Universidade Federal de Santa Maria — UFSM

Dissertação de Mestrado

UMA NOVA ABORDAGEM PARA REDUÇÃO DE MENSAGENS DE CONTROLE EM DETECTORES DE DEFEITOS

Rogério Corrêa Turchetti

Programa de Pós-Graduação em Engenharia de Produção — PPGEP

Santa Maria, RS, Brasil

2006

UMA NOVA ABORDAGEM PARA REDUÇÃO DE MENSAGENS DE CONTROLE EM DETECTORES DE DEFEITOS

por

Rogério Corrêa Turchetti

Dissertação apresentada ao Curso de
Mestrado do Programa de Pós-Graduação em
Engenharia da Produção, área de concentração
em Tecnologia da Informação, da Universidade
Federal de Santa Maria (UFSM, RS), como
requisito parcial para obtenção do grau de
Mestre em Engenharia da Produção

PPGEP

Santa Maria, RS, Brasil

2006

Universidade Federal de Santa Maria
Centro de Tecnologia
PPGEP

A Comissão Examinadora, abaixo assinada,
aprova a Dissertação de Mestrado

UMA NOVA ABORDAGEM PARA REDUÇÃO DE MENSAGENS DE CONTROLE EM DETECTORES DE DEFEITOS

elaborada por
Rogério Corrêa Turchetti

como requisito parcial para obtenção do grau de
Mestre em Engenharia da Produção

COMISSÃO EXAMINADORA:

Prof. Dr. Raul Ceretta Nunes
(Presidente/Orientador — Departamento de Eletrônica e Computação — UFSM)

Prof^a. Dr^a. Ingrid Jansch Porto
(Departamento de Computação Aplicada — UFRGS)

Prof^a. Dr^a. Roseclea Duarte Medina
(Departamento de Eletrônica e Computação — UFSM)

Prof. Dr. João Baptista dos Santos Martins
(Departamento de Eletrônica e Computação — UFSM)

Santa Maria, 15 de maio de 2006

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Corrêa Turchetti, Rogério

Uma Nova Abordagem Para Redução de Mensagens de Controle em Detectores de Defeitos / Rogério Corrêa Turchetti. – Santa Maria: PPGEP, 2006.

82 f.: il.

Dissertação (mestrado) – Universidade Federal de Santa Maria. PPGEP, Santa Maria, BR–RS, 2006. Orientador: Raul Ceretta Nunes.

1. Detectores de defeitos. 2. Tolerância a falhas. 3. Sistemas distribuídos assíncronos. 4. Explosão de mensagens. 5. Reaproveitamento de mensagens. I. Ceretta Nunes, Raul. II. Título.

✍ A confecção desta dissertação foi realizada através do uso de recursos de processamento de textos da ferramenta LaTeX 2_ε

AGRADECIMENTOS

Ao meu orientador e amigo, professor Raul Ceretta Nunes, pelo apoio, pelas longas horas de orientação e principalmente pela confiança depositada em mim. Os momentos de brilhante inspiração certamente foram fundamentais para o desenvolvimento deste trabalho.

Dedico este trabalho a minha esposa Patrícia que sempre foi compreensiva e companheira em todos os momentos, sem palavras para te agradecer. Amo você demais e agradeço a Deus por ter colocado você em meu caminho. E também por ter me dado a família maravilhosa que tenho. Meus sinceros agradecimentos a meu pai Altair, a minha mãe Lúcia e a minha irmã Maritê. Não importa o que eu faça, sei que sempre terei o apoio de vocês.

Durante a realização desta obra inúmeras amizades foram conquistadas. Agradeço aos meus amigos e colegas da UNIFRA e do GMICRO, que sabem o verdadeiro significado da palavra amizade. Outros amigos não se fazem mais presentes, mas deixam muita saudade. Agradeço a minha segunda mãe Geni pelo seu carinho.

Por fim, desculpo-me pela pouca atenção que pude dar a essas pessoas que estiveram ao meu lado, vibrando e sofrendo comigo em todos os momentos.

RESUMO

Detectores de defeitos não confiáveis são amplamente utilizados como bloco básico na implementação de técnicas de tolerância a falhas em sistemas distribuídos assíncronos. Sua utilização nestes ambientes é motivada pela impossibilidade de implementação de protocolos de acordo determinísticos, pois não há como distinguir processos defeituosos daqueles de acesso mais lento. Entretanto, o uso maciço de recursos computacionais exige soluções aplicáveis a sistemas distribuídos de larga escala. Neste contexto, algoritmos tradicionais de detecção de defeitos podem apresentar problemas de escalabilidade, tal como o de explosão de mensagens. O grande número de mensagens enviadas pode comprometer a qualidade de serviço do detector de defeitos e a escalabilidade do sistema.

Esta dissertação visa minimizar o problema da explosão de mensagens de controle geradas pelos algoritmos de detecção de defeitos em ações de monitoramento de processos. Para tal, propõe-se uma nova abordagem para redução do número de mensagens de controle através do reaproveitamento de mensagens. A abordagem explora a manipulação da periodicidade de envio das mensagens de controle, maximizando o reaproveitamento de mensagens, e é composta por duas estratégias: ATF (Adaptação da Taxa de Freqüência), a qual reaproveita mensagens dos próprios algoritmos de detecção para suprir mensagem de controle; e AMA (Aproveitamento de Mensagens da Aplicação), a qual reaproveita mensagens das aplicações clientes para o mesmo objetivo da ATF. Como resultado, têm-se uma abordagem *genérica*, no sentido que pode ser aplicada a qualquer algoritmo de detecção, e *prática*, no sentido que algoritmos tradicionais de detectores de defeitos necessitam apenas alterar a semântica das mensagens de controle para utilizá-la. Através de experimentos demonstra-se que sua aplicação reduz o número de mensagens de controle, minimizando o problema da explosão de mensagens, sem comprometer a qualidade de serviço do detector de defeitos.

Palavras-chave: Detectores de defeitos, tolerância a falhas, sistemas distribuídos assíncronos, explosão de mensagens, reaproveitamento de mensagens.

A New Approach to Reduce Control Messages in Failure Detectors

ABSTRACT

An unreliable failure detector is a basic building block widely used to implement fault tolerance techniques in asynchronous distributed systems. The use of failure detectors comes from the impossibility to implement deterministic agreement protocols in these environments, since it is not possible to distinguish a crashed process from a very slow process. However, the massive use of distributed computational resources claims for solutions applicable in large scale distributed systems. In these systems, traditional failure detector algorithms can present scalability problems, such as control message explosion problem, because a large number of messages could compromise the quality of service of failure detectors and the system scalability.

The goal of this dissertation is minimize the problem of control message explosion generated by failure detector algorithms in large scale processes monitoring. To do that, we propose a new approach to reduce the number of control messages from reusing messages. Our approach explores the manipulation of the interrogation period or heartbeat period, maximizing the reuse of messages, and it is organized by two strategies: ATF (Frequency Rate Adaptation), that reuses failure detector messages to suppress control messages; and AMA (Reusing of Application Message), that reuses client application messages to suppress control messages. As result, the resulting approach is *generic*, in the sense that it could be applied to any failure detector algorithm, and *practical*, in the sense that for its, the traditional failure detectors algorithms need only to change the semantic of control messages. From our experimental results, we demonstrate that our approach reduces the number of control messages, minimizing the message explosion problem, without compromising the quality of service of the failure detector.

Keywords: Failure detector, fault tolerance, asynchronous distributed systems, message explosion, reuse of messages.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

ack	*acknowledge*
AFDService	*Serviço de Detecção de Defeitos Adaptativo*
AMA	*Aproveitamento de Mensagens da Aplicação*
ATF	*Adaptação da Taxa de Freqüência*
CF	*Com Falhas*
DoS	*Denial of Service*
EA	*Expected Arrival*
FD	*Failure Detector*
FIFO	*First-In First-Out*
FD_L	*Lazy Failure Detector*
FS	*Failure Suspector*
FS	*Hewlett-Packard*
IP	*Internet Protocol*
LAN	*Local Area Network*
P_A	*Query accuracy probability*
QoS	*Quality-of-Service*
rtt	*round trip time*
St	*Stabilization time*
SF	*Sem Falhas*
TI	*Tecnologia de Informação*
T_D	*Detection time*
T_{FG}	*Forward good period duration*
T_G	*Good period duration*
T_M	*Mistake duration*
TMR	*Triple Modular Redundancy*
T_{MR}	*Mistake recurrence time*

UML	*Unified Modeling Language*
UDP	*User Datagram Protocol*
λ_M	*Average mistake rate*

SUMÁRIO

1 INTRODUÇÃO

Através da história é possível observar o desenvolvimento e o progresso da humanidade e suas constantes mudanças. As inovações surgem principalmente para agilizar tarefas e facilitar a comunicação. Tecnologias como o telegrama, o telefone, o computador, sistemas modernos de bancos, sistemas de alto desempenho, Internet, entre outros, são exemplos de infra-estruturas que revolucionaram costumes e a forma das pessoas interagirem. O computador foi e ainda é um dos principais responsáveis por revolucionar estes costumes e impulsionar diversas áreas de produção no seu desenvolvimento. A TI (Tecnologia de Informação) é um elemento que surge para designar o conjunto de recursos tecnológicos e computacionais para a geração e o uso da informação (REZENDE; ABREU, 2000), tendo como resultado melhoras significativas nos processos de produção das organizações, agregando valores, qualidade e reduzindo custos dos produtos produzidos.

Com o passar dos anos as soluções tecnológicas passaram a ser uma vantagem competitiva e a própria gestão da área de TI precisou passar por transformações, prova disso é o surgimento de diversas áreas de pesquisa que procuram melhorar o planejamento e o desenvolvimento na produção de *softwares*. Neste contexto, algumas características passaram a ser exigidas para que os *softwares* apresentassem garantias de qualidade, tais como: funcionalidade, usabilidade, eficiência, manutenibilidade, portabilidade e confiabilidade. Estas características são fundamentais para que o *software* siga, por exemplo, o modelo de qualidade estabelecido pela norma ISO 9126 (NBR-13596, 1996), e sejam responsáveis por atuações em ambientes críticos como mercado financeiro, sistemas que controlam aeronaves, mísseis, aviões e naqueles que fornecem suporte à vida humana.

Devido a importância de alguns sistemas, é essencial que eles sejam confiáveis, mantendo disponíveis os recursos e componentes responsáveis por executar cada tarefa. Para isso, precisa-se de alguma forma minimizar a possibilidade de ocorrência de defeitos. Entretanto, essa é uma tarefa difícil de ser alcançada, particularmente em sistemas distribuídos, onde a cooperação de componentes também deve ser considerada. Portanto, a solução não se restringe somente em tentar prevenir falhas (*fault prevention*), e sim, na sua ocorrência, tratá-las de forma com que o sistema se mantenha disponível e confiável (JALOTE, 1994), ou seja consiste em tolerar falhas (*fault tolerance*). Ressalta-se que diante da variabilidade de significados das palavras **falha**, **erro** e **defeito**, faz-se necessário, inicialmente, defini-las no contexto desta dissertação: uma **falha** é uma condição física que poderá causar um erro; um **erro** é uma informação corrompida que pode levar a um defeito; e um **defeito** é manifestado quando ocorrer um desvio de especificação, não podendo ser tolerado (AVIZIENIS; LAPRIE; RANDELL, 2000). Assim, quando uma falha for sensibilizada, ela causará um erro e este por sua vez poderá levar a manifestação de um defeito. Entretanto, se o sistema implementar técnicas de tolerância a falhas, um erro não necessariamente causará um defeito, pois valores redundantes podem ser obtidos de outros dispositivos (WEBER, 2001).

Assim, diversas técnicas para fornecer tolerância a falhas em sistemas distribuídos, tanto em componentes (*software*) quanto em recursos (*hardware*), foram propostas ao longo dos anos, como: redundância modular tripla (*Triple Modular Redundancy - TMR*), atualização e restauração de estados consistentes, comunicação em grupo confiável, entre outras. Assim, um sistema que implementa tais técnicas é chamado sistema tolerante a falhas, o qual parte do princípio da replicação de recursos e/ou componentes, ou seja, de redundância (GARTNER, 1999).

Entretanto, com a replicação de recursos e/ou componentes surgem outros problemas que devem ser tratados como, por exemplo, fazer com que eventos sejam executados na mesma seqüência em todas as réplicas do sistema, a fim de manter a consistência dos dados. Freqüentemente a coordenação consistente de tais eventos exige alguma forma de acordo distribuído, mas a realização de tal tarefa não é possível em sistemas puramente assíncronos sujeitos a falhas. Isso se deve à impossibilidade em determinar quando um processo está suspeito ou simplesmente mais lento que os demais (FISCHER; LYNCH; PATERSON, 1985).

Frente a essa limitação, detectores de defeitos (CHANDRA; TOUEG, 1996) surgem como uma solução, pois possibilitam que o consenso possa ser atingido em sistemas assíncronos adicionados com módulos de detecção de defeitos não confiáveis (susceptíveis a erros). Tais módulos de detecção de defeitos trabalham como um oráculo encapsulando o problema do indeterminismo, isto é, eles tentam descobrir os estados funcionais dos processos e fornecem informações suficientes para permitir soluções determinísticas.

Detectores de defeitos trabalham em função da formação e da manutenção de uma visão que consiste nos processos suspeitos. Esta visão é construída através de trocas de mensagens entre processos, os quais possuem um módulo de detecção de defeitos *atachado*. As propriedades dos detectores de defeitos como *completeness* e *accuracy* (CHANDRA; TOUEG, 1996) determinam respectivamente a abrangência e a exatidão na formação das tais visões, em outras palavras, a propriedade *completeness* diz respeito à capacidade do detector de defeitos descobrir todos os processos que estão falhos, ao passo que *accuracy* procura evitar que processos corretos tenham seus estados como suspeitos, ou seja, esta propriedade procura determinar a precisão das suspeitas observadas. Em síntese estas propriedades originam a implementação de diversas características de detecção de defeitos, bem como a possibilidade da formação de uma visão unificada do sistema.

Ao longo dos anos diversas propostas de algoritmos de detecção de defeitos surgiram. Conseqüentemente suas funcionalidades foram sendo melhoradas em diversos aspectos, dentre os quais pode-se destacar: a sua utilização como serviço independente da aplicação, o que possibilita a parametrização de interfaces (FELBER; GUERRAOUI; SCHIPER, 1998), a adaptação dinâmica do *timeout* e dos parâmetros de detecção (CHEN; TOUEG; AGUILERA, 2000; NUNES; JANSCH-PÔRTO, 2003; HAYASHIBARA, 2004), o que possibilita aos detectores de defeitos adaptarem-se às condições da rede e a requisitos da aplicação, e a otimização dos algoritmos para alcançar escalabilidade (RENESSE; MINSKY; HAYDEN, 1998; LARREA; ARÉVALO; FERNÁNDEZ, 1999; FETZER; RAYNAL; TRONEL, 2001; BERTIER; MARIN; SENS, 2003). Entretanto, a diversidade de modelos de sistema e de hipóteses consideradas ainda mantém fértil o campo de novas soluções, mesmo na área da economia de mensagens tratados entre os módulos de detecção de defeitos.

1.1 Definição do Problema

De acordo com Hayashibara, Cherif e Katayama (2002) em sistemas de larga escala algoritmos tradicionais de detecção de defeitos apresentam os seguintes problemas:

- **1. Escalabilidade:** Aplicações destinadas a sistemas de grande escala como Grid oferecem um grande número de recursos distribuídos na rede. Neste ambiente, um detector de defeitos deve ser capaz de eficientemente monitorar um grande número de processos, e de detectar defeitos rapidamente minimizando o número de falsas suspeitas;

- **2. Mensagens perdidas:** Detectores de defeitos devem ser sensivos às condições da rede. Numa rede global, como a Internet, mensagens perdidas ocorrem com grande probabilidade devido a falhas nas transmissões ou atenuação do sinal, as mensagens também podem ser arbitrariamente atrasadas. Estes problemas podem resultar em falsas suspeitas.

- **3. Explosão de mensagens:** De acordo com o grande número de processos que necessitam ser monitorados e da distribuição do sistema, os detectores de defeitos devem prevenir uma inundação de mensagens de controle nos canais de comunicação pelas ações de monitoramento.

- **4. Flexibilidade:** Em um Grid diferentes tipos de aplicação são executadas. Um serviço de detecção de defeitos deve ser capaz de adaptar-se a estas aplicações, isto irá reduzir o número de mensagens de controle enviadas na rede.

- **5. Dinamismo:** Um sistema em Grid é altamente dinâmico, com componentes saindo e entrando no ambiente a todo momento. Um detector de defeitos deve estar atento às reconfigurações e às adaptações do Grid.

- **6. Segurança:** Detectores de defeitos podem ser usados para ataques de DoS (*Denial of Service*), pois eles podem se comportar como um oponente e gerar suspeitas sempre no pior momento.

Esta dissertação explora o problema da explosão de mensagens causadas pelas ações de monitoramento dos detectores de defeitos, item 3 da relação realizada por Hayashibara.

Numa abordagem comum de detectores de defeitos, o monitoramento é executado entre todos os processos participantes, ou seja, cada processo executa o monitoramento trocando informações diretamente com os demais processos. Num ambiente controlado como, por exemplo uma rede local, a explosão de mensagens pode não ser um problema muito comum, pois o atraso é pequeno e a largura da banda (vazão) é grande, se comparados a uma rede de larga escala como a Internet. Em sistemas distribuídos de larga escala, onde o número de processos e os atrasos são imprevisíveis e a largura da banda é restrita, o problema da explosão de mensagens gerado pelo grande número de mensagens de controle enviadas, pode comprometer o serviço de detecção de defeitos e a escalabilidade do sistema.

1.2 Trabalhos Correlatos

Diversas soluções ao problema da explosão de mensagens, também referidas como dificuldade de escalabilidade, foram encontradas na literatura. Como evidência podem-se destacar os seguintes trabalhos: Sargent, Défago e Schiper (2001) propuseram detectores de defeitos implementados junto a protocolos de consenso (mesma aplicação), obtendo como resultado um algoritmo extremamente eficiente em termos de número de mensagens. Seguindo este mesmo objetivo, detectores de defeitos foram definidos em topologias lógicas de rede, por exemplo, Larrea propôs uma topologia em anel (LARREA; ARÉVALO; FERNÁNDEZ, 1999) e uma topologia em estrela (LARREA; FERNÁNDEZ; ARÉVALO, 2000). Outros trabalhos também foram propostos dispondo os processos numa topologia hierárquica (FELBER et al., 1999;

BERTIER; MARIN; SENS, 2003; BURNS; GEORGE; WALLACE, 1999). Já Fetzer et al. (FETZER; RAYNAL; TRONEL, 2001) estavam preocupados no custo gasto pelos detectores de defeitos quando estes são utilizados, neste sentido eles definiram um algoritmo que envia mensagens somente quando necessário. Outros trabalhos poderão ser visualizados no capítulo 3.

Ressalta-se que a abordagem proposta por este trabalho difere-se das demais principalmente pelo fato de ser genérica e poder ser combinada a diversos algoritmos de monitoramento, pois a abordagem parte do princípio do reaproveitamento de mensagens para suprir mensagens de controle.

1.3 Objetivos e Contribuições do Trabalho

Este trabalho tem por objetivo principal propor uma abordagem genérica para tentar contornar o problema da explosão de mensagens, facilitando a escalabilidade dos serviços de detecção de defeitos. Além disto, também tem como objetivo aplicar tal abordagem nos algoritmos de detecção de defeitos que compõem o $AFDService$ (Serviço de Detecção de Defeitos Adaptativo). Como resultado esta dissertação contribui:

- definindo principais estratégias observadas na literatura, que têm por objetivo, reduzir o número de mensagens de controle auxiliando no desenvolvimento de novos protocolos otimizados para este fim;
- propondo uma abordagem genérica, composta por duas estratégias que permitem reduzir o número de mensagens de controle através do reaproveitamento de mensagens;
- dispondo uma interface às aplicações clientes, que permita o reaproveitamento de mensagens da aplicação pelos algoritmos de detecção;
- incluindo os algoritmos de consenso e difusão confiável no AFDService, dando maior funcionalidade ao serviço; e
- demonstrando que a abordagem proposta realmente é eficiente para a redução do número de mensagens, quando comparada com outros algoritmos. Mostrando aspectos positivos e negativos obtidos pela abordagem proposta.

1.4 Estrutura do Texto

No capítulo 2 é apresentado o modelo de sistema, juntamente com definições e algumas notações. Neste capítulo também faz-se uma breve introdução sobre o consenso em sistemas distribuídos assíncronos e detectores de defeitos.

No capítulo 3, inicialmente, realiza-se um vasto estudo relacionado ao problema da explosão de mensagens, para depois definir algumas estratégias para solucionar o problema atacado nesta dissertação.

O capítulo 4 detalha a abordagem para a redução de mensagens, bem como os algoritmos das estratégias que a compõem e define um detector de defeitos *Eventually Perfect* utilizado para o monitoramento de processos.

Um serviço de detecção de defeitos adaptativo é abordado no capítulo 5. Detalhes e questões de implementação também são apresentados neste capítulo.

O capítulo 6 avalia a redução do número de mensagens de controle e o impacto na qualidade do serviço dos detectores quando utilizada a abordagem proposta.

Por fim, o capítulo 7 apresenta as conclusões do trabalho e alguns pontos para possíveis trabalhos futuros.

2 MODELO DE SISTEMA E DEFINIÇÕES

Na área de detectores de defeitos, uma série de hipóteses e formalismos são utilizados, e alguns destes serão definidos neste capítulo. Para contextualizar o ambiente em que os algoritmos de detecção de defeitos deste trabalho serão executados, este capítulo realiza um apanhado geral sobre: hipóteses e modelos utilizados (seção 2.1); problema para realização de acordo em sistemas distribuídos (seção 2.2); conceitos básicos dos algoritmos de detecção de defeitos, das suas classes e propriedades, das práticas de monitoramento, questões de implementação e qualidade de serviço (seção 2.3).

2.1 Modelo de Sistema

Considera-se um sistema distribuído composto por um número finito de processos $\Omega = \{p_1, p_2, ..., p_n\}$ onde cada processo pode se comunicar com qualquer outro processo do sistema. Para todo processo $p_i \in \Omega$ há um relógio interno que funciona independente dos demais relógios. A comunicação entre processos é realizada pelo envio e recebimento de mensagens, através de um canal de comunicação.

Alguns modelos de sincronismo e de comunicação definidos na literatura serão abordados a seguir, enfatizando-se os modelos utilizados neste trabalho.

2.1.1 Sincronismo

Definir o modelo de sincronismo é essencial para o desenvolvimento de algoritmos, pois a especificação do mesmo depende das hipóteses de temporização do sistema. Formalmente, sincronismo é definido nos modelos versando sobre as velocidades dos processos, *hosts* e atraso nas transmissões das mensagens (HAYASHIBARA, 2004).

A seguir apresenta-se uma classificação (DWORK; LYNCH; STOCKMEYER, 1988) de especial interesse para a especificação de algoritmos de detecção de defeitos.

Sistema Síncrono: com um sistema síncrono há um conhecimento temporal prévio do comportamento do sistema, ou seja, cada execução dos processos ou mensagens transmitidas é realizada dentro de um tempo limite previamente conhecido.

Sistema Assíncrono: um sistema assíncrono é caracterizado pela ausência de conhecimento ou limitações temporais de qualquer ordem, com respeito a velocidade dos processos, *hosts* e transmissões de mensagens.

Sistema Parcialmente Síncrono: este sistema trabalha em níveis intermediários de sincronismo (entre o síncrono e o assíncrono), e por isto é chamado de parcialmente síncrono. Em geral, ele assume a existência de um instante de tempo futuro t, desconhecido, em que a partir dele não ocorrem mais falhas de temporização. Vários tipos de modelos parcialmente síncronos foram desenvolvidos como: (i) processos síncronos e comunicação parcialmente síncrona;

(ii) processos e comunicação parcialmente síncronos; (iii) processos parcialmente síncronos e comunicação síncrona.

Neste trabalho assume-se o modelo parcialmente síncrono proposto por Chandra e Toueg (1996) (ii). O modelo assume que para toda execução ou comunicação existem limites temporais denotado por Δ_{lim}. Entretanto esses limites não são conhecidos antes de um tempo *St (Stabilization time)* desconhecido (como no modelo assíncrono). O limite de temporização torna-se conhecido (como no modelo síncrono) após o sistema atingir *St*.

2.1.2 Defeitos nos Processos e Propriedades do Canal de Comunicação

Um processo pode ser considerado correto ou incorreto. Processos corretos comportam-se de acordo com suas especificações, ao passo que processos incorretos podem apresentar defeitos de acordo com a seguinte classificação (JALOTE, 1994):

Defeito por colapso: um processo que falha por colapso cessa seu funcionamento eternamente.

Defeito por omissão: um processo falha intermitentemente por omitir o envio ou o recebimento de mensagens.

Defeito por temporização: um defeito de temporização ocorre quando um processo viola as especificações de temporização assumidas pelo modelo de sistema. Observe que este tipo de defeito é irrelevante em sistemas assíncronos, pois estes se abstraem de qualquer ordem de temporização.

Defeito bizantino: defeito bizantino apresenta um comportamento completamente arbitrário, ou seja, sua ocorrência pode: mudar o conteúdo de uma mensagem, duplicá-la, transmitir mensagens não solicitadas ou, de forma maliciosa, levar o sistema ao colapso.

Por outro lado, um processo correto é definido como um processo que nunca expressa qualquer comportamento defeituoso.

Os algoritmos propostos neste trabalho somente assumem falhas por colapso *(crash)*, ou seja, por suspender sua execução prematuramente. Será denominado *processo correto* aquele que nunca falhou e *processo incorreto/defeituoso* aquele que parou sua execução (*crashed*). O termo *processo suspeito* relata a percepção momentânea de um dado detector de defeitos referente a um determinado processo, o qual não tem certeza de seu estado.

De maneira similar, o canal de comunicação pode ostentar todos os diferentes tipos de defeitos apresentados anteriormente, ou seja, o canal pode apresentar (JALOTE, 1994): defeito por colapso, neste caso nenhuma atividade (transmissão ou recebimento) será realizada; pode perder mensagens (defeito por omissão); pode entregar mensagens depois de longos períodos de atrasos (defeito por temporização); ou ainda comportar-se de maneira arbitrária (defeito bizantino). Ressalta-se que defeitos nos canais de comunicação podem ser confundidos com defeitos nos processos, e vice-versa.

Neste trabalho, o algoritmo de detecção de defeitos assume que o canal de comunicação é confiável (*reliable channel*), ou seja, o canal não **cria**, **altera**, **duplica** e nem **perde** mensagens de controle. E ainda, que o canal não necessita ser FIFO (*First-In First-Out*).

Ressalta-se que a perda das mensagens das aplicações que são reaproveitadas pelos algoritmos de detecção, não implicará no funcionamento da abordagem proposta nesta dissertação. Maiores detalhes podem ser vistos no capítulo 4.

2.2 O Problema do Consenso

Num consenso, um conjunto de processos Π deve por unanimidade concordar numa decisão obtida. Esta decisão é relacionada ao grupo de valores inicialmente propostos por qualquer

processo $p_i \in \Pi$. Usualmente a decisão a ser obtida, se refere a um valor binário (por exemplo, 0 ou 1).

Em (CHANDRA; TOUEG, 1996), o problema do consenso é definido sobre duas primitivas, *propor(v)* e *decidir(v)* onde *v* é um valor qualquer proposto por cada participante. O valor decidido deve ser sentenciado por todos os processos corretos. Esta decisão deve satisfazer as seguintes propriedades:

- *Terminação:* todo processo correto decide por um valor em um número finito de rodadas.
- *Validade:* se um processo decidir por v, então v foi proposto por algum processo.
- *Integridade:* todo processo decide no máximo uma vez.
- *Acordo:* dois processos corretos não decidem diferentemente.

De fato, em sistemas síncronos ou isentos de falhas as propriedades podem ser garantidas e o consenso pode ser resolvido trivialmente. Entretanto, o principal impasse para resolvê-lo ocorre em ambientes assíncronos sujeitos a falhas, onde não há algoritmos determinísticos que possam solucionar o consenso (FISCHER; LYNCH; PATERSON, 1985). Uma alternativa é utilizar um modelo parcialmente síncrono formado por um assíncrono incrementado com detectores de defeitos não confiáveis. Para o protocolo de consenso, o detector de defeitos é um oráculo que permite encapsular o indeterminismo.

Neste trabalho será utilizado o protocolo de consenso proposto por Chandra e Toueg (1996) como uma aplicação. Os aspectos de implementação do algoritmo de consenso são abordados detalhadamente no Capítulo 5 e os resultados dos testes avaliados com este protocolo são mostrados no Capítulo 6.

2.3 Conceitos Básicos Sobre Detectores de Defeitos

Diversas dificuldades são impostas pelos modelos assíncronos (FISCHER; LYNCH; PATERSON, 1985). Tais dificuldades referem-se ao problema do indeterminismo, já comentado neste trabalho. Neste sentido, para a implementação de detectores de defeitos, assume-se neste trabalho o modelo parcialmente síncrono apresentado na seção 2.1.

Seguindo esta abordagem, um detector de defeitos inserido em ambiente parcialmente síncrono pode contar a passagem do tempo para os intervalos de comunicação ou para outras atividades. Com isso, pode-se fornecer informações suficientes para obter conhecimento sobre os estados dos processos; o estado pode ser operacional ou suspeito. Dessa maneira, um algoritmo de detecção de defeitos é projetado para realizar ações de monitoramento através de trocas de mensagens, onde se denomina *processo monitor* aquele que coleta informações de estados dos chamados *processos monitorados*.

2.3.1 Detectores de Defeitos Não Confiáveis

A noção de detectores de defeitos não confiáveis foi formalizada por Chandra e Toueg (1996). Um detector de defeitos pode ser visto como um grupo de módulos, onde cada processo p_i possui um módulo FD_i [1] anexado a ele, e p_i não pode ser considerado suspeito independentemente de seu FD_i. Um processo p_i pode consultar seu módulo de detecção FD_i com a finalidade de obter informações referentes aos estados de outros processos. A informação retornada por um detector de defeitos pode ser incorreta, ou seja, um processo correto pode ser

[1] Detector de defeitos do processo p_i.

considerado como suspeito e um processo incorreto pode ser considerado correto. Além disso, um detector de defeitos pode fornecer informações inconsistentes como: em um dado momento t, é possível que um FD_i suspeite de um processo p_x e FD_j não. Detectores de defeitos são classificados de acordo com duas propriedades (CHANDRA; TOUEG, 1996): (1) abrangência (*completeness*) e (2) precisão (*accuracy*). A abrangência caracteriza a capacidade do detector de defeitos suspeitar de todos os processos defeituosos *(crashed)*, enquanto que a precisão caracteriza a capacidade do detector de defeitos não suspeitar de processos corretos. De acordo com a propriedade do detector ele pode pertencer às seguintes classes:

- (1) STRONG COMPLETENESS: em algum instante futuro, *todo* processo falho será permanentemente suspeito por *todos* processos corretos.
- (1) WEAK COMPLETENESS: em algum instante futuro, *todo* processo falho será permanentemente suspeito por *algum* processo correto.
- (2) STRONG ACCURACY: *nenhum* processo é suspeito antes de ter falhado.
- (2) WEAK ACCURACY: existe *pelo menos um* processo correto que jamais será suspeito.
- (2) EVENTUAL STRONG ACCURACY: em algum instante futuro, *todos* os processos são considerados suspeitos somente após falharem.
- (2) EVENTUAL WEAK ACCURACY: em algum instante futuro, *algum* processo correto nunca é suspeito antes de ter falhado.

Realizando uma combinação entre os níveis (1) e (2), podem ser deduzidas oito classes de detectores de defeitos apresentadas na tabela 2.1.

Tabela 2.1: Relação das classes de detectores de defeitos

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

Cada classe pode ser identificada de acordo com suas características. Um detector $\mathcal{P}$ chamado *Perfect* deve satisfazer as propriedades de *Strong Completeness* e *Strong Accuracy*. Já o detector $\Diamond\mathcal{S}$ que contempla as propriedades de *Eventual Weak Accuracy* e *Strong Completeness*, refere-se a classe mais fraca para resolver o consenso. Entretanto, *Weak Completeness* e *Strong Completeness* podem ser equivalentes, ou seja, um detector que satisfaça a propriedade de *Weak Completeness* pode ser transformado para um detector que satisfaça *Strong Completeness*. Isto significa que o detector $\Diamond\mathcal{W}$ e $\Diamond\mathcal{S}$ são equivalentes para resolver o consenso. Esta ocorrência é devido a facilidade de propagação de uma suspeita para todos os outros detectores, fazendo com que todos suspeitem de um processo falho.

Fetzer (2001) mostrou como um detector de defeitos da classe $\Diamond\mathcal{S}$ pode ser transformado num detector que nunca comete erros (por exemplo $\mathcal{P}$). Esta habilidade foi garantida forçando o sistema a falhar a cada suspeita do detector $\Diamond\mathcal{S}$. O protocolo proposto faz uso de *watchdogs* (cães de guarda) que atuam como injetores de falhas, assegurando que o detector de defeitos jamais erre nos anúncios de suspeitas.

Neste trabalho é utilizado um detector de defeitos da classe $\Diamond\mathcal{P}$, e sua especificação é explanada no capítulo 4.

2.3.2 Modelos Tradicionais de Detectores de Defeitos e Questões de Implementação

Inicialmente foram projetados dois protocolos básicos para os detectores de defeitos em redes locais, ambos utilizando *timeouts* para controlar limites de tempo de espera. Estes algoritmos foram denominados de *Push* e *Pull* (FELBER; GUERRAOUI; SCHIPER, 1998), embora também possam ser referenciados por outros nomes como, por exemplo, *Heartbeat* ou *Ping-style*, respectivamente (HAYASHIBARA, 2004).

Ressalta-se que diante da nomenclatura definida aos protocolos de detecção de defeitos, existem algumas divergências que às vezes podem confundir o leitor iniciante. Por exemplo, o detector *Push* também é referido por *Heartbeat* devido a característica das mensagens enviadas, que são mensagens de vida (*I'm alive*) periódicas. Entretanto, não se deve confundir com o detector de defeitos *Heartbeat* proposto por Aguilera, Chen e Toueg (1997) que possui características diferenciadas do detector *Push*, onde a cada mensagem de vida recebida é incrementado um contador eliminando a necessidade de *timeouts*. Com a finalidade de diferenciá-los, neste trabalho, serão utilizados detector *Push*, detector *Pull* e detector *Heartbeat*. Mensagens *Heartbeat* e *I'm alive* serão consideradas cognatas.

2.3.2.1 Detector Push

No algoritmo de detecção *Push*, as mensagens de controle geradas pelos detectores seguem o mesmo sentido do fluxo das informações. Os processos monitorados por um detector de defeitos enviam periodicamente mensagens *Heartbeat* indicando que eles ainda estão operacionais. Caso o processo monitor não receba uma mensagem dentro de um limite de tempo especificado, na visão do detector o processo monitorado passa a ser suspeito. O detector *Push* destaca-se pelo fato de ser eficiente no número de mensagens trocadas, uma vez que o fluxo é unidirecional.

A figura 2.1 apresenta a troca de informações entre um processo monitor p e um processo monitorado q. A cada mensagem `IAmAlive` recebida por p, ele reinicia o *timeout* correspondente ao processo emissor. Dessa forma, existem certas restrições a serem efetuadas na definição dos parâmetros Δ_i (periodicidade de envio de mensagens) e Δ_{to} (*timeout*). Considerando a figura 2.1, pode-se observar que Δ_{to} deve ser maior que Δ_i (SERGENT; DÉFAGO; SCHIPER, 2001), caso contrário as mensagens *Heartbeat* não chegarão ao seu destino em tempo hábil.

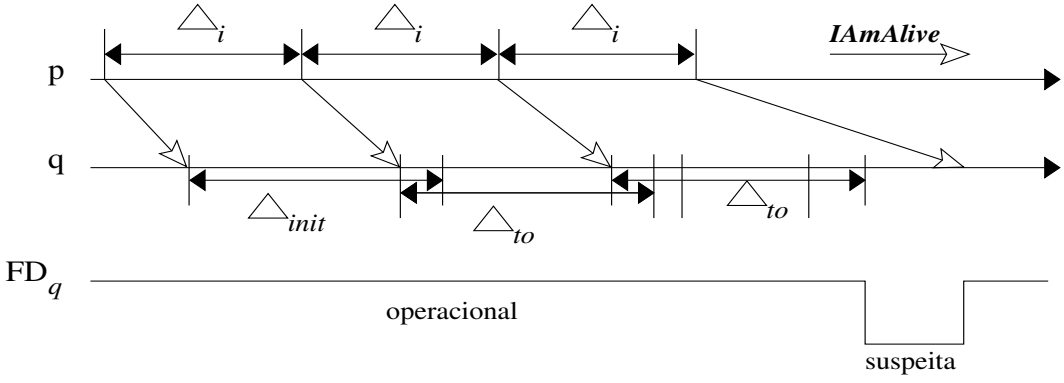


Figura 2.1: Detector de defeitos *Push*

2.3.2.2 *Detector Pull*

No algoritmo de detecção *Pull*, as mensagens de controle seguem no sentido oposto ao fluxo de controle. Os processos monitorados periodicamente são questionados pelo detector de defeitos com uma mensagem de *Liveness Request* (requisição de vida). Se um processo monitorado responder às requisições feitas pelo detector, dentro de um determinado tempo *(timeout)*, significa que ele está operacional. Entretanto se nenhuma resposta for recebida ou se a mensagem recebida não condizer com a respectiva requisição esperada, o processo monitor será considerado suspeito. Este cenário pode ser visualizado na figura 2.2.

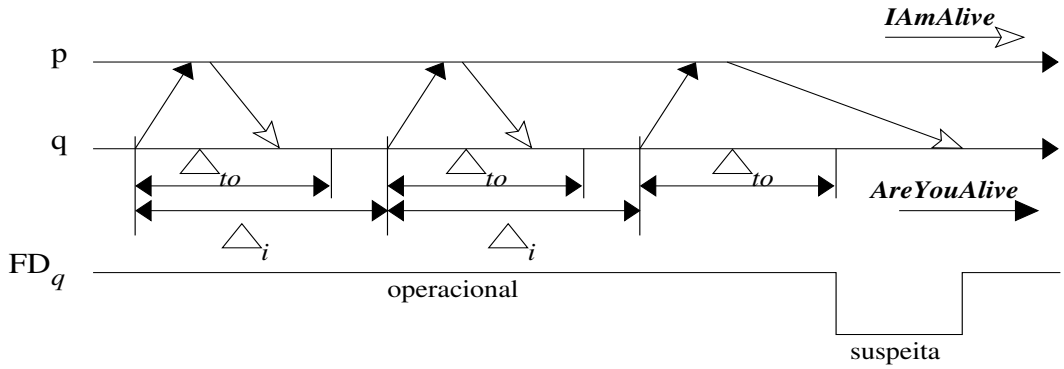


Figura 2.2: Detector de defeitos *Pull*

O algoritmo *Pull* também exige certas restrições a serem efetuadas na definição de seus parâmetros, ou seja, como a latência de detecção deste algoritmo envolve duas fases, onde a primeira corresponde ao envio de mensagens de questionamento e a segunda corresponde ao recebimento das mensagens requisitadas na primeira fase, isto indica que o tempo mínimo para a configuração do Δ_{to}, deverá ser o tempo de seu respectivo rtt (*round trip time*), caso contrário as mensagens de resposta jamais chegarão ao seu destino em tempo hábil.

2.3.3 Discutindo Aspectos de Interesse dos Detectores de Defeitos

Diversos algoritmos para detecção de defeitos são fortemente acoplados às aplicações. Entretanto, esta prática resulta num aumento da complexidade de implementação das mesmas e, como conseqüência, tem-se um serviço personalizado somente para determinada aplicação, o que impossibilita o reaproveitamento de tais serviços. Outra conseqüência nesta abordagem é que se pode ter serviços sendo aplicados de maneira redundante, uma vez que cada aplicação goza de seu próprio serviço.

Neste sentido, uma abordagem mais eficiente é implementar as técnicas de tolerância a falhas discernidas da camada de aplicação, como um serviço em separado (FELBER et al., 1999). Com isto haverá uma redução na complexidade de implementação das aplicações sem prejuízo à qualidade de serviço do detector de defeitos.

Por outro lado, implementar um serviço genérico e que satisfaça as necessidades de diferentes aplicações, passou a ser um problema a ser solúvel. A seguir serão referenciados alguns trabalhos propostos bem como algumas questões a serem discutidas.

2.3.4 Delegação de Responsabilidade

Aguilera, Chen e Toueg (1997) propuseram um detector de defeitos denominado *Heartbeat*. Naquele trabalho, o detector somente possui a função de contabilizar as mensagens recebidas, repassando para a aplicação uma matriz correspondente ao número total de mensagens já rece-

bidas. Por conseguinte, a aplicação deverá comparar os valores da matriz atual com os valores da requisição anterior e, quando a aplicação verificar que algum dos contadores não está mais sendo incrementado, esta passa a suspeitar do processo relativo àquele contador. Como efeito, a aplicação passa a ser responsável por identificar todos os estados dos processos do sistema. Hayashibara (2004) também propôs um serviço de detecção de defeitos denominado *Accural Failure Detector* que delega para a camada de aplicação a ação de interpretação dos dados monitorados, que usualmente é realizado pelo algoritmo de detecção de defeitos.

De fato, delegar principalmente a ação de decidir se um dado processo deverá ser suspeito ou não para a camada superior é uma estratégia bastante interessante, uma vez que, o detector não será intrusivo deixando as aplicações tomarem decisões conforme as suas necessidades. Entretanto é inevitável que a implementação da camada de aplicação se torne mais complexa, sendo que ela própria terá a responsabilidade de indicarem que instante ela passará a suspeitar de determinado processo. Um outro fato dentro desta linha de pesquisa é que, um detector de defeitos é dito não confiável, pois dizer que um dado processo está suspeito ou não, em um sistema distribuído assíncrono, é uma decisão arbitrária e não confiável pelos motivos já citados na seção 2.2. Neste sentido, quando se delega a decisão sobre os estados dos processos a camada de aplicação, pode-se concluir que um detector de defeitos deixa de ser não confiável, passando esta característica para a aplicação.

2.3.5 Conceitos e Definições de Métricas

Para a avaliação do impacto de abordagens/estratégias, um grupo de métricas devem ser criteriosamente levantado, de forma que os resultados possam transparecer o que realmente deseja-se comparar. Uma maneira para preparar este grupo de métricas é listar todos os possíveis pontos a serem avaliados e selecionar aqueles que estão de acordo com os objetivos (JAIN, 1991). Elementos chaves do sistema devem ser selecionados e avaliados exaustivamente, pois estes elementos podem representar gargalos, causando uma degradação na resposta do serviço como um todo. Em geral, procura-se definir limites como linhas mestras. Estas linhas podem ser extraídas das experiências no ambiente de produção, resultados de *benchmarks* e experiência pessoal. Por exemplo, Estefanel (2001) define limites do ambiente de produção para medir quais as taxas de envio e recebimento de mensagens que a aplicação e o ambiente de programação suportam. Esta experiência permite, por exemplo, detectar o tempo médio que uma mensagem leva para sair da aplicação origem atingir a aplicação destino e retornar a origem novamente rtt. Do ponto de vista de um algoritmo de detecção de defeitos do estilo *Pull*, isso significa que se for definido um *timeout* inferior ou aproximado ao valor médio obtido, o serviço corre sérios riscos de realizar detecções incorretas, o que causaria um impacto negativo para as análises.

No âmbito de detectores de defeitos, Chen, Toueg e Aguilera (2000) propuseram um grupo de métricas que permitem avaliar a qualidade de serviço dos algoritmos de detecção. As principais métricas são referentes à **velocidade** com que os serviços conseguem suspeitar de processos defeituosos e à **exatidão** destas suspeitas. Note que a **velocidade** corresponde aos processos que deixam de operar enquanto que a **exatidão** refere-se aos processos operacionais.

Segundo Chen, Toueg e Aguilera (2000), a **velocidade** de detecção de defeitos não é uma tarefa difícil de ser executada, pois envolve capturar o período entre o instante exato em que um processo p falhou e o instante em que o detector de defeitos inicia a suspeitar permanentemente de p. Esta métrica é denominada de **detection time** (tempo de detecção).

Entretanto, o impasse está em como medir a **exatidão** dos detectores de defeitos. Para evidenciar a complexidade desta tarefa, considere-se um sistema com dois processos p e q com o detector de defeitos localizado em q que monitora o processo p. O detector de defeitos deve oferecer duas possíveis saídas para determinar o estado de p: 'p *está operacional*' ou 'p *é*

suspeito'. Os estados podem alternar de tempos em tempos, e para o propósito de medir a **exatidão** do detector será assumido que o processo p não falha por colapso. Considere-se uma aplicação cliente que pergunta de forma randômica, ao detector de defeitos, o estado do processo p. Se o detector de defeitos retornar o estado correto de p obviamente ele estará assegurando a medida de **exatidão**. Esta métrica de QoS é denominada de **query accuracy probability** (probabilidade de exatidão - P_A). Por exemplo, na figura 2.3, a probabilidade do detector de defeitos FD_1 responder exatamente o estado do processo p é $12/(12+4) = 0.75$.

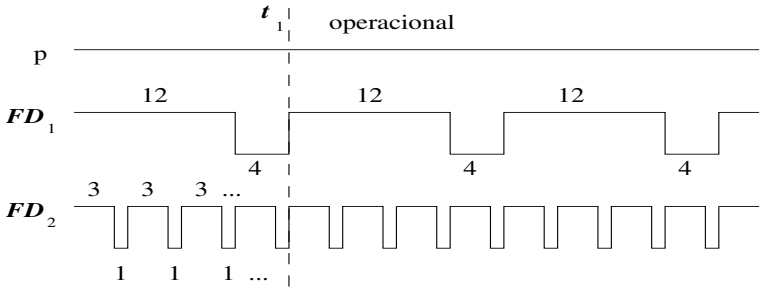


Figura 2.3: Os detectores FD_1 e FD_2 possuem a mesma probabilidade de exatidão (0.75), mas FD_2 erra 4 vezes mais do que FD_1

Por outro lado, sozinha P_A não é suficientemente segura para descrever a exatidão de um detector de defeitos, pois se for analisado o detector de defeitos FD_2 da figura 2.3 ver-se-á que os dois detectores possuem o mesmo valor para a métrica da **exatidão** (0.75) mas o número de **falsas suspeitas** (*mistake rate*- ocorre quando um detector de defeitos suspeita de um processo operacional) é na ordem de 4 vezes o valor do detector FD_1. Observe-se que até o instante t_1 o detector FD_2 suspeita erroneamente 4 vezes ao passo que para este mesmo período o detector FD_1 suspeita uma única vez.

Observe-se novamente que considerar somente a falsa suspeita não é o suficiente para caracterizar a **exatidão**, pois assim como foi demonstrado anteriormente que obtendo o mesmo valor para a métrica da **exatidão** não significa garantir o mesmo valor para as **falsas suspeitas**. Se for analisada a figura 2.4 ver-se-á que a recíproca é verdadeira, onde os detectores possuem a mesma taxa para a métrica de **falsas suspeitas** mas diferentes valores para a **exatidão**. No instante t_1 correspondente ao tempo de 16 obtém-se uma taxa de 1/16, ou seja, uma suspeita a cada intervalo de 16 tempos.

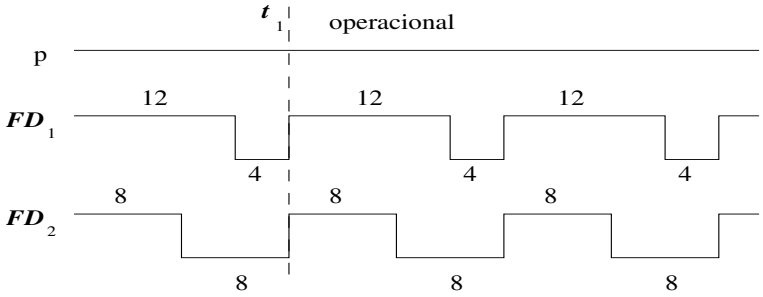


Figura 2.4: Os detectores FD_1 e FD_2 possuem a mesma taxa para falsas suspeitas (1/16), mas a exatidão de FD_1 é 0.75 e de FD_2 é 0.50

Em síntese pode-se concluir que o detector FD_1 em ambos cenários é melhor do que o FD_2. Por exemplo, FD_1 tem alta probabilidade para retornar uma resposta exata e baixa taxa

de falsas suspeitas. Mesmo assim, o FD_2 pode ser melhor que o FD_1 em outros aspectos. Por exemplo, considerando a figura 2.3 pode-se concluir que o tempo de **mistake duration** (duração de falsas suspeitas) do detector FD_2 é menor do que FD_1. Este ponto pode ser importante para determinadas aplicações. Assim, dada a hipótese de que existam dois processos p_i e p_j onde p_i monitora p_j, Chen, Toueg e Aguilera definem que, para especificar a QoS de detectores de defeitos, as seguintes métricas são necessárias:

Métricas Primárias

1. **Tempo de detecção (***Detection time* **-** T_D**).** Mede o tempo decorrido desde o instante em que p_i falha até o instante em que p_j suspeita permanentemente de p_i (vide figura 2.5).

2. **Tempo para recorrência ao erro (***Mistake recurrence time* **-** T_{MR}**).** Determina o tempo entre dois erros consecutivos cometidos pelo detector. O T_{MR} é variado e inicia no instante em que ocorrer a primeira suspeita errada até a próxima (vide figura 2.6).

3. **Duração de um erro (***Mistake duration* **-** T_M**).** Representa o tempo que um detector de defeitos permanece em situação de erro, ou seja, suspeitas incorretas (vide figura 2.6).

Métricas Derivadas

4. **Duração de um período bom (***Good period duration* **-** T_G**).** Representa o período no qual p_i deixa de suspeitar de p_j, ele pode ser expresso por $T_G = T_{MR} - T_M$ (vide figura 2.6).

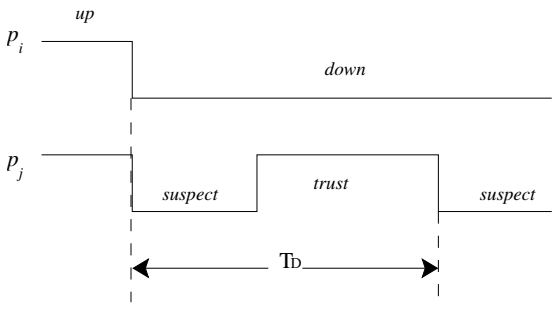


Figura 2.5: Tempo de detecção T_D

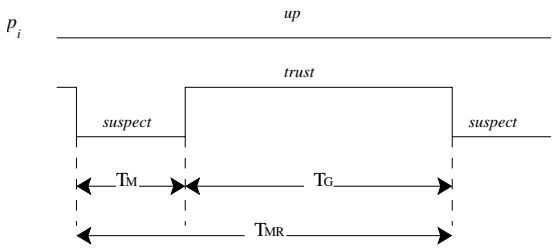


Figura 2.6: Duração de um erro T_M, tempo para recorrer ao erro T_{MR} e duração de um período bom T_G

5. **Média da taxa de erros (***Average mistake rate* **-** λ_M**).** Significa a taxa com que um dado detector de defeitos gera falsas suspeitas, podendo ser representado por $\lambda_M = \frac{1}{E(T_{MR})}$.

6. **Probabilidade de uma resposta exata (***Query accuracy probability* **-** P_A**).** É a probabilidade com que detectores de defeitos geram saídas corretas em instantes aleatórios.

7. **Duração de um período bom a frente (***Forward good period duration* **-** T_{FG}**).** Representa uma variável randômica que indica o tempo restante do período bom. Tempo que resta para p_i continuar a acreditar em p_j.

Salienta-se que as métricas primárias apresentadas anteriormente serão utilizadas neste trabalho e seus respectivos cálculos serão apresentados no capítulo 6.

2.4 Conclusões Parciais

No presente capítulo, foi abordado um apanhado geral sobre hipóteses e formalismos assumidos para a construção do modelo de sistema utilizado nesta dissertação. Definições como: o modelo de sincronismo, tipos de defeitos nos processos, tipos de defeitos no canal de comunicação, nomenclaturas utilizadas, entre outras, são necessárias para contextualizar as condições em que os algoritmos propostos obtêm um funcionamento adequado e para o entendimento dos demais capítulos.

Também foi apresentada uma breve descrição sobre o problema de atingir um consenso em sistemas distribuídos assíncronos sujeitos a falhas. Uma introdução sobre detectores de defeitos e suas propriedades foram expressas. Referindo-se ao aspecto de interesse dos detectores de defeitos, observou-se (seção 2.3.3) que alguns autores propuseram algumas soluções que retiram o papel de detecção de defeitos dos algoritmos de detecção. Entretanto ao retirar esta funcionalidade dos detectores e repassá-la a camada de aplicação, é inevitável o aumento da complexidade da aplicação que utiliza o referido serviço.

Questões de monitoramento, implementação e métricas para a avaliação dos detectores de defeitos também foram vistas. No próximo capítulo, um estudo detalhado sobre o problema da explosão de mensagens será abordado.

3 REDUÇÃO DO TRÁFEGO E TRABALHOS EM DETECTORES DE DEFEITOS

O problema da explosão de mensagens de controle nos canais de comunicação, causadas pelas ações de monitoramento de processos, foi e está sendo analisado por diversos pesquisadores. Como evidência disto, serão apresentados neste capítulo, diversos trabalhos que buscam a redução do número de mensagens em algoritmos de detecção de defeitos.

Inicialmente apresenta-se e discute-se um amplo referencial teórico a respeito do assunto da explosão de mensagens. Fundamentando-se nos trabalhos relacionados, define-se algumas estratégias observadas para possível minimização do problema abordado nesta dissertação.

3.1 Uma Análise Sobre Detectores de Defeitos

Na literatura sobre detectores de defeitos, encontram-se diversos projetos que podem ser relacionados ao problema da explosão de mensagens nos canais de comunicação. Nesta seção apresenta-se e discute-se tais trabalhos, evidenciando as estratégias utilizadas por cada projeto para a redução de mensagens de controle.

3.1.1 Modelos Tradicionais *Pull* e *Push*

Os detectores de defeitos *Pull* e *Push*, apresentados no Capítulo 2, possuem aspectos positivos e negativos. Por exemplo, no detector *Pull* os processos monitorados não necessitam ser ativos, o que proporciona ao detector maior flexibilidade no monitoramento dos processos. Considere o caso em que se está utilizando um detector *Push* e, por exemplo, há a necessidade de configurar o *timeout* dinamicamente de acordo com o tempo de detecção requisitado pela aplicação, ou pela variação observada no ambiente. Neste contexto, este algoritmo de detecção realizaria esta tarefa de forma ineficiente se comparado ao modelo *Pull*, uma vez que os parâmetros estão centralizados no processo monitor proporcionando maior flexibilidade, e assim, facilitando a prática do dinamismo, ou melhor, da implementação de um detector de defeitos adaptativo.

Por outro lado, considerando a questão de número de mensagens geradas no canal de comunicação, o detector de defeitos *Push* é consideravelmente mais eficiente, pois em um ciclo de detecção deverão ser geradas *n(n-1)* mensagens, enquanto que no detector *Pull* serão geradas *2n(n-1)* mensagens em cada ciclo.

Em síntese, a questão da explosão de mensagens é mais bem solucionada utilizando um modelo *Push*, pois nas mesmas condições este possui uma economia de mensagens de 50% em relação ao modelo *Pull*.

3.1.2 O Protocolo *Gossip*

O protocolo *Gossip* (RENESSE; MINSKY; HAYDEN, 1998) é conhecido por disseminar sua informação através de 'fofocas' (*Gossip*), ou seja, a cada informação nova que ele recebe, ele conta aos seus vizinhos. Os vizinhos podem ser uma sub-rede. Neste protocolo duas variações foram propostas, onde uma é denominada de *Gossip* básico e a outra de *Gossip* multi-nível. Na versão *Gossip* básico, um processo contém uma lista com o valor de um contador denominado de *heartbeat counter*, para cada um dos seus vizinhos. O membro ocasionalmente envia sua lista para um membro escolhido randomicamente. Quando recebe a lista, realiza a união com sua lista e adota o maior *heartbeat* para cada processo. Caso o contador *heartbeat* de um dado processo p_i, contido na lista local de um processo p_j, não for incrementado antes de um dado limite de tempo, o processo p_j suspeitará do processo p_i.

A escolha randômica para a disseminação da informação já é uma estratégia que permite reduzir o número de mensagens de controle se comparada à disseminação de mensagens de todos para todos. Entretanto na versão multi-nível este protocolo permite reduzir ainda mais o número de mensagens de controle permitindo maior escalabilidade. O multi-nível usa a estrutura de domínios e seu mapeamento baseia-se em endereço IP, o que possibilita identificar domínios e sub-redes e mapear diferentes níveis. Como a maioria das mensagens é enviada pelo protocolo básico na sub-rede e poucas mensagens são trocadas entre os domínios, esta versão é mais escalável.

Observa-se na literatura que este algoritmo é bastante utilizado com uma topologia hierárquica (FELBER et al., 1999; BERTIER; MARIN; SENS, 2003; BURNS; GEORGE; WALLACE, 1999) pois desta forma é possível realizar um limite na **abrangência de monitoramento**, obtendo como resultado uma grande flexibilidade, escalabilidade e redução de mensagens de controle.

3.1.3 Detector de Defeitos que Atende Requisitos da Aplicação

A preocupação com a inundação de mensagens nos canais de comunicação pelos algoritmos de detecção foi um problema já considerado por Cosquer, Rodrigues e Veríssimo (1995). Eles propuseram uma abordagem para dar suporte ao desenvolvimento de aplicações confiáveis sobre ambientes escalares (por exemplo, Internet).

A proposta é ter um algoritmo de detecção configurável, denominado de *Failure Suspector* (FS), onde as aplicações distribuídas possuem o privilégio de especificar parâmetros importantes a elas, satisfazendo assim algumas exigências como, por exemplo, o tempo para a detecção de um defeito. Esta abordagem tem como principal objetivo o desenvolvimento de aplicações confiáveis de uma forma eficiente, e endereça o problema da explosão de mensagens, através da **adaptabilidade dos parâmetros** como, por exemplo, o ajuste da periodicidade de envio de mensagens para o monitoramento dos processos. A periodicidade pode ser adaptada para alta ou baixa (*high or low*) conforme as exigências da aplicação.

Apesar desta solução ser interessante, ela é incapaz de oferecer suporte a diversas aplicações simultâneas com diferentes requisitos, o que a torna pouco flexível. Este projeto foi um alicerce para diversos outros trabalhos relacionados (CHEN; TOUEG; AGUILERA, 2000; NUNES, 2003; HAYASHIBARA, 2004).

3.1.4 Algoritmo Especializado *Silent*

No trabalho proposto por Sergent, Défago e Schiper (2001), foi realizada uma análise do impacto de detectores de defeitos nos protocolos de consenso. Desta análise surgiram algumas observações interessantes. Eles constataram que os algoritmos de detecção são necessários

somente em determinados pontos, denominados pontos críticos na execução de um protocolo de consenso e que fundidas à aplicação podem trocar poucas mensagens.

A idéia é especializar (*ad-hoc*) um detector de defeitos acoplado ao protocolo de consenso. Em outras palavras, criar um único algoritmo responsável tanto por detectar defeitos quanto por fornecer um serviço de acordo distribuído. Com esta abordagem é possível obter um serviço que reduz eficientemente o número de mensagens de controle geradas nos canais de comunicação e atingir o consenso em menor tempo, se comparado às condições em que se está utilizando os dois serviços dissociados.

A figura 3.1 ilustra a noção de mensagens críticas denominada por Sergent. Segundo a abordagem, uma mensagem crítica pode ser caracterizada por instantes em que uma dada mensagem enviada necessita de uma resposta dentro de um limite de tempo para a progressão do algoritmo de consenso. Este cenário pode ser visualizado na figura 3.1 onde o processo q envia mensagens críticas (m_q) para o processo p. Para que o processo q seja capaz de suspeitar de p, caso este não responda a uma dada entrada, q implementa o mecanismo de *timeout* Δ_{to}. Assim o processo q poderá suspeitar do processo p quando transcorrer o tempo máximo de espera sem receber uma resposta de p.

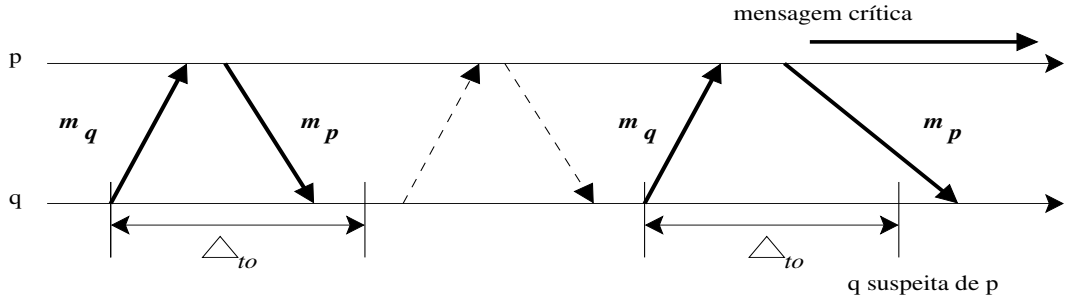


Figura 3.1: Mensagens críticas na implementação do detector de defeitos *Silent*

De fato, a solução proposta por Sergent é bastante atraente, principalmente pela definição utilizada por ela de que um protocolo de consenso só necessita de um algoritmo de detecção em determinados instantes da sua execução, e que utilizar um algoritmo que somente gere mensagens em instantes extremamente necessários, contribua para o desempenho do sistema. Entretanto o que leva ao questionamento é a carência de flexibilidade deste serviço, uma vez que um protocolo de detecção não poderá ser reaproveitado, pois ele é especializado ao serviço de consenso. Além disso, há outra desvantagem quanto ao serviço de detecção de defeitos, pois existe uma grande possibilidade de ocorrerem falsas suspeitas. Considere-se o caso da figura 3.2 em que um processo p depende das respostas m_q vinda do processo q e m_r vinda do processo r, para seguir sua execução normalmente. No entanto supondo que o processo r está lento e atrase para enviar m_r para p, o processo q pode transcorrer seu tempo de espera e atingir o limite do *timeout* passando este a suspeitar erroneamente do processo p.

Esta situação pode ser análoga ao protocolo de consenso no instante em que o coordenador, por exemplo, processo p, envia uma mensagem m_p com o valor de *propor* para os processos q e r e aguarda uma resposta para posteriormente decidir sobre o valor proposto.

O problema das falsas suspeitas foi solucionado com uma especialização ao algoritmo *Silent*. A principal vantagem do novo algoritmo é minimizar o número de falsas suspeitas dando continuidade a um serviço otimizado em termos de número de mensagens, embora este novo algoritmo não consiga ser tão eficiente quanto ao *Silent*.

A solução proposta neste algoritmo também é baseada em pontos críticos e no **reaproveitamento de mensagens** para suprir mensagens de controle dos detectores de defeitos. A figura

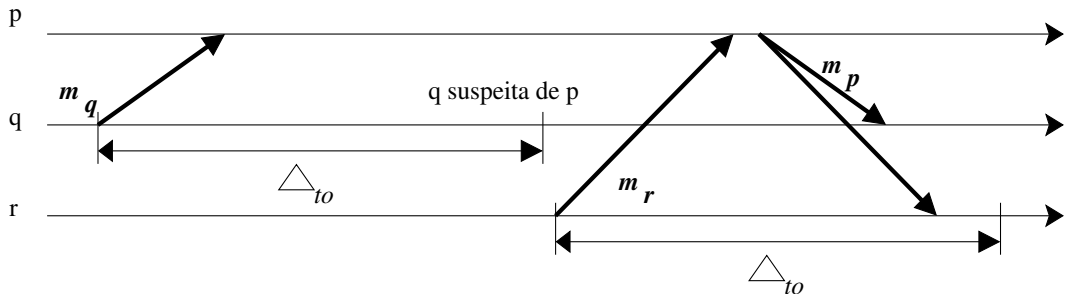


Figura 3.2: Execução do algoritmo *Silent* com suspeitas incorretas

3.3 apresenta a execução do algoritmo com três processos onde o processo q envia uma mensagem crítica ao processo p, no instante em que p recebe a mensagem crítica, ele inicia o envio de mensagens de *IAmAlive*, indicando ao processo q sua operacionalidade. A emissão dessas mensagens somente será cessada após p enviar a mensagem m_p, assim o processo q não irá gerar falsas suspeitas como ocorreria com o algoritmo *Silent*, ao perceber que apenas o processo p não teria ainda enviado sua mensagem crítica.

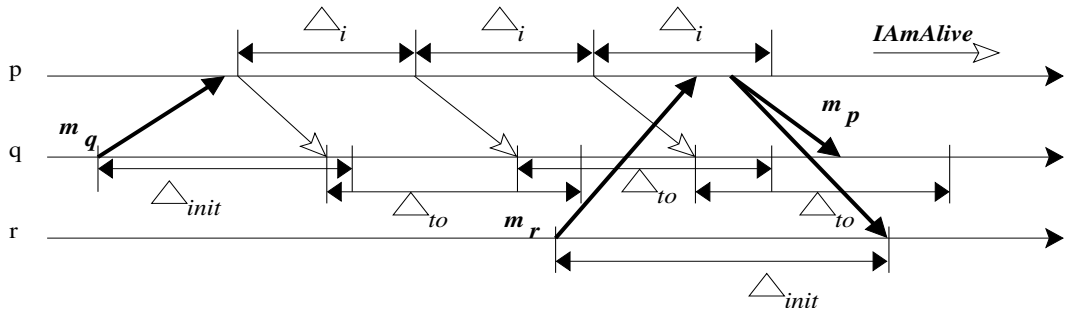


Figura 3.3: Execução do algoritmo *Heartbeat* evitando falsas supeitas

Em síntese, para a obtenção de um serviço confiável é inevitável a emissão de mensagens de detecção, no entanto o que se pode fazer para que as mensagens não passem a ser uma sobrecarga para o sistema é tentar de alguma forma controlar sua emissão. Entretanto, como reduzir a emissão de mensagens de controle sem perder na qualidade de serviço do detector de defeitos?

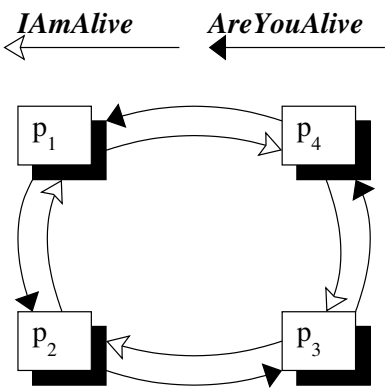
3.1.5 Fluxo de Monitoramento em Anel

Larrea, Arévalo e Fernández (1999) propuseram algoritmo para detecção de defeitos, onde os processos estão combinados num anel lógico. O algoritmo requer propriedades fortes, isto é, supõem-se que as mensagens nunca são perdidas e todos os canais de comunicação são, em algum instante futuro, temporizados (*eventually timely*), ou seja, em um tempo finito (*St*) os canais são temporizados em ambas direções do anel. Isto caracteriza um sistema parcialmente síncrono.

O algoritmo é familiar ao proposto por Chandra e Toueg (1996); no entanto, o algoritmo de Chandra e Toueg baseia-se na comunicação de todos para todos, onde, supondo a existência de n processos no sistema e C processos não defeituosos, pelo menos nC mensagens são periodicamente trocadas entre os processos. No algoritmo proposto por Larrea, a **abrangência de monitoramento** assegura um algoritmo mais otimizado, pois cada processo monitora somente um único processo em uma forma circular. Cada *polling* (consulta dos estados dos processos) envolve a troca de somente duas mensagens entre o processo monitor e o monitorado através

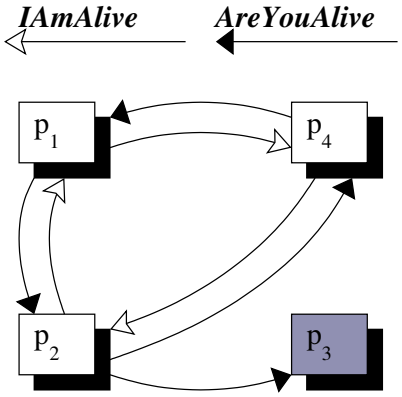
de uma estratégia de **monitoramento passivo**. Se os *pollings* forem feitos periodicamente, não mais do que $2C$ mensagens no total devem ser periodicamente trocadas, provendo assim, uma significativa melhora se comparado com o algoritmo proposto por Chandra e Toueg.

A figura 3.4 apresenta a formação do anel lógico, em que o processo p_i antecede e monitora o processo $p_{(i \bmod n)+1}$; desta forma, o anel é gerado e cada processo saberá qual processo deverá ser monitorado por ele.



2C mensagens enviadas

Figura 3.4: Formação do Anel sem ocorrência de falhas.



p_2 suspeita de p_3

Figura 3.5: Quebra no ciclo do Anel, processo p_3 falhou.

Em algoritmos que trabalham em anel, falhas podem caracterizar uma quebra no anel lógico impossibilitando a comunicação dos processos. Este cenário pode ser visualizado na figura 3.5, em que o processo p_2 envia uma mensagem *AreYouAlive* aguardando uma resposta vinda do processo p_3. Como p_2 não recebe nenhuma resposta, a expiração de um *timeout* indica uma suspeita. Neste caso, o processo p_2 inicia a monitorar o processo subseqüente do processo p_3 no anel, evitando uma partição lógica. Observe que mesmo sob falha o algoritmo mantém a lógica de economia de mensagens.

3.1.6 Fluxo de Monitoramento em Estrela

Em outro trabalho, Larrea, Fernández e Arévalo (2000), com o mesmo objetivo de reduzir o número de mensagens geradas pelos detectores de defeitos, propuseram um outro algoritmo, desta vez com os processos organizados numa topologia em estrela para limitar a **abrangência de monitoramento**, utilizando uma estratégia de **monitoramento ativo**.

Segundo os autores, o algoritmo proposto é melhor do que qualquer outro detector de defeitos da classe $\diamondsuit\mathcal{S}$, em termos do número de mensagens e do total de informações geradas a cada período. Para tal, o algoritmo utiliza-se de um líder chamado p_{leader}, e satisfaz as seguintes propriedades:

- *Eventual Weak Accuracy:* em um tempo futuro, mas finito, p_{leader} não será suspeito por nenhum processo correto;
- *Strong Completeness:* todo processo p_i suspeita de todos os processos do sistema, exceto p_{leader}.

No algoritmo se o processo p_i torna-se candidato a ser p_{leader} o processo p_i começa a enviar periodicamente mensagens do tipo *IAmAlive* para todos os processos p_{i+1},...,p_n, como mostrado

na figura 3.6. Pelo fato de ter apenas um processo no centro da estrela e somente este processo enviar mensagens indicando aos demais sua operacionalidade, este algoritmo irá emitir no máximo *n-1* mensagens de controle, uma vez que somente o processo p_{leader} envia mensagens periodicamente.

No entanto, no contexto de tolerância a falhas este algoritmo aparenta ser ineficiente pois somente um processo será monitorado no sistema, sendo ele o processo p_{leader}. Em caso de falhas dos demais processos do sistema $\{p_{i+1},...,p_n\}$, o algoritmo não será capaz de detectar tornando-o ineficiente como pode ser visualizado na figura 3.7.

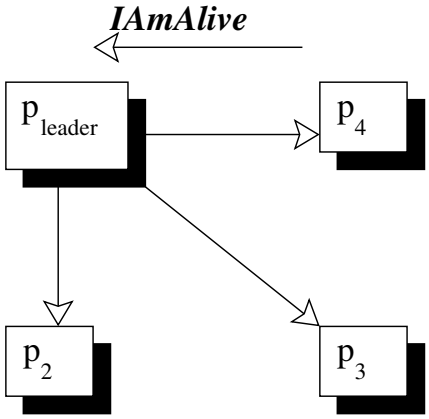


Figura 3.6: Somente o processo p_{leader} é monitorado por todos os demais processos.

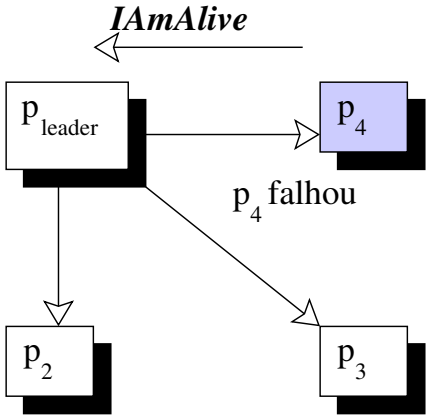


Figura 3.7: Neste caso nenhum processo irá detectar a falha do processo p_4.

3.1.7 Fluxo de Monitoramento Hierárquico

A estratégia para limitar a **abrangência de monitoramento**, por exemplo, dispor os processos de forma hierárquica foi uma alternativa muito utilizada na literatura, como será apresentado nesta seção, para propor serviços de monitoramento em ambientes escalares. Esta organização permite uma forma simples de gerenciamento e acentuada redução no fluxo de informações geradas nos canais de comunicação.

Neste ambiente os processos são divididos em níveis, como pode ser visualizado na figura 3.8. Os níveis crescem verticalmente e cada nível pode conter diversos domínios administrativos, que por sua vez, podem expandir horizontalmente. Cada domínio é gerenciado por um detector de defeitos responsável por monitorar e disseminar informações de estados dos processos do seu domínio para os demais níveis do sistema. Com isso, o número de mensagens entre níveis distintos é reduzido, uma vez que somente um processo se comunica com os demais níveis da hierarquia; esta é uma necessidade inerente aos ambientes escalares como, por exemplo, a Internet, sendo que a distância entre os níveis é imprevisível.

Felber

Referindo-se a projetos que propõem uma **abrangência de monitoramento** de processos através de uma topologia hierárquica, pode-se destacar o trabalho de Felber et al. (1999) onde é apresentada uma arquitetura modular que propõe customizar a comunicação entre diferentes detectores de defeitos.

O trabalho considera uma rede de larga escala em que os processos monitores estão dispostos hierarquicamente, a fim de contornar o problema da explosão de mensagens. A idéia

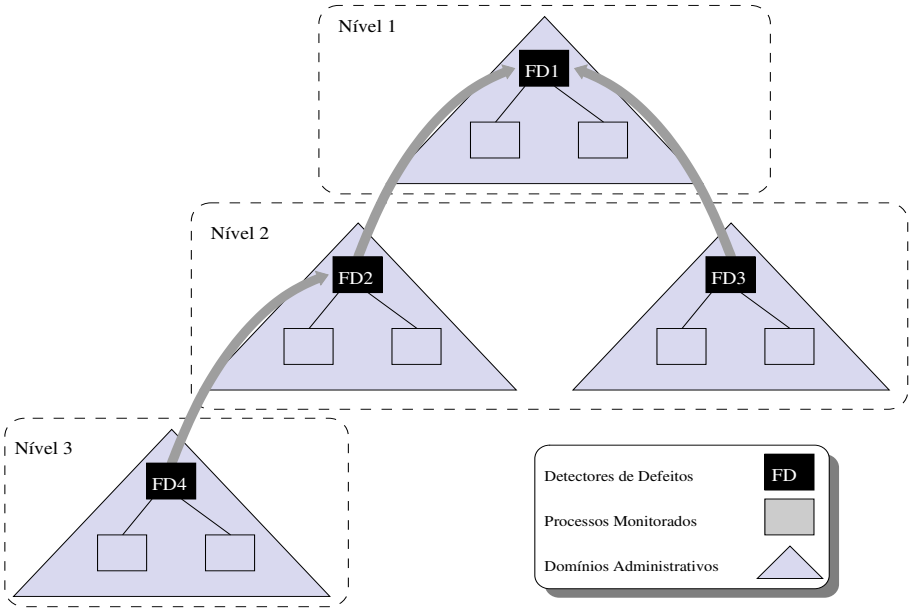


Figura 3.8: Organização Hierárquica

fundamental é reduzir o fluxo de informações trocadas entre processos distantes. Na topologia hierárquica proposta, um ou mais módulos de detecção de defeitos, independente do modelo utilizado (*Push*, *Pull* ou *Dual*), pode monitorar o estado de todos os processos contidos numa determinada LAN e transmitir essas informações de estado para outros módulos remotos localizados em LANs diferentes.

Neste modelo hierárquico reduz-se a comunicação entre LANs, mas se ocorrer algum defeito nos processos intermediários da hierarquia (ou particionamento de rede), grande parte dos processos estarão incapazes de se comunicarem. Para resolver esta questão, Felber et al. utilizaram o protocolo do estilo *Gossip* (RENESSE; MINSKY; HAYDEN, 1998). Este protocolo tende a combinar a flexibilidade hierárquica com sua característica intrínseca de disseminar a informação para seus vizinhos. A principal vantagem deste protocolo é que em vez de selecionar um alvo uniforme para monitorar, que o tornaria uma rede hierárquica fixa, os processos monitores selecionam aleatoriamente cada processo a ser monitorado, transformando o fluxo de monitoramento numa hierarquia dinâmica.

Bertier

Bertier, Marin e Sens (2003) avaliaram o desempenho de um serviço de detecção de defeitos seguindo uma estrutura hierárquica, esse novo serviço é uma versão mutável do detector de defeitos *heartbeat*. O detector de defeitos proposto é dividido basicamente por duas camadas, sendo elas: uma camada básica que fornece um 'bom' tempo de detecção, e uma camada que se adapta às necessidades das aplicações; estas camadas podem ser visualizadas na figura 3.9.

A camada básica (BERTIER; MARIN; SENS, 2002) é composta por um suposto tempo de chegada (*EA - Expected Arrival*) adicionado a uma margem de segurança (α), onde *EA* consiste em uma média dos n últimos *EAs* observados. Assim *EA* possibilita um pequeno tempo de detecção, mas aumenta a probabilidade de ocorrerem falsas suspeitas. A margem de segurança é calculada similarmente à utilizada por Jacobson (GROUP, 2000), ou seja, ela é adaptada de acordo com a carga de rede e baseada nos últimos erros amostrados. O mecanismo de adaptação propõe um intervalo para a periodicidade de envio das mensagens de acordo com as necessidades da aplicação, fixando o menor intervalo requerido para ter o menor tempo de detecção. Um

outro serviço desta camada é transmitir informações da aplicação por *piggy-backing* para assim reduzir as mensagens enviadas na rede; esta estratégia implica no **reaproveitamento de mensagens** para suprir mensagens de controle.

Em síntese, o serviço da camada básica é fornecer, através de um mecanismo denominado *Blackboard*, uma lista de processos suspeitos, um intervalo de emissão de mensagens, uma margem de segurança e a QoS observada.

A camada de adaptação, a qual fornece a **adaptabilidade dos parâmetros** está localizada entre a camada de aplicação e a camada básica. Os serviços fornecidos pela camada inferior (camada básica) é adaptado de acordo com a QoS da camada superior (aplicação), permitindo fornecer um serviço de detecção personalizado para cada aplicação.

O serviço de detecção proposto suporta ambientes de grande escala por meio de uma organização hierárquica que engloba dois grupos (figura 3.10): um grupo local, onde cada LAN (*Local Area Network*) é composta por processos e um líder, sendo a comunicação realizada por *IP-Multcast* (*Internet Protocol - IP*), e um grupo global, composto por todos os líderes de cada LAN, onde o protocolo de comunicação utilizado é o UDP (*User Datagram Protocol*). A comunicação *multicast* utiliza uma estratégia de *endereçamento* para suprir mensagens de controle, ou seja, uma única mensagem pode ser enviada a um grupo de processos participantes.

Esta estrutura proposta pelos autores permite reduzir a complexidade quanto ao problema da explosão de mensagens, pois cada líder monitora o seu grupo local e difunde informações para os demais líderes. Como resultado, tem-se uma redução no fluxo de mensagens.

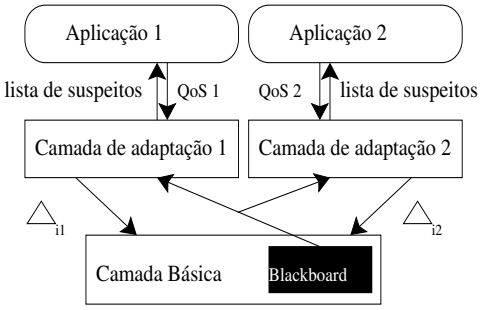


Figura 3.9: Disposição das camadas no FD

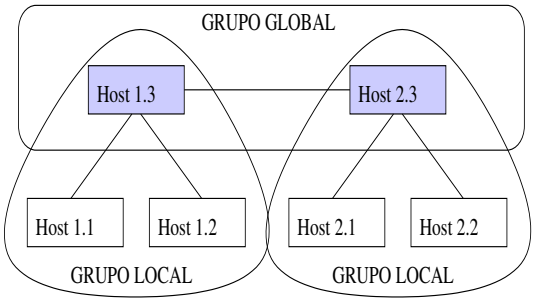


Figura 3.10: Disposição dos processos na rede

Burns

Outro trabalho relevante nesta mesma linha de pesquisa é o proposto por Burns, George e Wallace (1999), onde foi implementada variação do protocolo *Gossip* destinado a sistemas de alto desempenho como *clusters*. Com o objetivo de avaliar os protocolos de acordo com o tempo e o número de mensagens *Gossip* necessárias para atingir um consenso, foi concluído que o protocolo *Gossip* inserido numa topologia hierárquica e utilizando o **reaproveitamento de mensagens** (*piggy-back*) nas mensagens da aplicação, obteve desempenho e escalabilidade superiores sobre as demais variações deste protocolo.

A diferença proposta por Burns foi explorar domínios e subdomínios numa rede de grande escala, em que o protocolo *Gossip* não utiliza uma topologia hierarquia dinâmica como a proposta em Felber et al. (1999). A justificativa, segundo os autores, é de que o protocolo básico não beneficia pequenos grupos, tendo uma grande possibilidade das mensagens serem randomicamente enviadas a destinos inacessíveis; por isso as escolhas dos processos a serem disseminadas as informações são determinadas estaticamente.

3.1.8 Detector de Defeitos *Lazy*

Fetzer, Raynal e Tronel (2001) propõem um protocolo que utiliza a estratégia de **reaproveitamento de mensagens** das aplicações, para economizar mensagens geradas pelo algoritmo de detecção. Segundo os autores somente quando as aplicações não estão trocando informações[1] é que as mensagens de detecção serão necessárias, por isso o detector de defeitos é chamado de *Lazy Failure Detector* (FD_L), ou seja, detector de defeitos preguiçoso.

O trabalho relata também que, quando o sistema satisfaz suposições parcialmente síncronas (modelo proposto por Chandra e Toueg (1996)), o FD_L implementa um detector de defeito *Eventually Perfect* ($\Diamond\mathcal{P}$). Se a média no atraso da transmissão for finita, o detector torna-se *Perfect* ($\mathcal{P}$). As características intuitivas deste protocolo o tornam bastante interessante; entretanto, as mensagens da aplicação utilizadas pelo detector de defeitos têm um proveito de $n(appl)/2$, onde $n(appl)$ é o número de mensagem da aplicação enviada. Este aproveitamento de mensagens deve-se ao fato de que a cada mensagem da aplicação reaproveitada uma resposta ack (*acknowledge*) deve ser retornada como confirmação do recebimento.

A figura 3.11 apresenta um cenário em que o algoritmo utiliza três tipos de primitivas descritas a seguir:

- *Send appl*: primitiva utilizada por algum processo p para enviar uma mensagem m da aplicação para outro processo q;
- *Ack*: esta primitiva sempre é chamada para retornar uma confirmação ao processo emissor de uma mensagem *ping* ou *appl*;
- *Ping*: primitiva responsável por realizar consultas de estados feitas periodicamente pelo FD_L.

Query é um método utilizado pela aplicação para saber sobre o estado de um dado processo. Neste caso, quando ele é invocado, o FD_L retorna imediatamente o estado do processo e realiza uma consulta (*ping*) ao respectivo processo. Caso o processo não retorne um *ack* dentro de um limite de tempo (maior rtt já observado), o FD_L passa a suspeitar do processo consultado informando à aplicação.

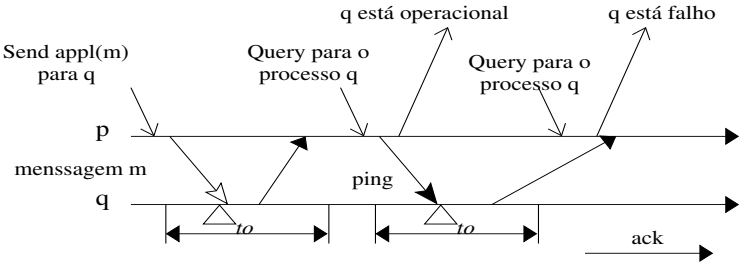


Figura 3.11: Execução do detector de defeitos *Lazy*

De fato, a idéia proposta no artigo obteve êxito na redução do número de mensagens de controle. Entretanto, mesmo quando as aplicações estão trocando informações, serão necessárias mensagens de controle, pois o algoritmo necessita de uma confirmação para cada mensagem da aplicação enviada, o que possibilita questionar o seguinte discurso dos autores: que "somente quando as aplicações não estiverem trocando informações é que as mensagens de detecção serão utilizadas". O fato da mensagem do tipo *ack* ser utilizada pelo FD_L para indicar que o processo

[1] É assumido um canal confiável, ou seja, o canal não **cria**, **altera**, **duplica** e nem **perde** mensagens.

emissor está operacional, torna-a uma mensagem de controle/detecção, o que contradiz a frase mencionada anteriormente.

3.2 Estratégias Para Redução do Tráfego

Nesta seção serão apresentadas algumas estratégias que poderão ser utilizadas como alternativas para otimizar o tráfego de mensagens de controle. Estas estratégias foram definidas baseadas no estudo de mecanismos de monitoramento já validados em outros serviços de detecção de defeitos, como pôde ser observado nas seções anteriores neste capítulo. Desta forma procura-se auxiliar no desenvolvimento de novos protocolos otimizados para este fim, uma vez que serão abordadas algumas possíveis maneiras de realizar esta tarefa.

Salienta-se que podem existir outras estratégias, provenientes das ações de monitoramento, que talvez reduzam a carga na rede de comunicação. Entretanto, as estratégias apresentadas a seguir, correspondem às encontradas na literatura abordada.

3.2.1 Abrangência de Monitoramento

A abrangência de monitoramento implica na sobrecarga da rede, devido ao fato de que, em ambientes onde todos detectores monitoram todos os processos existentes, será emitido um número maior de mensagens se comparado a uma rede dividida por limites de monitoramento, onde cada detector define uma gama de processos (domínio) a serem monitorados por ele, determinando assim uma abrangência de monitoramento. Os estados dos processos que são monitorados por determinado detector, são repassados aos demais módulos de detecção. Desta forma, todos os detectores possuem a informação dos estados de todos os processos, sem a necessidade de questioná-los diretamente.

Na prática, esta questão pode estar relacionada à definição de uma topologia de rede utilizada pelos algoritmos de detecção. Neste contexto pode-se destacar três topologias lógicas de rede:

Topologia em anel: normalmente é estruturada de forma ordenada onde um processo se comunica com outro processo subseqüente do anel; esta estrutura deve ser flexível na presença de falhas, caso contrário falhas podem impossibilitar a comunicação dos processos no anel.

Topologia em estrela: uma topologia em estrela possui a característica de um sistema centralizado, uma vez que sempre deverá existir um componente responsável por receber todas as informações enviadas na rede. Por este motivo esta organização não é muito adotada, além do que este centralizador será um "ponto único de falha". No entanto, endereçando a questão da explosão de mensagens, alguns autores a utilizaram obtendo bons resultados.

Topologia hierárquica: esta topologia é organizada em níveis, onde cada nível é monitorado por determinados processos e estes são responsáveis em fornecer uma visão de estado de todos os processos correspondentes ao seu nível para outros níveis da hierarquia. O algoritmo utilizado para monitoramento de processos numa topologia hierárquica também deve ser flexível na presença de falhas, caso contrário, uma falha em um dos processos pode se caracterizar como um particionamento de rede, impossibilitando a comunicação em diferentes níveis.

3.2.2 Adaptabilidade dos Parâmetros

De forma geral existem dois parâmetros utilizados pelos detectores de defeitos que são: *timeout* (Δ_{to}) e a periodicidade de envio de mensagens (Δ_i). A configuração destes parâmetros está intimamente ligada à qualidade de serviço dos detectores e interfere diretamente no número de mensagens de controle por unidade de tempo. Por exemplo, a periodicidade de envio pode ser controlada conforme as exigências das aplicações. Como resultado tem-se um serviço com melhor qualidade de serviço e emitindo mensagens somente conforme a necessidade da aplica-

ção, ou mesmo, de acordo com as condições do canal de comunicação. Se uma determinada aplicação exigir somente em determinados pontos uma detecção mais rápida (reduzindo o (Δ_{to}) e o (Δ_i)), em situações em que esta necessidade deixar de existir o detector que utilizar parâmetros fixos continuará emitindo mensagens numa periodicidade reduzida, o que resultará na emissão de mensagens desnecessárias.

3.2.3 Formas de Monitoramento

Tradicionalmente existem duas formas de monitoramento que são utilizadas pelos detectores de defeitos: **Monitoramento Ativo** (também chamado de modelo *Push*) e **Monitoramento Passivo** (também chamado de modelo *Pull*). No Monitoramento Ativo, processos monitores aguardam sinais (mensagens) de vida vindos dos processos monitorados. No Monitoramento Passivo, os processos monitores questionam os processos monitorados para detectar a sua operacionalidade.

Todos os algoritmos de monitoramento devem utilizar uma das formas abordadas anteriormente, pois são as únicas formas possíveis de monitoramento. Ou o monitor opta por aguardar um sinal de vida ou ele opta por questionar o estado sempre que achar necessário. Entretanto, optar por uma destas formas pode trazer conseqüências para a redução ou o acréscimo da sobrecarga na rede (vide seção 3.1.1).

3.2.4 Reaproveitamento de Mensagens

Em sistemas distribuídos, a comunicação e a sincronização entre processos ocorrem unicamente por trocas de mensagens, assim qualquer serviço que trabalhe neste ambiente deverá se comunicar através deste mecanismo. Desta forma, estas mensagens podem ser reaproveitadas suprindo assim mensagens de controle.

Em alguns casos, para o reaproveitamento de mensagens, os serviços devem interceptar as mensagens enviadas adicionando informações de controle junto às mensagens reaproveitadas, este mecanismo é denominado de *piggyback*. Observa-se que este mecanismo é uma espécie de 'carona' que as informações de controle pegam para viajar de um ponto ao outro sem consumir banda no canal de comunicação.

3.2.5 Endereçamento

A estratégia de endereçamento trata como a comunicação entre os processos é realizada, ou seja, para quantos processos a mensagem será endereçada. Caso a mensagem seja a mesma, pode-se realizar uma comunicação por difusão utilizando um *multcast* ou *broadcast*, reduzindo a necessidade da criação de diversas mensagens para enviá-la.

Entretanto, na implementação de comunicação em grupo, pode ser necessária a utilização de um protocolo confiável amplamente utilizado em algoritmos de acordo, e neste caso um novo serviço será inserido ao ambiente distribuído. Além disso, a questão de se optar por uma comunicação em grupo ou uma comunicação *unicast* está intimamente relacionada a necessidade de abrangência para cada mensagem enviada. Desta forma, em determinados algoritmos pode ser inconveniente a utilização de um protocolo que abrange vários processos, optando assim pela comunicação ponto-a-ponto.

3.3 Conclusões Parciais

Neste capítulo foi realizada uma revisão bibliográfica sobre alguns trabalhos que consideraram o problema da explosão de mensagens de monitoramento. Foram também levantada algumas possíveis estratégias utilizadas para contornar este problema referenciando aspectos práticos de implementação. Inicialmente o que se observa é uma tendência na utilização de uma topologia lógica para limitar a **abrangência de monitoramento**, tendo como resultado a redução no número de mensagens de controle. Tal tendência pode ser induzida pela necessidade de alterar pouca coisa em um protocolo original para se ter uma solução deste tipo. No entanto, dentre as soluções de topologias, a mais visada é a topologia hierárquica dividida em níveis, ressalta-se que neste trabalho não se utilizou topologia lógica para a redução de mensagens, mas que esta solução seria completamente plausível para uso nas estratégias que serão apresentadas neste trabalho.

Uma outra estratégia, a qual pode ser utilizada combinada com outras soluções é a questão da **adaptabilidade dos parâmetros**. Neste contexto, observa-se uma melhora não somente para a explosão de mensagens de controle, como também para aprimorar a qualidade de serviço dos detectores de defeitos de forma geral. O serviço proposto nesta dissertação utiliza esta estratégia para reaproveitar as mensagens das aplicações. Outra estratégia levantada é a questão da **forma de monitoramento** ativo e passivo, entretanto dentre estas duas formas nós procuramos propor uma solução que não apresentasse nenhuma tendência, ou seja, priorizar a redução de mensagens em ambas formas de monitoramento deixando livre a escolha por determinada forma.

Outra estratégia é o **reaproveitamento de mensagens**; neste contexto observam-se várias soluções que utilizam o conceito de *piggyback*. Esta abordagem é um tanto intrusiva, pois altera-se o pacote enviado pela aplicação para que possam ser adicionadas informações extras utilizadas para reaproveitamento de mensagens. Além disto, utilizar *piggyback* pode trazer ainda atraso na emissão das mensagens se comparada a envios normais (VOGELS; RE, 2003). Entretanto a solução proposta neste trabalho é reaproveitar mensagens das aplicações clientes ou mesmo as próprias mensagens de controle dos detectores, sem adicionar nenhuma informação ao pacote. A questão de **endereçamento** é válida para ambientes controlados que permitem o envio de mensagens por difusão, no entanto o ambiente utilizado nesta dissertação para a execução do serviço de detecção de defeitos é a Internet. Neste ambiente, serviços de difusão são bastante complicados de serem utilizados e a alternativa é utilizá-los combinados a comunicação ponto-a-ponto.

Entre os detectores vistos neste capítulo, o trabalho que mais se assemelha com a solução proposta nesta dissertação (estratégia AMA - vide seção 4.1.2) é o algoritmo *Lazy*, por este motivo serão realizadas outras comparações entre estas duas soluções no decorrer deste trabalho.

4 GERENCIANDO TROCA DE MENSAGENS EM ALGORITMOS DE DETECÇÃO DE DEFEITOS

Tradicionalmente, a ação de monitorar processos e *hosts* em sistemas distribuídos é baseada na troca de mensagens de controle (FELBER et al., 1999). Dependendo do algoritmo utilizado, do número de processos participantes ou mesmo da característica do ambiente, esta ação poderá ocasionar a sobrecarga dos canais de comunicação, devido a explosão de mensagens gerada (HAYASHIBARA; SHERIF; KATAYAMA, 2002).

Um serviço ideal para monitoramento de processos deveria utilizar um algoritmo que não adicionasse nenhuma mensagem extra no canal de comunicação e que provisse alta qualidade de serviço. Entretanto, na prática este algoritmo não pode ser implementado porque o número de mensagens geradas para monitoramento é proporcional à probabilidade de detectar defeitos mais rapidamente (CAMARGOS, 2003). Por isto encontra-se na literatura muitas estratégias para reduzir o número de mensagens de controle em algoritmos de detecção de defeitos, conforme descrito no capítulo anterior

Assim, para reduzir a carga da rede, neste capítulo explora-se o uso de uma nova abordagem que atrasa o envio das mensagens de controle sempre que possível.

4.1 A Nova Abordagem

A abordagem proposta inicialmente considera que quanto maior a carga de trabalho nos canais de comunicação menor será a vazão do serviço para monitoramento de processos. Considerando períodos de grande uso dos canais de comunicação em sistemas distribuídos de larga escala, mensagens dos detectores de defeitos contribuem para um aumento ainda maior. Neste ambiente, diversas aplicações comunicam-se constantemente por troca de mensagens e o propósito da nova abordagem é atrasar o envio das mensagens de controle/monitoramento reaproveitando as mensagens geradas pelas aplicações ou mesmo pelos próprios algoritmos de detecção, sempre que possível.

Assim, mensagens de controle que poderiam estar contribuindo para aumentar a carga da rede podem ser supridas através do reaproveitamento de mensagens possibilitando maior escalabilidade do sistema. A idéia que fundamenta a nova abordagem é simples, entretanto atrasar o envio de mensagens de controle é uma tarefa que deve ser cuidadosamente executada. A abordagem está dividida em duas estratégias distintas as quais serão apresentadas na seqüência.

4.1.1 Adaptação da Taxa de Freqüência (ATF)

A estratégia ATF consiste no reaproveitamento de mensagens de controle reutilizando mensagens dos próprios algoritmos de detecção de defeitos. A ATF altera a semântica das mensagens de detecção, ou seja, o significado das mensagens de controle de um detector que requisita

estados aos processos monitorados.

Tradicionalmente estes detectores de defeitos monitoram os processos enviando, a cada Δ_i unidades de tempo, mensagens de requisição de estado `AreYouAlive` para um processo monitorado e aguardam por uma resposta (`IAmAlive`). Normalmente, para garantir que o processo monitorado esteja operacional é necessário que a resposta seja recebida dentro de um certo período de tempo (modelo parcialmente síncrono assumido). Este estilo de monitoramento possui a característica de monitoramento ativo (vide capítulo 3).

Entretanto quando se aplica a estratégia ATF, alterando a semântica das mensagens nos referidos algoritmos, um processo monitor assume que qualquer mensagem recebida de um processo monitorado (`AreYouAlive` ou `IAmAlive`) indica a vivacidade/acessibilidade do processo emissor naquele instante. Além disto, ele atrasa o envio da requisição de estado reinicializando o temporizador do Δ_i. Apesar desta hipótese parecer óbvia, não se tem conhecimento de proposta similar à implementada neste trabalho, evitando mensagens desnecessárias e como resultado, será provado que a estratégia reduz significativamente o impacto de mensagens de controle.

Ressalta-se que, embora a nova abordagem seja designada para um propósito geral, como será apresentada na próxima seção, a estratégia ATF aplica-se a todos os algoritmos de detecção de defeitos que realizam requisições de estados aos seus processos monitorados. Em síntese, faz-se necessário a utilização de monitoramento ativo pelos detectores de defeitos.

Para demonstrar o funcionamento da estratégia ATF, assuma o seguinte cenário (figura 4.1): um processo p_i monitora, e é monitorado por outro processo p_j. A cada Δ_i unidades de tempo p_i e p_j deveriam enviar mensagens `AreYouAlive` para verificar o estado do processo vizinho. Entretanto, a cada instante que um dos processos receber uma mensagem `AreYouAlive`, o detector assume que o processo emissor está operacional e reinicializa o relógio que controla Δ_i. Como resultado, um processo monitor somente enviará uma mensagem `AreYouAlive` se e somente se ele não receber nenhuma mensagem de controle do processo monitorado em Δ_i instantes de tempo.

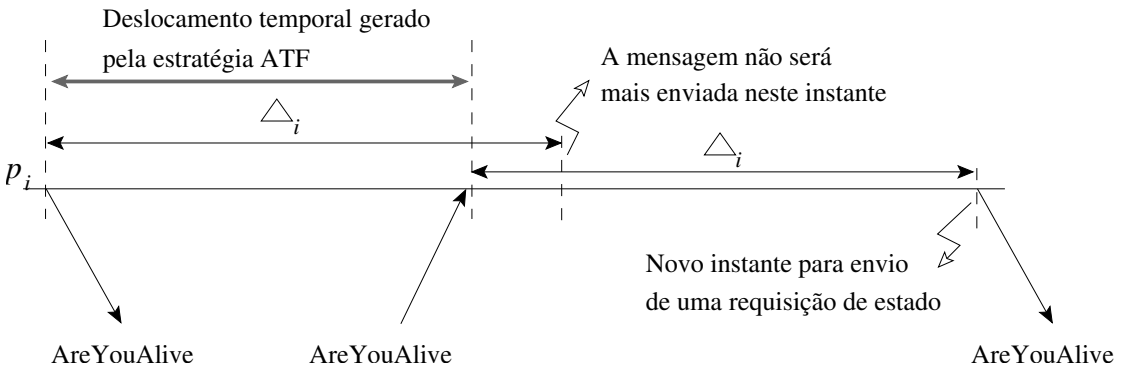


Figura 4.1: Reduzindo o número de mensagens de controle com a estratégia ATF

4.1.1.1 Algoritmo Para a Estratégia ATF

O algoritmo para a estratégia ATF é apresentado em detalhes na figura 4.2. Neste trabalho ele servirá como exemplo para o desenvolvimento de algoritmos de detecção com redução nas mensagens de controle. O algoritmo distribuído é dividido em duas etapas (*Task 1* e *Task 2*) distintas, onde processos monitores executam as tarefas *Task 1* e *Task 2* e processos monitorados executam somente a *Task 2*. Para facilitar a especificação do algoritmo, utiliza-se como exemplo ilustrativo dois processos p_i e p_j onde p_j é monitorado pelo processo p_i.

Inicialmente, todo processo p_i executa algumas inicializações. Considerando que $L^l_{p_i}$ representa a lista local de processos suspeitos do processo p_i e que Ω representa o conjunto dos processos participantes do sistema distribuído (vide capítulo 2), o algoritmo assegura que nenhum processo é suspeito pelos demais processos do sistema, $\forall\ p_i : L^l_{p_i} = \emptyset$, e que todo processo inicia sua execução monitorando os demais processos, garantindo que $\forall\ p_i \in \Omega$ há um módulo de detecção vinculado. Após inicializado, cada processo cumpre suas tarefas como descrito a seguir (figura 4.2):

```
1: Todo processo p_i executa:
2:
3: seqNumber ← 0
4: ∀ p_j ∈ Ω : Δ_i^{p_j} ← default frequency
5:       Δ_to^{p_j} ← default timeout
6: L^g_{p_i} ← ∅ and L^l_{p_i} ← ∅                    {lista global e lista local respectivamente}
7: received ← true
8: cobegin
9: Task 1:
10:      loop
11:         if Δ_i^{p_j} ← 0 then
12:            send (AreYouAlive, L^g_{p_i}, segNumber) to p_j
13:            seqNumber ← seqNumber + 1
14:            restart Δ_to^{p_j} and Δ_i^{p_j}
15:            received ← false
16:         end if
17:         if Δ_to^{p_j} ← 0 then
18:            if not received then
19:               L^l_{p_i} ← L^l_{p_i} ⋃ {p_j}
20:               L^g_{p_i} ← L^g_{p_i} ⋃ {p_j}
21:            end if
22:         end if
23:      end loop
24: Task 2:
25:      forever
26:      upon
27:         received message m from a process p_j
28:         at ← arrivalTime
29:         case m = AreYouAlive or m = IAmAlive
30:            if m = AreYouAlive then send (IAmAlive) to p_j
31:               Δ_i^{p_j} ← at + defaultfrequency      {atualiza o próximo envio}
32:               if p_j ∈ L^l_{p_i} then
33:                  L^l_{p_i} ← L^l_{p_i} - {p_j}
34:                  Δ_to^{p_j} ← Δ_to^{p_j} + 1
35:               end if
36:               L^g_{p_i} ← L^g_{p_j} ⋃ L^l_{p_i} - {p_i}
37:               received ← true
38:            end if
39:            if m = IAmAlive then
40:               if p_j ∈ L^l_{p_i} then
41:                  L^l_{p_i} ← L^l_{p_i} - {p_j}
42:                  Δ_to^{p_j} ← Δ_to^{p_j} + 1
43:               end if
44:               received ← true
45:            end if
46:         end case
47:      end forever
48: coend
```

Figura 4.2: Algoritmo com a estratégia ATF

Na *Task 1* (linhas 9-23) o processo p_i dispara as mensagens de monitoramento quando a periodicidade (Δ_i) atingir o valor de 0 (linha 11) aos seus alvos, no caso ele enviará uma *liveness*

request (linha 12) para p_j. No método *send* são passados por parâmetro três valores: o tipo da mensagem, o número de seqüência da mensagem e sua lista global indicando os processos suspeitos. Se nenhuma mensagem for recebida num período Δ_{to}, p_i inicia a suspeitar de p_j adicionando-o a sua lista de processos suspeitos (linhas 19-20).

Na *Task 2* (linhas 24-47) as mensagens (*m*) são recebidas pelos processos (linha 27). Caso m seja do tipo `AreYouAlive` (linha 30) p_i responde ao processo com uma mensagem `IAmAlive`, e assume que p_j está operacional. Por ter recebido uma mensagem de monitoramento, reajusta o temporizador $\Delta_i^{p_j}$ (linha 31). Esta estratégia permite reduzir o número de mensagens enviadas, simplesmente por fazer a seguinte analogia: uma mensagem `AreYouAlive` é equivalente a uma mensagem `IAmAlive`. Assim p_i verifica em sua lista local se p_j estava como suspeito, se sim ele retira o processo de sua lista de suspeitos local (linha 33) e incrementa o valor de $\Delta_{to}^{p_j}$ (linha 34). O incremento do *timeout* significa que o algoritmo cometeu um engano por ainda não ter atingido St (modelo parcialmente síncrono). O processo p_i extrai da mensagem `AreYouAlive` recebida, a lista global de processos suspeitos realizando uma fusão com sua lista global (linha 44).

Caso m seja do tipo `IAmAlive` (linha 38), p_i verifica em sua lista local se p_j estava como suspeito; se sim, ele retira o processo de sua lista de suspeitos local (linha 41) e incrementa o valor de $\Delta_{to}^{p_j}$(linha 42). Por fim, sempre que p_i indicar a operacionalidade do processo p_j no atual instante, ele passa a variável *received* a receber o valor de *true* (linhas 37 e 44).

Este algoritmo assegura as propriedades necessárias para a implementação de um detector de defeitos da classe $\diamondsuit\mathcal{P}$. Estas propriedades serão provadas posteriormente neste capítulo.

4.1.2 Aproveitamento de Mensagens da Aplicação (AMA)

A estratégia AMA segue a mesma filosofia da estratégia ATF, no que diz respeito ao atraso das mensagens de controle; no entanto, a estratégia AMA reaproveita mensagens geradas pelas aplicações para suprir mensagens de controle. Neste caso, a semântica das mensagens de controle também é alterada, uma vez que qualquer mensagem da aplicação recebida de um processo monitorado indica a vivacidade/acessibilidade do processo emissor naquele instante.

A solução proposta assume um serviço de detecção de defeitos trabalhando entre a aplicação cliente e o *kernel* do sistema operacional (figura 4.3). Tendo em vista que um módulo de detecção monitora processos através de duas primitivas básicas: *send* e *receive*, e que em sistemas distribuídos as aplicações trocam mensagens freqüentemente, pode-se oferecer às aplicações clientes uma interface que as permite utilizar as primitivas oferecidas pela camada de detecção toda a vez que uma aplicação desejar trocar informações com outra aplicação.

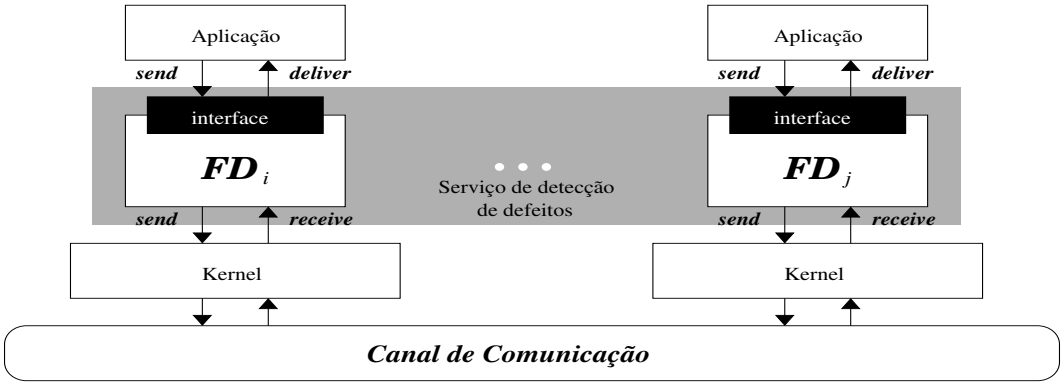


Figura 4.3: Arquitetura para troca de mensagens com a estratégia AMA

Ciente do fluxo de mensagens da aplicação, o detector pode utilizá-lo para suprir mensagens

do detector (estratégia AMA) atrasando a introdução de mensagens de controle sempre que possível, como na estratégia ATF. Como resultado o detector somente irá gerar mensagens de controle caso as aplicações não estiverem trocando mensagens.

A estratégia de aproveitar mensagens da aplicação para reduzir mensagens de controle é amplamente utilizada, mas a proposta neste trabalho possui algumas inovações como, por exemplo, combiná-la com a filosofia da estratégia ATF. Neste contexto, um algoritmo que implementa ATF+AMA assume que qualquer mensagem do tipo `AreYouAlive`, `IAmAlive` ou `ApplicationMessage` recebida de um processo monitorado indica a vivacidade do processo emissor naquele instante. Sempre que uma mensagem indicando a vivacidade chegar, o envio da próxima mensagem é postergado pela reinicialização do seu temporizador.

Particularmente observa-se que a estratégia de reaproveitamento de mensagens foi muito bem explorada no algoritmo de detecção denominado FD preguiçoso (*Lazy*) (seção 3.1.8). Nele, quando o FD associado a uma instância *q* da aplicação recebe uma mensagem proveniente de uma instância *p* ele deve retornar imediatamente uma mensagem de confirmação (*ack*), pois o protocolo depende de uma confirmação.

Na estratégia AMA o *ack* não é necessário, pois as mensagens da aplicação são utilizadas como mensagens `IAmAlive`, como pode ser visualizado na figura 4.4. Deste modo, as mensagens são contabilizadas somente no processo p_i que recebe a mensagem da aplicação (`ApplicationMessage`) localizada no processo p_j, tendo como resultado um serviço de monitoramento com menor custo. Ressalta-se que as mensagens de retorno para as mensagens `AreYouAlive` na figura 4.4 foram omitidas para fins de legibilidade.

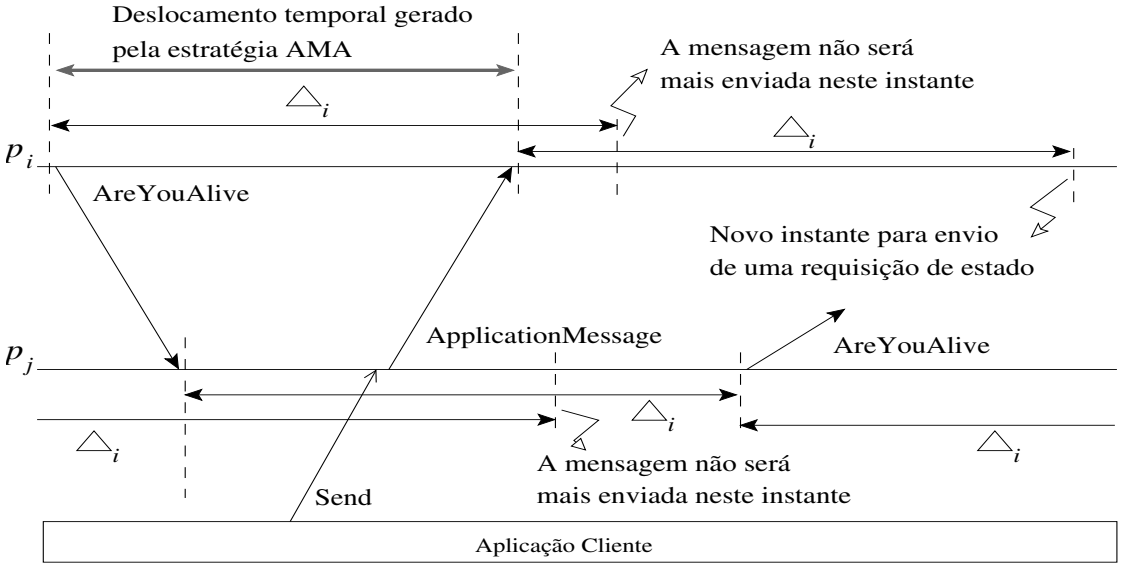


Figura 4.4: Reduzindo o número de mensagens com a combinação de ATF+AMA

Um aspecto importante da estratégia AMA é que as mensagens da aplicação são aproveitadas para detectar a operacionalidade dos processos e não defeitos como normalmente é utilizado (SERGENT; DÉFAGO; SCHIPER, 2001; BERTIER; MARIN; SENS, 2003), logo, não existem *timeouts* para as mensagens da aplicação. Além disto, nenhuma mensagem a mais é necessária para cada mensagem da aplicação enviada (economia de mensagens). O segundo aspecto é que esta estratégia pode ser estendida a diversos algoritmos de detecção. Neste trabalho a estratégia AMA foi aplicada a estilos de monitoramento ativo e passivo como será mostrado nos experimentos realizados no capítulo 6.

4.1.2.1 Algoritmo Para a Estratégia AMA

O algoritmo para a estratégia AMA segue a mesma especificação feita na seção anterior, ou seja, duas etapas (*Task 1* e *Task 2*) são executadas pelos processos, onde processos monitores executam as tarefas *Task 1* e *Task 2*, ao passo que processos monitorados executam somente a *Task 2*. Para fins ilustrativos dois processos são utilizados como exemplo onde: p_j é monitorado pelo processo p_i. Além disso, todo processo p_i executa algumas inicializações. Considerando que $L^l_{p_i}$ representa a lista local de processos suspeitos do processo p_i e que Ω representa o conjunto dos processos participantes do sistema distribuído, o algoritmo assegura que nenhum processo é suspeito pelos demais processos do sistema, $\forall\ p_i : L^l_{p_i} = \emptyset$, e que todo processo inicia sua execução monitorando os demais processos, garantindo que $\forall\ p_i \in \Omega$ há um módulo de detecção vinculado. Vale salientar que uma aplicação cliente é qualquer *software* localizado na camada superior que utiliza serviços da camada de detecção de defeitos.

Após a inicialização do algoritmo da estratégia AMA, cada processo cumpre suas tarefas como descrito a seguir (figura 4.5):

```
.
.                                                          {idem ao algoritmo da figura 4.2}
.
24: Task 2:
25:     forever
26:     upon
27:         received message m from a process p_j
28:         at ← arrivalTime
.
.                                                          {idem ao algoritmo da figura 4.2}
.
46:           if m = ApplicationMessage then
47:             Δ_i^{p_j} ← at + defaultfrequency          {atualiza o próximo envio}
48:             if p_j ∈ L^l_{p_i} then
49:               L^l_{p_i} ← L^l_{p_i} - {p_j}
50:               Δ_to^{p_j} ← Δ_to^{p_j} + 1
51:             end if
52:             received ← true
53:             deliver (m)
54:           end if
55:         end case
56:     end forever
57: coend
```

Figura 4.5: Algoritmo com a estratégia AMA

Na *Task 2* os processos trabalham da mesma forma que foi especificado na *Task 2* da estratégia ATF, exceto pelo fato de que esta estratégia também assume mensagens da aplicação. As linhas 46 à 54 demonstram como é tratada uma mensagem recebida de uma aplicação localizada no processo p_j.

Tão logo o processo p_i recebe uma mensagem do tipo `ApplicationMessage`, ele assume que p_j está operacional e, por ter recebido uma mensagem da aplicação, ele reajusta o temporizador $\Delta_i^{p_j}$ (linha 47) atrasando a próxima mensagem. Logo depois, p_i verifica o estado do processo emissor na lista local de processos suspeitos (linha: 48), caso o processo emissor constar na lista de processos suspeitos, ele será retirado, pois uma falsa suspeita haverá sido levantada. Devido a isto o detector de defeitos incrementa o valor de $\Delta_{to}^{p_j}$ (linha 50) em busca de atingir St para não ocorrer mais falhas de temporização. Por se tratar de uma mensagem da

aplicação cliente o detector de defeitos realiza um *deliver* (linha 53) para repassar e entregar a mensagem destinada a camada superior de aplicação.

As mensagens de retorno destinadas a aplicação são entregues pelo método *deliver* na forma de *callback*.

4.1.3 Algoritmo de Detecção *Eventually Perfect*

Nesta seção será mostrado que os algoritmos propostos na seção anterior implementam um detector de defeitos da classe $\diamondsuit\mathcal{P}$, nas condições em que o sistema esteja de acordo com as especificações do modelo de sistema adotado (vide seção 2.1).

Eventual Strong Accuracy. Para contemplar esta propriedade, após um tempo finito, processos corretos não devem mais ser considerados suspeitos por nenhum outro processo correto.

Lema 1. Depois de *St* (vide seção 2.1.1), qualquer processo correto p_i suspeitará de p_j no instante maior do que Δ_{lim}.

Lema 2. Qualquer processo correto p_i suspeitará erroneamente de um processo correto p_j um número finito de vezes.

Prova. Assumindo p_i e p_j corretos, após St p_i conhecerá Δ_{lim} para a comunicação com p_j ajustando o *timeout* para um valor ideal, onde p_j não mais será erroneamente considerado suspeito.

Teorema 1. *Eventual Strong Accuracy*. Existe um tempo depois no qual os processos corretos não serão mais considerados suspeitos por nenhum outro processo correto.

$$\exists t_0 : \forall t' > t_0, \forall p_i \in correct, \forall p_j \in correct, p_j \notin L_{p_i}(t')$$

Prova. Deixa t_0 ser o último instante em que p_i supeita de p_j, este instante existe no Lema 1. Então depois do instante t' nenhum processo correto p_i terá o processo correto p_j na lista L_{p_i}.

Colorário 1. O algoritmo da figura 4.2 fornece a propriedade de *Eventual Strong Accuracy* realizando $\Delta_{to} \leftarrow \Delta_{to}^{p_j} + 1$ a cada falsa suspeita.

Strong Completeness. Esta propriedade requer que, após um tempo finito, todos os processos que falharam sejam permanentemente suspeitos por **todos** processos corretos. Para isso, cada processo p_i possui uma lista local de processos suspeitos assegurando somente os estados dos processos que estão sendo monitorados por ele (visão local). Assim para assegurar um estado comum entre todos os processos do sistema e garantir a propriedade de *Strong Completeness* uma lista global é implementada por cada processo e enviada por *piggyback* junto com as mensagens `AreYouAlive` inerentes às ações de monitoramento (visão global).

Lema 3. $\forall p_i \in \Omega, \forall t, L_{p_i}^{l}(t) \subseteq L_{p_i}^{g}(t)$

Lema 4. $\exists t'' : \forall p_j \in crashed, \forall p_i \in correct, \forall t \geq t'', p_j \in L_{p_i}^{g}(t)$

Prova. Estando no instante t' definido no Teorema 1, qualquer processo $p_j \in crashed$ já falhou e tem seu estado permanentemente incluído em $L_{p_i}^{l}$. Considere que $p_j \in crashed$ e que $p_i \in correct$, por dedução p_j estará incluso permanentemente em $L_{p_i}^{g}$ num tempo finito. Neste caso se $p_j \in L_{p_i}^{l}$, do Lema 3 p_j estará permanentemente em $L_{p_i}^{g}$.

Teorema 2. Existe um tempo depois t em que p_i envia uma mensagem `AreYouAlive` e todos os outros processos contêm p_j em L^g.

Lema 5. $\forall p_i \in correct, \forall t, p_i$ em algum instante futuro entregará uma mensagem `AreYouAlive` depois de t.

Prova. A garantia de que p_i em algum instante futuro irá entregar uma mensagem `AreYouAlive` provém do canal confiável assumido no capítulo 2, da garantia que $\forall$ $p_i \in \Omega$ há um módulo de detecção vinculado, e que $n > 1$, onde n é o número de processos corretos.

Desta forma, depois da entrega da primeira mensagem `AreYouAlive` no instante maior que t, p_j será permanentemente incluído em qualquer processo $p_y \in \{p_i, ..., p_n\}$ na lista global $L^g_{p_y}$.

Colorário 2. O algoritmo da figura 4.2 provê *Strong Completeness* através da manutenção de uma lista global de processos suspeitos.

Colorário 3. Do Colorário 1 e Colorário 2 o algoritmo implementa um detector de defeitos da classe $\diamondsuit\mathcal{P}$.

A prova de que um detector da classe $\diamondsuit\mathcal{P}$ contempla as propriedades de um algoritmo de consenso pode ser encontrada em (CHANDRA; TOUEG, 1996).

4.2 Conclusões Parciais

No presente capítulo, foi proposta uma abordagem para reduzir o número de mensagens geradas por algoritmos de detecção de defeitos, através do reaproveitamento de mensagens. A abordagem está dividida em duas estratégias, ATF e AMA, as quais formam uma abordagem genérica podendo ser estendida a diversos algoritmos de detecção. A abordagem utiliza a estratégia de reaproveitamento de mensagens, mas nada impede a combinação das demais estratégias listadas na seção 3.2.

Dentre os trabalhos relacionados com a explosão de mensagens, o que mais se aproxima ao presente trabalho é o proposto por Fetzer, Raynal e Tronel (2001). Por este motivo algumas comparações foram realizadas com o objetivo de demonstrar a eficiência da abordagem proposta. Comparações mais precisas serão demonstradas no capítulo 6.

Também foram apresentados os algoritmos que implementam as estratégias ATF e AMA, bem como provas que garantem que tais algoritmos implementam um detector de defeitos da classe $\diamondsuit\mathcal{P}$. Os algoritmos garantem a classe $\diamondsuit\mathcal{P}$ por assegurarem, da seguinte forma, as propriedades:

Eventual Strong Accuracy: esta propriedade foi cumprida por aumentar o *timeout* a cada suspeita errada; e

Strong Completeness: foi garantida através da implementação de uma lista global de processos suspeitos.

Em síntese, um detector que satisfaz tais propriedades garante que todos os processos que falharam serão permanentemente suspeitos por todos processos corretos, e ainda, após atingir *St* os processos corretos não são mais considerados suspeitos por nenhum processo correto.

5 UM SERVIÇO DE DETECÇÃO ADAPTATIVO

No Capítulo 4 foram descritas as estratégias propostas para a implementação de um detector de defeitos otimizado em termos de número de mensagens. Neste capítulo apresenta-se um serviço para monitoramento de processos em sistemas distribuídos, o $AFDService$ (NUNES, 2003), que pode ser configurado, tanto de acordo com as necessidades da aplicação quanto ao comportamento da rede, e descreve-se como as estratégias propostas foram incorporadas nele. Também será apresentado como um algoritmo de consenso interage com o serviço em questão. A seguir será apresentada uma breve descrição sobre detectores de defeitos adaptativos.

5.1 Detectores de Defeitos Adaptativos

Nos algoritmos de detecção de defeitos é comum verificar que os serviços sofrem distorções quando são utilizados valores de *timeouts* fixos. Isto se deve ao fato de que há uma tendência maior de serem considerados erroneamente suspeitos os processos mais lentos ou mais distantes, pois estes têm maior possibilidade de ultrapassar os valores limites. No entanto, aumentar ou diminuir o *timeout* implica diretamente na qualidade de serviço do detector. Por exemplo, simplesmente aumentar o *timeout* em busca de abranger todos os processos monitorados irá também aumentar o tempo de detecção do serviço.

Assim definir um valor fixo para o *timeout* é uma decisão delicada, pois qualquer alteração no ambiente poderá influenciar na qualidade de serviço do detector. Para reduzir a complexidade de tentar encontrar um valor adequado que abranja todos os processos surge a idéia de detectores de defeitos adaptativos. Estes serviços, através de estimativas baseadas em modelos matemáticos, tentam corrigir o valor do *timeout* de acordo com o comportamento do ambiente ou necessidade da aplicação. Neste sentido, diversos trabalhos foram propostos (CHEN; TOUEG; AGUILERA, 2000) (BERTIER; MARIN; SENS, 2003) (SANTOS SÁ; ARAÚJO MACÊDO, 2005) entre eles, pode-se destacar os testes realizados por Nunes (NUNES; JANSCH-PÔRTO, 2004) onde foram utilizados modelos preditivos baseados em séries temporais para alcançar uma boa qualidade de serviço.

5.2 O AFDService

Nesta seção, será detalhado o serviço no qual foram implementadas e validadas as estratégias descritas no capítulo 4, o $AFDService$ (NUNES, 2003), um serviço de detecção de defeitos adaptativo.

A figura 5.1 apresenta o modelo do serviço organizado semanticamente em pacotes e representado na linguagem UML (*Unified Modeling Language*). Os algoritmos de detecção de defeitos são agrupados no pacote FD, enquanto que os algoritmos de predição dos *timeouts*

dinâmicos são apresentados no pacote TS. O pacote CS representa os algoritmos de consenso e foi projetado neste trabalho para avaliar as estratégias propostas. Este pacote será mais bem detalhado na seção 5.4.

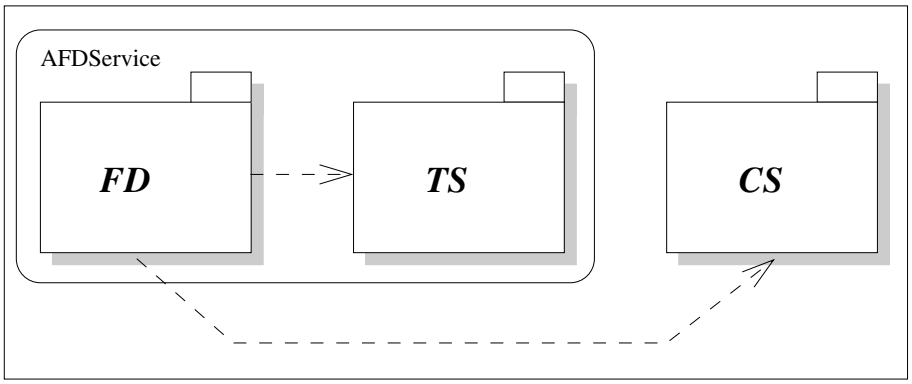


Figura 5.1: Modelo em pacotes UML

A arquitetura do $AFDService$ foi projetada para prover flexibilidade, ou seja, tanto o módulo de detecção (onde estão os detectores) quanto o de previsão (onde estão os preditores) mantêm um repositório de algoritmos que podem ser utilizados conforme a necessidade de cada aplicação. Ressalta-se que a seleção de preditores não interfere na seleção das estratégias para redução de mensagens, podendo ser utilizada qualquer combinação.

A figura 5.2 ilustra como a arquitetura do $AFDService$ está organizada. Basicamente o serviço é dividido em dois módulos, um de detecção, onde estão todos os algoritmos de detecção (atualmente implementados os algoritmos *Pull, Push, Dual e Gossip*) e um módulo de previsão, onde estão os algoritmos de previsão que são responsáveis pelos ajustes dos *timeouts* (atualmente sete preditores distintos).

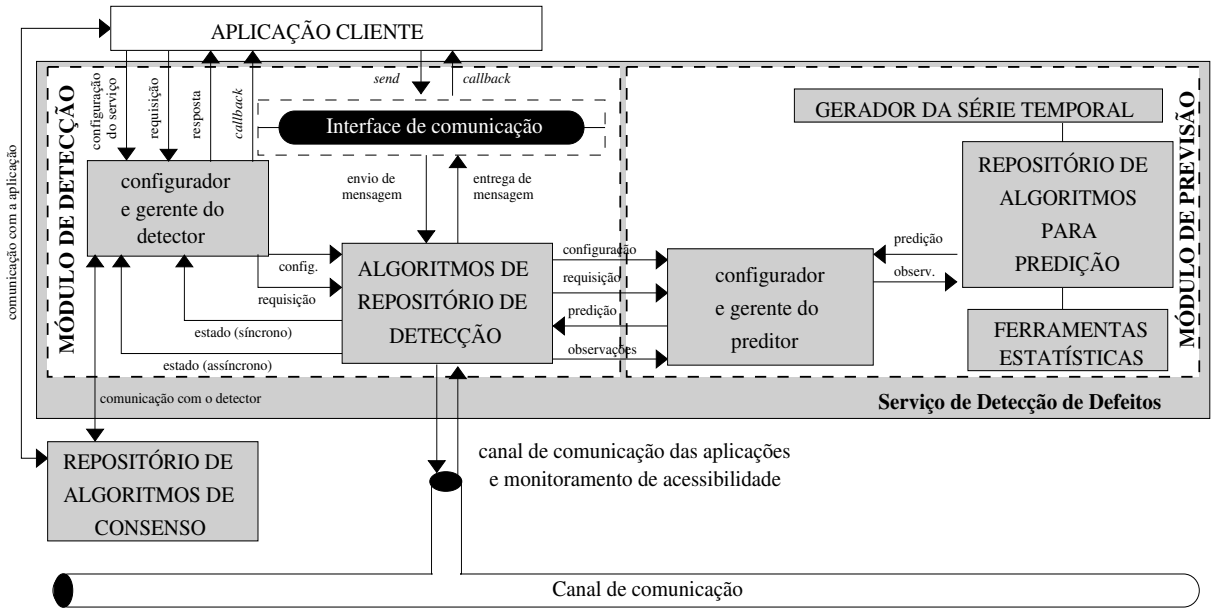


Figura 5.2: Arquitetura do *AFDService*

A aplicação cliente se comunica com o módulo de detecção para ajustar, por exemplo, o tipo de algoritmo de detecção, o tipo de algoritmo de predição, o suporte a algoritmo de consenso, a

estratégia para redução de mensagens, entre outros. As informações de estados podem ser prestadas por *callback*, a cada troca de estado, ou através de uma requisição efetuada pela aplicação cliente, quando desejado.

Para possibilitar a inserção das estratégias de redução de mensagens optou-se por disponibilizar à aplicação uma interface com primitivas que possibilite a ela trocar informações através do $AFDService$ (vide seção 5.3).

O repositório de algoritmos de consenso atualmente implementa o protocolo proposto por Chandra e Toueg (1996), este algoritmo ainda utiliza um serviço de difusão confiável para disseminar o valor decidido ao término de uma rodada. A integração destes algoritmos juntamente com o $AFDSevice$ serão apresentados posteriormente neste capítulo.

5.2.1 Serviço Baseado em Interfaces

O objetivo de disponibilizar um serviço disposto em interfaces é que a aplicação cliente somente necessita utilizar um conjunto de métodos, isolando detalhes de implementação aos componentes que a utilizam. Neste sentido, diversos serviços de detecção de defeitos (FELBER et al., 1999) (BERTIER; MARIN; SENS, 2003) (FRIEDMAN; RAYNAL, 2004) foram propostos empregando esta mesma abordagem. Assim, as funcionalidades dos serviços são observadas omitindo detalhes de implementação.

Antes de apresentar a arquitetura disposta em interfaces oferecida pelo serviço de detecção de defeitos proposto neste trabalho, alguns conceitos básicos definidos por Felber (FELBER et al., 1999) devem ser explorados. Felber propôs um serviço genérico composto por interfaces com o objetivo de permitir uma ampla aplicabilidade; neste serviço três componentes básicos foram definidos para a construção do serviço de detecção de defeitos:

- *Monitor (Detector de Defeitos):* são os componentes que colecionam informações sobre os estados dos processos.
- *Monitorável:* são processos que têm seus estados monitorados pelos processos monitores.
- *Notificável:* são normalmente aplicações clientes que desejam ser síncrona ou assíncronamente notificados sobre alterações de estados dos processos monitoráveis.

Felber mostrou que, trabalhando com os três componentes citados anteriormente orientados a aplicação, é possível obter um serviço adaptado para diversas topologias de redes (no capítulo 3 foram apresentados algoritmos de detecção em diversas topologias). E ainda, a aplicação pode ser responsável por indicar quais processos são monitores, monitorados e quais são notificados de acordo com seu domínio de visão. As interfaces que compõem o $AFDService$ baseam-se no serviço de monitoramento proposto por Felber et al. (1999)

O conjunto de interfaces do $AFDService$ pode ser visualizada na figura 5.3. Observa-se que as interfaces *Notifiable* e *Monitorable* devem ser implementadas pela aplicação cliente, ao passo que a interface *Monitor* é implementada pelos algoritmos de detecção. Com isso, a interface *Monitor* pode ser especializada em diversos algoritmos de detecção, resultando na reutilização da interface. O amplo conjunto de métodos disponíveis nesta interface permite tanto a consulta de estados dos processos monitoráveis quanto a configuração de parâmetros funcionais que são herdados da interface *FailureDetector*, por exemplo, tipo de algoritmo de detecção e margem de segurança.

A interface *FailureDetector* permite que cada algoritmo de detecção de defeitos ofereça à aplicação cliente a possibilidade de ajustar diversos parâmetros como o *timeout* ('setTimeout()') e o intervalo de envio de mensagens ('setCrlMsgInterval()'), bem como adicionar ou remover

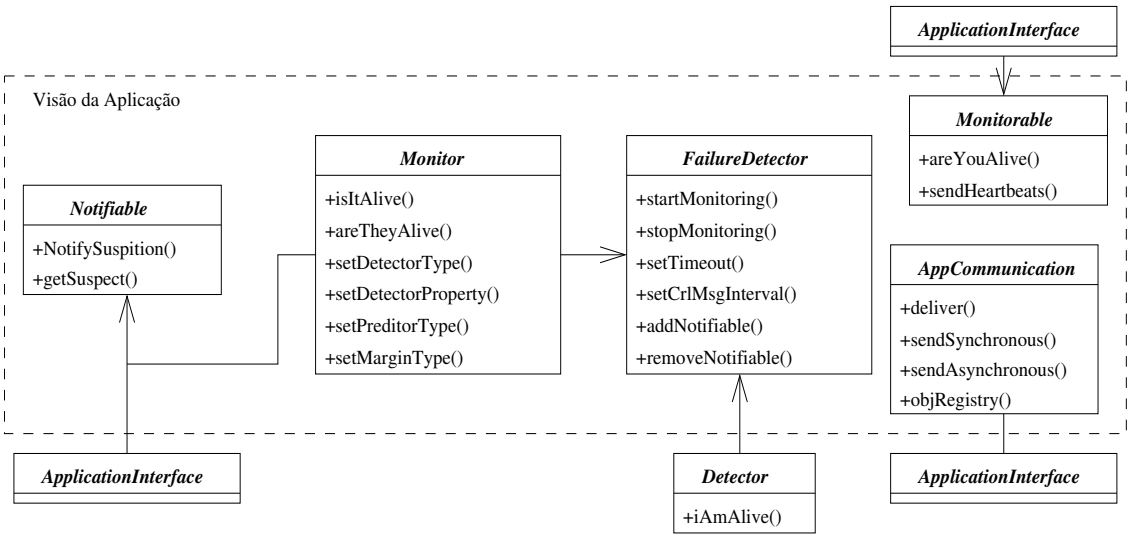


Figura 5.3: Interfaces do *AFDService*

processos notificáveis ('addNotifiable()' e 'removeNotifiable()') e iniciar ou cessar o monitoramento de processos ('startMonitoring()' e 'stopMonitoring()').

A interface *Monitorable* permite trabalhar tanto com um processo monitorável *Pull* ou *Push* facilitando a troca do tipo de detector. Salienta-se que o $AFDService$ possui outros algoritmos de detecção, entretanto como o escopo deste trabalho refere-se aos dois algoritmos citados anteriormente, restringir-se-á a eles.

A interface *Detector* é orientada aos detectores, ou seja, aos algoritmos de detecção suportados pelo serviço. Neste caso, para os estilos de detecção que envolvem estados, o método 'iAmAlive()' é responsável por indicar tal informação aos monitores. A interface *AppCommunication* será detalhada na seção 5.3, a qual descreve como as estratégias foram integradas ao seviço.

5.2.2 AFDService e Padrões de Projetos

O $AFDService$ é um serviço configurável, no sentido de que se pode alterar parâmetros como por exemplo a margem de segurança[1] do detector, e também flexível, pois permite a troca, tanto estática quanto dinâmica, dos algoritmos de detecção e/ou de predição, ou mesmo a inclusão de novos algoritmos.

Para garantir esta flexibilidade o $AFDService$ implementa em seus pacotes o padrão de projetos [2] *Strategy* (GAMMA; HELM; JOHNSON, 1995) onde pode-se definir uma família de algoritmos encapsulando-os de forma a tornar a escolha flexível, permitindo a implantação de diversos algoritmos/estratégias para o mesmo problema. Em outras palavras, este padrão organiza a implementação com uma interface comum que pode ser utilizada por todos os algoritmos.

Outro padrão de projeto utilizado pelo $AFDService$ é o *Singleton*, onde a idéia básica consiste em criar somente uma instância de uma dada classe. Este padrão pode ser utilizado para diferentes propósitos proporcionando diferentes funcionalidades as aplicações. Para o propósito deste trabalho, o *Singleton* foi adotado para evitar conflitos no momento da execução do serviço; ou seja, nos instantes em que duas instâncias do serviço tentarem ser executadas na mesma máquina, somente uma será criada, evitando o conflito que ocorre quando dois programas tentam

[1] A margem de segurança permite variações limitadas no atraso de comunicação e é considerada no cômputo do *timeout*

[2] São descrições formais para soluções simples e reconhecidamente eficientes em projetos orientado a objetos

utilizar a mesma porta de comunicação.

Originalmente este padrão foi implementado na classe *FDManager* a qual interage diretamente com a aplicação cliente, entretanto um problema foi observado pois nestas condições somente uma aplicação cliente poderia interagir com o serviço. Desta forma, para que o serviço fosse disponibilizado para diversas aplicações cliente o padrão foi movido para a classe *ReceiveServer*, responsável pela criação dos *sockets* de comunicação. No aspecto de implementação, para concretizar este padrão, o objeto que instancia a própria classe (ver figura 5.5) deve ser declarado como estático. Assim, para as chamadas subseqüentes, o construtor verifica o estado do objeto (igual ou diferente de *null*); se ele for diferente de *null*, significa que este já foi instanciado e, por ser estático, seu estado não mais será alterado tendo como retorno a instância já criada.

5.3 Integração das Estratégias AMA e ATF ao $AFDService$

No capítulo anterior foi apresentada a abordagem para a redução de mensagens, composta de duas estratégias (AMA e ATF). Já, neste capítulo, foi apresentada uma arquitetura disposta em interfaces (vide figura 5.3), dentre elas está a interface *AppCommunication*, a qual tem por objetivo permitir que as aplicações clientes se comuniquem com outras aplicações do sistema integrando a estratégia AMA ao $AFDService$.

A figura 5.4 apresenta a interface *AppCommunication* bem como os métodos que são disponibilizados às aplicações clientes. Tais métodos possuem as seguintes funcionalidades: 'sendSynchronous()' e 'sendAsynchronous()' são métodos utilizados pelas aplicações para envio de mensagens síncronas e assíncronas respectivamente; o método 'deliver()' é responsável pela entrega de mensagens por *callback* às aplicações, para isto, elas devem executar o método 'registerObj()' responsável por registrar os objetos de cada aplicação cliente na camada de detecção. Este registro faz-se necessário para que o serviço possa identificar para quem se destinam as mensagens recebidas pelo serviço. Em outras palavras, permite repassar a mensagem para a aplicação correta.

Com esta abordagem a camada de detecção de defeitos consegue observar todas as mensagens trocadas pela aplicação cliente com outras aplicações distribuídas. Salienta-se que existem outras formas para observar as mensagens trocadas pela aplicação, entretanto, como este não é o enfoque deste trabalho, optou-se por disponibilizar uma interface como forma de interação.

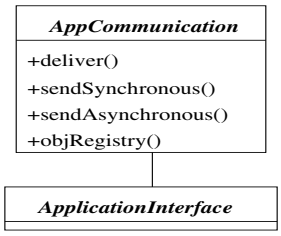


Figura 5.4: Interface que permite a integração da estratégia AMA ao $AFDService$

Para a integração da estratégia ATF ao $AFDService$, a única mudança necessária foi alterar as semânticas das mensagens de controle junto aos respectivos algoritmos de detecção de defeitos. Tais mudanças foram descritas no capítulo 4.

O diagrama de classes da figura 5.5 representa o pacote de software FD apresentado na figura 5.1. Ressalta-se que os métodos das interfaces foram omitidos, pois eles já foram descritos na figura 5.3.

A classe *AppEvent* implementa a interface *AppCommunication* e é responsável pelos eventos de comunicação entre as aplicações clientes. Para que uma aplicação envie uma mensagem para outra aplicação, esta mensagem será encapsulada num objeto serializável do tipo *Message* e será enviada pela aplicação através dos métodos 'sendSynchronous()' ou 'sendAsynchronous()'. A classe *Message*, por sua vez, possui alguns atributos para a formação do pacote a ser disseminado como: o tipo da mensagem; o endereço do processo destinatário e do processo emissor; o conteúdo da mensagem, neste caso este atributo é utilizado pelas aplicações clientes empacotarem dados para serem enviados; um contador para controle de mensagens perdidas para as mensagens de monitoramento.

Como as mensagens da aplicação são contabilizadas no processo receptor, a classe *DeliverBoy* ao receber uma mensagem deste tipo automaticamente cancela a próxima mensagem a ser enviada para o respectivo processo reiniciando o temporizador; esta tarefa é executada pela classe *RequestControler*. Ressalta-se que, para um maior esclarecimento de como o serviço $AFDService$ trabalha, serão definidas a seguir as demais classes apresentadas na figura 5.5.

A classe onde se concentram as informações mais importantes é a classe *FDManager*. Esta classe é responsável por receber as configurações exigidas pelas aplicações. Por exemplo, caso a aplicação cliente decida utilizar um determinado algoritmo de detecção, a classe *FDManager* ativa o algorimo escolhido delegando a funcionalidade a uma classe que implemente a interface *Detector*. A classe *FDManager*, por sua vez, implementa a interface *Monitor* para permitir a aplicação cliente consultar os estados dos processos monitorados.

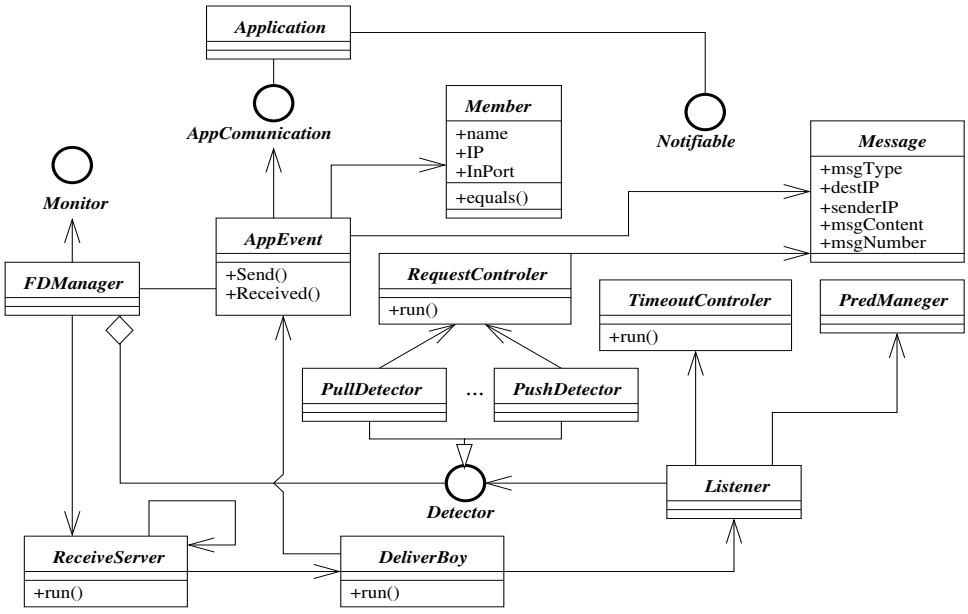


Figura 5.5: Diagrama de Classes do Pacote FD

Os processos monitorados que constituem o sistema distribuído são representados por objetos do tipo *Member*. Esta classe possui três atributos: nome do processo; um endereço IP; e a respectiva porta de entrada para a recepção de mensagens deste processo. Os objetos da classe *Member* devem ser instanciados a partir da aplicação cliente e repassados à classe *FDManager* para que então possam ser inicializadas as ações de monitoramento 'startMonitoring()'.

Para a operação de recebimento de mensagens, tanto para ações de monitoramento quanto para a comunicação entre aplicações, é gerada uma *thread* receptora pela classe *ReceiveServer*. Assim que as mensagens são recebidas, elas são repassadas à classe *DeliverBoy*. A classe *DeliverBoy* tem a responsabilidade de identificar, no campo *msgType*, os tipos de mensagens

recebidas. Deste modo esta classe serve como roteadora de mensagens para as *threads* de escuta do detector de defeitos.

Para cada processo monitorado será instanciado um objeto *Listener* para o controle de *timeouts*. Cada *timeout* é escalonado pela classe *TimeoutControler* e, a cada mensagem de estado recebida, a tarefa escalonada é cancelada. Caso uma mensagem não chegue antes de expirar o *timeout*, o *TimeoutControler* adiciona o processo na lista de suspeitos e chama o método 'addSuspect()' na classe *FDManager*, a qual chama o método `NotifySuspcion` declarado na interface *Notifiable*. Assim, para que a aplicação cliente receba a informação de estado pelo serviço de detecção, esta deverá implementar a interface *Notifiable*. A aplicação será notificada por *callback* através do método 'NotifySuspcion()' ou assincronamente através do método 'getSuspect()', que retorna uma lista de membros suspeitos. O ajuste dos *timeouts* é realizado através da classe *PredManager* realizando a previsão referente aos tempos de rtt enviados pela classe *Listener*.

A interface *Detector* instancia o estilo de detecção desejado pela aplicação cliente. Por exemplo, se for utilizado o detector *Pull*, a classe *RequestControler* será executada periodicamente conforme configurado o Δ_i. Esta periodicidade pode ser personalizada para cada processo monitorado, ou seja, um processo que está mais distante poderá ser consultado em períodos maiores do que processos que se encontram mais próximos do processo monitor. A classe *RequestControler* também pode ser utilizada para o processo monitorado enviar mensagens de vida ao monitor. Neste caso isto ocorre se a aplicação optar pelo estilo de detecção *Push*.

5.4 Algoritmo de Consenso e Aspectos de Implementação

Para avaliar as estratégias propostas neste trabalho, sobre o ponto de vista do algoritmo de consenso, foi implementado um pacote junto ao $AFDService$, como mostrado na figura 5.2. O algoritmo de consenso definido para fazer parte do serviço foi o de Chandra e Toueg (1996), devido ao fato deste algoritmo ser amplamente difundido. O algoritmo trabalha em quatro fases que constituem uma rodada. Informalmente estas fases são descritas da seguinte forma:

- *Fase 1 ($F1$):* os processos estimam seus valores enviando-os ao coordenador do consenso.
- *Fase 2 ($F2$):* o coordenador aguarda pela proposição da maioria e escolhe um dos valores para ser proposto aos processos.
- *Fase 3 ($F3$):* os processos esperam pela proposta do coordenador. Quando recebem adotam o valor enviado pelo coordenador retornando uma confirmação (*ack message*) ao mesmo. Entretanto, se um processo suspeitar do coordenador antes de receber o valor proposto, ele emitirá uma mensagem negativa (*nack message*) e irá para a próxima rodada.
- *Fase 4 ($F4$):* o coordenador aguarda até receber as mensagens de confirmação (*ack ou nack*) da maioria dos processos. Se a proposição da maioria dos processos é *ack*, o valor proposto torna-se a decisão. Neste caso, o coordenador envia o valor decidido por difusão confiável a todos os processos. Se, por outro lado, o coordenador receber pelo menos um *nack* de algum processo, ele não envia a decisão e procede para a próxima rodada.

Uma execução, livre de falhas, do algoritmo de consenso de Chandra e Toueg exemplificando o seu padrão de envio de mensagens e as quatro fases seqüenciais pode ser visualizada na figura 5.6. Como pode-se ver, em casos onde nenhuma suspeita é gerada, o consenso pode ser concluído numa única rodada.

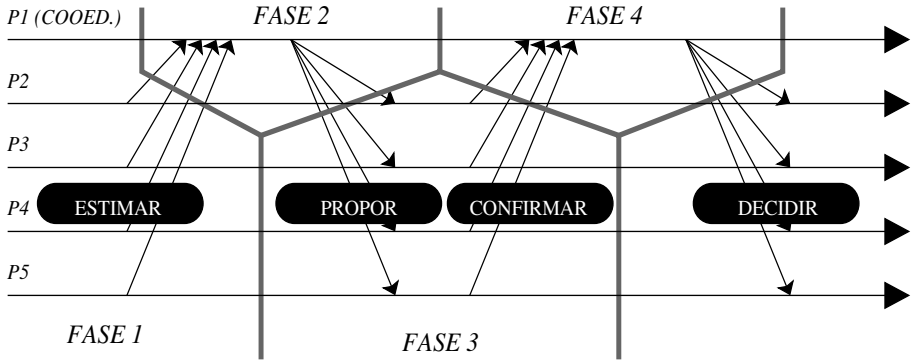


Figura 5.6: Troca de mensagens no algoritmo de consenso

A interação do algoritmo de consenso com o serviço $AFDService$ pode ser visualizada na figura 5.7. Esta combinação foi denominada como arquitetura confiável, onde a aplicação pode interagir com tal arquitetura de forma a fornecer a aplicaçê-la maior confiabilidade para as operações que necessitarem de referidos serviços. A interação do consenso com o FD é realizada de forma similar a qualquer outro cliente do sistema. As informações trocadas referem-se ao estado dos processos e podem ser indicadas por *callback* a cada alteração de estado observada pelo módulo de detecção, ou por requisição do estado de algum processo específico. Tais informações também são úteis ao algoritmo de difusão confiável (*Reliable Broadcast*) (CRISTIAN; BEIJER; MISHRA, 1994). Como resultado o protocolo de difusão confiável ao difundir uma mensagem envia somente para os processos considerados ativos pelo detector de defeitos.

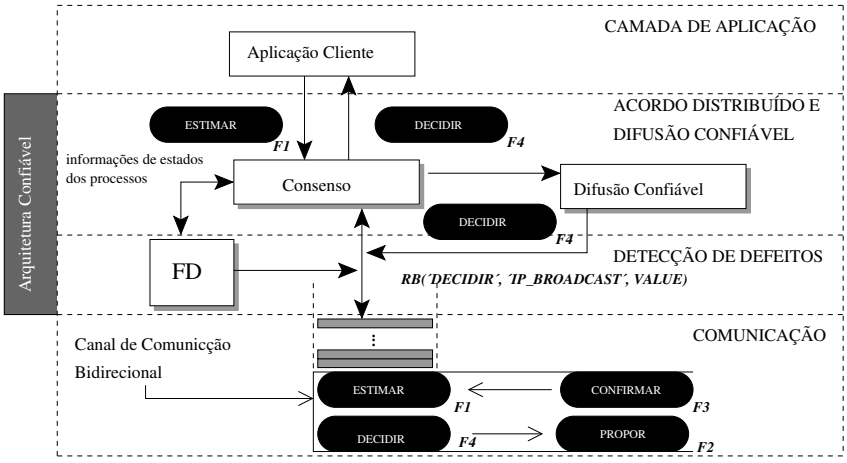


Figura 5.7: Arquitetura de serviços em camadas

A figura 5.7 ilustra uma execução do consenso em que cada rodada é inicializada no instante em que é enviada uma mensagem do tipo `ESTIMAR` ao coordenador corrente ($F1$). Esta mensagem ativa no coordenador uma nova rodada, de modo que este não necessite realizar nenhuma previsão, a saber, quem é o coordenador, uma vez que, a mensagem deste tipo já o indica como o coordenador da rodada que se inicia. O coordenador escolhe um dos valores para ser proposto a todos os integrantes; para isto ele aguarda a proposição da maioria, lembrando que os instantes de espera foram definidos por Sergent como os momentos críticos, onde o detector de defeitos é fundamental para a terminação do consenso. Neste caso, para que a espera não passe a ser infinita, o detector de defeitos deve estar ativo para reportar qualquer alteração de estado dos processos observado no sistema. Depois da emissão do valor a ser proposto ($F2$),

o coordenador novamente aguarda pela confirmação dos participantes ($F3$), para então enviar a decisão por difusão confiável. O algoritmo de difusão confiável irá garantir que todos os processos, não falhos do sistema, receberão a mensagem `DECIDIR` ($F4$). Informalmente difusão confiável pode ser definida por duas primitivas *R-broadcast(m)* e *R-deliver(m)* (URBÁN, 2003), onde *m* representa uma mensagem qualquer. Dessa forma, um protocolo de difusão confiável pode ser definido pelas seguintes propriedades (DÉFAGO; SCHIPER; URBÁN, 2004):

Validade Uniforme: Se um processo correto realizar *R-broadcast m*, em algum instante futuro ocorrerá *R-deliver m*.

Integridade Uniforme: todo processo irá realizar *R-deliver m* no máximo uma vez, se somente se *m* foi *R-broadcast* por algum processo anteriormente.

Acordo Uniforme: Se um processo *R-deliver m* então todo processo correto, em algum instante futuro, realizará um *R-deliver m*.

Observe que as propriedades de Validade e Integridade são hipóteses que devem assegurar qualquer serviço de broadcast, ao passo que a propriedade de Acordo garante a confiabilidade do protocolo.

No contexto de implementação, inicialmente a difusão confiável foi implementada sobre uma rede local onde a utilização de endereço de *broadcast* era suficiente para disseminar uma mensagem a todos os participantes que compunham o consenso. Entretanto ao transladar a solução proposta para a Internet, esta funcionalidade ainda não é possível de ser utilizada fazendo com que alguns aspectos da implementação sejam alterados. Para contornar este impasse o *broadcast* é realizado de uma forma 'simulada', ou seja, ao enviar uma mensagem por difusão, esta mensagem é criada e são realizados n envios consecutivos onde n é o número de processos participantes do consenso. Ciente de que esta alternativa utilizada não pode ser igualada a realização de um único envio por *broadcast*, ela foi utilizada pelo fato de que não encontrou-se outra possibilidade para realizar tal tarefa.

5.5 Conclusões Parciais

Neste capítulo inicialmente foi argumentada a necessidade de um serviço de detecção de defeitos adaptativo e na seqüência foi apresentado um serviço denominado $AFDService$; sua arquitetura, diagrama de classes e interfaces foram detalhados com o objetivo de facilitar o entendimento da integração da abordagem proposta junto ao serviço. Logo, foi apresentado como as estratégias ATF e AMA foram incorporadas ao $AFDService$. Neste contexto pode-se observar a facilidade da integração da estratégia ATF, necessitando apenas alterar a semântica das mensagens de controle. Já para a integração da estratégia AMA, além da alteração das semânticas das mensagens de controle, uma interface foi disponibilizada para a comunicação da aplicação cliente e conseqüentemente possibilitar o reaproveitamento das mensagens geradas pela mesma.

Uma arquitetura em camadas foi descrita com o objetivo de demonstrar a integração do serviço junto a outros algoritmos como, por exemplo, o algoritmo de consenso, utilizados como experimentos na seção 6.4. Por fim, foi apresentada uma descrição sucinta da utilidade do algoritmo de difusão confiável (útil ao protocolo de consenso em questão) e suas respectivas propriedades.

Para reportar a funcionalidade do serviço detalhado neste capítulo, serão apresentados no capítulo que segue os experimentos utilizando o $AFDService$ e a abordagem para redução de mensagens de controle.

6 AVALIAÇÃO, VALIDAÇÃO E TESTES

O propósito geral deste capítulo é apresentar os experimentos e detalhes sobre a realização de testes, com o intuito de avaliar o impacto da abordagem de redução de mensagens de controle, proposta no capítulo 4. Métricas para avaliação da qualidade de serviço do detector de defeitos, como as propostas por Chen, Toueg e Aguilera (2000) foram utilizadas, com a finalidade de fornecer uma visão consistente do impacto na QoS dos algoritmos propostos.

Uma breve descrição sobre o cenário utilizado para a execução dos testes também será apresentada. As definições das métricas, algumas hipóteses assumidas e os resultados dos testes são apresentados na seqüência.

6.1 Sobre a Escolha do Ambiente para Execução de Testes

Avaliar e comparar serviços não é uma tarefa trivial, principalmente em ambientes distribuídos. A dificuldade não está somente em determinar quais métricas devem ser utilizadas e sim como aplicá-las. Montar um ambiente distribuído envolve ter disponível uma gama de dispositivos, o que muitas vezes se torna inviável. Assim, uma alternativa bastante utilizada é explorar o comportamento dos componentes através de simulação. Entretanto ainda há uma necessidade por plataformas que simulam um ambiente distribuído; e o que se observa nas plataformas existentes é que há um alto grau de dificuldade para o entendimento do funcionamento dos simuladores. Isto se torna ainda mais complexo quando estes simuladores não são acompanhados por documentações.

Um serviço que permite avaliar algoritmos distribuídos, através de simulação ou execução em ambiente real, é a plataforma Neko (URBÁN; DÉFAGO; SCHIPER, 2002). Devido à flexibilidade desta ferramenta, este trabalho foi inicialmente avaliado através de simulação fazendo uso da plataforma Neko (TURCHETTI; NUNES, 2005). Os resultados preliminares permitiram, de uma maneira ágil, obter resultados que possibilitaram avaliar a abordagem proposta. Com os resultados das análises, verificou-se a viabilidade da abordagem e assim conseguiu-se ver a possibilidade da implementação da mesma no serviço $AFDService$.

Entretanto, montar um cenário real para a execução do serviço tornou-se um desafio, pois haveria a necessidade de se ter disponíveis diversos recursos. Este problema tornou-se ainda maior com a necessidade de se avaliar a abordagem em ambientes de larga escala onde o controle foge do alcance, se comparado a um laboratório tradicional com uma LAN interna onde tudo pode ser rigorosamente controlado.

O uso do laboratório mundial denominado PlanetLab (PETERSON et al., 2005) foi a solução encontrada. O PlanetLab é constituído por diversas Universidades e algumas das maiores empresas tecnológicas do mundo, como a Intel e a HP (*Hewlett-Packard*), e caracteriza-se por fornecer aos usuários um laboratório virtual que permite a execução de novos serviços em redes de grande escala. Neste contexto optou-se por aplicar os experimentos no PlanetLab, uma vez

que o problema da explosão de mensagens de monitoramento ganha a proporção desejada neste ambiente.

A figura 6.1 apresenta os nodos utilizados, bem como a distribuição deles no laboratório virtual. O $cluster$ é composto por 8 processos, um por nodo, todos executando no mesmo sistema operacional e mesmo $hardware$. Os respectivos nomes e endereços podem ser visualizados no apêndice A.

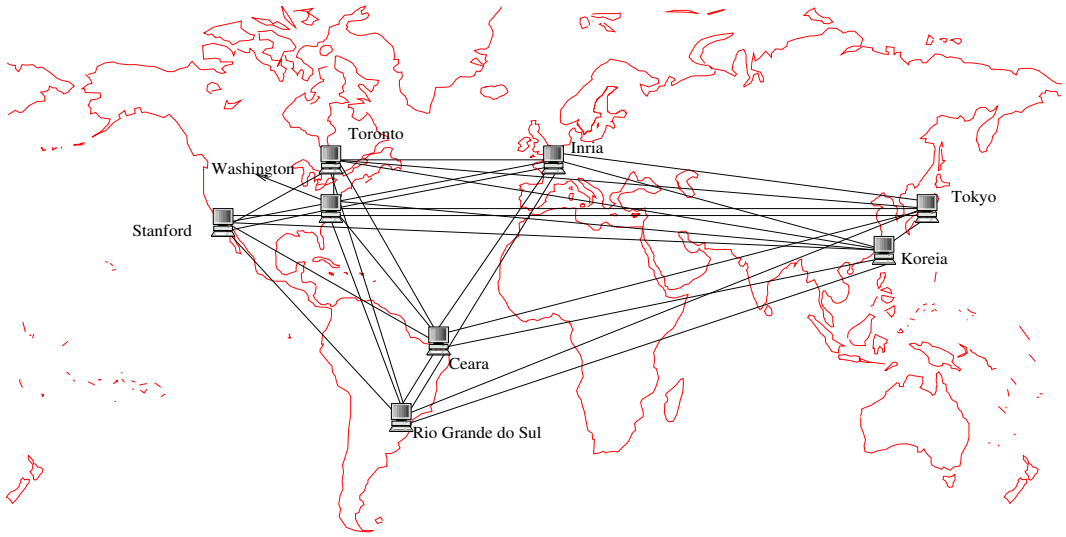


Figura 6.1: Cluster no Planet-Lab

6.2 Redução de Mensagens de Controle

Para os experimentos realizados neste trabalho, inicialmente define-se dois cenários para a execução dos testes: SF- sem falhas; CF- com falhas. Para avaliar o impacto dos detectores de defeitos quanto ao número de mensagens de controles geradas assume-se um cenário SF. O motivo da definição deste cenário é que, em situações de ocorrência de falhas dos processos, pode-se ter uma variação indesejada no número de mensagens de controle enviadas, tendo como conseqüência valores inconsistentes.

A definição destes cenários é importante, não somente para o cômputo das mensagens, mas para outras métricas inclusive. Por exemplo, o cenário CF permite medir o tempo de detecção na ocorrência de uma falha/*crash* para cada um dos algoritmos. O cenário SF também será utilizado para computar a terminação do algoritmo de consenso. O tempo de duração e a freqüência de falsas suspeitas podem ser calculados em ambos cenários (SF ou CF).

A seguir, define-se a métrica para o cômputo do número de mensagens de controle geradas na rede pelos algoritmos *Push* e *Pull* tradicionais e as estratégias ATF e AMA. Na seqüência apresentar-se-ão as análises dos resultados obtidos nos experimentos. Os comparativos entre diferentes detectores foram testados sobre situações idênticas, visando a consistência dos resultados, bem como conclusões obtidas.

6.2.1 Número de Mensagens de Controle Enviadas NM

O número de mensagens enviadas permite avaliar o tráfego na rede gerado pelas mensagens de controle. O número de mensagens trocadas para cada processo utilizando o algoritmo de detecção *Pull* é igual a $2(n-1)$ mensagens, onde n é o número de processos. Para nossos cálculos isto corresponde a $2(n_{SF}-1)$, onde n_{SF} corresponde ao número de processos sem a

ocorrência de falhas. O modelo *Push* em cada ciclo envolve a troca de $n-1$ mensagens por processo, ou $n_{SF}-1$. O cálculo para o total de mensagens enviadas por processo é:

$$NM_p = \sum M\_send$$

M_send corresponde a cada mensagem enviada. Como no modelo *Pull* cada envio de mensagem gera um retorno, o total de mensagens enviadas e recebidas por um processo é dado por:

$$NM_p = (\sum M\_send).2$$

6.2.1.1 NM Enviadas pelas Estratégias Propostas Comparadas a seus Modelos Tradicionais

Nesta seção são realizados os experimentos relacionados ao número de mensagens enviadas no canal de comunicação em detrimento das ações de monitoramento dos algoritmos de detecção de defeitos. Para as primeiras análises variou-se apenas a periodicidade de envio das mensagens. O objetivo é verificar se as estratégias propostas obtêm melhores resultados em períodos com maior ou menor carga na rede.

Os intervalos de amostras para todos os experimentos foi de 60 minutos, sendo que a amostragem foi realizada em diferentes horários no decorrer do dia e os valores que serão apresentados correspondem a dados médios. Para a análise dos experimentos com a estratégia AMA, fez-se necessário a implementação de uma aplicação sintética. Esta aplicação implementa as interfaces disponibilizadas pelo serviço $AFDService$. Para os primeiros experimentos a aplicação sintética troca informações com outras aplicações em períodos constantes de 10000 ms. O objetivo é avaliar o reaproveitamento de mensagens pelo serviço de detecção de defeitos.

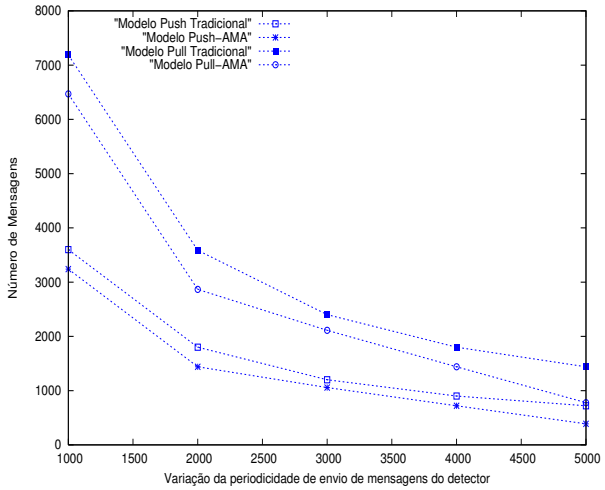


Figura 6.2: Número de mensagens enviadas variando o Δ_i, estratégia AMA

A figura 6.2 apresenta o experimento aplicando a estratégia AMA aos algoritmos *Pull* e *Push* comparados aos seus modelos tradicionais. Este experimento revela que o reaproveitamento da estratégia AMA está relacionada a proporção da periodicidade de envio da aplicação juntamente com a periodicidade de envio do algoritmo de detecção. Observando o gráfico da figura 6.2 pode-se analisar que a tendência maior para um melhor reaproveitamento de mensagens da aplicação ocorre em situações em que a periodicidade de envio do detector de defeitos se aproxima com a periodicidade de envio da aplicação sintética.

Observe que para um Δ_i igual a 1000 ms tem-se um ganho aproximado de 10% para ambos modelos e conforme o Δ_i cresce a estratégia obtém resultados ainda melhores, ou seja, para Δ_i

igual a 5000 ms obtém-se um ganho de 46% aproximadamente. Em síntese o melhor caso para a estratégia AMA é ter uma aplicação que envia mensagens em períodos menores que o Δ_i. Este caso pode ser visualizado na figura 6.3, onde fixou-se a periodicidade de envio do detector em Δ_i igual a 3000 ms e variou-se a periodicidade de envio da aplicação. Para os instantes em que a aplicação envia mensagens em períodos menores que o detector de defeitos observa-se um reaproveitamento de mensagens de 100%. Para instantes em que a aplicação envia mensagens em períodos superiores a periodicidade do detector de defeitos, o experimento revela que o grau de reaproveitamento de mensagens se relaciona à quantidade de tempo em que se pode atrasar uma mensagem futuramente; ou seja, quanto mais próximo do instante em que o detector for gerar uma mensagem de controle, ele reaproveita uma mensagem melhor. Evidência disto pode ser observado no gráfico da figura 6.3 onde se tem um melhor reaproveitamento nos instantes em que a periodicidade de envio da aplicação é de 5000 ms do que em períodos menores, como por exemplo em períodos de 3500 ms.

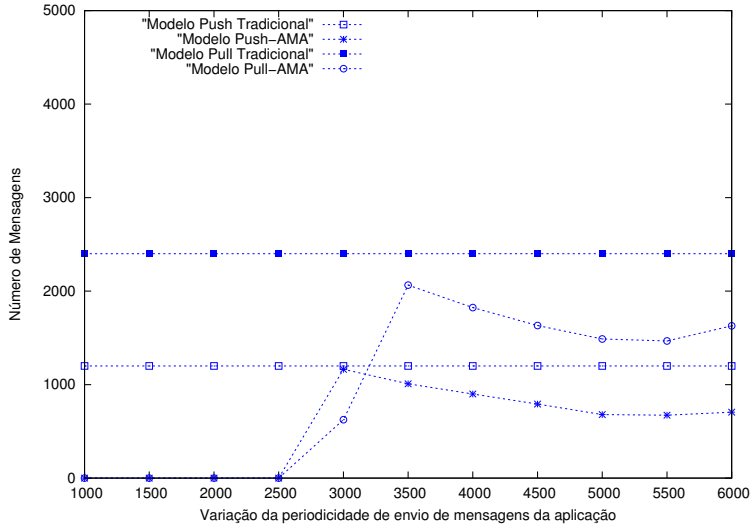


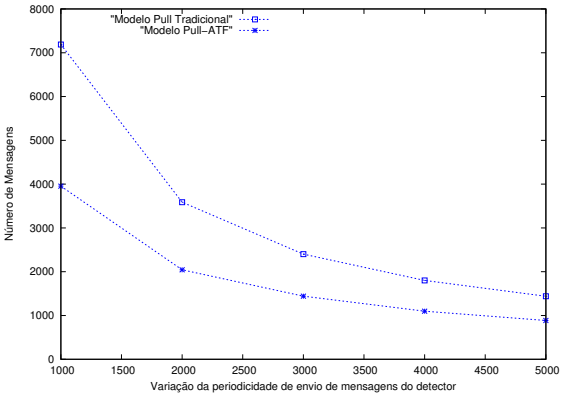
Figura 6.3: Número de mensagens enviadas variando a periodicidade da aplicação, estratégia AMA

Entretanto, a tendência do gráfico é diminuir o reaproveitamento de mensagens conforme se aumenta a periodicidade de envio da aplicação.

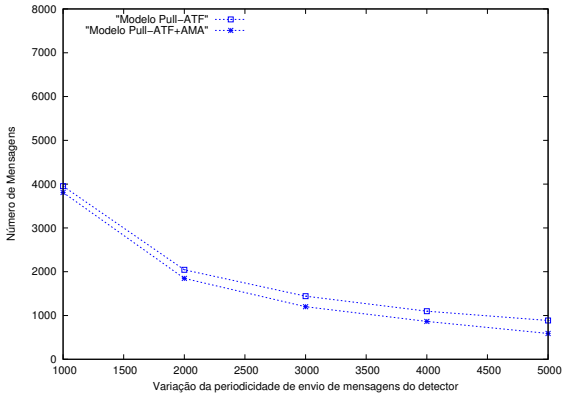
Os experimentos com a estratégia ATF e a combinação de ATF+AMA, aplicadas ao modelo *Pull* são apresentados na figura 6.4. Ressalta-se que a estratégia ATF só se aplica a modelos de monitoramento que requisitam estados aos processos monitorados (vide capítulo 4). Analisando os resultados da figura 6.4(a) pôde-se observar que em períodos com maior sobrecarga do detector, por exemplo, Δ_i igual a 1000 ms, obteve-se os melhores resultados tendo um ganho aproximado de 45% no número de mensagens enviadas, ao passo que para um Δ_i igual a 5000 ms, tem-se um ganho aproximado de 38%. Este experimento indica que a estratégia ATF tem um melhor proveito em períodos de sobrecarga do canal de comunicação, sendo o inverso do que foi observado nos experimentos com a estratégia AMA.

Aplicando-se ambas estratégias (AMA e ATF) ao modelo *Pull*, observa-se (figura 6.4(b)) que a estratégia AMA consegue obter um ganho extra de 11,8% somado ao ganho da estratégia ATF. Além disto, observa-se novamente que o ganho da estratégia AMA aumenta conforme se eleva o Δ_i. Por exemplo, para os resultados com Δ_i igual a 1000 ms obteve-se um ganho aproximado de 4%, enquanto que para os experimentos com o Δ_i igual a 5000 ms observa-se um ganho de 33%. Combinando as duas estratégias obteve-se (figura 6.4(c)) os melhores

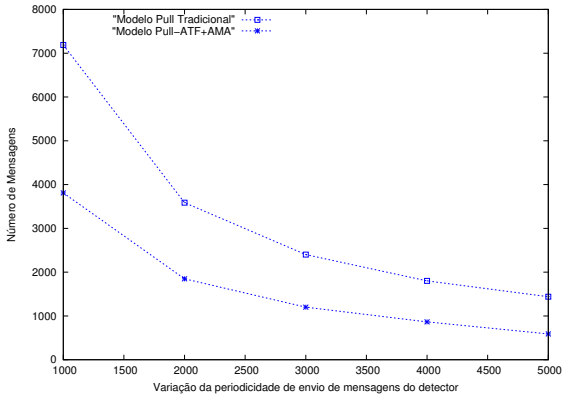
resultados para a redução do número de mensagens de controle, atingindo um ganho médio de 55%, se comparado ao modelo *Pull* tradicional.



(a) Ganho da estratégia ATF



(b) Ganho extra aplicando a estratégia AMA



(c) Ganho da combinação ATF+AMA comparada ao modelo *Pull* tradicional

Figura 6.4: Número de mensagens enviadas variando o Δ_i, estratégia ATF e combinação ATF+AMA

Um outro experimento foi realizado com o objetivo de verificar o comportamento das estratégias quando há crescimento do número de processos, ou seja, situação onde a explosão de mensagens pode impedir a escalabilidade do sistema. Para obter este resultado utilizou-se todos os processos/nodos do *cluster* do PlanetLab conforme apresentado na figura 6.1. O resultado deste experimento, o qual pode ser visualizado na figura 6.5, permitiu observar que com o aumento do número de processos no sistema as estratégias mantiveram-se com o mesmo ganho médio, se comparado aos experimentos com dois processos. Além disto, apesar da estratégia AMA depender do comportamento (troca de informações) da aplicação cliente, os melhores resultados obtidos, em termos de número de mensagens, foi aplicando esta estratégia ao modelo *Push*. Entretanto, em termos de porcentagem de ganhos, se comparado aos seus respectivos modelos tradicionais, a estratégia ATF obteve um resultado surpreendente, ou seja, um ganho de aproximadamente 45% quando aplicada ao modelo *Pull*, quase se igualando ao modelo *Push* tradicional.

Em síntese, os resultados mostram que as estratégias AMA e ATF reduzem o número de mensagens de controle se comparadas aos modelos tradicionais. Além disto, foi mostrado no capítulo 4, que aplicá-las aos modelos tradicionais exige pequenas alterações que resumem-se em alterar a semântica das mensagens.

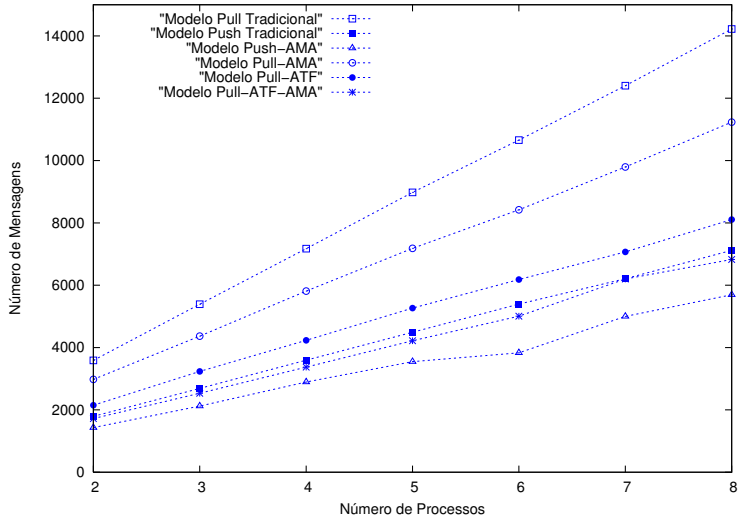


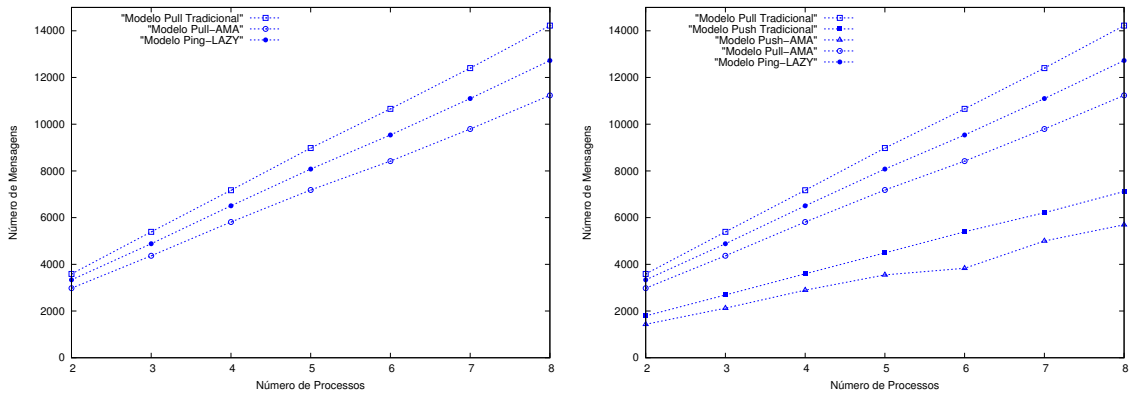
Figura 6.5: Número de mensagens enviadas variando o número de participantes

6.2.1.2 NM Enviadas pelas Estratégias Propostas Comparadas ao Detector Lazy

Comparar o algoritmo Lazy com as estratégias propostas, assumindo as mesmas condições é uma tarefa que necessita de algumas suposições que serão apresentadas nesta seção, pois aquele algoritmo, como já apresentado no capítulo 3, utiliza mensagens do detector somente quando uma aplicação cliente necessitar saber o estado de um determinado processo; os autores denominam esta ação de *query*. Entretanto isto fará com que a aplicação cliente se responsabilize em perguntar de tempos em tempos o estado de um determinado processo, caso contrário, poderá haver situações em que nunca será reportada à aplicação cliente um defeito ocorrido em um processo qualquer. Por exemplo, assumindo que uma aplicação cliente q não está mais se comunicando com nenhuma outra aplicação do sistema, mas ela aguarda a resposta de um processo p que tenha falhado, se nestas condições a aplicação q não perguntar o estado de p para o detector ele nunca irá reportar tal falha. Em síntese, pode-se afirmar que, para esta situação não ocorrer, tem-se que implementar uma periodicidade de *querys* realizada pela aplicação cliente ao detector, tal periodicidade será equivalente ao Δ_i utilizado pelos detectores propostos nesta dissertação.

Para avaliar o reaproveitamento de mensagens das aplicações pelos algoritmos de detecção, foi utilizada uma aplicação sintética que troca mensagens no transcorrer de cada 10000 ms. Estes experimentos podem ser visualizados na figura 6.6. Analisando o gráfico 6.6(a) o qual apresenta uma comparação do modelo *Pull* tradicional, *Pull* AMA e o algoritmo Lazy, pode-se verificar que ambas variações do modelo *Pull* que reaproveitam mensagens das aplicações obtiveram êxito se comparado ao modelo tradicional. Entretanto a estratégia AMA demonstrou-se mais eficiente garantindo um ganho aproximado de 11,4% de economia nas mensagens. Este ganho está agregado à ausência de confirmação para cada mensagem da aplicação enviada; além disso, como a estratégia AMA contabiliza somente as mensagens emitidas pela aplicação no processo receptor, para estas mensagens o canal de comunicação não necessita ser confiável. Em contrapartida, o algoritmo Lazy, devido ao fato deste contabilizar as mensagens no emissor e por depender de uma confirmação, seus autores supõem dipor de um canal confiável, onde as mensagens não podem ser criadas, duplicadas, alteradas e nem perdidas (FETZER; RAYNAL; TRONEL, 2001).

Outra diferença que vale ser salientada é que, como apresentado no gráfico 6.6(b), a estratégia AMA pode ser estendida ao modelo *Push* inclusive, e para tal combinação foi obtido o



(a) Lazy comparado ao modelo Pull

(b) Análise de todos os algoritmos

Figura 6.6: Comparação do número de mensagens de controle incluindo o algoritmo Lazy

melhor resultado entre todos os algoritmos experimentados. Entretanto o algoritmo Lazy somente pode ser aplicado a algoritmos que trabalham na forma de monitoramento bidirecional (*two-ways*).

6.3 Influência das Estratégias na QoS dos Detectores de Defeitos

Na seção 2.3.5 foi apresentado um grupo de métricas definidas por Chen, Toueg e Aguilera (2000). Tal grupo permite avaliar a velocidade com que os serviços geram suspeitas e a exatidão destas. Nesta seção são realizados experimentos utilizando as métricas propostas por Chen, com o objetivo de avaliar o impacto na QoS dos detectores de defeitos quando estes fizerem uso das estratégias AMA e ATF. A seguir define-se cada métrica utilizada, bem como seus respectivos experimentos.

6.3.1 Tempo de Detecção T_D

O tempo de detecção é uma métrica fundamental para avaliar um detector de defeitos. Este tempo é o transcorrido desde o instante que determinado processo falhou até a percepção de tal falha pelo detector de defeitos. Tendo em vista que um detector de defeitos observa falhas em seus componentes através da chegada de mensagens vindas dos processos monitorados e que a periodicidade destas mensagens varia de acordo com parâmetros do algoritmo de detecção, tem-se que quanto menor for o período para o envio de mensagens, mais rápido um detector conseguirá ter o conhecimento de falhas (CAMARGOS, 2003). Entretanto, esta rapidez tem um custo pois a velocidade de detecção consome largura de banda da rede e recursos computacionais.

A seguir será detalhado (seção 6.3.2) como foi realizado o cômputo do T_D para o pior caso, e analisado (seção 6.3.2.1) como as estratégias ATF e AMA o influenciam.

6.3.2 Cálculo do T_D

Para cálculo do tempo médio do T_D foi realizado um histórico de todos os T_D já computados. Baseando-se neste histórico, foi proposta a seguinte fórmula dada por:

$$\overline{T_D} = \frac{\sum_{i=1}^{n} T_{D_i}}{n}$$

onde n é o número de vezes que o T_D foi computado para uma dada execução. O cálculo do tempo médio do T_D foi aplicado para todos os casos de testes.

Pull **Tradicional**

Para computar a métrica T_D age-se como se todas as falhas no processo monitorado ocorressem tão logo ele responda a uma requisição de vida, situação que é considerada como sendo o pior caso. Além disto, foi assumido que os atrasos de comunicação entre uma mensagem *liveness request* e sua respectiva resposta são idênticos. Desta forma, o tempo de detecção é calculado da seguinte forma:

$$T_D = \Delta_i + \Delta_{to} - \frac{rtt_{seq}}{2}$$

onde Δ_i é o intervalo para o início de uma requisição de vida, Δ_{to} corresponde ao tempo limite de espera para a correspondente requisição e rtt_{seq} é o último rtt computado. A representação ilustrativa do cômputo do T_D é dado na figura 6.7 que ilustra uma suspeita detectada pelo processo q, o qual monitora o estado do processo p, e mostra a origem dos componentes do cálculo.

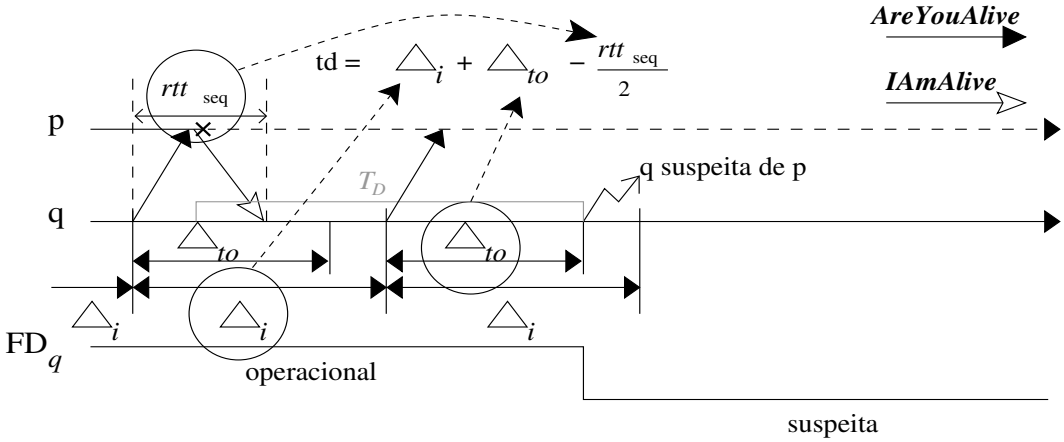


Figura 6.7: Cálculo de pior caso para o T_D no modelo *Pull* tradicional

Pull **adicionado das estratégias de aproveitamento de mensagens (ATF ou AMA)**

Como apresentado em capítulos anteriores, às estratégias ATF e AMA possuem o objetivo de reduzir o número de mensagens de controle utilizando mensagens do detector e da aplicação cliente respectivamente. Especificamente, a ATF opera dilatando o Δ_i de acordo com a percepção de sobrevivência do monitorado. Entretanto esta dilatação não afeta o T_D porque, para fins do seu cômputo, a indicação de sobrevivência se sobrepõe, mantendo o Δ_i no cálculo. Por outro lado, para o cálculo desta métrica, um processo p que monitora e é monitorado por um processo q age como se todas as falhas no processo q ocorrem tão logo ele enviasse uma requisição de vida ou a aplicação cliente trocasse informações com o processo p. Estas duas situações são consideradas no cálculo da seguinte forma:

$$T_D = \Delta_i + \Delta_{to} + \frac{rtt_{seq}}{2}$$

onde Δ_i corresponde ao Δ_i previamente ajustado e $\frac{rtt_{seq}}{2}$ corresponde ao atraso de comunicação. Fazendo uma analogia ao pior caso para o cálculo do T_D no estilo tradicional, percebe-se que o aumento utilizando as estratégias ATF ou AMA é de rtt_{seq} se comparado ao modelo tradicional, este cenário pode ser visualizado na figura 6.8.

A figura 6.8 apresenta um cenário em que ocorre um reaproveitamento de mensagem da aplicação (estratégia AMA), entretanto esta mesma situação pode ocorrer para o reaproveitamento de mensagens do tipo `AreYouAlive` (estratégia ATF).

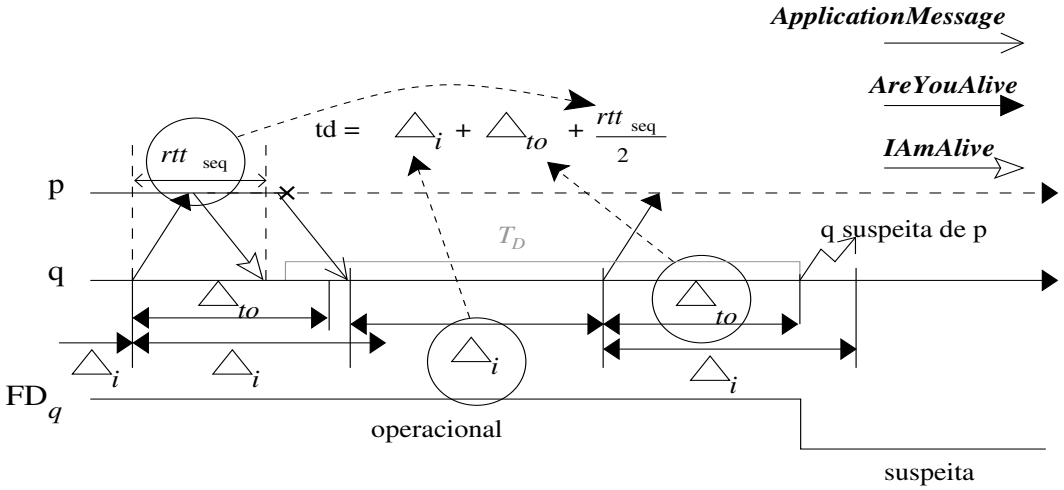


Figura 6.8: Cálculo de pior caso para o T_D no modelo *Pull* para as estratégias AMA e ATF

Push **tradicional ou estratégia AMA**

O pior caso considerado a este modelo é tão logo uma mensagem de vida seja enviada ao processo monitor o processo emissor falhe. Mesmo que a mensagem enviada seja do tipo `IAmAlive` ou `ApplicationMessage` o tempo para a detecção será dado por:

$$T_D = \Delta_{to} + Heartbeat_{Delay}$$

onde $Heartbeat_{Delay}$ é o tempo em que uma dada mensagem é enviada do processo monitorado até atingir o processo monitor. Junto a esta mensagem é enviada o instante Ts de envio da respectiva mensagem. Salienta-se que o modelo de sistemas assumido no capítulo 2 não fez nenhuma hipótese sobre a necessidade de sincronização de relógios para o funcionamento dos algoritmos de detecção de defeitos. Entretanto, somente para o cálculo do T_D a solução encontrada foi a sincronização de relógios uma vez que o cômputo do $Heartbeat_{Delay}$ é realizado da seguinte forma:

$$Heartbeat_{Delay} = Tr - Ts$$

onde Tr significa o tempo em que a mensagem atinge seu destino.

6.3.2.1 Análise do Tempo de Detecção

Para demonstrar o tempo de detecção de cada estratégia, foram retiradas amostras referentes a ocorrência de suspeitas, dentro de períodos de 24 horas, todas baseadas no pior caso como foi apresentado neste capítulo. Um *timeout* fixo com o valor de 4000 ms foi utilizado.

A tabela 6.1 apresenta os tempos de detecção para execuções com o detector *Pull* no estilo tradicional e utilizando as extensões com as estratégias AMA e ATF. Os resultados mostram que o tempo médio de detecção foi de $\overline{T_D} = 7305ms$ para o modelo *Pull* tradicional, ao passo que o pior resultado foi obtido com a estratégia AMA referente a $\overline{T_D} = 9903ms$. A estratégia ATF plicada ao modelo *Pull* obteve $\overline{T_D} = 9606ms$ pouca coisa inferior a estratégia AMA. Com o modelo tradicional obteve-se um tempo de detecção 26% menor se comparado a média entre o pior e melhor tempo obtido com as estratégias, isto implica num ganho médio de 2295 ms.

O melhor resultado obtido com o modelo *Pull* foi a combinação de ATF + AMA com um resultado de $\overline{T_D} = 9201ms$. O aumento no tempo de detecção do *Pull* pelo uso das estratégias já era esperado. Entretanto este não é o único fator que indica QoS. Outros experimentos (mostrados em seguida) mostram que a precisão pode ser melhorada.

Tabela 6.1: Comparação do $\overline{T_D}$ para as extensões utilizando o modelo *Pull*

	Algoritmos utilizados			
Métrica	$Pull$	$PullAMA$	$PullATF$	$PullATFAMA$
$\overline{T_D}$	7305	9903	9606	9201

Por outro lado, quando aplicada a estratégia AMA ao modelo *Push* e comparado ao seu respectivo modelo tradicional (tabela 6.2), observou-se que esta estratégia não piorou o tempo de detecção. A média obtida pelo modelo tradicional é de aproximadamente $\overline{T_D} = 5391ms$, o menor tempo se comparado com os experimentos utilizando o modelo *Pull*. Entretanto a média pertencente a estratégia AMA é de aproximadamente $\overline{T_D} = 5080ms$ o que corresponde uma melhora de aproximadamente 5,8% se comparado ao seu correspondente na forma tradicional.

Tabela 6.2: Comparação do $\overline{T_D}$ para as extensões utilizando o modelo *Push*

	Algoritmos utilizados	
Métrica	$Push$	$PushAMA$
$\overline{T_D}$	5391	5080

6.3.3 Tempo de Duração do Erro T_M

Para o cálculo do T_M é assumido que um detector FD_q que monitora um processo p no instante t tem o estado como S ou T, significando que o processo p está suspeito ou operacional respectivamente. Uma transição *S-transition* ocorre quando o estado do processo p altera-se de T para S, e uma transição *T-transition* ocorre quando o estado do processo p altera-se de S para T (CHEN; TOUEG; AGUILERA, 2000). Baseando-se nas transições de estados, um histórico contendo os instantes exatos de transições é armazenado para o cálculo de T_M. Para cada par de transição, a seguinte fórmula é aplicada no cálculo do T_M:

$$T_M = t_T - t_S$$

onde t_T significa o instante de uma transição de estado de S para T e t_S o instante de uma transição de T para S.

Diferente do cálculo do T_D, o qual exigiu cálculos personalizados para cada caso utilizado, o cômputo do T_M pode ser aplicado a todos os casos e o tempo médio de duração de um erro é calculado fazendo-se:

$$\overline{T_M} = \frac{\sum_{i=1}^{n} T_{M_i}}{n}$$

onde n é o somatório das falsas suspeitas já registradas.

Esta métrica é tão importante quanto T_D, uma vez que ela é uma métrica primária juntamente com o T_{MR} que indicam a precisão (*accuracy*) do detector.

6.3.3.1 Análise do Tempo de Duração do Erro

Nesta seção será avaliada a precisão dos detectores através da métrica T_M. Para este experimento foram retiradas amostras em períodos de 24 horas, onde desta foi calculado o tempo médio para a duração de erros. A periodicidade de envio de mensagens foi de Δ_i = 5000

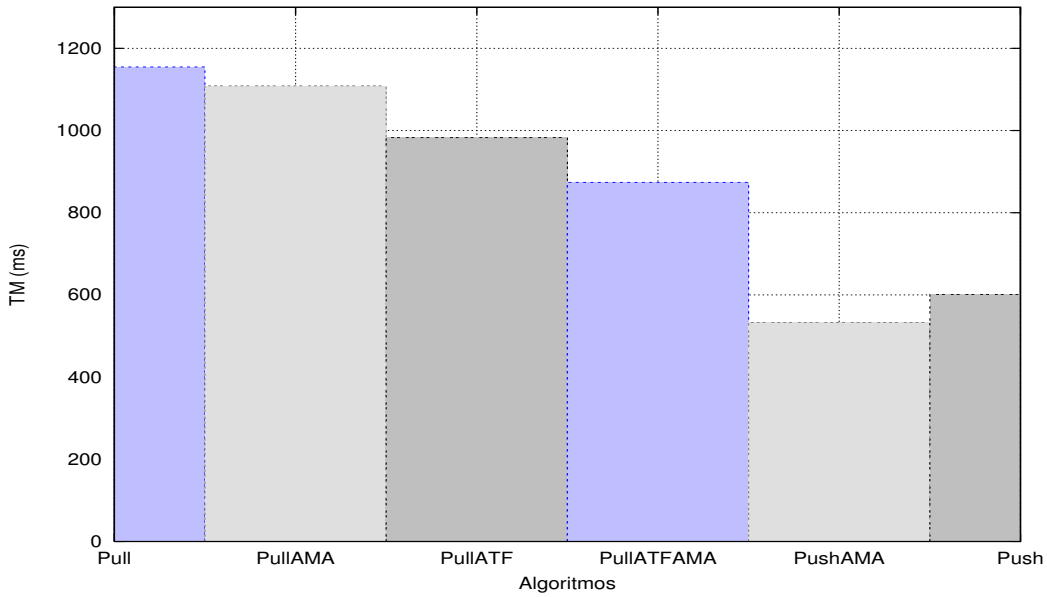


Figura 6.9: Tempo do T_M para todos os algoritmos

ms. Ocuparam-se todos os processos do *cluster* para a execução dos experimentos. O comportamento das estratégias propostas, para este experimento, demonstrou resultados bastante eficientes, que podem ser visualizados na figura 6.9.

Para o modelo *Pull* o menor $\overline{T_M}$ obtido foi com a combinação da estratégia ATF + AMA onde o valor médio observado é 24,3% menor do que o *Pull* tradicional. Já para o modelo *Push* a estratégia AMA obteve um ganho relativo de 11% se comparada ao seu estilo tradicional.

O bom desempenho das estratégias no que diz respeito ao T_M pode ser explicado pelos "tempos mínimos"obtidos, ou seja, pelos menores T_M obtidos em cada caso. As estratégias AMA e ATF aplicadas ao modelo *Pull* conseguiram atingir valores mínimos de $T_M = 1ms$, enquanto que a estratégia AMA aplicada ao *Push* atingiu valores mínimos de $T_M = 3ms$. Já para os experimentos aplicados aos modelos tradicionais observaram-se valores mínimos bem superiores. Por exemplo, o valor mínimo para reconhecer uma falsa suspeita para o modelo *Pull* tradicional é no mínimo igual ao seu respectivo rtt, entretanto esta analogia não pode ser aplicada a estratégia ATF, pois caso o respectivo processo suspeito envie uma mensagem do tipo `AreYouAlive`, automaticamente este processo será retirado da lista de processos suspeitos. Neste sentido, o valor mínimo observado pelo modelo *Pull* tradicional foi de $T_M = 108ms$, enquanto que o mínimo atingido pelo modelo *Push* tradicional foi de $T_M = 69ms$.

6.3.4 Tempo para Recorrência ao Erro T_{MR}

O tempo de recorrência ao erro T_{MR} mede o tempo entre dois erros consecutivos, ou seja entre duas falsas suspeitas. Assim, para o cálculo do T_{MR} um histórico contendo os instantes exatos de cada transição dos estados *S-transition* (apresentado na seção anterior), será utilizado. O tempo para a recorrência ao erro é calculado da seguinte forma:

$$T_{MR} = t_{S(i)} - t_{S(i-1)}$$

onde as transições $t_{S(i)}$ e $t_{S(i-1)}$ são adjacentes no tempo. Logo a fórmula para o calculo do tempo médio para a recorrência ao erro é dada por:

$$\overline{T_{MR}} = \frac{\sum_{i=1}^{n-1} T_{MR_i}}{n-1}$$

onde n é o número de falsas suspeitas já computadas.

6.3.4.1 Análise do Tempo para a Recorrência ao Erro

Nesta seção será avaliada a precisão dos detectores através da métrica T_{MR}. Para este experimento foram retiradas amostras durante períodos de 24 horas interruptos. Da mesma forma que para os experimentos do T_M, o comportamento das estratégias para o cálculo do T_{MR} demonstrou resultados bastante eficientes, que podem ser visualizados na figura 6.10.

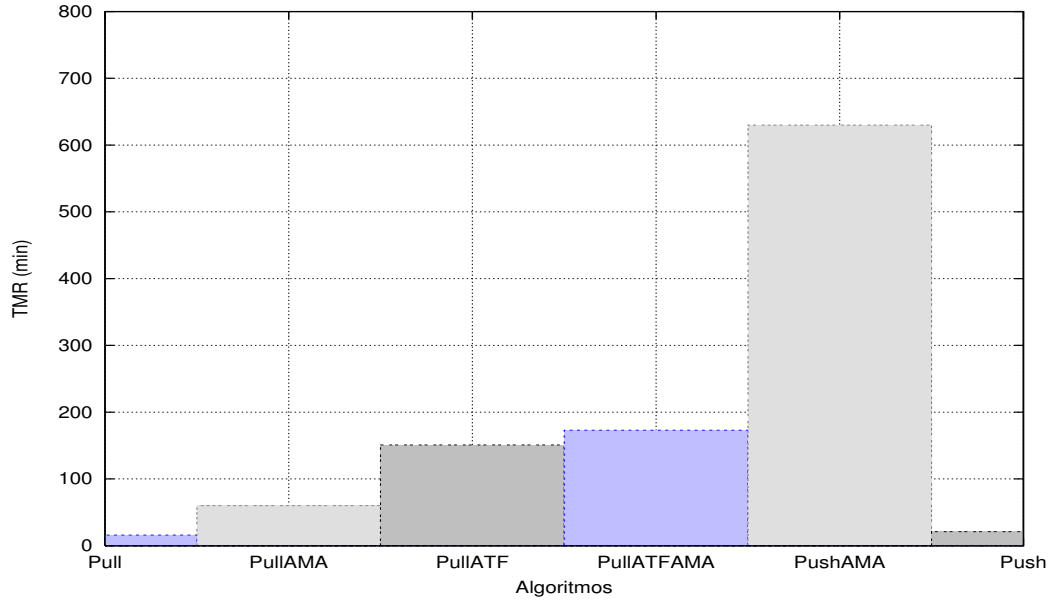


Figura 6.10: Tempo do T_{MR} para todos os algoritmos

Para todos os resultados as estratégias propostas obtiveram ganho se comparadas aos modelos tradicionais. O $\overline{T_{MR}}$ para o pior resultado obtido corresponde a 16,98 min atingido pelo modelo *Pull* tradicional, ao passo que a combinação de ATF + AMA obteve o melhor resultado para este mesmo modelo, chegando a média de 173,49 min o que equivale a um ganho aproximado de 921,73%. A estratégia AMA estendida ao modelo *Push* obteve o melhor resultado para todos os experimentos, sendo de aproximadamente $\overline{T_{MR}}$ = 630,65 min, equivalente a um ganho aproximado de 2834,62% se comparado ao modelo *Push* tradicional.

Em síntese pode-se dizer que a estratégia AMA aplicada ao modelo *Push* obteve o melhor resultado em termos de precisão, pois este apresenta o menor T_M e o maior T_{MR}. Logo este algoritmo obtém a maior rapidez para a recuperação de um erro e a menor taxa de recorrência ao mesmo. A tabela 6.3 confere a eficiência das estratégias comparadas aos modelos tradicionais, e principalmente demonstra em termos do número de falsas suspeitas que realmente a estratégia AMA obteve hegemonia sobre as demais.

Tabela 6.3: Comparação entre o n^o de falsas suspeitas para todas as extensões

	Algoritmos utilizados					
Métrica	$Pull$	$PullAMA$	$PullATF$	$PullATFAMA$	$Push$	$PushAMA$
$N^o suspeitas$	5089	1436	569	498	4021	137

Estes experimentos demonstram que as estratégias propostas além de reduzirem o número de mensagens de controle, também garantem precisão para as suspeitas observadas.

6.4 Tempo para a Execução do Consenso

Analisar o impacto dos detectores de defeitos sobre o protocolo de consenso tem sua relevância, pois cada detector possui vantagens e desvantagens sob o ponto de vista do próprio algoritmo de consenso. Uma métrica, utilizada inicialmente por Sergent, Défago e Schiper (2001) é avaliar o tempo de determinação do consenso. Esta métrica passou a ser utilizada em diversos trabalhos pois este protocolo exige agilidade do detector para mobilizar o término de execução do consenso.

Os experimentos de Estefanel (2001) comprovaram que os maiores fatores que contribuem para degradar o desempenho dos protocolos de consenso são as detecções incorretas, os atrasos nas transmissões das mensagens, as limitações físicas no canal de comunicação e a sobrecarga no processamento das mensagens. Já Sergent, Défago e Schiper (2001) definem como um dos principais fatores o número de mensagens enviadas pelos detectores para realizar o monitoramento. Baseando-se nesta justificativa, eles propuseram os algoritmos *Silent* e o *Heartbeat* especializado, apresentados no capítulo 3.

Para garantir que todas as rodadas do consenso tenham exatamente o mesmo comportamento, o cenário considerado foi o SF. Desta maneira somente as execuções livres de falhas foram computadas, a média de tempo de execução foi sobre 1000 execuções do protocolo de consenso. As execuções possuem rodadas onde o coordenador propõe um valor para todos os processos participantes: se a maioria aceitar o valor, o coordenador responde com a decisão tomada; neste instante a execução do consenso é terminada e o valor de execução é armazenado. O algoritmo de consenso utilizado para os experimentos foi o proposto por Chandra e Toueg (1996); a integração do mesmo com o serviço $AFDService$ foi detalhado na seção 5.4.

O cômputo do tempo de execução é inicializado quando um processo envia uma mensagem do tipo `ESTIMAR` sugerindo um novo valor a ser decidido. Como mostrado na seção 5.4, esta mensagem corresponde à fase 1 do protocolo de consenso e sua execução é terminada quando um valor é decidido. Esta decisão é indicada pela mensagem do tipo `DECIDIR`, correspondente à fase 4. O tempo médio para o cálculo da execução do consenso é dado por:

$$\overline{CS_{Exec}} = \frac{\sum_{i=1}^{n} CS_{Exec_i}}{n}$$

onde n é o número de execuções livres de suspeitas.

Observe-se que o cálculo do CS_{Exec} inclui os tempos de processamento e de comunicação, sendo o tempo de comunicação muito superior ao tempo de execução.

6.4.1 Análise do Tempo para Execução do Consenso

Os experimentos realizados sobre o consenso foram executados considerando o cenário SF. Logo, nas execuções com ocorrência de falhas, os tempos não foram computados. O objetivo deste experimento é avaliar os detectores de defeitos, bem como as estratégias propostas, sob o ponto de vista do protocolo de consenso, uma vez que esta combinação é amplamente utilizada em sistemas replicados tolerantes a falhas. Os resultados representam uma média para a tomada de tempo de terminação do consenso, onde são retiradas amostras de 1000 execuções seqüenciais.

Como apresentado neste capítulo, as execuções são computadas quando uma aplicação sugere um novo valor a ser decidido. Este valor é repassado ao protocolo de consenso e este inicia a execução enviando uma mensagem do tipo `ESTIMAR` sugerindo o valor que a aplicação propôs.

Nas execuções do protocolo de consenso, utilizou-se uma aplicação cliente para estimar valores a serem decididos iniciando assim o serviço de consenso. A proposição é feita em ciclos

iguais a 5000 ms, e os tempos para execução podem ser visualizados na figura 6.11 onde varia-se o número de processos participantes do consenso.

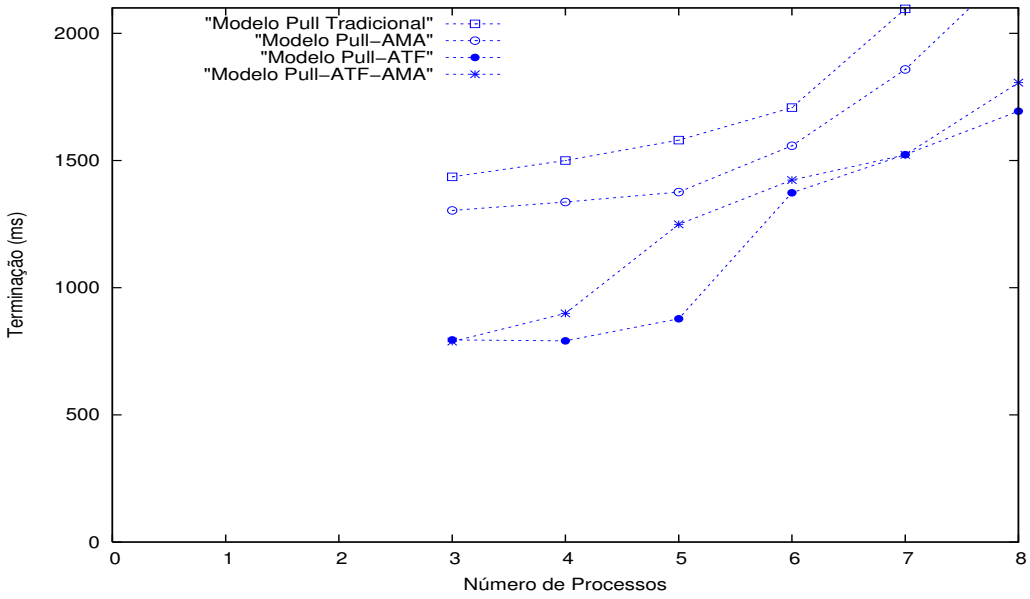


Figura 6.11: Tempo para terminação do consenso, modelo *Pull* e estratégias

Para a prática dos testes o número de processos varia de 3 a 8. Medidas para 2 processos foram omitidas, pois o algoritmo de consenso não suporta falhas neste caso. Apesar do cálculo para a terminação do consenso assumir somente execuções livres de falhas, optou-se por este cenário, uma vez que o cômputo do consenso para somente dois processos não tem muito sentido (HAYASHIBARA et al., 2002) [1].

Observando o gráfico da figura 6.11, pode-se verificar que o melhor desempenho, em termos da terminação do consenso, foi obtido pela estratégia ATF atingindo um valor médio de $1175ms$. Este valor equivale ao ganho percentual de aproximadamente 35% se comparado ao *Pull* Tradicional que obteve um valor correspondente de $1821ms$.

Um ponto favorável apontado pelas estratégias propostas é que todas elas conseguiram reduzir o tempo de terminação do consenso, confirmando a tese de que reduzindo o número de mensagens obtém-se uma redução no tempo de terminação do consenso, embora a estratégia que obteve melhor desempenho em relação ao consenso não corresponda a que obteve maior redução do número de mensagens.

A combinação de ATF + AMA degrada seu desempenho conforme o número de processos participantes aumenta, ou seja, a combinação de ATF + AMA atinge um ganho aproximado de 45% com o número de processos igual a 3, ao passo que obtém um ganho aproximado de 30% com o número de participantes igual a 8, tomando por base os valores obtidos pelo modelo *Pull* tradicional. O mesmo pode ser dito da estratégia ATF, que obtém um ganho aproximado de 45% com o número de processos igual a 3 e um ganho de 35% com o número de participantes igual a 8. Estes valores permitem concluir que, em situações onde o número de participantes cresce, outros fatores podem ser mais influentes para o desempenho do consenso do que somente o número de mensagens como, por exemplo, o coordenador torna-se um gargalo onde o processamento, 'bufferização', memória entre outras variáveis podem ser mais relevantes. Entretanto estes fatores não foram avaliados neste trabalho.

[1] Por exemplo, se um algoritmo de consenso necessitar de pelo menos 50% dos processos livres de falhas para tomar uma decisão, neste caso 2 processos não ofereciam suporte a execuções com uma única falha.

Na figura 6.12 são mostrados os tempos de terminação do consenso fazendo uso do modelo *Push* tradicional e a estratégia AMA, variando o número de participantes do consenso. O gráfico permite observar que a estratégia AMA obtém pequenas melhoras para o tempo de terminação do consenso, ou seja, o tempo médio atingido por esta estratégia foi de $1145ms$. Este valor equivale a um ganho percentual de aproximadamente 4%, se comparado ao *Push* Tradicional $1195ms$. Em síntese a estratégia AMA obteve resultados melhores se aplicada ao modelo *Pull* do que ao modelo *Push*, uma vez que esta estratégia aplicada ao modelo *Pull* obteve um ganho aproximado de 11%.

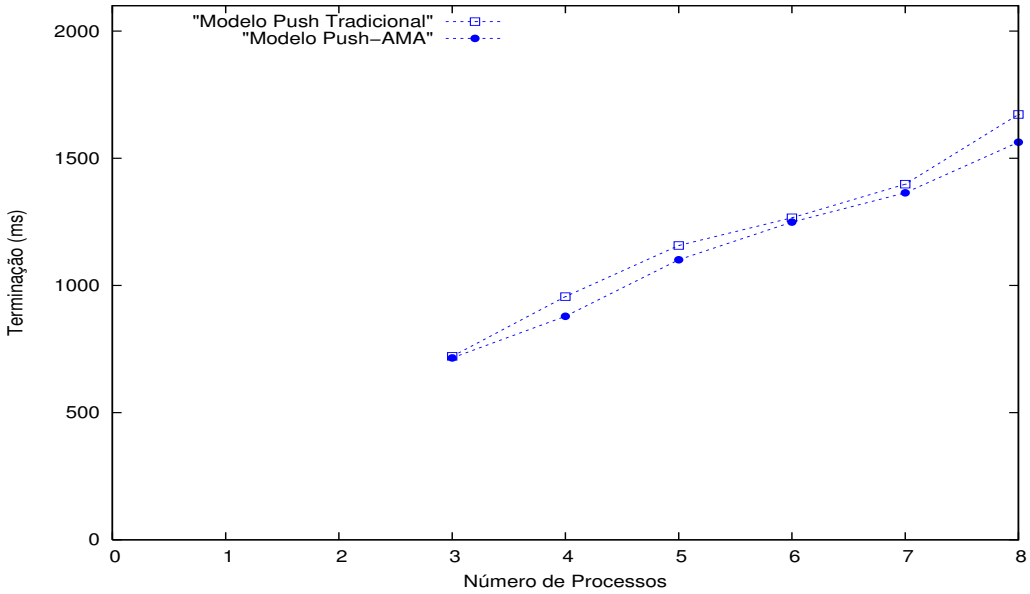


Figura 6.12: Tempo para terminação do consenso, modelo *Push* e estratégias

6.5 Conclusões Parciais

Neste capítulo, foram apresentados os resultados dos experimentos executados em um *cluster* distribuído sobre a Internet. Este ambiente foi utilizado com a finalidade de caracterizar o ambiente onde o problema da explosão de mensagens aparece com mais intensidade. Neste contexto, foram realizados experimentos com as estratégias AMA e ATF com o objetivo de demonstrar tanto a redução do número de mensagens de controle enviadas quanto o impacto na qualidade de serviço do detector.

Para estes experimentos foi inicialmente executado o algoritmo variando a periodicidade de envio das mensagens de controle. Neste ambiente, observou-se um ganho acentuado pela estratégia ATF que tira proveito das próprias mensagens dos detectores; em contrapartida foi analisado um aumento no tempo de detecção quando executada tal estratégia considerando o pior caso. Contudo, este aumento no T_D é em ordem de grandeza bem menor se considerados os ganhos logrados por esta estratégia, uma vez que o algoritmo de consenso obteve o melhor resultado em termos de tempo de terminação quando executado em conjunto com a estratégia ATF.

Considerando o reaproveitamento de mensagens das aplicações clientes, a estratégia AMA, no geral, obteve ótimos resultados, uma vez que sua abordagem demonstrou-se extensível a ambos estilos de detecção *Push* e *Pull*. Além disso, a sua utilização demonstrou ser melhor, em diversos aspectos como, por exemplo, número de mensagens enviadas, aplicabili-

dade/extensibilidade, entre outros, quando comparada ao algoritmo Lazy. Outra observação refere-se aos resultados obtidos com diferentes estilos de monitoramento; por exemplo, apesar desta estratégia reduzir o número de mensagens para ambos estilos, observou-se, no geral, um desempenho melhor na combinação de Push-AMA, por exemplo, ao passo que a combinação Pull-AMA aumenta o T_D, observou-se uma redução para esta mesma métrica quando implementada a estratégia AMA ao modelo *Push*. Além disto, a combinação das estratégias ATF-AMA juntamente ao modelo *Pull*, conseguiu reduzir o número de mensagens de controle ao ponto de obter resultados melhores que o modelo *Push* tradicional; este experimento demonstrou o desempenho das estratégias e sua relevância.

Particularmente a abordagem proposta poderia ter sido avaliada com outros algoritmos de detecção, além disso, experimentos que não obtiveram bons resultados poderiam ter sido mais explorados, entretanto estas tarefas serão deixadas para trabalhos futuros.

7 CONCLUSÃO

Focado no uso de detectores de defeitos em ambientes distribuídos de larga escala, este trabalho explorou uma nova abordagem para minimizar o problema da explosão de mensagens. A seguir faz-se uma síntese do problema, da solução e dos resultados alcançados.

Um detector de defeitos não confiável é uma abstração que viabiliza a especificação e prova de algoritmos determinísticos sobre ambientes distribuídos e assíncronos, possibilitando a especificação de sistemas tolerantes a falhas. O detector encapsula o problema da impossibilidade de distingüir um processo falho de um processo lento, normalmente resolvendo-o através de mecanismos que impõem limitações temporais para a comunicação e as execuções das tarefas. Em outras palavras, seu propósito é tentar distingüir processos falhos dos lentos com velocidade e precisão, mas por utilizar soluções aproximadas são não confiáveis por definição. Assim, usualmente detectores de defeitos são projetados para monitorarem recursos através da troca periódica de mensagens de controle, levantando uma suspeita quando não for mais observada troca de mensagens por longos períodos de tempo. Entretanto, em sistemas distribuídos de larga escala, o número de processos pode crescer de tal forma que detectores de defeitos podem comprometer seus serviços devido a explosão de mensagens de controle causada pelas inúmeras ações de monitoramento. Conseqüentemente tal explosão pode restringir a escalabilidade do sistema, comprometendo o algoritmo e a aplicação.

Avaliando os trabalhos relacionados, observou-se cinco possíveis estratégias para contornar o problema da explosão de mensagens: *i* - escolher de uma topologia lógica conveniente (anel, estrela, hierárquica, etc); *ii* - ajustar os parâmetros do detector; *iii* - escolher uma forma de monitoramento adequada (algoritmo); *iv* - método de endereçamento (*unicast* ou *multicast*); *v* - reaproveitamento de mensagens. Este trabalho explorou a estratégia de reaproveitamento de mensagens aplicando-a tanto às mensagens das aplicações clientes como às mensagens dos próprios algoritmos de detecção de defeitos.

A solução proposta, chamada nova abordagem para redução de mensagens de controle, é composta por duas estratégias: ATF e AMA. A idéia chave da estratégia ATF é trocar o significado semântico da mensagem de requisição de estado para servir também como mensagem de informação de estado. Com esta alteração, detectores de defeitos que utilizam monitoramento ativo podem reaproveitar mensagens do próprio algoritmo de detecção para suprir mensagens de controle. Por outro lado, a idéia chave da estratégia AMA é reaproveitar mensagens das aplicações clientes. Salienta-se que a estratégia AMA por si só não é inovadora, pois outros trabalhos já foram propostos nesta mesma linha de raciocínio, como por exemplo no detector Lazy proposto por Fetzer, Raynal e Tronel (2001), o qual explora muito bem tal estratégia. Entretanto, a estratégia AMA pode ser combinada com a estratégia ATF para considerar toda e qualquer mensagem como mensagem de informação de estado. Finalmente, observa-se que as idéias chaves das estratégias e a possibilidade de combinação conferem a abordagem proposta a possibilidade de ser genérica, ou seja, aplicável a diversos algoritmos.

A praticidade para a integração da nova abordagem em serviços de detecção de defeitos foi testada (capítulo 5) junto ao AFDService (NUNES, 2003). A integração demonstrou que ambas estratégias são facilmente integradas a serviços já existentes, uma vez que a implementação e a validação das estratégias ATF e AMA num ambiente real foram realizadas sem dificuldades.

Em síntese, a solução proposta é genérica e prática. Genérica no sentido que pode ser aplicada a qualquer algoritmo de detecção de defeitos, independente da forma de comunicação, parâmetros ou topologia adotada, e prática no sentido que pode ser facilmente incorporada a algoritmos já implementados, bastando trocar a semântica das mensagens de controle.

Os experimentos práticos foram realizados com o objetivo de avaliar os ganhos das estratégias em termos do número de mensagens de controle enviadas nos canais de comunicação. Para estes experimentos obteve-se bons resultados se comparados aos seus modelos tradicionais. Para a estratégia ATF foi analisado seu comportamento sob períodos adversos, ou seja, mediante períodos de sobrecarga na rede e mediante períodos mais amenos. Os resultados mostram que a estratégia ATF destaca-se em períodos de maior sobrecarga no canal de comunicação, o que revela a eficiência da estratégia uma vez que a explosão de mensagens implica diretamente na sobrecarga da rede. Já as análises com a estratégia AMA mostram que o ganho na redução do número de mensagens de controle está vinculado à periodicidade de envio do detector de defeitos em relação à periodicidade de envio da aplicação cliente. Se a aplicação cliente enviar mensagens em períodos menores que o detector de defeitos, tem-se 100% das mensagens de controle supridas (melhor caso). Por fim, a combinação das duas estratégias (ATF+AMA) chega a economizar 55% das mensagens de controle se aplicadas ao modelo *Pull*, o que o torna uma abordagem de monitoramento ativa (*Pull*) tão eficiente quanto uma abordagem de monitoramento passiva (*Push*).

Dos experimentos pode-se concluir também que para ambas estratégias os melhores resultados estão relacionados aos instantes em que o reaproveitamento de mensagens ocorre. Ou seja, o grau de reaproveitamento de mensagens diz respeito à quantidade de tempo que se consegue atrasar uma mensagem. Em outras palavras, quanto mais próximo do instante de envio de uma mensagem de controle for observado uma mensagem reaproveitável melhor.

Para avaliar comparativamente o reaproveitamento de mensagens das aplicações com algoritmos de detecção da literatura, a estratégia AMA foi comparada ao algoritmo Lazy, considerado até então o mais eficiente neste item. Verificou-se que a estratégia AMA obteve uma economia de mensagens em torno de 14% para ambos algoritmos tradicionais (*Pull* e *Push*), quando comparada ao Lazy. Este êxito, basicamente deriva da ausência de confirmação para cada mensagem da aplicação enviada. Além disso, como a estratégia AMA contabiliza as mensagens da aplicação no processo receptor, e não no emissor como no Lazy, para este fluxo o canal de comunicação não necessita ser confiável. No Lazy é assumido um canal confiável (FETZER; RAYNAL; TRONEL, 2001).

Outro objetivo dos experimentos foi avaliar a influência das estratégias propostas na QoS dos detectores. Neste sentido, três métricas primárias foram analisadas: T_D, T_M e T_{MR}. A estratégia AMA praticamente não influenciou no cômputo do T_D, inclusive observou-se uma pequena melhora. Já para a estratégia ATF verificou-se um aumento no tempo de detecção. Mas tal aumento no T_D é em ordem de grandeza bem menor se considerados os ganhos logrados por esta estratégia em termos de economia de mensagens e precisão. Por exemplo, considerando a precisão dos detectores, tanto para as avaliações do T_M quanto do T_{MR}, a abordagem demonstrou resultados eficientes.

Finalmente, pode-se afirmar que o presente trabalho apresentou uma solução inovadora para o problema da explosão de mensagens. Sua contribuição permite que novos projetos para detectores de defeitos em ambientes distribuídos e de larga escala possam ser construídos. Entretanto,

esta dissertação deixa assuntos em aberto que podem ser explorados em trabalhos futuros. Por exemplo, podem ser explorados:

- a combinação da nova abordagem às outras estratégias citadas no capítulo 3;
- procurar métodos para minimizar o aumento do T_D observado com a aplicação da estratégia ATF;
- realizar novos experimentos com pelo menos três tipos de aplicação diferentes, uma vez que neste texto utilizou-se apenas um serviço de consenso;
- a exploração do uso da estratégia em domínios específicos como o de redes de sensores.

REFERÊNCIAS

AGUILERA, M. K.; CHEN, W.; TOUEG, S. **On the Weakest Failure Detector for Quiescent Reliable Communication**. Cornell University, Ithaca, NY, USA: [s.n.], 1997.

AVIZIENIS, A.; LAPRIE, J.; RANDELL, B. Fundamental concepts of dependability. In: ISW 2000. 34TH INFORMATION SURVIVABILITY WORKSHOP, 2000. **Proceedings...** [S.l.: s.n.], 2000. p.7–12.

BERTIER, M.; MARIN, O.; SENS, P. Implementation and Performance Evaluation of an Adaptable Failure Detector. In: DSN, 2002. **Anais...** [S.l.: s.n.], 2002. p.354–363.

BERTIER, M.; MARIN, O.; SENS, P. Performance Analysis of a Hierarchical Failure Detector. In: DSN, 2003. **Anais...** [S.l.: s.n.], 2003. p.635–644.

BURNS, M. W.; GEORGE, A. D.; WALLACE, B. A. Simulative performance analysis of gossip failure detection for scalable distributed systems. **Cluster Computing**, [S.l.], v.2, n.3, p.207–217, 1999.

CAMARGOS, L. J. **DisCusS**: desenvolvendo um serviço de consenso genérico, simples e modular. 2003. Dissertação (Mestrado em Ciência da Computação) — Instituto de Computação, Universidade Estadual de Campinas, Campinas.

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

CHEN, W.; TOUEG, S.; AGUILERA, M. K. On the Quality of Service of Failure Detectors. In: DSN '00: PROCEEDINGS OF THE 2000 INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS (FORMERLY FTCS-30 AND DCCA-8), 2000, Washington, DC, USA. **Anais...** IEEE Computer Society, 2000. p.191.

COSQUER, F.; RODRIGUES, L.; VERÍSSIMO, P. Using Tailored Failure Suspectors to Support Distributed Cooperative Applications. In: IASTED/ISMM INTERNATIONAL CONFERENCE ON PARALLEL AND DISTRIBUTED COMPUTING AND SYSTEMS, 7., 1995, Washington (DC), USA. **Proceedings...** [S.l.: s.n.], 1995. p.352–356.

CRISTIAN, F.; BEIJER, R. de; MISHRA, S. A performance comparison of asynchronous atomic broadcast protocols. **Distributed Systems Engineering**, [S.l.], v.1, n.4, p.177–201, 1994.

DÉFAGO, X.; SCHIPER, A.; URBÁN, P. Total order broadcast and multicast algorithms: taxonomy and survey. **ACM Comput. Surv.**, New York, NY, USA, v.36, n.4, p.372–421, Dec 2004.

DWORK, C.; LYNCH, N.; STOCKMEYER, L. Consensus in the presence of partial synchrony. **Journal of the ACM**, New York, NY, USA, v.35, n.2, p.288–323, Apr 1988.

ESTEFANEL, L. A. **Avaliação dos Detectores de Defeitos e sua Influência nas Operações de Consenso**. 2001. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

FELBER, P.; DÉFAGO, X.; GUERRAOUI, R.; OSER, P. Failure Detectors as First Class Objects. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS (DOA'99), 1999, Washington, DC, USA. **Proceedings...** IEEE Computer Society, 1999. p.132–141.

FELBER, P.; GUERRAOUI, R.; SCHIPER, A. The Implementation of a CORBA Object Group Service. **Theory and Practice of Object Systems**, [S.l.], v.4, n.2, p.93–105, 1998.

FETZER, C. Enforcing Perfect Failure Detection. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS, 21., 2001, Phoenix, AZ. **Proceedings...** [S.l.: s.n.], 2001.

FETZER, C.; RAYNAL, M.; TRONEL, F. An Adaptive Failure Detection Protocol. In: PRDC '01: PROCEEDINGS OF THE 2001 PACIFIC RIM INTERNATIONAL SYMPOSIUM ON DEPENDABLE COMPUTING, 2001, Washington, DC, USA. **Anais...** IEEE Computer Society, 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, NY, USA, v.32, n.2, p.374–382, 1985.

FRIEDMAN, R.; RAYNAL, M. On the Benefits of the Functional Modular Approach in Distributed Data Management Systems. In: IEEE WORKSHOP ON DEPENDABLE DISTRIBUTED DATA MANAGEMENT, 2004, Florianópolis, BR. **Proceedings...** IEEE Computer Society, 2004. p.1–6.

GAMMA, E.; HELM, R.; JOHNSON, R. **Design Patterns. Elements of Reusable Object-Oriented Software**. [S.l.]: Addison-Wesley, 1995. (Addison-Wesley Professional Computing Series).

GARTNER, F. C. Fundamentals of fault-tolerant distributed computing in asynchronous environments. **ACM Comput. Surveys**, New York, NY, USA, v.31, n.1, p.1–26, 1999.

GROUP, N. W. **RFC 2988**: computing tcp's retransmission. http://www.rfc-editor.org/rfc/rfc2988.txt - último acesso em janeiro 2006.

HAYASHIBARA, N. **Accrual Failure Detectors**. 2004. Tese (Doutorado em Ciência da Computação) — Japan Advanced Institute of Science and Technology, Ishikawa, Japan. http://takilab.k.dendai.ac.jp/ haya/research/papers/dissertation/dissertation.pdf.gz - último acesso em junho 2005.

HAYASHIBARA, N.; SHERIF, A.; KATAYAMA, T. Failure Detectors for Large-Scale Distributed Systems. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS (SRDS-21), THE INTERNATIONAL WORKSHOP ON SELF-REPAIRING AND SELF-CONFIGURABLE DISTRIBUTED SYSTEMS (RCDS'2002), 21., 2002, Ōsaka, Japan. **Proceedings...** IEEE Computer Society, 2002. p.404–409.

HAYASHIBARA, N.; URBÁN, P.; SCHIPER, A.; KATAYAMA, T. Performance comparison between the Paxos and Chandra-Toueg consensus algorithms. In: INT'L ARAB CONF. ON INFORMATION TECHNOLOGY (ACIT), 2002. **Proceedings...** [S.l.: s.n.], 2002. p.526–533.

JAIN, R. **The Art of Computer Systems Performance Analysis**: techniques for experimental design, measurement, simulation, and modeling. 1st.ed. [S.l.]: John Wiley and Sons, INC, 1991. 685p.

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

LARREA, M.; ARÉVALO, S.; FERNÁNDEZ, A. Efficient Algorithms to Implement Unreliable Failure Detectors in Partially Synchronous Systems. In: INTERNATIONAL SYMPOSIUM ON DISTRIBUTED COMPUTING, 13., 1999, London, UK. **Proceedings...** Springer-Verlag, 1999. p.34–48.

LARREA, M.; FERNÁNDEZ, A.; ARÉVALO, S. Optimal Implementation of the Weakest Failure Detector for Solving Consensus. In: IEEE SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS (SRDS2000), 19., 2000, Nürnberg, Germany. **Proceedings...** IEEE Computer Society Press, 2000.

NBR-13596. **Tecnologia de Informação**: avaliação de produto desoftware ? características de qualidade e diretrizes para o seu uso - características de qualidade e diretrizes para o seu uso - nbr 13596/96. [S.l.]: Editora ABNT, 1996. (Rio de Janeiro, RJ).

NUNES, R. C. **Adaptação Dinâmica do Timeout de Detectores de Defeitos Através do Uso de séries Temporais**. 2003. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

NUNES, R. C.; JANSCH-PÔRTO, I. A Lightweight Interface to Predict Communication Delays Using Time Series. In: LADC, 2003. **Anais...** Lecture Notes in Computer Science, 2003. p.254–263.

NUNES, R. C.; JANSCH-PÔRTO, I. QoS of Timeout-Based Self-Tuned Failure Detectors: the effects of the communication delay predictor and the safety margin. In: DSN, 2004. **Anais...** IEEE Computer Society, 2004. p.753–761.

PETERSON, L.; BAVIER, A.; FIUCZYNSKI, M.; MUIR, S.; ROSCOE, T. **Towards a Comprehensive PlanetLab Architecture**. [S.l.]: PlanetLab Consortium, 2005. (PDN–05–030).

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

REZENDE, D. A.; ABREU, A. F. d. **Tecnologia da Informação Aplicada a Sistemas de Informação Empresariais**: o papel estratégico da informação e dos sistemas de informação nas empresas. [S.l.]: Editora Atlas, 2000. (São Paulo, SP).

SANTOS SÁ, A. dos; ARAÚJO MACÊDO, R. J. de. An Adaptive Failure Detection Approach for Real-Time Distributed Control Systems Over Shared Ethernet. In: INTERNATIONAL CONGRESS OF MECHANICAL ENGINEERING (COBEM2005), 18., 2005, Ouro Preto, BR. **Proceedings...** [S.l.: s.n.], 2005.

SERGENT, N.; DÉFAGO, X.; SCHIPER, A. Impact of a Failure Detection Mechanism on the Performance of Consensus. In: IEEE PACIFIC RIM SYMP. ON DEPENDABLE COMPUTING (PRDC), 2001, Seoul, Korea. **Proceedings...** [S.l.: s.n.], 2001.

TURCHETTI, R.; NUNES, R. C. Uma Nova Abordagem Para Redução de Mensagens de Controle em Detectores de Defeitos. In: LADC. 3TH WORKSHOP ON THESES AND DISSERTATIONS, 2005, Salvador-Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.49–54.

URBÁN, P. **Evaluating the Performance of Distributed Agreement Algorithms**: tools, methodology and case studies. 2003. Tese (Doutorado em Ciência da Computação) — École Polytechnique Fédérale de Lausanne, Switzerland. http://ddsg.jaist.ac.jp/en/pub/Urb03.html - último acesso em agosto 2005.

URBÁN, P.; DÉFAGO, X.; SCHIPER, A. Neko: a single platform to simulate and protype distributed al gorithms. **Journal of Information Science and Engineering**, [S.l.], v.17, n.6, Nov. 2002.

VOGELS, W.; RE, C. WS-Membership - Failure Management in a Web-Services World. In: WORLD WIDE WEB CONFERENCE SERIES, 2003, Budapest, Hungary. **Anais...** [S.l.: s.n.], 2003.

WEBER, T. S. **Tolerância a Falhas**: conceitos e exemplos. www.inf.ufrgs.br/ taisy/disciplinas/textos/ConceitosDependabilidade.PDF - último acesso em dezembro 2005.

APÊNDICE A ARQUIVO DE CONFIGURAÇÃO

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% hosts monitorados

% sintaxe: <host> <porta>

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%planetlab1.pop-ce.rnp.br

200.129.0.161 2020

%planetlab1.pop-rs.rnp.br

200.132.0.69 2020

%planetlab1.inria.fr

138.96.250.222 2020

%planetlab1.iii.u-tokyo.ac.jp

133.11.240.56 2020

%planetlab1.iis.sinica.edu.tw

140.109.17.180 2020

%pl1.csl.utoronto.ca

142.150.238.12 2020

%planetlab01.cs.washington.edu

128.208.4.197 2020

%planetlab-1.stanford.edu

171.64.64.216 2020

%bubbles 2020

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% host monitor

%%%%%%%%%%%%%%%%%%%%%%%%%%%%

```
Receive_Port: 2020
% Failure Detector Type = PULL, PUSH, ...
FD_Type: PULL
% Predictor Models = OFF, LAST, MEAN, WINMEAN, SMM, BROWN, ARIMA or LPF
Predictor_Type: OFF
% Consensus Type = OFF or CT
Consensus_Type: CT
```