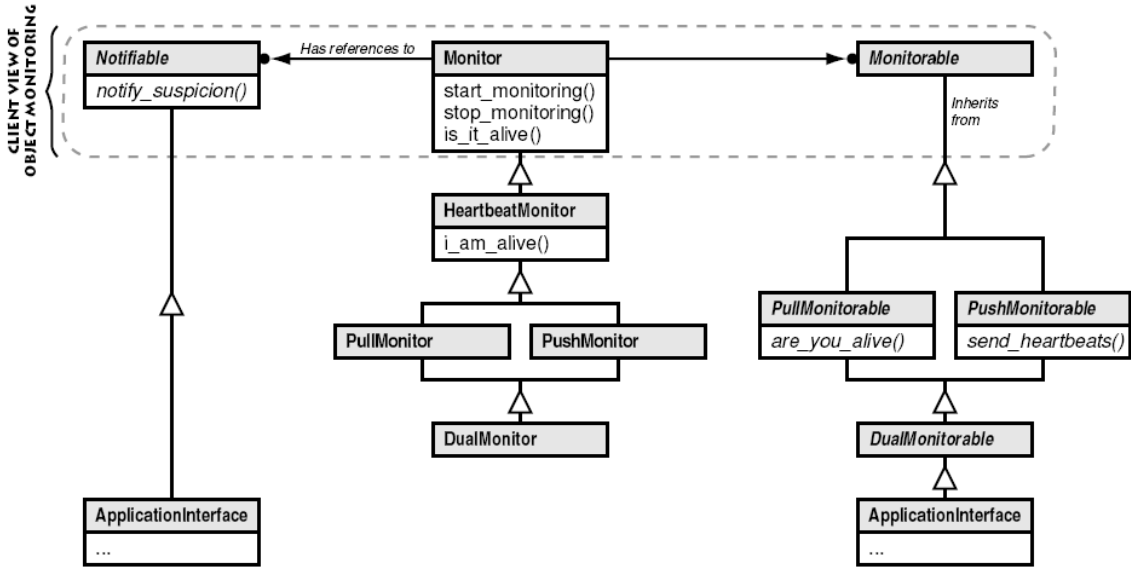

Figura 4.1: Diagrama de classes das interfaces do serviço de monitoramento

A interface alternativa oferecida por Nunes, como ilustra a Figura 4.2, extraída de seu trabalho, altera algumas interfaces orientadas à aplicação, adicionando métodos para a consulta de estado e configuração de parâmetros funcionais do detector. Os métodos de consulta de estados adicionados foram: o método de consulta de um componente específico (`isItAlive()`) e o método de consulta de um conjunto de componentes (`areTheyAlive()`) e; os métodos de configuração adicionados foram: por exemplo, o método de ajuste do *timeout* (`setTimeout()`), o ajuste de intervalo de interrogação ou envio de mensagens (`setCrlMsgInterval()`), entre outros.

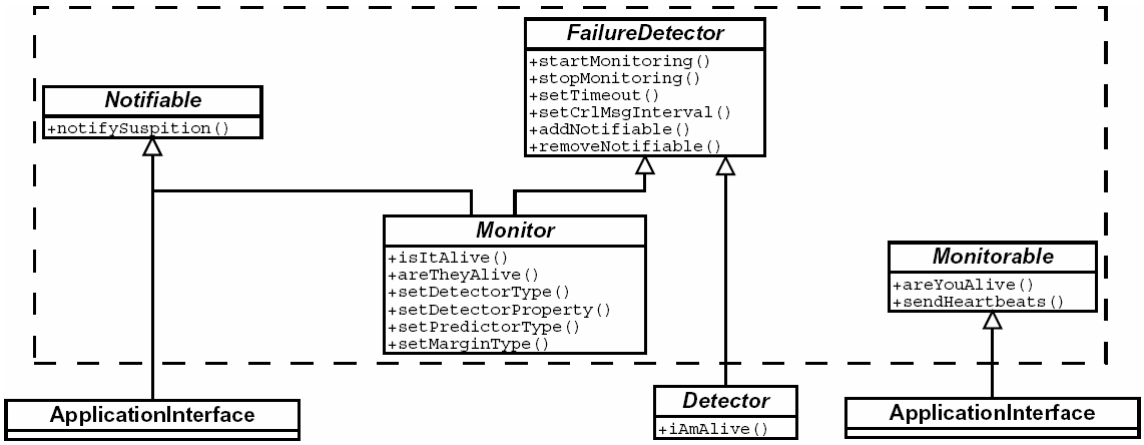

Figura 4.2: Arquitetura de interfaces para um serviço de detecção configurável

Segundo Nunes (2003), algoritmos de predição têm por objetivo prever valores futuros com base em alguma abordagem de previsão, e podem ser dissociados dos algoritmos de detecção, pois, conceitualmente, pertencem a categorias distintas. Com base nessa consideração, o serviço de detecção de Nunes é composto por dois módulos: um módulo de detecção e um módulo de predição que, semanticamente, podem ser representados nos pacotes FD e TS, como ilustra a Figura 4.3.

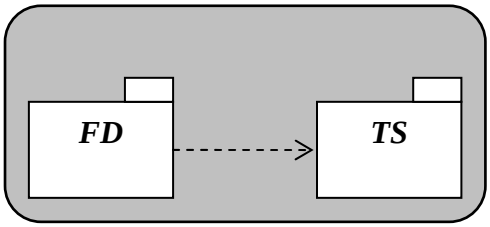

Figura 4.3: AFDService modelado em pacotes

A Figura 4.4 ilustra a arquitetura básica do serviço de detecção composto pelos dois módulos, bem como as relações entre módulos locais do serviço.

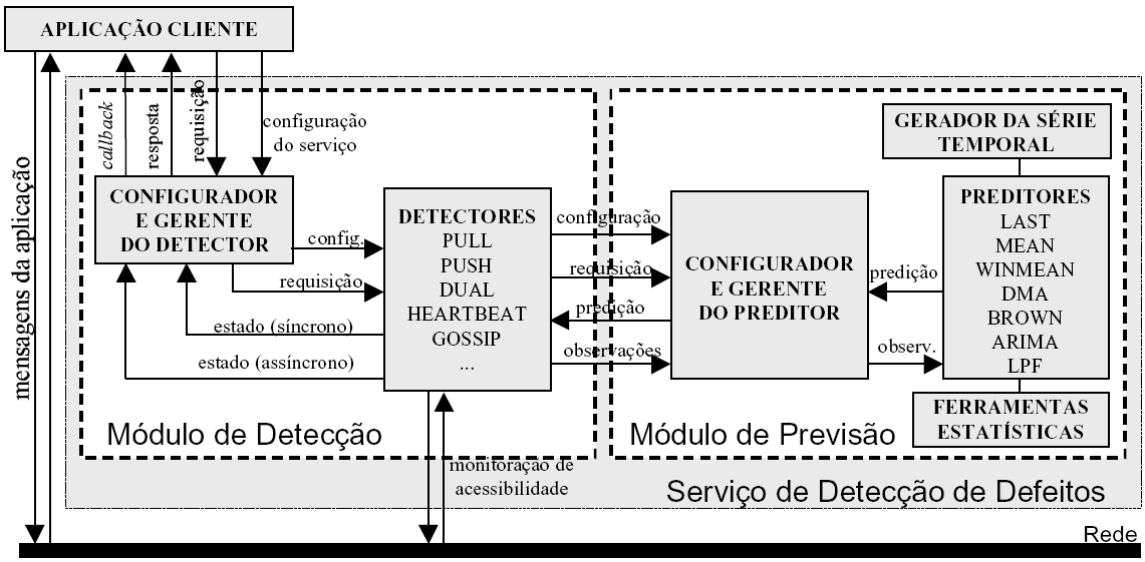

Figura 4.4: Arquitetura do AFDService (NUNES, 2003)

4.6 Proposição de um Modelo para um Serviço de Detecção de Defeitos

Oferecer a detecção de defeitos como um serviço com interface de configuração e acesso (possivelmente para operar em uma camada *middleware*) é uma das principais características que tornam o serviço flexível quanto ao uso e operação. O objetivo do serviço de detecção é monitorar e oferecer informações sobre um conjunto de entidades, especificamente em nível de máquinas, que compõem um sistema distribuído.

A monitoração, para o serviço proposto, é efetuada com base nas necessidades configuradas pelas aplicações que o utilizam, seguindo o modelo de operação descrito na subseção 4.4.2. As aplicações têm a possibilidade de trocarem informações de forma independente do serviço. A interação entre aplicações e serviço é feita também com base na troca de mensagens. Desta forma, as aplicações têm a possibilidade de configurar e fazer requisições ao serviço.

Em uma visão de alto nível, o serviço pode ser visto como sendo formado por três componentes básicos: monitores, monitoráveis e notificáveis, conforme definido por Felber *et al.* (1999) e descrito no capítulo 1. No entanto, no modelo proposto, os componentes notificáveis são as próprias aplicações que utilizam o serviço, e as máquinas que possuem uma instância do serviço assumem, simultaneamente, os papéis de componentes monitoráveis e monitores.

Algumas das principais funcionalidades que um serviço de detecção de defeitos pode oferecer para as aplicações de modo a ser configurável e flexível poderiam incluir as relacionadas a seguir.

- Configuração, adição e remoção de entidades monitoráveis;
- Inicialização e parada de monitoração de entidades monitoráveis;
- Informação de estado de entidades monitoradas (mediante a requisição da aplicação ou na ocorrência de mudança de estado);
- Configuração de parâmetros de detecção (intervalos de requisição e *timeouts*).

Em cada máquina, em que se deseja prover o serviço, o mesmo deve ser ativado, ou seja, executar uma instância do serviço. Em seguida, é iniciado um servidor exclusivo de envio e recebimento de mensagens, vinculado a uma porta de comunicação padrão (pré-estabelecida). Iniciado o serviço, a máquina fica disponível para atender aplicações que se desejam configurar e solicitar a monitoração de uma ou mais entidades pertencentes ao escopo do serviço[8].

O monitor, além de fornecer informações sobre as máquinas que está monitorando, oferece a possibilidade de consultas sobre o estado de máquinas que não estejam diretamente sendo monitoradas por ele, mas possivelmente por outro monitor de outra rede local pertencente ao serviço de detecção. Para isso, é necessária a comunicação com demais entidades ou módulos monitores.

4.6.1 Troca de informações entre monitores

A comunicação entre módulos é um dos principais diferenciais do serviço proposto com relação ao AFDService. Tal comunicação, até então, era feita apenas como uma monitoração de acessibilidade, na qual um módulo detector poderia assumir o papel de monitorado e responder a requisições de outro monitor. A troca de informações entre monitores ainda pode atender a outros dois objetivos: o primeiro, construir uma visão global do sistema (de todas as entidades monitoradas pelo serviço) em todos os monitores e, dessa forma, o serviço opera independente das aplicações (subseção 4.4.1). A construção da visão global ainda pode ser feita, utilizando e/ou adaptando as alternativas de disseminação, descritas no capítulo 3; e o segundo, atender interesses específicos das aplicações (monitoração de máquinas) (subseção 4.4.2). Este segundo objetivo é o modo de operação escolhido para a implementação do serviço proposto neste trabalho.

4.6.2 Organização hierárquica das entidades

Para favorecer a escalabilidade do serviço, decidiu-se utilizar uma configuração lógica hierárquica em dois níveis, independente da topologia física da rede de comunicação. Isso significa que os monitores apenas monitoram diretamente máquinas que pertencem à mesma rede local (mesmo nível). O primeiro nível da hierarquia é formado pelas redes locais (interno) e outro entre redes locais (externo, isto é, WAN), como visto na Figura 3.2.

---

[8] Pertencer ao escopo do serviço significa que a máquina está executando uma instância do serviço de detecção de defeitos, estando apta a monitorar e ser monitorada.

A adoção dessa organização restringe a monitoração direta das máquinas por monitores pertencentes à mesma rede local. A rede local pode ter um ou mais monitores responsáveis pela monitoração das máquinas configuradas pela aplicação. O fato de não necessitarem monitorar todas as máquinas da sua rede local, ou de outras redes locais diretamente, visa, além da economia de processamento e memória, a diminuir o número de mensagens que trafegam na rede de comunicação, favorecendo a escalabilidade do serviço.

Outra particularidade do serviço, baseada no trabalho de Bertier, Marin e Sens (2003), é a adoção de um monitor líder em cada rede local. As comunicações entre redes locais (inter-*cluster*) ocorrem por intermédio dos monitores líderes, de modo que o fluxo de troca de mensagens inter-*cluster* é controlado e centralizado por eles (nível mais alto da hierarquia). Dessa forma, se uma aplicação solicita a um monitor a monitoração de entidades que não estejam localizadas na mesma rede local do monitor, o monitor troca informações com o monitor líder do seu nível; caso não seja ele mesmo, o líder irá delegar a tarefa de monitoração para o líder da rede local onde a máquina de interesse se localiza.

Outra questão que surge é a formação e manutenção de grupos de entidades. O serviço de gerenciamento de *membership* pode ser essencial para adaptar um detector de defeitos de forma transparente a mudanças no sistema. Operações distribuídas, tais como gerenciamento de réplicas, coordenação e disseminação baseada em *gossip,* necessitam que cada membro do grupo mantenha uma lista local dos outros membros, pertencentes aquele grupo, chamada de lista de *membership*. Em um grupo dinâmico, com membros constantemente entrando, saindo e parando por defeito do tipo colapso (*crash*), a lista é mantida e atualizada por um protocolo de gerenciamento de grupo *membership*.

O protocolo de gerenciamento de *membership* auxilia o detector de defeitos, efetuando trocas transparentes de membros entre grupos; procura manter a distribuição dos membros em grupos de forma balanceada, seguindo limiares de mínimo e máximo de membros por grupo, incorporando novos membros no sistema, redistribuindo membros de grupos que excedam o máximo ou fiquem menores que o mínimo (JAIN; SHYAMASUNDAR, 2004).

Ainda no trabalho de Jain e Shyamasundar (2004), a detecção de defeitos é feita através de um mecanismo de *heartbeat*, no qual cada membro envia uma mensagem de *heartbeat* para o líder do seu grupo em intervalos regulares de tempo. Se um líder não receber uma mensagem de um membro em particular dentro de um número máximo de intervalos de espera (MAX_HB_TIMEOUTS), uma suspeita de defeito recai sobre o membro em questão. A diferença para com os métodos tradicionais é que os membros enviam mensagens de *heartbeat* apenas para o líder do grupo ao qual pertencem, de forma que ele gerencia uma tabela de *heartbeats* recebidos de cada membro do seu grupo. Essa tabela tem um intervalo de tempo de espera para o recebimento das mensagens de *heartbeat*. Toda vez que uma mensagem não chega dentro desse intervalo, uma variável é incrementada até um limiar máximo, a partir do qual o membro será considerado com defeito (SUSPECTED).

Porém, na implementação do modelo proposto neste trabalho, a formação de um grupo restringe-se a dois níveis, como no trabalho de Bertier, Marin e Sens (2003), onde cada rede local é um grupo de detecção com um líder local. A topologia lógica é formada manualmente na inicialização do serviço e permanece fixa. Dessa forma, não é

contemplada a manutenção de balanceamento de quantidade de monitores por nível, nem o dinamismo de entrada e saída de monitores.

4.6.3 Gerenciamento de entidades e escolha de líder

Quando o serviço é iniciado em uma máquina, antes de poder monitorar e/ou ser monitorada, ela deve anunciar-se para as demais entidades da rede local, caso existam. Uma maneira simples é efetuar um *broadcast* na rede local para saber se já existe algum outro monitor do serviço executando. Caso não receba a resposta em tempo, ela anuncia-se como líder daquela rede local. Caso receba a resposta de um ou mais monitores ela os adiciona na sua lista de monitores, incluindo o líder. Os monitores que existiam anteriormente atualizam suas listas com a informação da nova entidade. Nota-se que todos os monitores de uma rede local devem possuir uma lista de monitores que permite a todos se conhecerem. Através desse conhecimento, além de trocarem informações entre si, podem e devem monitorar o líder. Isso é útil para possibilitar uma rápida detecção e escolha de um novo líder no caso do atual apresentar defeito, como entrar em colapso.

Detectar defeitos, geralmente, é mais rápido no grupo local que no grupo global. Assim, quando um líder entra em colapso, o grupo local é capaz de detectar e iniciar a escolha de novo líder. Como o algoritmo de escolha de líder pode ser custoso, deve-se evitar executá-lo no caso de falsas suspeitas. Deste modo, a escolha só deve ocorrer quando pelo menos $((n + 1)/2)$ membros do grupo suspeitam do líder.

O algoritmo de escolha de líder funciona da seguinte forma: como todos os monitores conhecem os demais através de suas listas, quando um deles suspeitar de falha do monitor líder, ele notifica a suspeita enviando uma mensagem de *nomeação* para o próximo candidato a líder da lista, segundo uma ordem de identificação. A identificação dos detectores pode ser feita de forma única com base nos endereços IP das máquinas onde executam. Assim, todos podem escolher o próximo monitor candidato, seguindo uma ordem crescente. Não considerando o caso de particionamento, quando um monitor receber mais do que $((n/2) + 1)$ mensagens de *nomeação*, ele se torna líder e envia mensagens de *decisão* aos demais. Os demais, ao receberem essa mensagem, atualizam a identificação do líder. A Figura 4.5 ilustra uma seqüência de funcionamento para eleição do líder. Quando um novo líder é escolhido, todos os membros do grupo global devem ser informados (BERTIER; MARIN; SENS, 2003).

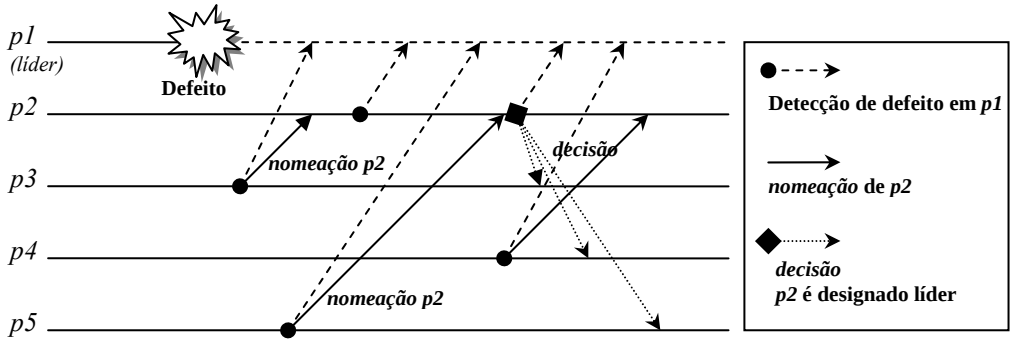

Figura 4.5: Escolha de líder

Um líder, que esteja lento e venha a ser suspeito de estar com defeito pelos demais processos, pode, assim, vir a receber uma mensagem de *nomeação* de um novo líder.

Caso isso ocorra, ele dará início a uma nova eleição, na tentativa de cancelar a atual. Por um curto período de tempo, existirão dois líderes no mesmo grupo. Para limitar este problema, assim que um novo líder entra num grupo global (nível mais alto), os demais líderes deste grupo ignoram as mensagens do líder anterior (BERTIER; MARIN; SENS, 2003).

Outro problema diz respeito de como inserir um novo líder no grupo global (nível mais alto). Após eleito, ele ainda não conhece os outros líderes e não tem como comunicar-se com eles. Para resolver esse problema, pode-se usar um mecanismo de subscrição, o qual permitirá a um novo líder comunicar-se com os demais. Cada novo líder eleito envia, periodicamente, mensagens de *broadcast* para todas as demais redes locais. Quando um líder recebe uma mensagem de registro, ele envia para o novo líder uma mensagem de identificação. Após receber essa mensagem, o novo líder pára de enviar mensagens de *broadcast* para aquela rede local e insere o líder que a enviou na sua lista de conhecidos. Se uma rede local falhar ao responder a mensagem de registro, uma confirmação de colapso naquela rede local é requisitada pelos demais líderes. Esse mecanismo impõe que todas as máquinas do sistema saibam previamente o endereço de *broadcast* de cada rede local (BERTIER; MARIN; SENS, 2003).

4.6.4 Arquitetura do serviço de detecção de defeitos

Tomando-se como base o AFDService de Nunes (2003), resumidamente descrito na seção 4.5, é proposta uma nova arquitetura para um serviço de detecção de defeitos. O diferencial desse novo modelo é agregar a comunicação interna entre módulos detectores, utilizando uma organização hierárquica e o gerenciamento dos grupos que formam a hierarquia, visando a tornar o serviço mais completo.

A proposta da nova arquitetura para um serviço de detecção de defeitos é ilustrada na Figura 4.6. Nesta arquitetura, o serviço é formado por um conjunto de cinco módulos: um de predição, um de detecção, um de comunicação entre detectores, um de gerenciamento de grupo e um módulo gerente de envio/recepção. As funções básicas dos módulos são descritas a seguir:

- **Módulo de predição**: responsável por oferecer algoritmos de predição, que são responsáveis por realizarem previsões de *timeout*, baseados no fluxo de amostragem (observação do atraso de comunicação - RTT). Observando os atrasos de comunicação, esses algoritmos calculam o próximo *timeout*. A previsão pode ser feita após uma requisição explícita, quando ocorrer uma nova observação, ou de acordo com uma política estabelecida.
- **Módulo de detecção**: responsável pela monitoração direta das entidades pertencentes à rede local onde o módulo de detecção está operando. O módulo de detecção efetua a monitoração seguindo um algoritmo de detecção, e este algoritmo pode operar conforme os modelos *push*, *pull* ou *dual*.
- **Módulo de comunicação entre detectores**: responsável por obter informações sobre entidades que as aplicações tenham interesse, mas somente sobre entidades que não pertençam à mesma rede local. Ao atender uma requisição deste tipo, o módulo se comunica com o líder da sua rede local. A partir de então, o líder da rede local comunica-se com o líder da outra rede local, intermediando a troca de informações entre as entidades de interesse. A comunicação entre os detectores (líderes) pode utilizar um algoritmo de disseminação, como o *gossip* ou *Heartbeat,* ou então ser feita sob demanda das aplicações.

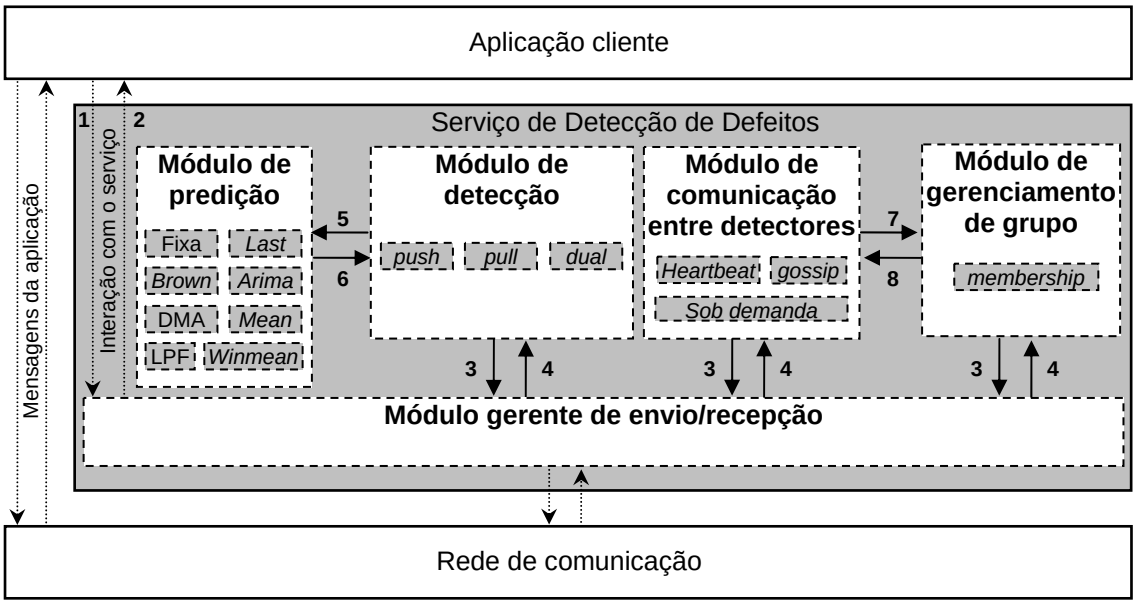

Figura 4.6: Arquitetura do serviço de detecção de defeitos proposto

- **Módulo de gerenciamento de grupo**: responsável pela manutenção das entidades pertencentes à mesma rede local. O módulo é responsável por anunciar a entidade a ser inserida no sistema de detecção, manter atualizada a lista de entidades participantes e identificar o líder da rede local. A monitoração e o gerenciamento (escolha) do líder, em caso de defeito, também são funções deste módulo. A manutenção da visão local dos módulos pode ser feita, utilizando uma estratégia como as discutidas na subseção 4.6.3, ou um mecanismo de *membership*.

- **Módulo gerente de envio/recepção**: representa um servidor de envio e recebimento de mensagens, usado pelo serviço para troca de informações entre as máquinas. Desta forma, mensagens de detecção, de comunicação e de controle utilizam o mesmo servidor de envio e recepção. Esse módulo também é responsável pela interação das aplicações com o serviço. As aplicações enviam mensagens para registro, configurações, consultas e trocam informações sobre os componentes monitorados, de forma temporizada (requisição/resposta) ou não temporizada (*callback*).

A seguir são descritas, de forma resumida, as principais interações entre os módulos que formam o serviço de detecção de defeitos proposto, conforme a numeração indicada na Figura 4.6:

1. **Aplicações para o módulo gerente de envio/recepção (interação com o serviço)**
   1.1. A aplicação registra-se no serviço.
   1.2. Configura o serviço, fornecendo:
      - Listas de entidades a serem monitoradas;
      - Estilo de detecção: *push*, *pull* ou *dual*;
      - Intervalo de monitoração (fluxo de mensagens);
      - Predição de *timeout*: fixa ou algoritmo de predição;
      - Margem de segurança: fixa, EP ou IC.
   1.3. Consulta o estado de entidades que estão sendo monitoradas.
   1.4. Pára a monitoração de entidades.

2. **Módulo gerente de envio/recepção para as aplicações**
   2.1. Informa o estado de entidades consultadas:
      - Temporizado (informa de tempos em tempos);
      - Não temporizado (sob solicitação da aplicação ou na ocorrência de mudança de estado).
3. **Demais módulos para módulo gerente de envio/recepção**
   3.1. Repassam as mensagens a serem encaminhadas pela rede de comunicação.
4. **Módulo gerente de envio/recepção para outros módulos**
   4.1. Repassa as mensagens para os respectivos módulos.
5. **Módulo de detecção para módulo de predição**
   5.1. Configura o tipo de predição de *timeout* e margem de segurança;
   5.2. Envia informações de atraso;
   5.3. Consulta nova previsão.
6. **Módulo de predição para módulo de detecção**
   6.1. Informa sobre nova previsão consultada.
7. **Módulo de comunicação entre detectores para módulo de gerenciamento de grupo**
   7.1. Consulta quem é o líder da rede local.
8. **Módulo de gerenciamento de grupo para módulo de comunicação entre detectores**
   8.1. Informa quem é o líder da rede local:
      - Temporizado (informa de tempos em tempos);
      - Não temporizado (sob solicitação do módulo de comunicação ou na mudança de líder).

Como já mencionado na seção 4.6.1, a comunicação entre módulos do serviço de detecção proposto é feita para a delegação de monitoração de componentes por outros módulos, bem como para a notificação de estados de componentes monitorados. Em oposição à monitoração direta por todos os módulos, esta estratégia de comunicação, constrói uma hierarquia de dois níveis que é responsável por delegar a monitoração de componentes pertencentes a outra rede local para o monitor líder localizado na mesma.

A arquitetura proposta é flexível pelo fato de oferecer independência entre os algoritmos de detecção, os algoritmos de predição, como no AFDServie e, também, por oferecer modularização em relação a comunicação entre módulos detectores e o gerenciamento do grupo.

4.7 Conclusões Parciais

Neste capítulo, foi sugerido oferecer a detecção de defeitos como um serviço, e as principais aplicabilidades do mesmo. Foram relacionados os principais desafios e problemas de implementar um serviço de detecção de defeitos para atender a sistemas de larga escala e que opere em redes de longa distância. Ainda referente a esses problemas foram tabeladas comparativamente as estratégias analisadas. Foi apresentada a arquitetura do AFDService que foi utilizada como base para a proposição de uma nova arquitetura e modelo de funcionamento para a construção de um serviço de detecção. Foi feita a descrição das principais funcionalidades dos módulos do serviço proposto e como esses módulos interagem para fornecer o serviço de detecção às aplicações. O próximo capítulo descreve a implementação de um protótipo da arquitetura proposta e as peculiaridades desta etapa.

5 IMPLEMENTAÇÃO

Este capítulo descreve a implementação do protótipo da arquitetura do serviço proposto na seção 4.6. A seção 5.1 descreve o ambiente de implementação, e a seção 5.2 a arquitetura do *framework* de simulação Neko (URBÁN; DÉFAGO; SCHIPER, 2001). A seguir, a seção 5.3 descreve como foi mapeada a arquitetura proposta para a implementação do protótipo sobre o Neko, suas peculiaridades e o funcionamento de cada camada implementada. A seção 5.4 descreve a estratégia para inferir suspeitas, empregada nos algoritmos de detecção de defeitos implementados (*push* e *pull*). A seção 5.5 descreve a flexibilidade da implementação ao usar o padrão de projeto *Strategy* (GAMMA *et al.*, 1994). Finalizando, a seção 5.6 apresenta as conclusões parciais do capítulo.

5.1 Ambiente de Implementação

Idealmente, o modelo do serviço de detecção proposto na seção 4.6 poderia ser implementado e testado em ambientes distribuídos de larga escala e WANs. Porém, além da complexidade de avaliar um serviço em ambientes reais, não controlados, é importante antes avaliar as funcionalidades e as alternativas adotadas para o funcionamento do mesmo.

A inviabilidade de implementação e teste em ambientes reais forçou a adoção de um simulador. O *framework* de simulação de algoritmos distribuídos Neko (URBÁN; DÉFAGO; SCHIPER, 2001) foi o escolhido, por oferecer suporte à descrição e simulação de cenários e à programação em linguagem de alto nível – Java.

A decisão pelo Neko implicou a escolha da linguagem de programação Java (SUN MICROSYSTEMS, 2006), que tem a seu favor outros fatores, como: simplicidade de expressão, portabilidade, suporte à programação concorrente, ser orientada a objetos, permitir a reutilização de classes desenvolvidas e prover estruturas de alto nível, que facilitam o entendimento do fluxo de execução.

Especificamente para este trabalho, Java foi importante, pois permitiu o reuso do pacote de algoritmos de predição de *timeout* (TS) desenvolvido por Nunes (2003). Uma vez que a análise da implementação é qualitativa, o desempenho absoluto da linguagem não constitui um fator essencial.

5.2 Arquitetura do *framework* Neko

O *framework* Neko permite simular algoritmos distribuídos em uma única máquina ou em várias (numa rede local, por exemplo), usando o mesmo código-fonte. A arquitetura, como ilustra a Figura 5.1, é dividida em duas partes: aplicação e rede. Os

processos são programados em múltiplas camadas hierárquicas, facilitando o entendimento e a modelagem do sistema. As mensagens são enviadas através das camadas por meio do método `send()`, e são entregues através do método `deliver()`.

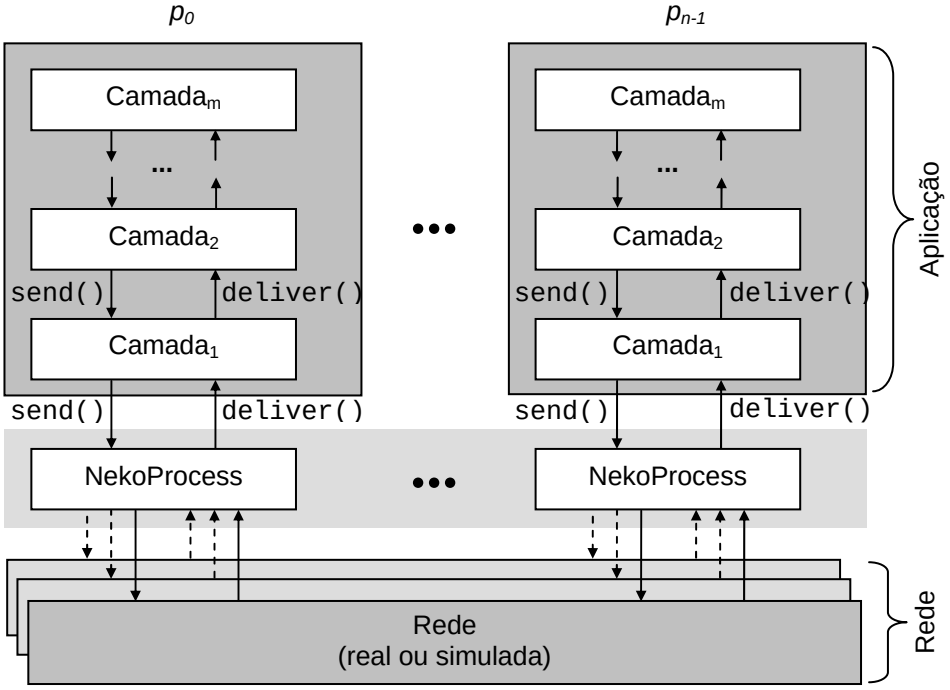

Figura 5.1: Arquitetura do Neko (URBÁN; DÉFAGO; SCHIPER, 2001)

Cada processo de aplicação possui um objeto `NekoProcess` associado, localizado entre as camadas de aplicação e rede. Esse objeto é responsável por enviar e receber mensagens da rede de comunicação, registrar as mensagens no arquivo de registro (*log*), criar a pilha de camadas do processo no início da simulação e manter o identificador do processo ao qual está vinculado. Numa simulação, todos os processos estão localizados em uma única máquina, enquanto que numa execução distribuída (emulação) cada processo pode estar localizado em uma máquina diferente.

A plataforma de comunicação do Neko não é uma caixa preta: a comunicação pode ser controlada de diversas formas. Primeiro, a rede pode ser instanciada a partir de uma coleção predefinida de tipos de redes, tais como uma rede real TCP ou Ethernet simulada. Segundo, o Neko pode gerenciar redes em paralelo. Terceiro, as redes que estendem o *framework* podem ser facilmente programadas e adicionadas.

5.3 Mapeamento da Arquitetura do Serviço para Camadas do Neko

A implementação procura refletir o modelo proposto para o serviço de detecção de defeitos apresentado na seção 4.6, seguindo as funções específicas descritas para cada módulo que compõe a arquitetura (subseção 4.6.4).

O módulo de gerenciamento de grupo, descrito anteriormente, tem como função o gerenciamento dinâmico de entrada e saída das máquinas, pertencentes ao escopo do serviço (*membership*), e, também, é responsável pela manutenção do líder do grupo (monitoração e escolha). Apesar dessas funções influenciarem no desempenho e na escalabilidade finais do serviço, por simplicidade, e para não fugir do objetivo e do escopo do trabalho, foi implementado o gerenciamento para ser feito de forma estática.

A topologia das máquinas (formação de grupos em dois níveis hierárquicos) e informações sobre os líderes são estática e previamente definidas antes da inicialização do serviço. Futuramente, poderão vir a ser desenvolvidas ou mesmo incorporadas soluções já prontas para desempenhar as funcionalidades deste módulo.

Os módulos (e suas respectivas funcionalidades) que compõem o modelo de arquitetura do serviço de detecção de defeitos proposto na Figura 4.6, foram mapeados em camadas, como ilustra a Figura 5.2, de acordo com o modelo de implementação exigido pelo Neko. As camadas estendem a classe `Layer` do Neko, e podem utilizar os métodos `send()` e `deliver()` para enviar e receber mensagens, respectivamente. Vale salientar que a troca de mensagens entre camadas obedece à pilha de formação das camadas, ou seja, se duas camadas não adjacentes necessitarem trocar mensagens, essas mensagens passam pelas camadas intermediárias.

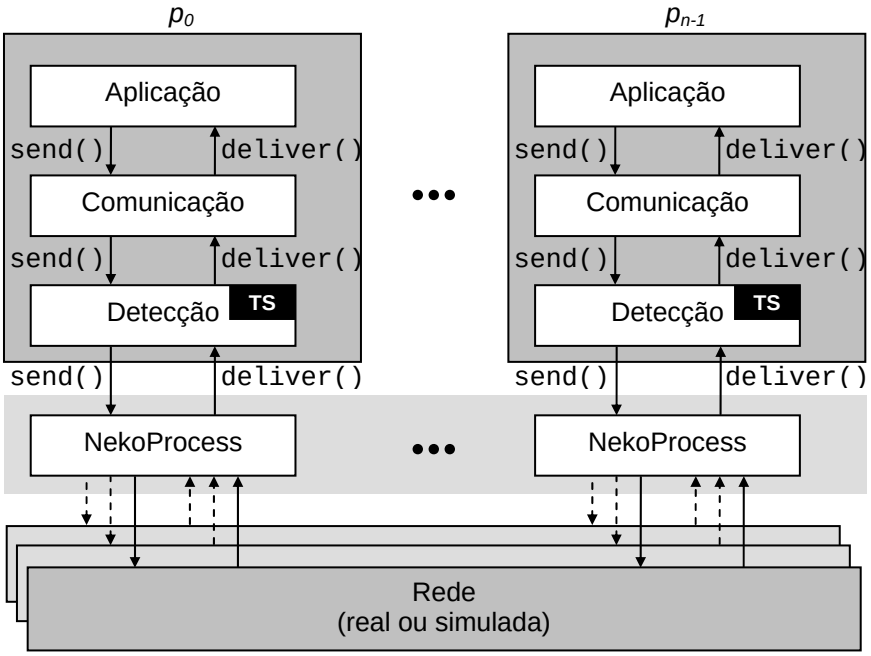

Figura 5.2: Arquitetura em camadas

Analisando o mapeamento da arquitetura em módulos da Figura 4.6 para a arquitetura em camadas da Figura 5.2, as seguintes observações são feitas:

- A aplicação é traduzida em uma camada específica chamada `aplicação`.
- Por não ter sido implementado, o módulo de gerenciamento de grupo foi omitido do mapeamento.
- As funções do módulo de comunicação entre detectores foram mapeadas para serem desempenhadas pela camada `comunicação`.
- As funções do módulo de detecção foram mapeadas para serem desempenhadas pela camada `detecção`.
- As funções do módulo de predição são desempenhadas pelo pacote de predição de *timeout* (TS) de Nunes (2003) e atua na mesma camada `detecção`.
- As funções de comunicação do módulo gerente de envio/recepção são desempenhadas pela camada `NekoProcess`, oferecida pelo Neko.

As próximas subseções descrevem mais detalhadamente as funções e peculiaridades das camadas implementadas.

5.3.1 Camada `aplicação`

A aplicação é implementada como uma camada. As aplicações que usam o serviço podem ser tanto locais quanto distribuídas. Numa execução real, a forma de comunicação entre os processos distribuídos da aplicação é livre, podendo ser por troca de mensagens, memória compartilhada ou outra. O que se exige das aplicações é que elas conheçam o local onde o serviço está sendo executado. No ambiente real, a interação da aplicação com o serviço (comunicação) pode ser feita via *sockets* (UDP ou TCP), usando endereçamento IP. Já no caso da simulação, é preciso apenas conhecer o identificador de processo do serviço. A este serviço é que a aplicação delegará a tarefa de monitoração.

A aplicação não precisa executar na mesma máquina do serviço, pois a interação entre ambos, aplicação e serviço, se dá por troca de mensagens. Cada máquina que deva monitorar ou ser monitorada precisa, necessariamente, executar o serviço. Para uma dada máquina *i*, que tenha uma instância do serviço de detecção de defeitos, o serviço será referenciado como SDD_i. Quanto às aplicações (AP_i), sejam elas locais ou distribuídas, podem ser executadas em qualquer máquina, como ilustra a Figura 5.3.

Na classe `SDDInitializer`, responsável por montar a pilha de camadas, é possível escolher quais processos (máquinas) executarão as aplicações.

No ambiente simulado, a aplicação envia uma mensagem do tipo START para o serviço, contendo uma lista de máquinas a serem monitoradas. Na lista, cada monitorável é associado à aplicação interessada, e são configurados o intervalo de monitoração, o *timeout* inicial (em ms) para a inferência de suspeitas, o algoritmo de predição de *timeout* e o tipo de margem de segurança. As aplicações podem parar a monitoração de uma ou mais máquinas a qualquer tempo, enviando uma mensagem do tipo STOP com a identificação das que não devem ser mais monitoradas.

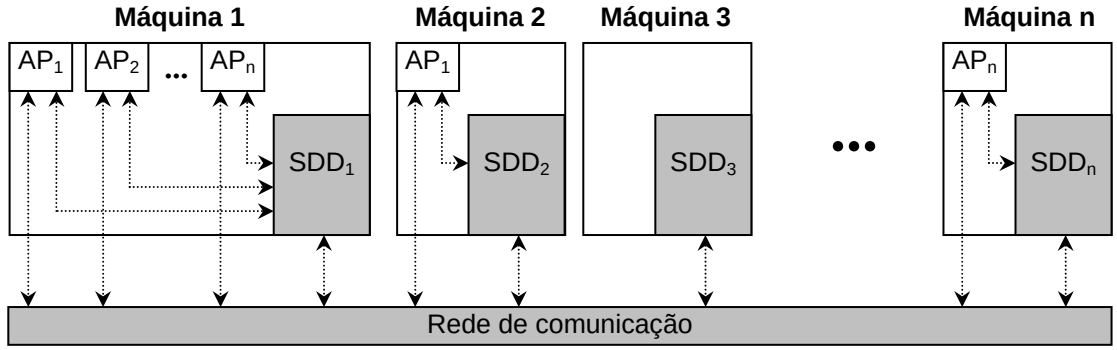

Figura 5.3: Modelo de comunicação e interação com o serviço

Uma aplicação pode escolher um dentre sete algoritmos de predição de *timeout* (LAST, MEAN, WINMEAN, DMA, BROWN, ARIMA e LPF), ou usar um valor fixo. Esses algoritmos vêm do pacote de predição implementado por Nunes (2003). Adicionalmente, o pacote possui três tipos de margens de segurança implementadas (fixa, variável em função do erro de previsão - EP, ou variável em função do intervalo de confiança da previsão – IC), que também podem ser escolhidas pelas aplicações para o ajuste dinâmico do *timeout* de monitoração de cada máquina. O pacote de algoritmos de predição é utilizado na camada `detecção` descrita na subseção 5.3.3.

5.3.2 Camada `comunicação`

Visando a atender o requisito de escalabilidade, foi decidido utilizar uma estratégia hierárquica de comunicação. Para tanto, a camada `comunicação` implementa as funcionalidades do módulo de comunicação entre detectores. Nessa camada, estão definidas as informações sobre topologia e sobre os líderes. Baseada nessas informações, ela delega a monitoração das entidades de interesse da aplicação para os módulos de detecção responsáveis.

A camada `comunicação` também recebe a lista de máquinas e, segundo o conhecimento da topologia, grupos e líderes, intermedia ou delega a tarefa de monitoração para os detectores responsáveis. De acordo com a topologia hierárquica, um detector apenas monitora máquinas da sua rede local. Para as demais, delega a monitoração para o seu próprio líder que, por sua vez, delega para o líder da rede à qual a máquina de interesse pertença. Um caso especial ocorre quando uma aplicação deseja monitorar o líder de outra rede local. Nessa situação, o líder da sua rede irá monitorar o outro diretamente. A camada `comunicação` utiliza mensagens específicas do tipo START_C e STOP_C para delegar, a outros módulos detectores, o início e parada da monitoração de um conjunto de máquinas.

A utilização dessa camada possibilita controlar o tráfego de mensagens entre redes, restringindo-o a mensagens de delegação e informações de alteração de estado ou monitoração entre líderes.

5.3.3 Camada `detecção`

A camada `detecção` implementa um ou mais algoritmos de detecção de defeitos. Para este trabalho, foi implementado o algoritmo que segue o modelo *pull* e o que segue o modelo *push* para monitoração das máquinas.

É na camada `detecção` que é utilizado o pacote de predição de *timeouts* (TS), desenvolvido por Nunes (2003). O pacote TS contempla sete tipos de políticas de ajuste e três opções de margem de segurança. O algoritmo de detecção de defeitos, que atua na camada `detecção`, fornece informações sobre os atrasos de comunicação (RTT) das mensagens de controle e monitoração.

Os algoritmos de detecção *push* e *pull* (`FailureDetectorPush` e `FailureDetectorPull`, respectivamente) implementam a interface `FailureDetector` como ilustrado na Figura 5.4. Eles são simétricos e atuam tanto como monitores quanto como monitoráveis, sendo que as atividades de monitoração são dependentes das requisições das aplicações. A classe `FailureDetector` é uma camada do Neko e também estende a camada `Layer` para o envio e recepção de mensagens.

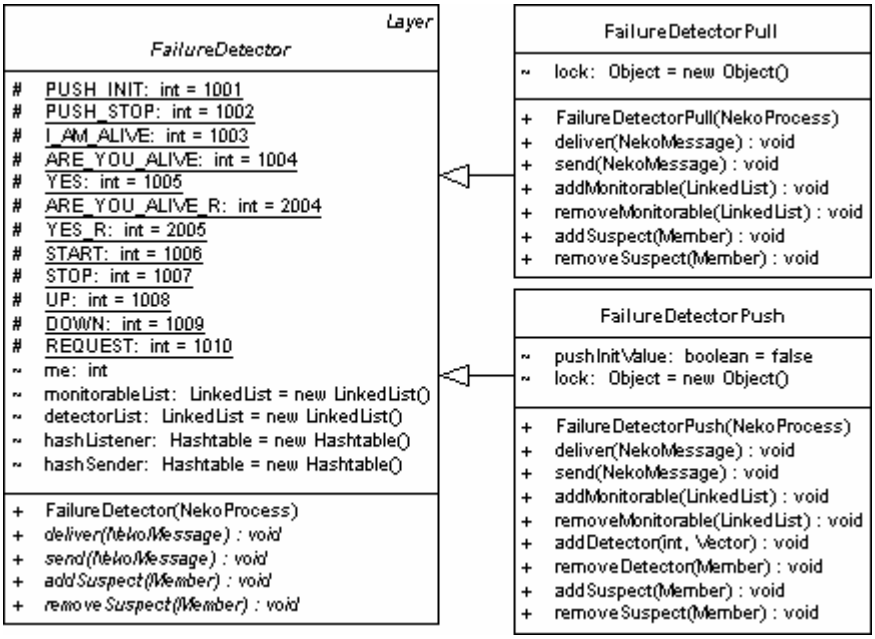

Figura 5.4: Diagrama de classes dos detectores de defeitos *push* e *pull*

Pode-se notar pela Figura 5.4 que os algoritmos de detecção utilizam estruturas do tipo *hash* para gerenciar a lista de instâncias `Listener` e a lista de instâncias `Sender`, associadas às máquinas monitoradas e monitoras, respectivamente.

Um detector, ao iniciar a monitoração de um conjunto de máquinas, inicializa uma classe chamada `Listener` para cada uma delas. Essa classe é responsável pelo gerenciamento de envio e recepção das mensagens de monitoração. A classe `Listener`, por sua vez, instancia outra classe chamada `TimeoutControler`, que estende a classe Java `TimerTask`. A classe `TimeoutControler` nada mais é do que um contador de tempo que, após expirar o tempo configurado, invoca uma função pré-definida, neste caso, a função `addSuspect()`, implementada pelos algoritmos de detecção. Como o nome sugere, `addSuspect()` indica uma suspeita para a aplicação. Quando o monitorado responde dentro do tempo esperado, o contador da classe `TimeoutControler` é reiniciado (`startTimeout()`). Conforme as estratégias do algoritmo de detecção e do *timeout*, são feitas as inferências das suspeitas e a aplicação é avisada na ocorrência de mudança de estado. A mudança de estado compreende a de não suspeito para suspeito e de suspeito para não suspeito, com mensagens do tipo DOWN e UP, respectivamente. As classes `Listener` e `TimeoutControler` são ilustradas na Figura 5.5. A classe `Listener` é abstrata e cada algoritmo de detecção pode estendê-la e modificar as funções para o tratamento de mensagens recebidas.

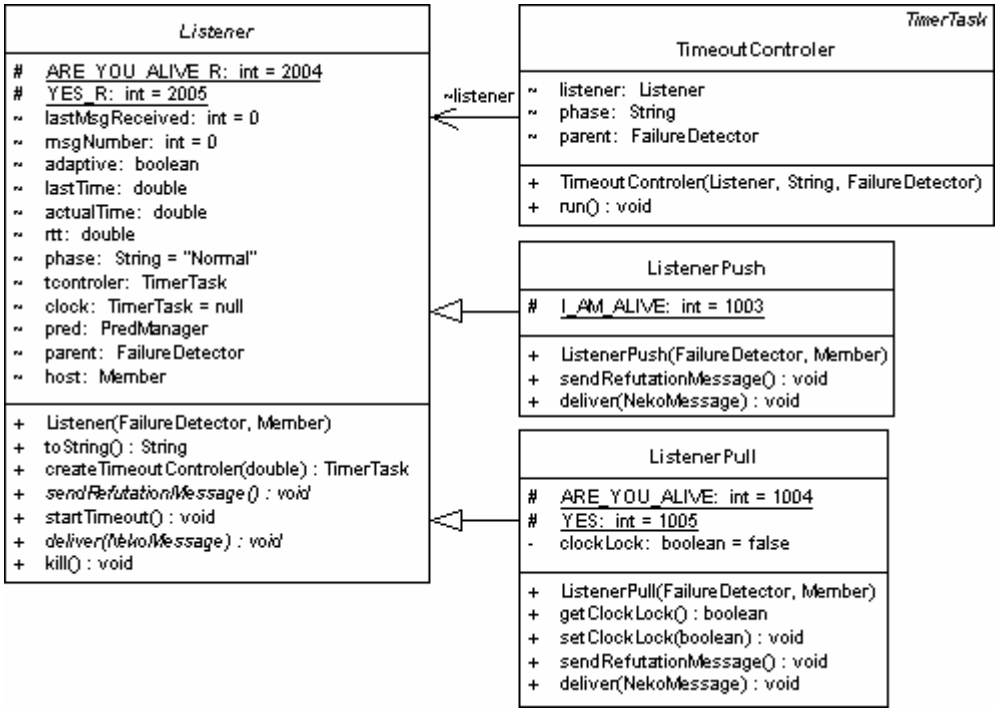

Figura 5.5: Classes `Listener` e `TimeoutControler`

A identificação das máquinas monitoradas e dos parâmetros associados a cada uma é feita por um objeto da classe `Member` ilustrada na Figura 5.6.

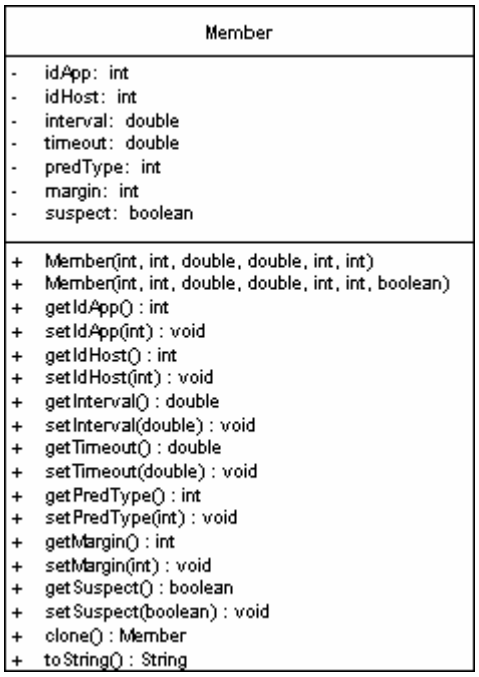

Figura 5.6: Classe `Member`

5.3.3.1 Algoritmo push

Para iniciar a monitoração de uma máquina, o detector envia uma mensagem do tipo PUSH_INIT com os devidos parâmetros, incluindo a taxa de envio de mensagens do tipo I_AM_ALIVE que a máquina monitorada deve enviar. Nesse momento, a máquina monitorada instancia uma classe `Sender` (que estende a classe Java `TimerTask`),

ilustrada na Figura 5.7, para o envio de mensagens para o detector. Uma máquina pode ser monitorada por vários detectores simultaneamente, cada qual com seus parâmetros. Para que essa monitoração possa ocorrer, a máquina monitorada instancia uma classe de envio de mensagens (`Sender`) para cada um dos detectores. Essas classes são periodicamente escalonadas, de acordo com a configuração do intervalo, para enviar as mensagens I_AM_ALIVE.

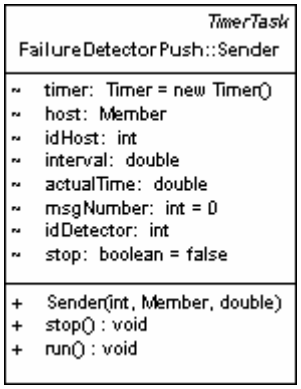

Figura 5.7: Classe `Sender` do *push*

A monitoração feita por um detector estilo *push* é essencialmente um mecanismo que controla o intervalo de recebimento das mensagens das máquinas que estão sendo monitoradas. O controle é feito a partir do seu relógio interno, o que significa que as máquinas monitoradas não sabem como os detectores vão julgar suas mensagens. A inferência sobre suspeitas é disparada toda vez que o detector não receber uma mensagem da máquina monitorada dentro do limite de tempo pré-estabelecido (que considera o *timeout* configurado ou estimado), ou seja, toda vez que o *timeout* estourar a máquina será considerada suspeita.

Uma taxa de envio de mensagens I_AM_ALIVE, configurada com freqüência muito alta, pode acarretar a sobrecarga do sistema de detecção e da rede de comunicação. Para terminar a monitoração (fluxo de mensagens), o detector envia uma mensagem do tipo PUSH_STOP para a máquina que está monitorando.

5.3.3.2 Algoritmo pull

No detector tipo *pull*, as máquinas monitoradas não necessitam guardar informações sobre seus detectores, elas apenas respondem às mensagens de requisição do tipo ARE_YOU_ALIVE com uma mensagem do tipo YES, a qualquer detector que as faça. O controle da monitoração, início, parada e taxa, é feito pelo detector, que usa uma classe `Sender` específica para mandar as mensagens de requisição. A classe é ilustrada na Figura 5.8.

O detector envia as mensagens segundo a taxa de intervalo de requisição configurada pela aplicação, e espera a resposta de acordo com o *timeout*, configurado ou estimado. Não é necessário enviar mensagens de início e parada de monitoração para as máquinas monitoradas, devido aos monitorados serem passivos e o controle ser feito pelo detector.

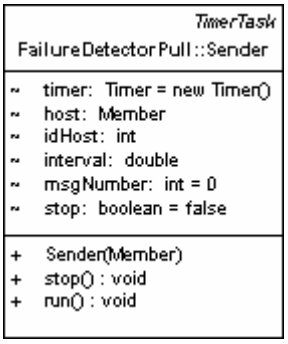

Figura 5.8: Classe `Sender` do *pull*

5.4 Estratégia para Inferência de Suspeita

Em um ambiente real, desconhecer o atraso de comunicação entre as máquinas do sistema distribuído que esteja executando o serviço pode gerar defeitos de temporização e, além disso, mensagens perdidas podem gerar defeitos de omissão. Nunes (2003) propõe o uso de uma estratégia global de adaptação do *timeout* no serviço de detecção de defeitos, estruturada em três fases: normal, refutação e recuperação. O serviço implementado, apesar de executar em um ambiente simulado, contempla a fase normal e a fase de refutação, e infere suspeitas com maior qualidade, evitando as falsas suspeitas. A fase de recuperação não foi implementada, pois somente aumentaria o número de mensagens e a complexidade de análise dos algoritmos.

As três fases[9] são descritas a seguir:

5.4.1 Fase normal

Na fase normal, pode-se ajustar dinamicamente o *timeout*, usando um dos algoritmos configurados. O ajuste baseia-se no atraso de comunicação e na margem de segurança previamente configurada pela aplicação. Essa fase é caracterizada pela ausência de suspeitas. Quando o *timeout* expira, o detector considera que pode ter havido uma perda de mensagem (defeito de omissão) ou um atraso expressivo (defeito de temporização) e entra na fase de refutação.

5.4.2 Fase de refutação

Nessa fase, é reenviada uma mensagem de controle (ARE_YOU_ALIVE_R) à máquina monitorada com a intenção de refutar a possível falsa suspeita. Esse comportamento é sugerido no algoritmo de detecção estilo *dual*, proposto por Felber *et al.* (1999). Se durante essa fase uma mensagem válida de controle for recebida (YES, I_AM_ALIVE ou YES_R), volta-se à fase normal e não é gerada alteração de estado do monitorado. Se o *timeout* dessa fase expirar, o monitorado é considerado suspeito. O uso de refutação aumenta o número de mensagens de controle, mas reduz a probabilidade de uma falsa suspeita, especialmente importante para sistemas que consideram possibilidade de defeitos de omissão e que não desejam *timeouts* com valores muito elevados.

---

[9] Maiores detalhes e a validação da estratégia global de adaptação do *timeout* podem ser encontrados no trabalho de Nunes (2003).

5.4.3 Fase de recuperação

Para garantir a recuperação de um monitorado erroneamente considerado suspeito, o monitor periodicamente requisita o estado de todas as máquinas monitoradas. Deste modo, se um suspeito não estiver realmente com defeito, ele responderá em um tempo finito (considera-se que o atraso de comunicação é desconhecido, mas finito), possibilitando ao monitor a sua recuperação.

5.5 Flexibilidade da Implementação

De acordo com o modelo de sistema distribuído, descrito na seção 2.1, o serviço implementado oferece suporte a defeitos do tipo colapso, omissão e temporização, como no AFDService (Nunes, 2003). Outra característica herdada do trabalho de Nunes (2003) é a implementação de uma interface para os algoritmos de detecção de defeitos, seguindo o padrão de projeto *Strategy* (GAMMA *et al.*, 1994). Isto permite inserir facilmente novos algoritmos.

No padrão *Strategy*, define-se uma interface abstrata que deve ser implementada por estratégias concretas (implementações de algoritmos específicos). São os algoritmos implementados que efetivamente provêem funcionalidades. Além dos algoritmos de detecção seguirem o padrão, como mostrado no diagrama de classes da Figura 5.4, os do pacote de predição também o seguem, facilitando a inserção ou a modificação de algoritmos de detecção, predição e margem de segurança no serviço.

5.6 Conclusões Parciais

Este capítulo apresentou os aspectos de implementação e do mapeamento da arquitetura para o modelo em camadas exigido para implementar sobre o Neko. Foram descritas as principais estratégias utilizadas e implementadas, bem como suas funcionalidades e características. Partindo desta implementação, o capítulo seguinte apresenta os testes de simulação que avaliam as estratégias utilizadas no protótipo de serviço de detecção de defeitos implementado.

6 TESTES E AVALIAÇÃO

Este capítulo apresenta os testes e avaliação da implementação do protótipo do serviço de detecção de defeitos. A seção 6.1 apresenta os principais objetivos que conduziram à construção dos cenários e à realização dos testes sobre esses cenários. A seção 6.2 define o cenário de teste, apresentando as configurações do ambiente de simulação. A seção 6.3 descreve o perfil de dois tipos de aplicações sintéticas criadas, a configuração do conjunto de testes, além de apresentar os resultados obtidos com os testes e a avaliação relativa ao número e tipos de mensagens trocadas. A seção 6.5 avalia a qualidade de detecção do serviço frente a defeitos de omissão e colapso. A seção 6.6 avalia a abordagem dos desafios de implementação descritos na seção 4.3. Por fim, a seção 6.7 apresenta as conclusões parciais do capítulo.

6.1 Objetivos dos Testes de Simulação

Os cenários foram construídos basicamente para a comparação do serviço de detecção de defeitos, operando sob uma organização plana e hierárquica de seus módulos detectores. Pontualmente, os principais objetivos dos testes são relacionados a seguir:

- Testes do funcionamento dos algoritmos de detecção implementados (*push* e *pull*).
- Contabilização das trocas de mensagens efetuadas pelos algoritmos de detecção implementados.
- Comparação entre o número de mensagens de controle sobre o serviço operando sob uma organização plana, com monitoração direta, frente a uma organização hierárquica, com a delegação de tarefas de monitoramento entre líderes.
- Avaliação do serviço frente a defeitos de omissão e colapso.
- De um mundo geral, avaliação das funcionalidades, configurabilidade e flexibilidade do serviço.

6.2 Configurações de Cenários para Testes de Simulação

Visando a simplificar a simulação, os testes foram executados num cenário composto por 15 máquinas distribuídas. Essas máquinas conheciam a localização das demais e estavam aptas a monitorar e serem monitoradas pelo serviço.

A análise das monitorações considerou os dois algoritmos de detecção, *push* e *pull*. Para analisar a estratégia hierárquica, as máquinas foram organizadas logicamente de duas formas: uma plana, sem o conhecimento da topologia, na forma de um grafo

totalmente conectado, e outra hierárquica, organizada em 3 grupos locais (LANs) de 5 máquinas, cada um com seu líder local (L_i). Os líderes locais formam entre si um grupo global (WAN), o segundo nível da hierarquia.

6.2.1 Organização plana (*flat*)

Na organização plana cada máquina pôde comunicar-se diretamente com as demais. Se todas necessitassem monitorar todas (comunicação *todos-para-todos*), elas podem fazê-lo diretamente, mas as mensagens de monitoração e controle podem inundar a rede, se o número de máquinas for muito grande ou mesmo se a taxa de monitoração for muito freqüente. Sem o prévio conhecimento da topologia física, muitas mensagens podem trafegar entre LANs distintas. Esta comunicação tem maior custo e características indesejáveis, tais como maior atraso e maior taxa de perda de mensagens, se comparada com a comunicação local. A Figura 6.1 ilustra uma organização plana do cenário com 15 máquinas.

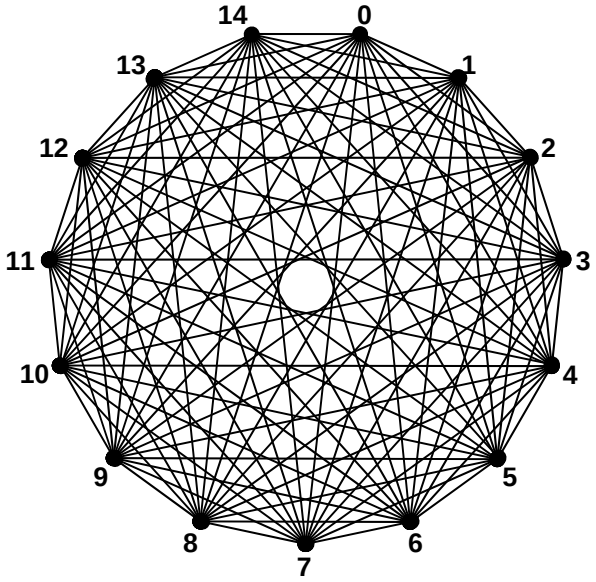

Figura 6.1: Organização plana (*todos-para-todos*)

6.2.2 Organização hierárquica

Na organização hierárquica, as máquinas estão organizadas em grupos. A topologia do cenário foi organizada em dois níveis lógicos, um LAN e outro WAN. Isso implica a possibilidade das mensagens passarem por diferentes níveis administrativos e diferentes *gateways*, *switches* ou *hubs*, porém, logicamente, a comunicação é feita pelo serviço como se estivesse em um mesmo nível hierárquico (ou domínio administrativo). Há 3 grupos locais (LANs), com 5 máquinas cada, e um grupo global (WAN), com 3 máquinas. Cada grupo local possui um líder (L_i), e esses três líderes formam o grupo global. A Figura 6.2 ilustra a organização hierárquica do cenário, identificando as redes locais e seus respectivos líderes.

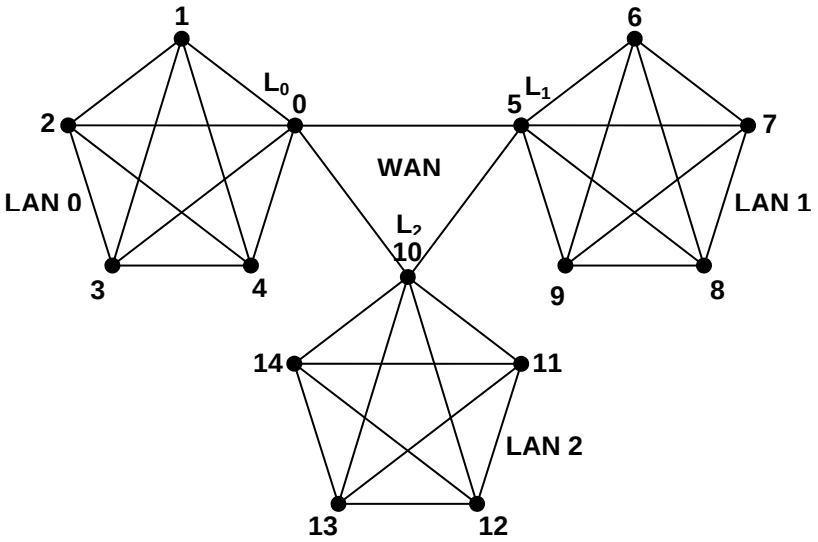

Figura 6.2: Organização hierárquica

6.2.3 Outras características da implementação da simulação e testes

Para aproximar o ambiente simulado de um ambiente real de sistema distribuído, os atrasos de comunicação entre máquinas que estivessem na mesma rede local foram diferenciados dos atrasos entre máquinas que estivessem em redes locais diferentes. Como o Neko permite configurar modelos variados de redes simuladas simultaneamente, foram criadas duas redes de atrasos fixos, uma para comunicação intra-LANs e outra inter-LANs (WAN).

Uma nova camada de simulação, chamada `Topologia`, precisou ser implementada para que fosse possível alternar o uso das duas redes definidas. Essa camada, localizada logo acima da camada `NekoProcess`, como ilustra a Figura 6.3, é responsável por verificar a origem e o destino das mensagens enviadas e, com base na topologia, escolher qual rede seria utilizada para entregá-las.

O serviço foi testado em dois cenários para fins de comparação: um sem o conhecimento da topologia (organização plana), e outro com o conhecimento prévio da topologia e utilizando a organização hierárquica. No primeiro, uma máquina pôde monitorar qualquer outra, e a comunicação estava sujeita a atrasos, conforme a distância entre elas. No segundo, a monitoração foi efetuada de forma direta se o monitor e a máquina estivessem localizados na mesma rede local, e de forma indireta, intermediada pelos líderes, se a máquina de interesse estivesse localizada em outra rede local. Na monitoração indireta, o monitor delega a monitoração ao líder de sua rede local (caso não seja ele próprio) que, por sua vez, delega ao líder onde a máquina de interesse se localiza. A partir daí, o líder, que está monitorando diretamente a máquina, somente envia informações entre redes sobre requisição da aplicação interessada ou na ocorrência de mudança de estado da máquina monitorada.

Um caso especial ocorre quando uma aplicação solicita a um monitor de uma rede local a monitoração de uma máquina de outra rede local, mas, esta máquina em questão, é líder. Nesse caso, a monitoração entre os líderes se dá de forma direta, isto é, mensagens de monitoração trafegam entre as redes. Para alternar os dois cenários, sem e com organização hierárquica, bastou utilizar ou não a camada `comunicação` descrita na subseção 5.3.2.

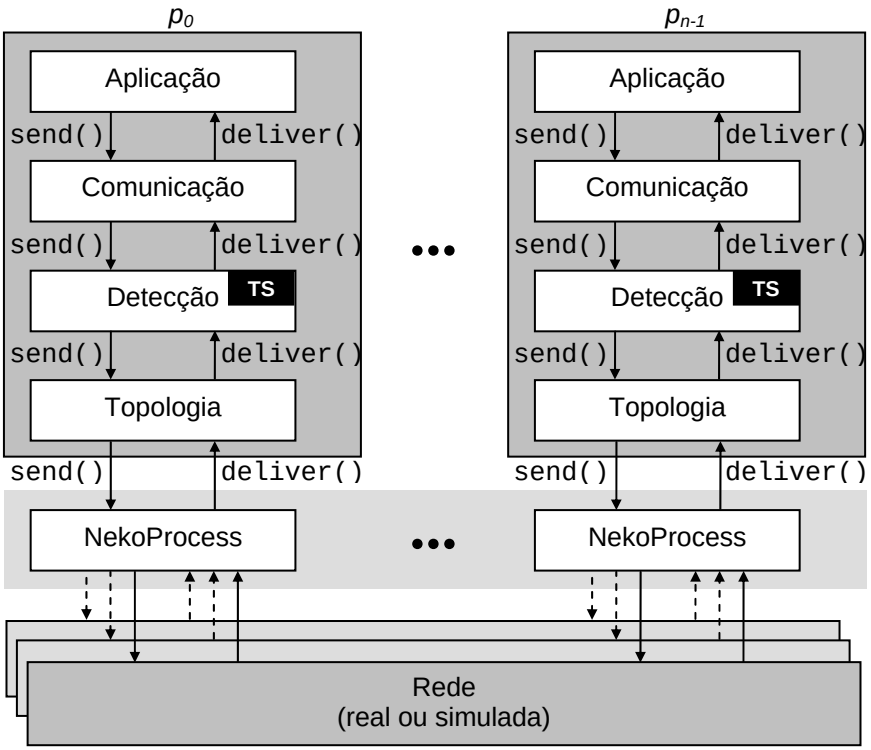

Figura 6.3: Hierarquia das camadas implementadas

6.3 Perfil das Aplicações para Teste e Avaliação

O serviço funciona, segundo o modo de operação, sob demanda das aplicações, como descrita na subseção 4.4.2. Nesse modo de operação, a monitoração somente é iniciada ou encerrada, segundo as requisições das aplicações. Para executar os testes, dois tipos de aplicações (cenários) foram definidos:

- 1) uma aplicação local, localizada em uma máquina, possui interesse em outra máquina remota (de outra LAN) e, solicita ao serviço a monitoração dessa máquina remota (monitoração *um-para-um*). Para efeitos de simulação, a máquina 1 executa uma aplicação que tem interesse em monitorar a máquina 9. As duas estão em LANs diferentes (vide Figura 6.4), o que permite observar saltos (*hops*) e atrasos, conforme é usada a organização plana ou a hierárquica;
- 2) uma aplicação distribuída, abrangendo todas as máquinas, e cada uma delas tem interesse em monitorar as demais (monitoração *todos-para-todos*). Esse segundo cenário caracteriza um ambiente de comunicação intensa, para o qual é importante avaliar a escalabilidade do modelo. Devido à monitoração ser feita em nível de máquina e não de processo, não se considera o caso de uma máquina monitorar a si mesma.

Foram definidas duas redes básicas (`BasicNetwork`) no Neko para a comunicação entre as máquinas. Uma, com atraso fixo de 1 ms, para comunicação local (LAN), e outra com atraso fixo de 5 ms, para comunicação global (WAN), como mostra a Figura 6.4. Por exemplo, se a máquina 1, utilizando a organização plana, deseja enviar uma mensagem para a máquina 9, a mensagem é entregue em 5 ms devido aos atrasos impostos pela rede de comunicação (um salto da máquina 1 para a máquina 9 com atraso de 5 ms, como ilustrado pelo arco tracejado da Figura 6.4). Se for utilizada a

organização hierárquica, a mensagem é entregue em 7 ms (um salto da máquina 1 para a 0 (L_0) com atraso de 1 ms, um salto da 0 para a 5 (L_1) com atraso de 5 ms e um salto da 5 para a 9, com atraso de 1 ms, como ilustrado pelos arcos sólidas da Figura 6.4).

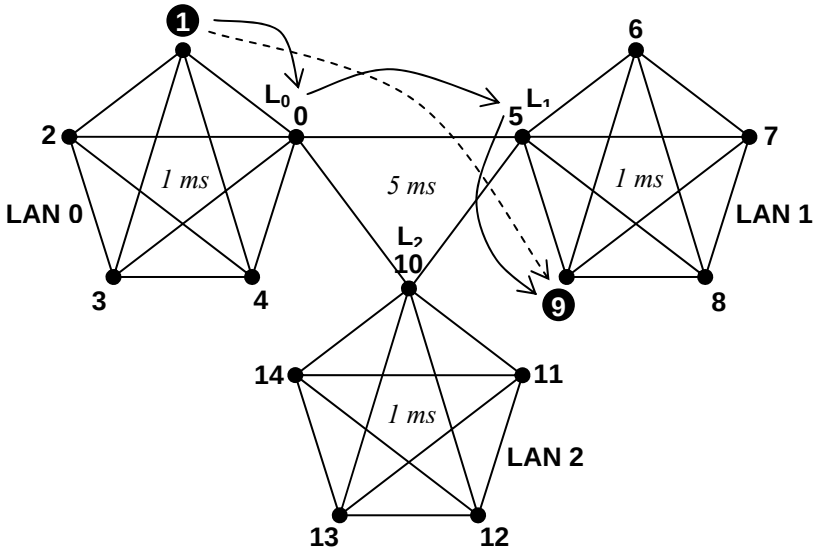

Figura 6.4: Atraso de comunicação

6.4 Avaliação dos Tipos e Número de Mensagens

A simulação dos dois tipos de aplicações, anteriormente definidas, foi executada, segundo a organização plana e segundo a organização hierárquica. Para ambos os casos foram usados, separadamente, os dois algoritmos de detecção, *push* e *pull*. O uso de um e outro permitiu avaliar os tipos e a diferença no número total de mensagens trocas no período. Tem-se, portanto, um total de quatro simulações cujos resultados podem ser observados nas tabelas 6.1, 6.2, 6.3 e 6.4. Nas tabelas de resultados, além dos tipos e quantidades de mensagens geradas, também pode ser observada qual a camada que gerou as mensagens.

Para a simulação, foi considerado um sistema distribuído com ausência de defeitos e perda de mensagens. Os parâmetros de monitoração configurados pela aplicação foram os seguintes:

- **Tempo de monitoração**: 149 ms;
- **Intervalo de monitoração**: 10 ms;
- ***Timeout* inicial**: 10 ms;
- **Algoritmo de predição**: LAST;
- **Margem de segurança**: fixa.

O tempo de monitoração foi escolhido para não ser múltiplo do intervalo de monitoração, apenas para evitar que uma mensagem de monitoração fosse enviada no mesmo instante que o recebimento de uma mensagem de parada de monitoração, para evitar, assim, uma mensagem perdida. O *timeout* inicial foi escolhido para ser maior que o atraso máximo, apenas para não gerar falsa suspeita no início da monitoração. A partir da chegada da primeira mensagem da máquina monitorada, o algoritmo de predição LAST ajusta os próximos *timeouts*, conforme os atrasos das mensagens. Na

implementação do protótipo, o pacote de predição efetuou o cálculo do *timeout* a cada recebimento de mensagens da máquina monitorada.

De acordo com a Tabela 6.1, percebeu-se que o algoritmo *push* totalizou em um menor número de mensagens enviadas durante o período. Teoricamente, o algoritmo enviaria a metade do número de mensagens do *pull*, mas, considerando as mensagens adicionais de controle, PUSH_INIT e PUSH_STOP, o total ultrapassou um pouco a metade. Ambos os algoritmos tiveram duas mensagens adicionais por entrarem uma vez na fase de refutação (ARE_YOU_ALIVE_R e YES_R).

Tabela 6.1: Mensagens *um-para-um* na organização plana

Camada	**Tipo**	**N° de mensagens**	
		push	***pull***
Aplicação	START	1	1
Detecção	PUSH_INIT	1	0
Detecção	I_AM_ALIVE	15	0
Detecção	ARE_YOU_ALIVE	0	15
Detecção	YES	0	15
Detecção	ARE_YOU_ALIVE_R	1	1
Detecção	YES_R	1	1
Aplicação	STOP	1	1
Detecção	PUSH_STOP	1	0
Total de mensagens		**21**	**34**

A mesma configuração de monitoração *um-para-um* aplicada à organização hierárquica, resultou no conjunto de mensagens listadas na Tabela 6.2. Comparativamente à organização plana, o uso da estratégia não causou um incremento significativo, foram apenas quatro mensagens adicionais (duas de START_C e duas de STOP_C). O algoritmo *push*, neste caso, entrou duas vezes na fase de refutação.

Tabela 6.2: Mensagens *um-para-um* na organização hierárquica

Camada	**Tipo**	**N° de mensagens**	
		push	***pull***
Aplicação	START	1	1
Comunicação	START_C	2	2
Detecção	PUSH_INIT	1	0
Detecção	I_AM_ALIVE	15	0
Detecção	ARE_YOU_ALIVE	0	15
Detecção	YES	0	15
Detecção	ARE_YOU_ALIVE_R	2	0
Detecção	YES_R	2	0
Aplicação	STOP	1	1
Comunicação	STOP_C	2	2
Detecção	PUSH_STOP	1	0
Total de mensagens		**27**	**36**

O segundo cenário, em que a monitoração foi do tipo *todos-para-todos*, exercita exaustivamente a comunicação. A Tabela 6.3 e a Tabela 6.4 listam o conjunto de mensagens trocadas, durante a simulação, para um sistema, formado por quinze máquinas.

A monitoração ocorreu em nível de máquina, logo foi descartada a situação de uma máquina monitorar a si própria. Assim, o número de mensagens START, enviadas pela

aplicação é dado por $n(n-1)$, sendo n o número de máquinas. Para o exemplo considerado, onde n é igual a 15, resultaria em 210 mensagens START. Essa quantidade foi constatada tanto na organização plana quanto hierárquica, como mostra a Tabela 6.3 e a Tabela 6.4. Para o cenário de 15 máquinas, cada uma delas monitorou e ao mesmo tempo foram monitoradas pelas demais máquinas.

Tabela 6.3: Mensagens *todos-para-todos* na organização plana

Camada	**Tipo**	**N° de mensagens**	
		push	***pull***
Aplicação	START	210	210
Detecção	PUSH_INIT	210	0
Detecção	I_AM_ALIVE	3150	0
Detecção	ARE_YOU_ALIVE	0	3150
Detecção	YES	0	3150
Detecção	ARE_YOU_ALIVE_R	120	150
Detecção	YES_R	120	150
Aplicação	STOP	210	210
Detecção	PUSH_STOP	210	0
Total de mensagens		**4230**	**7020**

Pode-se observar, na Tabela 6.3, que o algoritmo *push* entrou 120 vezes na fase de refutação, devido aos atrasos de comunicação, e o *pull* 150 vezes. Nos resultados de simulação, apresentados na Tabela 6.4 o *push* entrou 360 e o *pull* 30 vezes na fase de refutação.

Tabela 6.4: Mensagens *todos-para-todos* na organização hierárquica

Camada	**Tipo**	**N° de mensagens**	
		push	***pull***
Aplicação	START	210	210
Comunicação	START_C	49	49
Detecção	PUSH_INIT	210	0
Detecção	I_AM_ALIVE	3150	0
Detecção	ARE_YOU_ALIVE	0	3150
Detecção	YES	0	3150
Detecção	ARE_YOU_ALIVE_R	360	30
Detecção	YES_R	360	30
Aplicação	STOP	210	210
Comunicação	STOP_C	49	49
Detecção	PUSH_STOP	210	0
Total de mensagens		**4808**	**6878**

Utilizando a organização hierárquica, constatou-se que o número de mensagens adicionais (desconsiderando as de refutação) foi de 98 (START_C e STOP_C). Para essas simulações as mensagens adicionais são referentes a delegações de monitoração.

Utilizar a política LAST de ajuste dinâmico de *timeout*, que tem como parâmetro de inferência somente o valor absoluto do último atraso de comunicação amostrado, permitiu observar que, em nenhum dos casos, foram inferidos e repassados à aplicação indicativos de falsas suspeitas decorrentes de atrasos. Pode-se também observar que foram geradas algumas mensagens de refutação, devido à estratégia global de adaptação de *timeout* implementada. Isso decorreu do ajuste do *timeout* inicial, quando a configuração ainda estava inadequada aos atrasos de comunicação da rede.

6.5 Avaliação da Qualidade de Detecção

Um segundo tipo de simulação do serviço foi feito na presença de defeitos, permitindo, dessa forma, que fossem avaliados o funcionamento e a qualidade de detecção dos algoritmos *push* e *pull*, utilizando organização plana e hierárquica.

Originalmente, o Neko não possui suporte para simular defeitos, mas o trabalho de Rodrigues (2006) estendeu a ferramenta de modo a permitir (facilitar) a inserção de defeitos de omissão, de colapso e de rede nas simulações.

Apesar de Chen, Toueg e Aguilera (2002) terem definido métricas para a avaliação da qualidade de serviço de detectores de defeitos, como descrito na seção 2.5, neste trabalho foram definidas outras três métricas específicas para avaliação do detector e dos atrasos impostos pelo uso de uma estratégia hierárquica. As três métricas são descritas a seguir.

- **Tempo de detecção de suspeita**: compreende o intervalo entre uma máquina monitorada começar a apresentar defeitos e o monitor detectá-los (DOWN).
- **Tempo de detecção de recuperação**: é o intervalo entre a máquina monitorada retornar às atividades normais e o detector tomar conhecimento (UP).
- **Tempo de aviso**: é o intervalo entre a observação de uma mudança de estado pelo detector (suspeito (DOWN) para não suspeito (UP) e vice-versa) e a aplicação ser notificada do fato.

Por simplificação, a análise dessas métricas foi efetuada como base em dois processos (perfil da aplicação 1 apresentada na seção 6.3) sendo: um processo representa uma aplicação localizada em uma determinada máquina que solicita ao serviço a monitoração de uma máquina de seu interesse; a máquina monitorada, por sua vez, possui um outro processo, que é uma instância do serviço para poder ser monitorada.

Nos testes executados, a máquina em que se localizava a aplicação, também possuia uma instância do serviço, o qual atendeu as requisições da aplicação. Porém, como a interação entre as aplicações e o serviço foi feita via troca de mensagens, a aplicação não necessariamente precisaria executar na mesma máquina do serviço.

Para este trabalho, foram testados somente defeitos de omissão e colapso, já que defeitos de temporização são pouco relevantentes em sistemas assíncronos, pois, os sistemas assíncronos se abstraem de qualquer ordem de temporização.

Como a monitoração foi efetuada no nível de máquina, para esse conjunto de testes não foi considerado o caso de uma aplicação solicitar a monitoração da máquina em que ela se encontra.

A próxima seção apresenta duas configurações de testes com defeitos de omissão da máquina monitorada: uma aplicação na máquina 1 monitora a máquina 9, uma aplicação na máquina 1 monitora a máquina 5, que é a líder da rede local 1.

6.5.1 Defeito de omissão

Para simular o defeito de omissão de mensagens foi inserida na máquina monitorada uma camada adicional. Essa nova camada foi inserida entre a da camada `Topologia` e a camada de `Detecção`. Essa camada de omissão, chamada `SendOmissionLayer`, interceptou as mensagens de saída da máquina monitorada e as descartou. Isso foi

efetuado conforme a política de descarte e por um período de tempo pré-estabelicido, que foram informados no arquivo de configuração do Neko. Um exemplo de arquivo de configuração é mostrado na Figura 6.5.

Como o conjunto de mensagens a serem trocadas era pequeno, a política de descarte adotada foi a de Bernoulli, com taxa de 100% durante 10 ms (iniciando no tempo 11 ms e terminando no tempo 21 ms), porém esses parâmetros poderiam ser facilmente variados, dependendo do objetivo do teste. Nesse período, todas as mensagens enviadas pela máquina monitorada foram descartadas. Estes parâmetros podem ser vistos nas linhas 24 a 27 da Figura 6.5. Definir o intervalo de tempo em que devem ocorrer defeitos é importante para evitar que sejam afetadas as mensagens iniciais e finais de sincronização.

```
 1  # Indica que é uma simulação
 2  simulation = true
 3
 4  # Número de processos
 5  process.num = 15
 6
 7  # Classe que inicializa a pilha de protocolos para cada processo
 8  process.initializer = SDDInitializer
 9
10  # Redes utilizadas para comunicação
11  Network = lse.neko.networks.sim.BasicNetwork,
    lse.neko.networks.sim.BasicNetwork
12  BasicNetwork.lambda = 1
13
14  # Escreve o log para arquivo (registro de eventos de comunicação)
15  Handlers = java.util.logging.FileHandler,
    java.util.logging.ConsoleHandler
16
17  # Define o nome do arquivo de registro
18  java.util.logging.FileHandler.pattern = pullhierar.log
19
20  # Faz o log de todas as mensagens trocadas entre processos
21  messages.level = FINE
22
23  # Política de descarte
24  process.9.sendomission.dropmodel = BernoulliDropModel
25  process.9.sendomission.dropmodel.params = 1.0
26  process.9.sendomission.start = 11
27  process.9.sendomission.finish = 21
```

Figura 6.5: Arquivo de configuração do Neko

Os parâmetros de monitoração, configurados pela aplicação, para a execução dos testes apresentados nas próximas subseções foram os seguintes:

- **Tempo de monitoração**: 30 ms;
- **Intervalo de monitoração**: 4 ms;
- ***Timeout* inicial**: 3.5 ms;
- **Algoritmo de predição**: fixo;
- **Margem de segurança**: fixa.

O tempo de monitoração escolhido foi 30 ms para melhor visualizar os *screenshots* de toda a simulação, incluindo o início e a parada. O *timeout* inicial foi escolhido

diferente do intervalo de monitoração, apenas para que no algoritmo *pull*, uma mensagem de detecção ARE_YOU_ALIVE e uma mensagem de refutação ARE_YOU_ALIVE_R, decorrente de um "estouro" de *timeout*, não fossem enviadas no mesmo instante. Essa configuração foi feita apenas para melhor visualização. O *timeout* permaneceu fixo até o final da simulação devido a não utilização de uma política de ajuste dinâmico. Essa configuração foi feita para que a omissão de mensagens da aplicação fosse detectada pelo monitor mais rapidamente, para evitar os ajustes dinâmicos.

Quando uma aplicação solicitou serviço de monitoração ao detector da sua máquina e o serviço operou com organização plana, as mudanças de estado foram imediatamente entregues à aplicação. Como é uma simulação, esse tempo de aviso foi considerado igual à zero.

Os resultados das simlações são apresentados na forma de *screenshots*, oriundos do arquivo de registro (*log*) da execução da simulação, gerados pela ferramenta *LogView*, que faz parte do pacote do Neko.

Para, visualmente, melhor identificar os tipos de mensagens trocadas na simulação, a ferramenta *LogView* permitiu, através de um arquivo de configuração XML (*Extensible Markup Language*), configurar cores específicas para os diferentes tipos. Além disso, as mensagens foram numeradas manualmente conforme seu tipo, para melhor visualizá-las nos *screenshots* dos testes. As mensagens descartadas pela camada de omissão foram indicadas com apóstrofo, conforme seus tipos (por exemplo, 4' e 6'), porém graficamente, ou invés de setas, são representadas por segmentos de reta (de cor preta). A Tabela 6.5 descreve os tipos de mensagens numeradas e suas respectivas cores.

Tabela 6.5: Tipos de mensagens utilizadas pelo serviço

N° da mensagem	Cor da mensagem	Tipo da mensagem	Descrição da mensagem
1	*ciano*	START_C	Delegação de início de monitoração
2	*ciano*	STOP_C	Delegação de parada de monitoração
3	*amarelo*	PUSH_INIT	Início da monitoração
4	*azul*	I_AM_ALIVE	Indicação de atividade
5	*cinza*	ARE_YOU_ALIVE	Requisição de atividade.
6	*azul*	YES	Resposta de indicação de atividade
7	*rosa*	ARE_YOU_ALIVE_R	Requisição de refutação
8	*marrom*	YES_R	Resposta de refutação
9	*vermelho*	DOWN	Indicativo de suspeita
10	*verde*	UP	Indicativo de recuperação
11	*amarelo*	PUSH_STOP	Parada de monitoração

6.5.1.1 Aplicação na máquina 1 monitora a máquina 9

Como ilustra a Figura 6.6, o detector na máquina 1 suspeitou de defeito na máquina 9 no tempo 7 ms e imediatamente repassou para aplicação. Esta suspeita originou-se do atraso imposto pela rede de comunicação, porém, no tempo 10 ms, foi corrigida. A omissão de mensagens, iniciada no tempo 11 ms pela máquina monitorada, gerou uma segunda suspeita que foi detectada no tempo 21 ms, e apesar de a máquina monitorada retornar as suas atividades normais no tempo 21 ms, a recuperação foi detectada aos 27.5 ms.

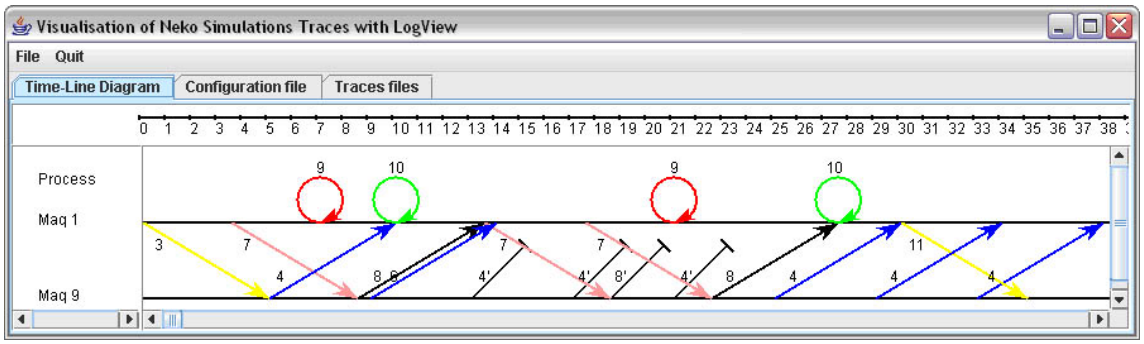

Figura 6.6: *Push* com organização plana

Na Figura 6.7, observa-se que no tempo 7 ms detector da máquina 1 suspeitou de defeito na máquina 9 e imediatamente repassou para aplicação. Esta suspeita, da mesma forma como foi observada usando o algoritmo *push*, originou-se do atraso imposto pela rede de comunicação, e foi corrigida no tempo 10 ms. A segunda suspeita decorrente da perda de mensagens foi detectada no tempo 23 ms, e a recuperação foi detectada aos 26 ms.

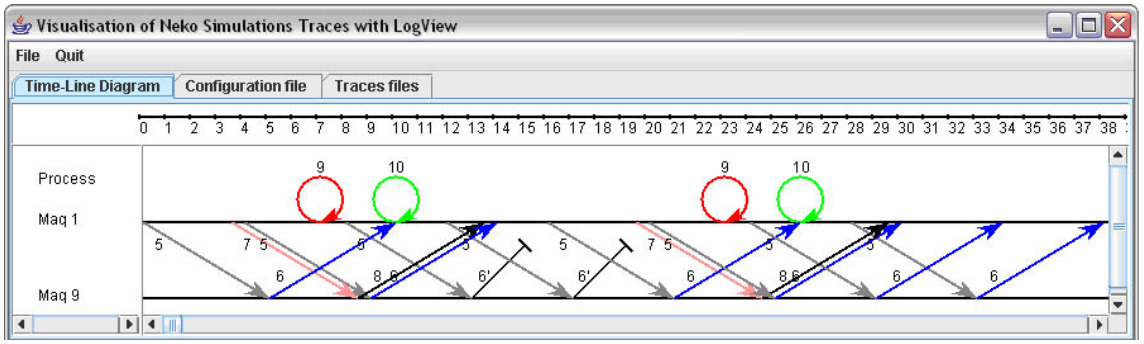

Figura 6.7: *Pull* com organização plana

Como ilustra a Figura 6.8, o detector da máquina 5 (líder da LAN 1) suspeitou de defeito na máquina 9 no tempo 19 ms. A suspeita foi repassada para aplicação, que a recebeu aos 25 ms. A recuperação foi percebida pelo detector no tempo 24 ms, e a aplicação recebeu o indicativo aos 30 ms.

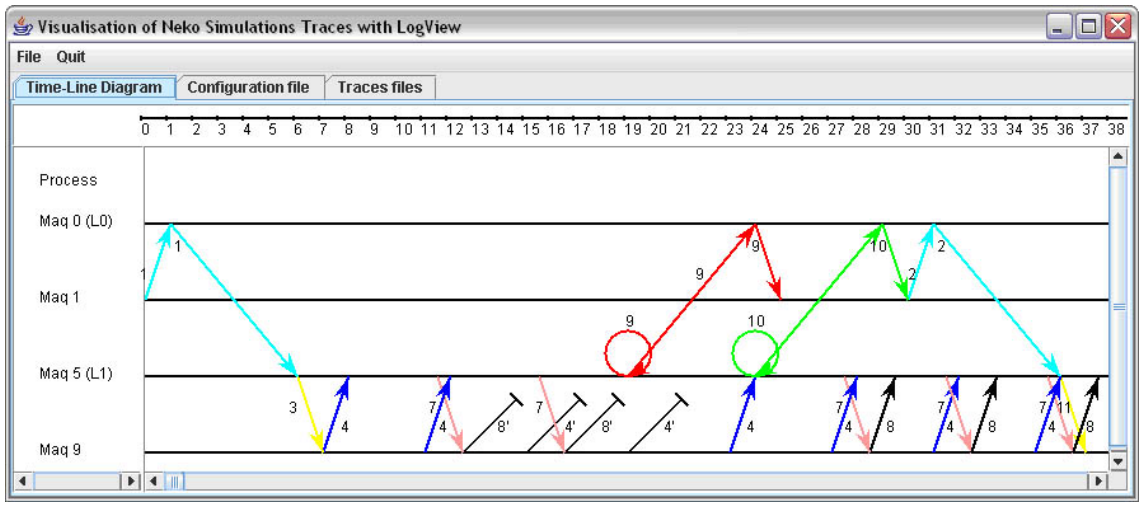

Figura 6.8: *Push* com organização hierárquica

Como ilustra a Figura 6.9, o detector da máquina 5 (líder da LAN 1) suspeitou de defeito na máquina 9 no tempo 17 ms. A suspeita foi repassada para aplicação, que a recebeu aos 23 ms. A recuperação foi percebida pelo detector no tempo 24 ms, e a aplicação recebeu o indicativo aos 30 ms.

Os atrasos finais percebidos nestes tempos decorreram dos saltos, que foram dependentes da topologia e do tempo de comunicação. A freqüência de monitoração (ou intervalo de monitoração) também influenciou no tempo de suspeita e recuperação. Por exemplo, uma freqüência elevada de monitoração resultaria em uma detecção mais rápida.

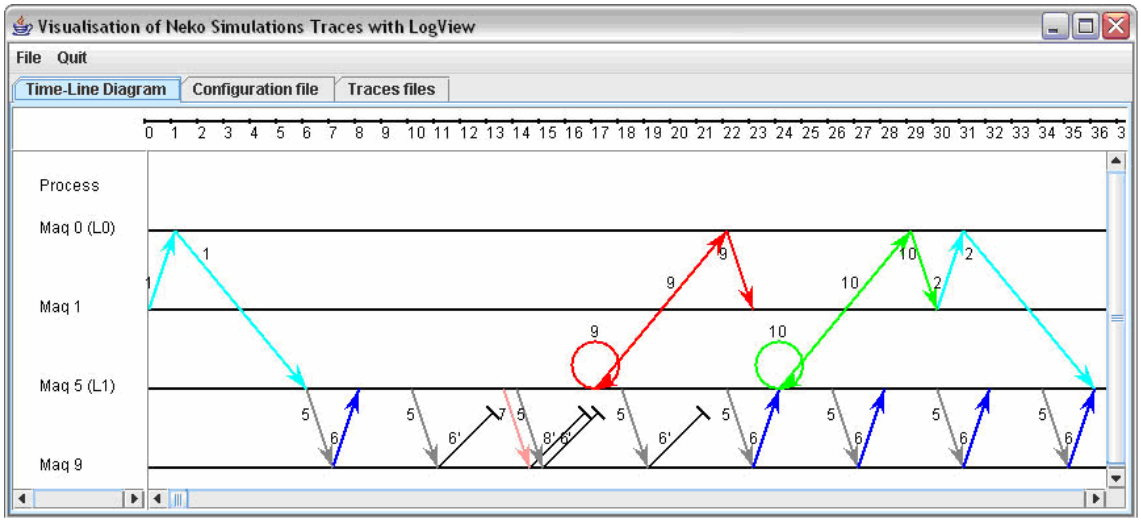

Figura 6.9: *Pull* com organização hierárquica

Para obter os tempos de suspeita e recuperação, foram considerados os tempos "reais" em que a máquina apresentou defeito de omissão, conforme a configuração (início de defeitos de omissão aos 11 ms e retono às atividades normais aos 21 ms). Os resultados para a organização plana e hierárquica, com o algoritmo *push* e *pull* são apresentados na Tabela 6.6.

Tabela 6.6: Tempos de detecção de omissão

		Organização plana			**Organização hierárquica**		
		Tempo de Detecção	**Tempo de Aviso**	**Total**	**Tempo de Detecção**	**Tempo de Aviso**	**Total**
Push	**Suspeita**	10 ms	0 ms	**10 ms**	8 ms	6 ms	**14 ms**
	Recuperação	6.5 ms	0 ms	**6.5 ms**	3 ms	6 ms	**9 ms**
Pull	**Suspeita**	12 ms	0 ms	**12 ms**	6 ms	6 ms	**14 ms**
	Recuperação	5 ms	0 ms	**5 ms**	3 ms	6 ms	**9 ms**

Obviamente, que o tempo de aviso na organização hierárquica foi dependente do caminho das mensagens imposto pela hierarquia, e do tempo de comunicação entre cada salto. A ênfase desses testes foi avaliar a organização hierárquica. Não foram efetuados testes sobre a qualidade do serviço de detecção, variando os algoritmos de predição e margem de segurança, pois não é objetivo do trabalho avaliar a influência desses algoritmos.

6.5.1.2 Aplicação na máquina 1 monitora a máquina 5 (líder LAN 1)

Quando uma aplicação na máquina 1 necessitou monitorar uma máquina de outra rede local, mas esta máquina era o próprio líder, a monitoração foi feita de líder para líder. Os resultados para a organização plana foram os mesmos apresentados na subseção anterior. Porém, na organização hierárquica, os resultados foram os seguintes:

Como ilustra a Figura 6.10, devido ao tempo de comunicação entre redes, o detector da máquina 0 (líder da LAN 0) suspeitou de defeito na máquina 5 (líder da LAN 1) no tempo 8 ms, e no tempo 9 ms a aplicação da máquina 1 foi avisada. No tempo 11 ms o detector detectou a recuperação e avisou a aplicação que recebeu no tempo 12 ms. Uma segunda suspeita, agora sim decorrente das mensagens descartadas pela camada de omissão, foi percebida pelo detector no tempo 22 ms e recebida pela aplicação no tempo 23 ms. A recuperação foi percebida pelo detector no tempo 27 ms, e a aplicação recebeu tal indicação no tempo 28 ms.

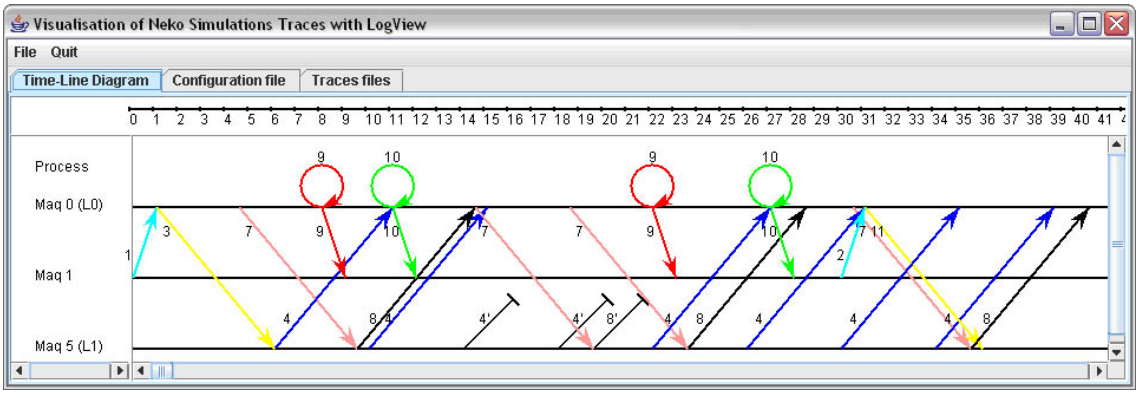

Figura 6.10: *Push* com organização hierárquica

Na Figura 6.11, observa-se que o detector da máquina 0 (líder da LAN 0) suspeitou de defeito na máquina 5 (líder da LAN 1) no tempo 8 ms e, no tempo 9 ms, a aplicação da máquina 1 foi avisada. No tempo 11 ms, o detector detectou a recuperação e avisou a aplicação que recebeu no tempo 12 ms. Uma segunda suspeita, decorrente das mensagens descartadas pela camada de omissão, foi percebida pelo detector no tempo 24 ms e recebida pela aplicação no tempo 25 ms. A recuperação foi percebida pelo detector no tempo 27 ms, e a aplicação recebeu tal indicação no tempo 28 ms.

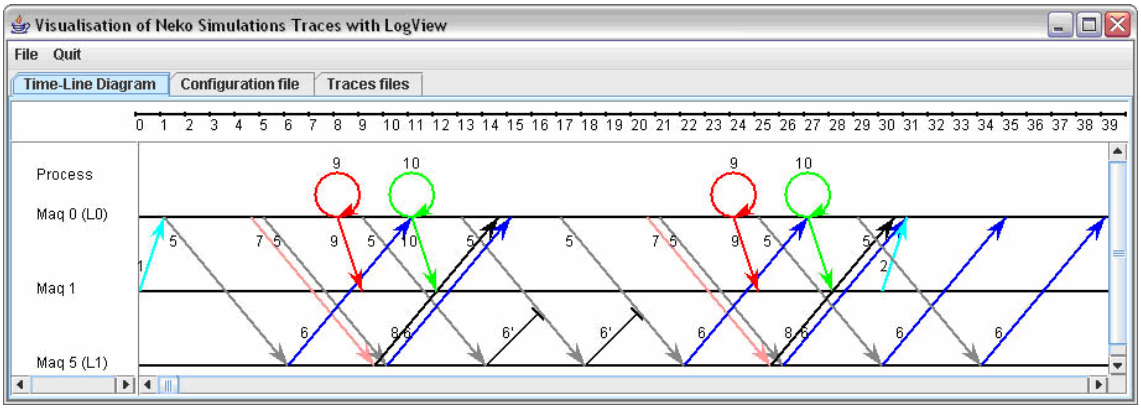

Figura 6.11: *Pull* com organização hierárquica

Os resultados para a organização hierárquica, com o algoritmo *push* e *pull*, são apresentados na Tabela 6.7.

Tabela 6.7: Tempos de detecção de omissão na organização hierárquica

		Organização hierárquica		
		Tempo de Detecção	**Tempo de Aviso**	**Total**
Push	**Suspeita**	11 ms	6 ms	**17 ms**
	Recuperação	6 ms	6 ms	**12 ms**
Pull	**Suspeita**	13 ms	6 ms	**19 ms**
	Recuperação	6 ms	6 ms	**12 ms**

6.5.2 Defeito de colapso

Para a simulação de defeitos de colapso foi utilizada uma camada adicional, chamada `CrashLayer`, localizada na máquina monitorada. A camada `CrashLayer` é semelhante à camada para a simulação de omissão `SendOmissionLayer`, porém faz a interceptação de todas as mensagens enviadas e recebidas pela máquina monitorada e as descarta durante o período de defeito.

Para a observação da simulação de defeitos de colpaso, foi configurada uma aplicação distribuída localizada em 14 máquinas (0 até 13) do cenário. As 14 máquinas solicitaram monitoração da máquina 14. Foram utilizados os mesmos parâmetros de monitoração da simulação de defeitos de omissão. Porém, foi configurada uma simulação de 3000 ms em que a máquina 14 (máquina monitorada) entrou em colapso no tempo 1500 ms.

Os tempos de detecção e de aviso do colapso percebidos pelas demais máquinas são mostrados na Tabela 6.8.

Tabela 6.8: Tempos de detecção de colapso

		Organização plana			Organização hierárquica		
	Aplicação	**Tempo de Detecção**	**Tempo de Aviso**	**Total**	**Tempo de Detecção**	**Tempo de Aviso**	**Total**
Push	**0**	9 ms	0 ms	**9 ms**	6 ms	5 ms	**11 ms**
	1	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	2	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	3	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	4	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	5	9 ms	0 ms	**9 ms**	6 ms	5 ms	**11 ms**
	6	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	7	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	8	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	9	9 ms	0 ms	**9 ms**	7 ms	6 ms	**13 ms**
	10	5 ms	0 ms	**5 ms**	5 ms	0 ms	**5 ms**
	11	5 ms	0 ms	**5 ms**	5 ms	0 ms	**5 ms**
	12	5 ms	0 ms	**5 ms**	5 ms	0 ms	**5 ms**
	13	5 ms	0 ms	**5 ms**	5 ms	0 ms	**5 ms**
Pull	**0**	11 ms	0 ms	**11 ms**	8 ms	5 ms	**13 ms**
	1	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	2	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	3	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**

	4	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	5	11 ms	0 ms	**11 ms**	8 ms	5 ms	**13 ms**
	6	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	7	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	8	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	9	11 ms	0 ms	**11 ms**	9 ms	6 ms	**15 ms**
	10	7 ms	0 ms	**7 ms**	7 ms	0 ms	**7 ms**
	11	7 ms	0 ms	**7 ms**	7 ms	0 ms	**7 ms**
	12	7 ms	0 ms	**7 ms**	7 ms	0 ms	**7 ms**
	13	7 ms	0 ms	**7 ms**	7 ms	0 ms	**7 ms**

Pode-se observar que, tanto na organização plana quanto na hierárquica, o algoritmo *pull* demorou 2 ms a mais para detectar o defeito de colapso. O tempo de aviso na organização hierárquica independe do algoritmo de detecção utilizado, mas é dependente da localização das máquinas. Nota-se, também, que as aplicações que estão na mesma LAN da máquina monitorada detectaram o defeito mais rapidamente.

6.6 Abordagens aos Desafios do Serviço de Detecção de Defeitos

Na seção 4.3, foram citados alguns desafios que surgem na implementação de um serviço de detecção de defeitos, principalmente quando o serviço deve operar sobre redes de longa distância e, possivelmente, atendendo a sistemas de larga escala. A seguir é discutido como a modelagem, a escolha das estratégias e a implementação do protótipo foram conduzidas, visando a enfrentar (ou ignorar) esses desafios.

- **Explosão do número de mensagens:** um dos principais diferenciais do serviço proposto é que ele opera sob demanda das aplicações. Se elas solicitarem a monitoração de muitas entidades simultaneamente e com parâmetros inadequados, pode ocorrer a explosão do número de mensagens. Adotar uma estratégia hierárquica concentra o fluxo de mensagens de monitoração internamente às redes locais. Dessa forma, entre redes locais trafegam apenas mensagens de controle (delegação de monitoração entre líderes, indicativo de mudança de estado de monitoráveis) e, possivelmente, mensagens de monitoração, para o caso especial de uma aplicação monitorar um líder de outra rede local;
- **Perda e atraso de mensagens:** em relação às redes locais, as de longa distância apresentam maior variabilidade de atrasos (*jitter*). Para enfrentar esse problema foram implementadas no protótipo do serviço as duas primeiras fases da política de ajuste global de *timeout* em três fases. Esse ajuste global de *timeout* em três fases foi proposto no trabalho de Nunes (2003). Além disso, as aplicações podem escolher de forma independente, para cada monitorável, um algoritmo de predição e uma margem de segurança para o cálculo dinâmico de *timeout*, ou podem escolher um valor fixo. A perda de mensagens geralmente é tratada pelos protocolos de nível mais baixo, como o de nível de rede (TCP);
- **Escalabilidade:** o uso da estratégia hierárquica visa a reduzir o tráfego, principalmente entre redes locais, que é mais oneroso. Isso contribui para melhorar a escalabilidade do serviço quando este monitora um número elevado de máquinas em diferentes redes locais. A monitoração, sob demanda, também procura evitar o desperdício na utilização de recursos;

- **Flexibilidade**: para atender diferentes tipos ou classes de aplicações, o serviço oferece configuração de grão fino. Ele permite a cada aplicação configurar parâmetros específicos de monitoração para cada máquina de interesse. Permite também o uso de diferentes algoritmos de detecção, predição de *timeout* e margem de segurança, além de seguir um padrão de projeto que facilita a inserção de novos algoritmos no serviço. Essas características tornam o serviço flexível e configurável;
- **Dinamismo:** o protótipo implementado assume uma topologia fixa de grupos e líderes, porém, a arquitetura proposta no trabalho inclui um módulo que gerencia a entrada e saída de máquinas dinamicamente, além da resiliência de defeitos nos líderes locais;
- **Segurança:** o serviço não deve se comportar de maneira inadequada, comprometendo a segurança do sistema. Todavia, as questões relativas aos aspectos de segurança não foram analisadas, pois fogem do escopo do trabalho;
- **Abrangência e precisão:** para inferir suspeitas com um bom nível de precisão e abrangência, o serviço usa o pacote de políticas de ajuste dinâmico de *timeout* (NUNES, 2003);
- **Baixa sobrecarga:** a avaliação da sobrecarga do serviço requer executá-lo num ambiente real, observando o consumo de recursos computacionais (processamento e memória) e de rede (sobrecarga). Na simulação, foi possível apenas avaliar o número e os tipos de mensagens de controle e monitoração gerados;
- **Aspectos temporais:** o serviço infere suspeitas baseado nas políticas configuradas e avisa as aplicações sobre mudanças de estado. Essa identificação também está sujeita aos atrasos da rede e da hierarquia de comunicação.

6.7 Conclusões Parciais

O capítulo mostrou os cenários de simulação testados, as configurações de monitoração e as implicações em usar o serviço sob uma organização plana e outra hierárquica. Os algoritmos de monitoração implementados, *push* e *pull*, funcionaram de acordo com suas especificações e trocaram o número de mensagens esperado. Com o mesmo conjunto de configurações, os dois algoritmos foram executados com o serviço operando sob uma organização plana e a hierárquica e, puderam ser observados o número e tipos de mensagens trocadas. Para duas aplicações sintéticas desenvolvidas, outro tipo de análise foi feito em relação ao tempo de suspeita, recuperação e aviso na presença de defeitos de omissão e colapso. Por fim, a modelagem, as estratégias e o protótipo do serviço proposto foram analisados conceitualmente no que diz respeito aos desafios enumerados em capítulo anterior. A seguir, é apresentada a conclusão do trabalho.

7 CONCLUSÃO

Sistemas conectados por redes de longa distância, geralmente, oferecem um ambiente mais hostil do que as LANs e *clusters*, devido aos atrasos longos e variáveis e à maior probabilidade de perda de mensagens, impondo um desafio na concepção de mecanismos que detectem defeitos de forma completa e precisa.

A proliferação de sistemas distribuídos fez com que o projeto e desenvolvimento das aplicações distribuídas exigissem requisitos de *dependabilidade,* conseqüentemente necessitando de mecanismos eficientes de detecção de defeitos. Um serviço de detecção de defeitos pode ser considerado uma abstração fundamental para a construção de sistemas distribuídos tolerantes a falhas. Entretanto, ainda são poucos os serviços que oferecem suporte à detecção e ao gerenciamento de defeitos e que operem em WANs.

Neste trabalho, foram avaliadas alternativas e estratégias que poderiam ser empregadas para a implementação de um serviço de detecção de defeitos. A avaliação gerou uma tabela comparativa para ilustrar como cada uma das estratégias selecionadas abordava, ou não, os requisitos exigidos por um sistema distribuído de larga escala e que operasse sobre WANs.

Posteriormente, foi proposto um modelo de arquitetura para a detecção de defeitos na forma de um serviço, localizado na camada *middleware*. Baseado nesse modelo, foi desenvolvido um protótipo do serviço sobre o *framework* de simulação Neko.

A modularização do protótipo e o uso da linguagem Java possibilitaram reutilizar o pacote de predição de *timeout* de Nunes (2003). Além do reúso de código do pacote, as funcionalidades do pacote auxiliam na melhora da qualidade de detecção do serviço, evitando falsas suspeitas, decorrentes dos atrasos de comunicação ou decorrentes da organização hierárquica adotada.

As simulações avaliaram as principais estratégias adotadas para o modelo e permitiram a comparação dos algoritmos de monitoração implementados, *push* e *pull*. A avaliação considerou o tipo e número de mensagens trocadas pelo serviço durante sua operação, atendendo a dois tipos de aplicações sintéticas. A primeira, uma aplicação local que exigia a monitoração *um-para-um*, e a segunda, uma aplicação distribuída que exigia a monitoração *todos-para-todos*. Variando-se a organização entre plana e hierárquica, e o algoritmo entre *push* e *pull*, também foi avaliada a qualidade da detecção na presença de defeitos de omissão e colapso.

A escolha da estratégia hierárquica para a organização dos módulos detectores mostrou ser adequada, pois, segundo os resultados das simulações realizadas, um mínimo de mensagens adicionais de controle foi observado, e a qualidade de detecção também não sofreu degradação ou atrasos expressivos. Adicionalmente, pôde-se observar que adotar a hierarquia em dois níveis (LAN e WAN) resultou em um número

menor de mensagens trafegando entre redes locais, além do que, as mensagens inter-LANs podem conter um conjunto maior de informações para reduzir o tráfego. Por exemplo, uma mensagem pode carregar as configurações de delegação de monitoração não só sobre uma, mas sobre um conjunto de máquinas que uma determinada aplicação tenha interesse. Desta forma, o serviço tira proveito do conhecimento da topologia da rede quando um número maior de máquinas é utilizado.

Por ser um fator essencial na economia de recursos computacionais e de rede, a monitoração, sob demanda das aplicações foi adotada como modo de operação para o serviço. Se nenhuma aplicação tem interesse na monitoração de outras máquinas, o serviço não troca mensagens entre seus módulos.

A prototipação do modelo também serviu para mostrar a configurabilidade e a flexibilidade do serviço, visto que preditores, margens de segurança e algoritmos de monitoração podem ser escolhidos ou inseridos de maneira simples. Adicionalmente, cada aplicação pode configurar o conjunto de parâmetros específicos para a monitoração de cada máquina de interesse.

O serviço proposto oferece suporte para detectar defeitos de omissão, temporização e colapso. Porém, a discriminação entre eles por um detector remoto é difícil. Se nenhuma informação sobre uma máquina é obtida, então, não é possível decidir se é devido ao colapso dessa máquina ou da rede. Todavia, isso não constitui um problema para as aplicações, pois, para essas o conhecimento de que a máquina está indisponível ou inalcançável é mais importante do que saber o motivo.

Sobre o protótipo desenvolvido ainda podem ser inseridos novos algoritmos, ou diferentes implementações além dos que já existem, de detecção, predição e margem de segurança, permitindo que novos experimentos, testes comparativos e avaliações sejam executados, se necessário. Novos cenários, mais complexos também podem ser construídos para este fim.

Com o objetivo de avaliar as funcionalidades do serviço de detecção e devido aos testes realizados terem sido feitos através de simulações, que geram resultados determinísticos, neste trabalho, optou-se em construir um cenário formado por um conjunto relativamente pequeno de máquinas.

Os resultados obtidos referentes aos tipos e número de mensagens e referentes à qualidade de deteção, se necessário, podem ser projetados para considerar um número maior de máquinas, já que, por se tratar de simulação, os tempos de comunicação independem de quantidade de mensagens que trafegam na rede. Portanto, a menos que os testes sejam executados em um ambiente real, não se justifica criar um cenário de larga escala para simular o protótipo desenvolvido.

Como trabalho futuro, o modelo do serviço proposto além de simulado, ainda pode ser emulado, já que a ferramenta Neko possibilita a execução distribuída. Pode-se ainda implementar o serviço em um ambiente real e disponibilizá-lo na forma de um *daemon*[10], como uma biblioteca ou como um serviço do próprio sistema operacional, dependendo dos níveis de intrusividade e desempenho desejados.

[10] Programa ativo em segundo plano pronto a receber instruções/pedidos de outros programas para executar determinada ação.

Na execução ou emulação do serviço em um ambiente real, é possível executar testes para reavaliar a escalabilidade do serviço e avaliar a sobrecarga imposta ao ambiente, além da influência nas aplicações que o utilizam.

O serviço de detecção desenvolvido pode ser usado diretamente por aplicações distribuídas, ou até mesmo servir como suporte para outros serviços de *middleware*, tais como gerenciamento, replicação, balanceamento de carga, comunicação de grupo e serviços de *membership*.

REFERÊNCIAS

AGUILERA, M. K.; CHEN, W.; TOUEG, S. Heartbeat: a Timeout-Free Failure Detector for Quiescent Reliable Communication. In: INTERNATIONAL WORKSHOP ON DISTRIBUTED ALGORITHMS, WDAG, 11., 1997. **Proceedings…** Saarbrücken, Germany: [s.n.], 1997. p. 126-140.

AKYILDIZ, I.F. *et al*. Wireless Sensor Networks: A Survey. **Computer Networks**, [S.l.], v. 38, n. 4, p. 393-422, May 2002.

BAUER, P.; SICHUTIU, M.; PREMARATNE, K. On Nature of Time-Variant Communication Delays. In: IASTED CONFERENCE MODELING, IDENTIFICATION AND CONTROL, MIC, 20., 2001. **Proceedings…** Innsbruck, Austria: [s.n.], 2001. p. 792-797.

BERTIER, M.; MARIN, O.; SENS, P. Implementation and performance evaluation of an adaptable failure detector. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 24., 2002. **Proceedings...** Bethesda, Maryland, USA: IEEE Computer Society Press, 2002. p. 354-363.

BERTIER, M.; MARIN, O.; SENS, P. Performance Analysis of a Hierarchical Failure Detector. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 25., 2003. **Proceedings...** San Francisco, California, USA: IEEE Computer Society Press, 2003. p. 635-644.

BIRMAN, K. P. **Building Secure and Reliable Network Applications**. Greenwich: Manning, 1996.

BORTOLÁS, A. O. **Agregação de Funcionalidades ao AFDService**. 2004. 66 f. Trabalho de Conclusão (Bacharelado em Ciência da Computação) – Curso de Ciência da Computação, UFSM, Santa Maria.

BURNS, M. W.; GEORGE, A. D.; WALLACE, B. A. Simulative Performance Analysis of Gossip Failure Detection for Scalable Distributed Systems. **Cluster Computing**. Hingham, Massachusetts, USA, v. 2, n. 3, p. 207-217, 1999.

CATÃO, B. G.; BRASILEIRO, F: V.; OLIVEIRA, A. C. A. Engineering a Failure Detection Service for Widely Distributed Systems. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS; SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 23., 2005. **Anais...** Fortaleza, CE, Brasil: SBC, 2005. p. 111-122.

CHANDRA, T. D.; HADZILACOS, V.; TOUEG, S. The Weakest Failure Detectors for Solving Consensus. **Journal of the ACM**, New York City, NY, EUA, v. 43, n. 4, p. 685-722, July 1996.

CHANDRA, T. D.; TOUEG, S. Unreliable Failure Detectors for Reliable Distributed Systems. **Journal of the ACM**, New York City, NY, EUA, v. 43, n. 2, p. 225-267, Mar. 1996.

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

CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems. **Communications of the ACM**, New York City, NY, EUA, v.34, n.2, p. 56-78, Feb. 1991.

DAHLGREN, J. **Efficient Failure Detection Protocols for Point-to-Point Communication Networks**. 2004. 110 p. Thesis (Master in Computer Science) – Department of Computer Science, Rochester Institute of Technology, New York.

DELICATO, F. C. **Middleware Baseado em Serviços para Redes de Sensores Fio**. 2005. 197 p. Tese (Doutorado em Engenharia Elétrica) – Coordenação dos Programas de Pós-graduação de Engenharia, COPPE/UFRJ, Rio de Janeiro.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the Presence of Partial Synchrony. **Journal of the ACM**, New York, NY, EUA, v. 35, n. 2, p. 288-323, Apr. 1988.

ESTEFANEL, L. A. B. **Avaliação dos Detectores de Defeitos e sua Influência nas Operações de Consenso**. 2001. 128 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

FALAI, L.; BONDAVALLI, A. Experimental Evaluation of the QoS of Failure Detectors on Wide Area Network. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, DSN, 27., 2005. **Proceedings...** Yokohama, Japan: IEEE Computer Society Press 2005. p. 624-633.

FELBER, P. *et al*. Failure Detectors as First Class Objects. In: IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS, DOA, 1999. **Proceedings...** Edinburgh, Scotland: IEEE Computer Society Press 1999. p. 132-141.

FELBER, P.; GUERRAOUI, R.; FAYAD, M. E. Putting OO Distributed Programming to Work. **Communications of the ACM**, New York City, NY, EUA, v. 42, n. 11, p. 97-101, Nov. 1999.

FELBER, P. **The CORBA Object Group Service**. 1998. 158 p. Thesis (Degree of Doctor of Philosophy in Computer Science) – *Département D'Informatique, École Polytechnique Fédérale de Lausanne*, Suíça.

FETZER, C.; RAYNAL, M.; TRONEL, F.: An Adaptive Failure Detection Protocol. In: PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, PRDC, 8., 2001. **Proceedings...** Seoul, Korea: [s.n.], 2001. p. 146-153.

FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of Distributed Consensus with One Faulty Process. **Journal of the ACM**, New York City, NY, EUA, v. 32, n. 2, p. 374-382, Apr. 1985.

GARTNER, F. C. Fundamentals of Fault-Tolerant Distributed Computing in Asynchronous Environments. **ACM Computing Surveys**, New York, NY, USA, v.31, n.1, p.1-26, Mar. 1999.

GAMMA, E. *et al*. **Design Patterns, Elements of Reusable Object-Oriented Software**. Massachusetts: Addison Wesley Professional, 1994. p. 395.

GORENDER, S.; MACEDO, R. Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS. In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 20., 2002. **Anais …** Búzios, RJ, Brasil: [s.n.], 2002. p. 277-292.

GUERRAOUI, R.; SCHIPER, A. Consensus: The Big Misunderstanding. In: IEEE COMPUTER SOCIETY WORKSHOP ON FUTURE TRENDS IN DISTRIBUTED COMPUTING SYSTEMS, FTDCS, 6., 1997. **Proceedings...** Tunis, Tunisia: IEEE Computer Society Press, 1997. p. 183-188.

GUERRAOUI, R.; SCHIPER, A. Software-Based Replication for Fault Tolerance. **IEEE Computer**, Los Alamitos, CA, v. 30, n. 4, p. 68-74, Apr. 1997.

GUO, K. H. Failure Detection. In: GUO, K. H. **Scalable Message Stability Detection Protocols**. 1998. 183 p. Thesis (Degree of Doctor of Philosophy in Computer Science) – Department of Computer Science, Cornell University, Ithaca.

HAYASHIBARA, N. **Accrual Failure Detectors**. 2004. 96 p. Thesis (Degree of Doctor of Philosophy in Computer Science) – School Information Science, Japan Advanced Institute of Science and Technology, Ishikawa, Japan.

HAYASHIBARA, N.; CHERIF, A.; KATAYAMA, T. Failure Detectors for Large-Scale Distributed Systems. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 21., 2002. **Proceedings...** Suita, Japan: IEEE Computer Society Press, 2002. p. 404-409.

HAYASHIBARA, N.; DÉFAGO, X.; KATAYAMA, T. Two-ways Adaptive Failure Detection with the φ-Failure Detector. In: WORKSHOP ON ADAPTIVE DISTRIBUTED SYSTEMS, WADiS, 2003. **Proceedings...** Sorrento, Italy: [s.n.], 2003. p. 22-27.

HAYASHIBARA, N. *et al*. The φ Accrual Failure Detector. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, SRDS, 23., 2004. **Proceedings...** Florianópolis, Santa Catarina, Brasil: IEEE Computer Society Press, 2004. p. 66-78.

HAYASHIBARA, N. *et al*. Information Propagation on the φ Failure Detector. In: INTERNATIONAL WORKSHOP ON DATABASE AND EXPERT SYSTEMS APLICATIONS, DEXA, 16., 2005. **Proceedings…** Copenhagen, Denmark: IEEE Computer Society Press, 2005. p. 72-76.

JAIN, A.; SHYAMASUNDAR, R. K. Failure Detection and Membership Management in Grid Environments. In: INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 5., 2004. **Proceedings...** Pittsburgh, Pennsylvania, USA: IEEE/ACM Computer Society Press, 2004. p. 44-52.

JALOTE, P. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice Hall, 1994.

LARREA, M.; ARÉVALO, S.; FERNÁDEZ, A. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, DISC, 13., 1999. **Proceedings…** Bratislava, Slovak Republic: [s.n.], 1999. p. 34-48.

LAPRIE, J. C. Dependable Computing and Fault Tolerance: Concepts and terminology. In: INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, FTCS, 15., 1985. **Proceedings…** Ann Arbor, Michigan, USA: IEEE Computer Society Press, 1985. p. 2-11.

LOUREIRO, A. A. F. *et al*. Redes de Sensores Sem Fio (Minicurso). In: SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 21., 2003. **Anais...** Natal, RN, Brasil: [s.n.], 2003. p. 179-226.

VERÍSSIMO, P.; LEMOS, R. **Confiança no Funcionamento:** Proposta para uma Terminologia em Português. [S.l.]: INESC e LCMI/UFSC. 1989. Disponível em: <http://www.cs.ukc.ac.uk/people/staff/rdl/CoF/inesc.ps>. Acesso em: set. 2005.

MEDEIROS, R.; CIRNE, W.; BRASILEIRO, F.; SAUVÉ, J. Faults in Grids: Why are they so bad and What can be done about it?. In: INTERNATIONAL WORKSHOP ON GRID COMPUTING, GRID, 4., 2003. **Proceedings...** Phoenix, Arizona, USA: IEEE/ACM Computer Society Press, 2003. p. 18-24.

NUNES, C. R. **Adaptação Dinâmica do *timeout* de Detectores de Defeitos através do Uso de Séries Temporais**. 2003. 164 f. Tese (Doutorado em Ciência da Computação) - Instituto de Informática, UFRGS, Porto Alegre.

RANGANATHAN, S. *et al*. Gossip-Style Failure Detection and Distributed Consensus for Scalable Heterogeneous Clusters. **Cluster Computing**, [S.l.], v. 4, n. 3, p. 197-209, 2001.

RODRIGUES, L. A. **Extensão do Suporte para Simulação de Defeitos no Neko**. 2006. 90 f. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

SABEL, L. S.; MARZULLO, K. **Election Vs. Consensus in Asynchronous Systems**. New York City, NY, USA: Department of Computer Science, Cornell University, 1995. (Technical Report TR95-1488).

SAMPAIO, L. M. R.; BRITO, A. E. M.; OLIVEIRA, E. W. A., Detectores de Falhas em Sistemas Assíncronos (Tutorial). In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS; SIMPÓSIO BRASILEIRO DE REDES DE COMPUTADORES, SBRC, 21., 2003. **Anais...** Natal, RN, Brasil: [s.n.], 2003. p. 2-29.

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

SUN MICROSYSTEMS. **The Source for Java Developers**. Disponível em: <http://java.sun.com>. Acesso em: jan. 2006.

STELLING, P. *et al*. A Fault Detection Service for Wide Area Distributed Computations. In: INTERNATIONAL SYMPOSIUM ON HIGH PERFORMANCE DISTRIBUTED COMPUTING, HPDC, 7., 1998. **Proceedings...** Chicago, Illinois, EUA: IEEE Computer Society Press, 1998. p. 268-278.

TUREK, J.; SHASHA, D. The Many Faces of Consensus in Distributed Systems. In: READINGS IN DISTRIBUTED COMPUTING SYSTEMS. **Proceedings…** New York City, NY, EUA: IEEE Computer Society Press, 1994. p. 83-99.

URBÁN, P.; DÉFAGO, X.; SCHIPER, A. Neko: A single environment to simulate and prototype distributed algorithms. In: IEEE INTERNATIONAL CONFERENCE ON INFORMATION NETWORKING, ICOIN, 15., 2001. **Proceedings…** Beppu City, Japan: [s.n.], 2001. p. 503-511.

VAN RENESSE, R.; MINSKY, Y.; HAYDEN, M. A Gossip-Style Failure Detection Service. In: IFIP INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS PLATFORMS AND OPEN DISTRIBUTED PROCESSING, 1998. **Proceedings...** The Lake District, England: [s.n.], 1998. p. 55-70.

VASCONCELOS, S. R. A. **Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos**. 1997. 178 f. Dissertação (Mestrado em Informática) – Coordenação de Pós-Graduação em Informática, UFPB, Campina Grande.

XU, N. A Survey of Sensor Network Applications. **IEEE Communications Magazine**, [S.l.], v. 40, n. 8, p. 102-114, Aug. 2002.