Detectores de Defeitos Não Confiáveis

Luiz Angelo Barchet-Estefanel

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO
CURSO DE MESTRADO EM CIÊNCIA DA COMPUTAÇÃO

Detectores de Defeitos Não Confiáveis

por

LUIZ ANGELO BARCHET ESTEFANEL
T.I. 880 - PPGC - UFRGS

Trabalho Individual I

Profª. Ingrid Eleonora Schreiber Jansch Pôrto
Orientadora

Porto Alegre, janeiro de 2000

Sumário

Lista de Abreviaturas

I/O	*Input/Output* (entrada/saída)
CG	Comunicação de Grupo
CORBA	*Common Object Request Broker Architecture*
FD	*Failure Detector* (Detector de Defeitos)
IP	*Internet Protocol*
MIB	*Management Information Base*
OGS	*Object Group Service*
ORB	*Object Request Broker*
TCP	*Transport Control Protocol*
UDP	*User Datagram Protocol*
UFRGS	Universidade Federal do Rio Grande do Sul

Lista de Figuras

Lista de Tabelas

Resumo

O protocolo de consenso, largamente utilizado para a coordenação de sistemas tolerantes a falhas, sofre uma grave restrição quando implementado em ambientes assíncronos, pois segundo a Impossibilidade FLP, este não pode ser resolvido de forma determinística em tais ambientes, devido à impossibilidade de distingüir processos falhos de processos lentos. Os detectores de defeitos constituem-se em uma das formas de contornar tal problema, pois embora imprecisos, aumentam a quantidade de informações sobre o sistema, auxiliando a realização do consenso e de outros protocolos que apresentam o mesmo problema. Este trabalho faz uma revisão sobre os detectores, demonstrando seus princípios e algumas propostas para sua implementação, sob diversos modelos de falhas nos sistemas. Além disso, este trabalho faz uma avaliação dos aspectos práticos dos detectores pouco presentes na bibliografia como, por exemplo, a granularidade da detecção e a interação com outros protocolos. Também é implementado um protótipo de uma das propostas estudadas, utilizando a linguagem Java, com o objetivo de verificar seu comportamento durante o funcionamento e avaliar quais são os principais pontos a serem melhorados, aumentando o desempenho e a transparência da sua utilização.

Palavras-chave: Tolerância a Falhas; Sistemas Distribuídos Assíncronos; Consenso; Detectores de Defeitos; Impossibilidade FLP.

Abstract

The consensus protocol, widely used to the coordination of fault-tolerant systems, have a big restriction when used in asynchronous environments. The impossibility of distributed consensus in asynchronous systems shown in 1985, proved that the consensus can not be solved in a deterministic way in such environments, due to the impossibility of distinguishing faulty processes from slow ones. The failure detectors represent a way to circumvent this problem, and even being inexact, they augment the information's amount on the system, assisting the accomplishment of the consensus and other protocols with the same problem. This work reviews the literature on detectors, presenting their principles and some proposals to their implementation, under different models of faults in the system. Moreover, this work makes an evaluation on the practical aspects from the detectors mainly ignored by the bibliography, as for example, the detection's granularity and the interaction with other protocols. A prototype of one of the studied proposals was also implemented, using the Java language, objectifing to verify its functional behavior. We also envisage to evaluate which are the main points to be improved, to increase its performance and transparency of usage.

Key-words: *Fault Tolerance; Asynchronous Distributed System; Consensus; Failure Detector; FLP Impossibility.*

1. INTRODUÇÃO

O desenvolvimento e a coordenação de aplicações distribuídas são consideradas tarefas de grande complexidade, especialmente quando se considera a possibilidade de falhas no sistema. Um dos mais difundidos e importantes modelos de acordo para o uso em sistemas distribuídos tolerantes a falhas é o consenso, que possibilita aos processos atingirem uma decisão comum, apesar da ocorrência de falhas. O uso do consenso torna mais simples a solução de problemas comuns em uma situação prática, tais como a definição de um novo líder ou um acordo sobre um resultado, essenciais aos sistemas distribuídos. Além disso, as operações de consenso são essenciais ao paradigma de comunicação de grupo, uma das mais expressivas formas de controle e cooperação em sistemas distribuídos, e que é muito utilizado para prover as abstrações necessárias ao desenvolvimento de sistemas tolerantes a falhas.

Por ser um dos mais simples protocolos de acordo, o consenso, mesmo implicitamente, é utilizado para a construção de operações mais complexas, como a difusão atômica e a eleição. Isso representa um grande problema, pois um consenso não pode ser solucionado deterministicamente, em sistemas assíncronos sujeitos a uma única falha de c*rash* [FIS 85]. Essencialmente, esta impossibilidade resulta da dificuldade em determinar se um processo participante de um consenso está realmente falho ou se é apenas mais lento que os demais. Uma vez que um sistema assíncrono não estabelece limites de ordem temporal, não há um limite máximo que possa ser usado para obter tais informações.

Felizmente, a impossibilidade da realização do consenso pode ser contornada. Diversos estudos propõem técnicas que funcionam em certos casos, empregando modelos de sistemas com sincronismo parcial ou assincronismo temporizado. Tais soluções, entretanto, não são suficientemente abrangentes para facilitarem a utilização de técnicas de acordo como o consenso entre processos, quando se considera o uso de sistemas mais heterogêneos.

Este trabalho estuda uma outra técnica para a solução do problema do consenso em sistemas distribuídos assíncronos, proposta por Chandra e Toueg [CHA96a], que envolve a colaboração entre os protocolos de acordo com um mecanismo de detecção de defeitos. Basicamente, um detector de defeitos é um módulo local a cada máquina, que monitora através de troca de mensagens um grupo ou sub-grupo de processos no sistema, mantendo uma lista de processos considerados suspeitos. O detector é considerado não confiável pois pode suspeitar erroneamente de um ou mais processos no sistema, sem no entanto prejudicar a segurança das operações. Os mecanismos detectores de defeitos foram descritos através de duas propriedades, *completeness* (abrangência) e *accuracy* (precisão), e ao variar os níveis de exigências para tais propriedades, obtêm-se versões de detectores de defeitos com diferentes flexibilidades e complexidades.

Neste trabalho são apresentadas diversas propostas de detectores de defeitos, baseadas na proposta de Chandra e Toueg, e que foram estendidas pelos seus autores para operar em sistemas distribuídos com características e modelos de falhas diversos. Através deste estudo pretende-se observar suas características comuns, esclarecendo algumas dúvidas que envolvem o funcionamento dos detectores e a sua integração ao consenso. Para aprimorar tal avaliação também será implementado um protótipo escolhido entre os modelos estudados, e com ele pretende-se verificar certos aspectos práticos que não costumam ser tratados na bibliografia.

A divisão deste trabalho segue a seguinte estrutura: o capítulo 2 será composto de uma introdução à comunicação de grupo em sistemas distribuídos assíncronos, apresentando a importância das operações de consenso para a coordenação das atividades e as dificuldades para obter o consenso em sistemas assíncronos.

O capítulo 3 apresentará a proposta de Chandra e Toueg para a realização do consenso em sistemas assíncronos, através do uso de detectores de defeitos não confiáveis, e além da descrição das suas propriedades básicas, serão comentados aspectos importantes quanto à precisão e implementabilidade das possíveis variações.

O capítulo 4 irá apresentar as abordagens de implementação estudadas. Como a proposta dos detectores de Chandra e Toueg não restringe a forma ou a lógica da implementação, existem diversas maneira de construir um detector de defeitos. Neste trabalho estão reunidos alguns exemplos de detectores, de forma a abranger diversos modelos de falhas.

O capítulo 5 traz uma análise sobre o papel de um detector de defeitos no momento da integração a outros protocolos e serviços que dão suporte à comunicação de grupo, em especial o protocolo de consenso. Este capítulo faz uma revisão sobre as propostas do capítulo 4, questionando os aspectos práticos dos detectores, e apresentando considerações sobre aspectos práticos que são pouco esclarecidas nas publicações.

O capítulo 6 apresenta um protótipo de detector de defeitos, desenvolvido para validar e testar o modelo dos detectores de defeitos. Neste capítulo, são apresentadas a estrutura do protótipo, os motivos que levaram à escolha do detector *Heartbeat* como a proposta utilizada, as dificuldades encontradas no momento de traduzir as especificações da proposta escolhida em um sistema com as características desejadas e a avaliação do detector implementado.

Finalmente, as conclusões obtidas a partir deste estudo estão reunidas no capítulo 7.

2. PRINCÍPIOS E DESAFIOS

2.1 Tolerância a Falhas em Sistemas Distribuídos

Cada vez mais a sociedade torna-se dependente de sistemas computacionais, o que leva ao questionamento sobre os limites de sua confiabilidade, e à identificação dos aspectos a serem reforçados para aumentar as garantias de tais sistemas [LAP 98]. Sistemas bancários, telefonia, fornecimento de energia elétrica e muitos outros serviços essenciais ao nosso quotidiano necessitam oferecer um alto grau de *dependability* (dependabilidade, ou segurança de funcionamento).

De fato, de acordo com as necessidades dos sistemas podem ser explorados diversos aspectos da *dependability* [LAP 98]:

- ***Availability*** **(disponibilidade)** - define a prontidão do sistema para o uso.
- ***Reliability*** **(confiabilidade)** - define a continuidade do oferecimento de um serviço.
- ***Safety*** **(segurança)** - define os esforços para que não ocorram conseqüências catastróficas no ambiente, decorrentes de alguma falha no sistema.
- ***Confidentiality*** **(sigilo)** - define a proteção de informações com relação aos acessos não autorizados.
- ***Integrity*** **(integridade)** - define a proteção de informações com relação à alterações impróprias.
- ***Maintenability*** **(manutenção)** - define as técnicas de manutenção e atualização do sistema.

O uso de técnicas de *dependability* permite aumentar a confiança do usuário em um sistema além daquela que seria possível usando apenas métodos tradicionais de projeto e desenvolvimento. A tolerância a falhas, assim como a prevenção, a remoção e a antecipação de falhas são conjuntos de técnicas utilizadas para prover a *dependability* em todos os aspectos do funcionamento de um sistema. O objetivo da tolerância a falhas é assegurar o funcionamento adequado de um sistema mesmo na presença de falhas.

As técnicas de tolerância a falhas utilizadas em um sistema podem ser implementadas em *hardware* ou *software.* Uma técnica implementada em *hardware* costuma utilizar dispositivos tais como discos, barramentos, placas controladoras, especialmente preparados para suportar falhas (e provavelmente utilizados de forma replicada). Essas soluções, entretanto, podem não ser suficientes. De acordo com as exigências da aplicação e do projeto do sistema, pode ser preciso garantir características que ultrapassem as possibilidades do *hardware* utilizado, tornando-se necessário adicionar camadas de *software* tolerante a falhas. Em especial, nos casos onde são exigidos distribuição geográfica e/ou processamento distribuído, ou onde o custo de uma replicação passiva em *hardware* torna-se proibitivo, pode-se dispor de um sistema distribuído adicionado de técnicas de tolerância a falhas.

Fisicamente, um sistema distribuído consiste de sistemas computacionais autônomos, conectados através de uma rede de comunicação, mas que não é necessariamente percebida por essa camada. Essas características levam a diversas considerações: sistemas autônomos conectados por

uma rede provavelmente não podem trocar informações através de memória compartilhada, assim como estabelecer um relógio global para todas as máquinas; a transparência necessária às camadas superiores (aplicação e usuário) deve ser provida pelo sistema operacional distribuído, que estabelece protocolos e serviços capazes de realizar as tarefas em um ambiente distribuído da mesma forma como se executasse apenas em uma única máquina; essa transparência também leva o sistema distribuído à necessidade de suportar falhas, as quais, dependendo das especificações, podem ser nos processos, na rede de comunicação ou em ambos simultaneamente. A forma como se apresentam as falhas (ou a forma como são identificadas) é conhecida como seu modelo de falhas. Tais modelos de falhas de processos foram descritos por Cristian (*apud* [JAL 94]) como:

- ***Crash* (colapso)** - a falha faz o componente suspender sua operação ou perder seu estado interno. Quando isso ocorre, o componente cessa totalmente sua atividade, não executando nenhuma atividade de forma incorreta.

- **Falhas por omissão** - são falhas que fazem com que um componente não responda a certos tipos de entradas.

- **Falhas de temporização** - quando um componente responde a uma requisição muito cedo ou muito tarde.

- **Falhas bizantinas** - quando um componente comporta-se de maneira totalmente arbitrária, durante a falha.

De acordo com as características de cada modelo de falhas, é estabelecida uma hierarquia, representada na Fig. 2.1. Há uma relação entre as classes mais internas e as externas de forma que, sob o ponto de vista do projeto, escolhe-se qual tratamento é necessário.

Jalote acrescentou ao modelo apresentado por Cristian uma outra classe de falhas, caracterizadas quando um processo obtém resultados incorretos a partir de entradas corretas, sendo assim chamada de falha por computação incorreta. As falhas por computação incorreta podem ser consideradas como um sub-tipo das falhas bizantinas, embora não mantêm relação com os demais modelos de falhas contidas nas falhas de temporização (o processo que falhou não entrou em *crash*, não foi omisso, etc.). A fig. 2.1 tenta representar tais modelos de falhas, estendendo a representação de Cristian.

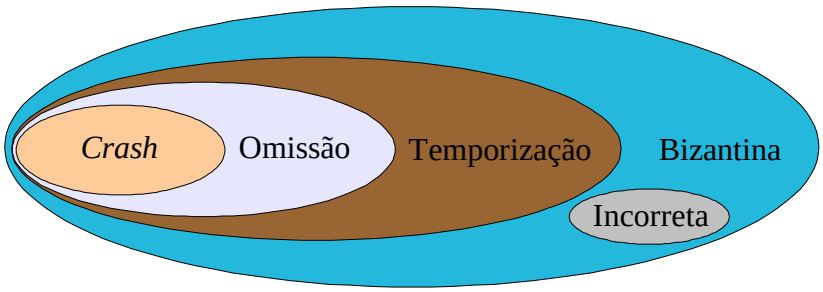

FIGURA 2.1 - Classes de falhas

Uma rede de comunicação também pode apresentar diferentes tipos de falhas. Segundo Jalote [JAL 94], essas falhas podem ser de *crash* (não há envio nem recepção de mensagens), computação incorreta (pelo corrompimento do conteúdo das mensagens), falhas por omissão (perda de mensagens), falhas de temporização (atraso na entrega) ou bizantinas, por registrar um comportamento imprevisível.

Uma aplicação distribuída (ou um sistema distribuído, do ponto de vista da aplicação) consiste de um conjunto de processos executando concorrentemente, mas colaborando entre si para

a realização de uma tarefa. Os processos cooperam através da troca de informações, seja por compartilhamento de memória ou troca de mensagens. No caso dos sistemas distribuídos, usualmente apenas a troca de mensagens é tolerada. Conceitualmente, costuma-se associar compartilhamento de memória física com processamento paralelo ou multiprocessamento, cujos objetivos podem diferenciar dos de um sistema distribuído.

Schroeder [SCH 93] estende a definição de um sistema distribuído, ao defini-lo a partir de características e desafios. As características primárias que um sistema distribuído precisa seguir são:

- **Múltiplos computadores** - um sistema distribuído contém mais de um computador "físico" com todos atributos necessários a um funcionamento autônomo.

- **Interconexões** - através de conexões de entrada e saída (*I/O*), um computador pode (e deve) comunicar-se com outras máquinas.

- **Estado compartilhado** - os computadores cooperam para manter um estado compartilhado. Para isso, deve haver uma coordenação entre eles a fim de preservar esse estado.

Ao construir um sistema com computadores interconectados, existem quatro desafios principais a serem tratados:

- **Defeitos independentes** - devido à presença de computadores distintos, quando um apresenta uma falha, os demais devem continuar funcionando. Freqüentemente deseja-se que o sistema continue operacional mesmo com falhas isoladas de alguns componentes.

- **Comunicação não-confiável** - como, na maioria dos casos, a interconexão entre os computadores não é delimitada a um ambiente rigidamente controlado, essa interconexão pode não funcionar sempre. A conexão pode ser rompida, mensagens podem ser perdidas ou alteradas, de forma que um computador não pode contar com a possibilidade de comunicar-se claramente com outro, se ambos estiverem operando.

- **Comunicação insegura** - as mensagens transmitidas entre os computadores podem ser alvo de escuta ou modificação não autorizada.

- **Custo da comunicação** - a intercomunicação entre computadores normalmente apresenta uma largura de banda reduzida e grande latência, comparado aos disponíveis para a comunicação entre processos em uma única máquina.

Como apresentado na definição acima, freqüentemente não é possível controlar a forma (e principalmente a velocidade) com que as operações e transmissões serão realizadas. Assim, quando um sistema ou protocolo é projetado, é comum estabelecer se o sistema será síncrono ou assíncrono. Em um sistema assíncrono, não é feita nenhuma exigência sobre características tais como: velocidade de execução de um processo, atraso da transmissão pelas redes de comunicação, atrasos na entrega para as aplicações, etc. Ou seja, um sistema assíncrono é considerado como não tendo nenhuma determinação de natureza temporal. Como não existem limites pré-determinados, as técnicas tradicionais de detecção de defeitos em um sistema tornam-se não-determinísticas, e o máximo que um processo pode fazer é suspeitar da falha em outro processo, sem certeza absoluta (pode ser que o processo suspeito esteja funcionando normalmente). Já em um sistema síncrono,

por oposição, tais parâmetros são considerados. São especialmente observados, em um sistema síncrono, os limites para as velocidades relativas dos processos e para os atrasos associados com os canais de comunicação; para um processo ser considerado falho pelos demais, basta não respeitar esses limites de tempo. Mishra (*apud* JAL 94]) define um canal de comunicação síncrono como um canal de comunicação onde o atraso máximo de transmissão de uma mensagem é conhecido e limitado; um processador síncrono é um processador onde o tempo de execução de uma instrução é finito e limitado.

Na realidade, considerar um sistema como assíncrono é não fazer exigências de ordem temporal, uma vez que todo sistema é inerentemente assíncrono. A definição de limites sobre um sistema inerentemente assíncrono, baseado no conhecimento específico dos componentes do sistema, é que possibilita atender a um sistema síncrono. Essa "especialização" que um sistema assíncrono faz sobre um sistema síncrono é que determina tanto a complexidade dos sistemas de suporte de uma aplicação distribuída, quanto a portabilidade desses códigos. Um protocolo desenvolvido para um sistema síncrono provavelmente vai utilizar as características deste sistema específico, realizando as operações de uma forma eficiente e otimizada. Entretanto, este protocolo estará restrito a um sistema com características peculiares e, ao contrário de um protocolo assíncrono, não poderá ser utilizado em qualquer sistema distribuído.

Um sistema síncrono apresenta ainda outras possíveis desvantagens. Em teoria, todos componentes utilizados na construção devem ter suas características e atrasos estabelecidos, o que possibilita construir um sistema síncrono. Mas caso haja saturação no sistema, e um ou mais elementos sofrerem degradação em seu desempenho, essa degradação propagar-se-á para todo sistema, podendo gerar situações de falhas caso seja ultrapassada a diferença relativa entre o desempenho dos componentes para o qual o sistema foi desenvolvido.

2.2 Comunicação de Grupo

Controlar todos os aspectos relativos à tolerância a falhas em um único processo já é uma atividade complexa, que tende a aumentar quando se implementa um sistema distribuído. Processos rodando concorrentemente, ausência de relógio global, imprevisibilidade do meio de comunicação e da velocidade de execução dos processos, acabam fazendo com que esse desenvolvimento fique bastante prejudicado, se forem utilizados os métodos tradicionais de projeto e controle das atividades dos processos.

Torna-se necessário, então, utilizar uma metodologia ou ferramenta (alguns autores têm referido esta metodologia através do termo paradigma, mas Nunes [NUN 98] discute o seu uso) que abstraia grande parte das estruturas comuns ao desenvolvimento de sistemas distribuídos, facilitando a tarefa do desenvolvedor. Uma das necessidades mais imediatas para este desenvolvedor é controlar eficientemente e de forma confiável todos os processos do sistema, e uma boa alternativa de fazer isso é através da programação orientada a grupos.

O conceito de grupos de processos associa-se a um conjunto de processos agrupados para cooperativamente fornecer um serviço [NUN 98]. O impacto inicial desse conceito é o de permitir aos projetistas e desenvolvedores a realização de aplicações de forma mais regular e em um nível de abstração mais alto. Birman (*apud* [NUN 98]) também expôs a importância do conceito da programação orientada a grupos para a disponibilização de outros serviços, como a manipulação de dados replicados, técnicas de tolerância a falhas ou balanceamento de carga.

Esta metodologia de trabalho também vai ao encontro das necessidades dos

desenvolvedores de sistemas distribuídos tolerantes a falhas, onde a associação entre a programação orientada a grupos e o suporte à *multicast* confiável originam a chamada Comunicação de Grupo (CG).

Powell [POW 96] faz uma analogia entre um grupo de pessoas e um sistema distribuído com comunicação de grupo. Ele descreve um grupo de pessoas em uma reunião, apresentando as vantagens e desvantagens de uma abordagem síncrona ou assíncrona:

> "Imagine que um comitê está se reunindo, e os membros deste comitê estão sentados ao redor de uma mesa, em uma sala de reuniões. É uma reunião longa, de forma que os membros ficam muito cansados e alguns deles chegam até a cochilar. Entretanto, quando acordam, eles podem facilmente verificar quem mais está acordado, pois todos estão sentados ao redor da mesa. Com um pouco de organização (um protocolo) todos membros podem saber quem mais está acordado, o que escutaram, e qual sua percepção do contexto.
>
> Com essa analogia, podem ser obtidos alguns pontos característicos dessa reunião:
>
> - existem ao menos três grupos de pessoas a serem considerados. Os membros convocados para a reunião, os membros presentes na reunião e os membros que participaram dos trabalhos em um dado instante, porque estavam acordados;
>
> - a sala de reuniões é análoga a um sistema síncrono, onde a comunicação é confiável e bem delimitada no tempo, e é fácil para as pessoas (processos) detectar quais membros dormiram (falharam). Então, é possível fazer uma forte definição sobre quem está acordado (os atuais integrantes do *membership*), o que eles escutaram (as mensagens entregues), e o que isto significou para eles (mudanças no estado interno resultantes da ordem de entrega das mensagens);
>
> - há a necessidade de planejar como novos membros (pessoas que se juntam à reunião após o seu início) e membros que se recuperam (pessoas que dormiam e depois acordaram) serão informados sobre o que foi decidido enquanto estavam ausentes ou dormindo.
>
> Agora será considerada outra maneira de realizar a reunião do comitê. Supondo que, ao invés de se reunir ao redor de uma mesa, em uma sala silenciosa, o comitê tenta conduzir suas decisões em um grande e movimentado *hall* de um hotel. Devido à movimentação e ao barulho do local, um membro nem sempre pode ver ou falar diretamente com um outro membro. Mesmo quando um membro pode ver um outro, não é certo que aquele está prestando atenção no primeiro, e alguns dos membros podem pegar no sono ou ir embora para casa sem que os outros saibam. É óbvio que desta maneira, o comitê terá muito mais dificuldade para realizar suas tarefas de forma consistente. Pode-se supor que os membros tentem reunir-se para realizar algum trabalho, mas para fazer isso eles precisam alcançar algum tipo de acordo sobre o que todos eles pensam sobre um determinado assunto (por exemplo, quem vai atuar como condutor da reunião).
>
> Como as pessoas entram e saem da vista dos demais, eles têm que sucessivamente fazer novas decisões sobre quem está no grupo de discussão. Além disso, podem haver diferentes grupos de discussão em diferentes partes do *hall* do hotel ao mesmo tempo.

Neste caso, os diferentes grupos de membros podem realizar decisões contraditórias. O único modo para evitar tais conflitos é impor uma regra onde, por exemplo, somente grupos com a maioria dos membros têm permissão para tomar decisões. Outra alternativa é a de que o comitê pode tentar reconciliar as decisões conflitantes sempre que dois ou mais grupos puderem se unir e formar um novo grupo.

Este cenário de reunião é análogo ao sistema distribuído assíncrono, e apresenta os seguintes pontos:

- como as pessoas (processos) não podem detectar com certeza quando outras estão ausentes, ou similarmente, estão dormindo (falhos), não se pode decidir exatamente quem está presente à reunião;

- é impossível evitar que os membros se dividam em diferentes sub-reuniões (agrupamentos). Tais grupos de participantes são às vezes chamados "visões" da reunião, desde que os membros considerem que seu grupo é a reunião. Esta partição lógica de reuniões é praticamente inevitável, então os protocolos de grupos assíncronos devem estar aptos a lidar com essas situações;

- quando um membro se junta a um grupo de discussões existente (criando dessa forma um novo grupo, maior), os participantes devem estabelecer o que este novo integrante sabe sobre o trabalho desenvolvido, e atualizá-lo. Esta contextualização tem que ser feita tanto se o novo integrante acordou do cochilo (recuperou-se de falha) quanto se ele perdera contato com seu grupo anterior (desconectou-se). Em um sistema assíncrono, é difícil dizer a diferença. Se os integrantes não desejam ter que constantemente informar aos novos integrantes tudo que foi feito desde o início das atividades, estes novos membros deverão lembrar (manter em armazenamento estável) o que eles fizeram em grupos anteriores, e o protocolo precisa assegurar que todo trabalho realizado nos diversos agrupamentos é feito de uma forma consistente."

Como pode-se observar neste exemplo, trabalhar com sistemas assíncronos induz a um aumento de complexidade nas operações, e as operações essenciais à comunicação dentro deste grupo podem ser definidas como a comunicação confiável entre os membros, a determinação dos elementos falhos, o acordo entre os elementos sobre uma decisão, a gerência dos membros ativos e possivelmente a política de reintegração dos grupos (ou a coordenação da consistência sobre os grupos dispersos).

2.3 O Consenso

Como pode-se observar com o exemplo de sistema assíncrono da seção 2.2, uma das ações fundamentais a serem tratadas pelo sistema distribuído refere-se à forma com que os membros entram em acordo sobre uma determinada decisão. Um protocolo de consenso é uma das operações de acordo entre processos mais importantes para um sistema distribuído. Sua grande vantagem é assegurar que processos corretos decidam em um mesmo valor, apesar da existência de falhas. Além disso, diversos protocolos podem ser construídos com base no consenso, tais como a difusão atômica [GUE 97], eleição [SAB 95] e outros.

Devido à essa importância do consenso para as operações de acordo entre processos, é

essencial poder dispor desse protocolo quando está sendo desenvolvido um sistema distribuído, especialmente no desenvolvimento de uma ferramenta de comunicação de grupo.

Uma das definições mais simples de um protocolo de consenso encontra-se a seguir. Considera-se um conjunto de processos que podem falhar por *crash*. Um processo não falho é chamado de correto. Cada processo tem um valor de entrada, 0 ou 1, que é proposto para os outros processos (através da troca de uma mensagem *propose(v)*). O problema do consenso consiste em desenvolver um protocolo tal que todos processos corretos unanimamente e irrevogavelmente decidam um valor comum para a saída, que faz parte do conjunto de valores da entrada (mensagem *decide(v)*). Para isso, um protocolo de consenso tem que seguir as seguintes propriedades [RAY 96]:

- **Terminação** - todos processos corretos irão decidir um valor, em um dado momento (em Inglês, essa situação é expressa pelo termo *eventually*, que significa algo que necessariamente irá ocorrer, mas sem um tempo predeterminado).

- **Integridade** - um processo decide no máximo uma vez.

- **Acordo** - não existem dois processos corretos que decidem diferentemente.

- **Validade** - se um processo decide v, então v foi proposto por algum processo.

Ao conjunto dos possíveis valores que podem ser decididos, constituído dos valores de entrada iniciais, pode ser acrescido um valor especial, chamado NU (*no unanimity*, sem unanimidade) [HAD 93], que indica quando foi impossível chegar a uma decisão unânime, ou seja, dois ou mais processos corretos propuseram valores diferentes.

Respeitando-se essas características, um protocolo de consenso pode assumir qualquer forma, normalmente determinada de acordo com o modelo de falhas às quais o protocolo se destina tratar.

Como exemplo, aqui serão apresentadas três alternativas de implementação para um protocolo de consenso que suporta falhas por *crash*, apresentadas por Guerraoui e Schiper [GUE 97]. Os autores apresentaram formas de resolver consenso com três, duas e até mesmo uma etapa de comunicação entre os processos, em situações de funcionamento normal. O protocolo de consenso com três etapas de comunicação, apresentado na fig. 2.2, segue os seguintes passos:

1. inicialmente, um processo p_1 difunde seu valor inicial v_1 para os demais processos (p_1 propõe o valor v_1).

2. um processo aceita a proposta de p_1 ao enviar uma mensagem de *ack* para p_1.

3. uma vez que p_1 recebeu mensagens de *ack* da maioria dos processos, ele difunde uma mensagem *decide(v)*.

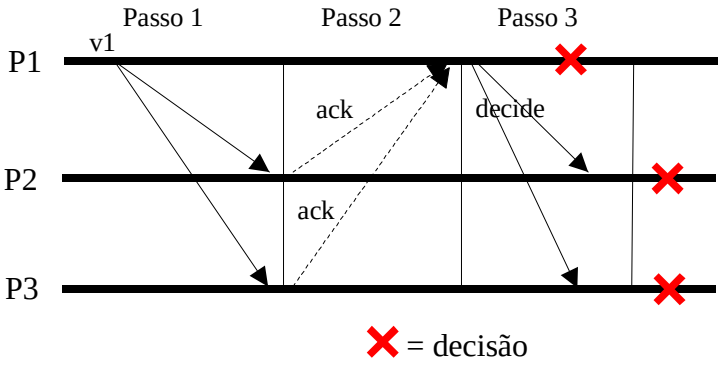

FIGURA 2.2 - Consenso em três passos

Para realizar um consenso em duas etapas, conforme a fig. 2.3, os passos podem ser os seguintes:

1. inicialmente, o processo p_1 difunde seu valor inicial v_1.

2. cada processo aceita a proposta de p_1 ao reenviar v_1 para todos os processos. Um processo decide após ter recebido v_1 da maioria dos processos.

Nota-se que neste algoritmo não é estabelecida uma garantia para a terminação da operação, como ocorre no algoritmo com três etapas de comunicação, mas sob certas condições e características do ambiente, a ausência dessa garantia pode ser aceita.

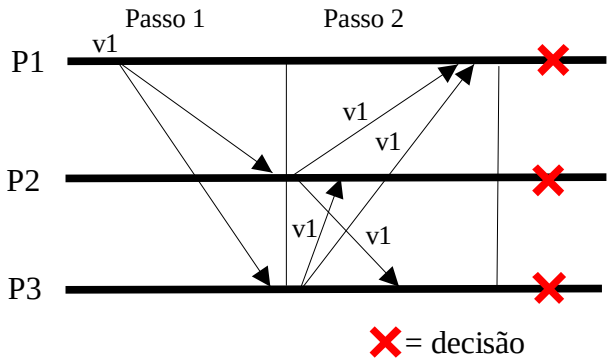

FIGURA 2.3 - Consenso em dois passos

Para determinar se é possível construir um algoritmo de consenso com apenas um único passo de comunicação, Guerraoui e Schiper analisaram as possibilidades de falhas de um algoritmo dessa natureza, e propuseram soluções sobre esses casos.

Como um processo p_1, após ter enviado sua mensagem *decide*(v_1), não pode ter certeza se os demais processos já receberam (e decidiram) sobre v_1, pode-se determinar o seguinte cenário de falha (fig. 2.4):

- o processo p_1 envia v_1 por difusão para todos os processos;

- simultaneamente, um defeito na rede separa os processos em duas partições, sendo que uma inclui os processos p_1 e p_2, e a outra inclui os processos p_3, p_4 e p_5. Por

hipótese, somente p_2 e p_1 receberam e decidiram v_1. Os processos p_3, p_4 e p_5 similarmente decidem um valor (diferente de v_1), violando a propriedade de Acordo.

Uma possibilidade de resolver esse problema é através da adoção de uma temporização, da seguinte forma:

- toda vez que um processo p_i recebeu um valor inicial v_1 de p_1, o processo p_i espera por um tempo Δ antes de decidir (Δ é o tempo necessário para detectar se ocorreu um defeito na rede). Após esse tempo Δ, se p_i não foi notificado de defeito, então p_i decide sobre v_1.

Este algoritmo resolve o problema do cenário de falhas proposto, e pode ser usado para resolver o consenso com um único passo de comunicação. Entretanto, como o intervalo de tempo Δ deve ser ajustado de tal forma a propiciar segurança na decisão e o processo tem que esperar até o fim do tempo Δ para decidir, o custo final, para situações normais de operação, pode acabar sendo mais elevado do que o custo de um consenso resolvido através de dois passos de comunicação.

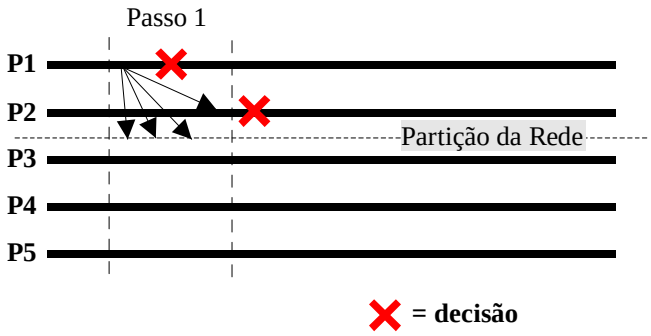

FIGURA 2.4 - Problemas com o consenso um passo

Embora as características do consenso apresentadas anteriormente sejam as mais usadas, de acordo com a bibliografia consultada, e por isso tenham sido utilizadas neste trabalho, existem certas variações, consideradas por alguns autores, e serão apresentadas a seguir.

Uma das variações do problema do consenso é chamado Consenso Uniforme [RAY 96]. Nele, além das propriedades de Terminação, Integridade e Validade presentes no consenso, é adicionada a seguinte propriedade de Acordo:

- Acordo Uniforme - Não existem dois processos que decidem diferentemente.

Essa propriedade significa que tanto um processo correto quanto um processo que decidiu um valor imediatamente antes de falhar não podem decidir diferentemente. Essa característica (uniformidade) é importante para a recuperação (*recovery*). Por exemplo, considera-se uma situação onde um processo decide usando o Acordo Uniforme e então falha. Se mais adiante esse processo for recuperado, ele irá naturalmente concordar com os outros processos corretos, sem violar a propriedade de Integridade.

Outra variação é o chamado Consenso Fraco (*Weak*) ou Não-Trivial [CHA 96a]. Nele, a propriedade Validade do consenso é substituída pela propriedade de Não-Trivialidade:

- Não-Trivialidade: há um ciclo de consenso no qual os processos corretos decidem 0,

e um ciclo onde eles decidem 1.

Ao contrário da Validade, esta propriedade não restringe explicitamente as condições sob as quais os processos devem decidir 0 ou 1, mas apenas indica que é possível que ambos valores sejam decididos, evitando que um protocolo mal projetado somente decida um único valor por toda sua duração.

2.4 O Problema do Consenso em Sistemas Assíncronos

As diversas formas de implementar um consenso, conforme exposto na seção 2.3, podem diferir quanto à flexibilidade que dão aos processos (o que provavelmente irá interferir sobre quais outros protocolos podem ser construídos sobre o consenso), mas todas têm a propriedade de resolver um acordo entre processos. Sua construção é relativamente simples, em sistemas distribuídos síncronos, mas apresenta sérias restrições quando implementado em sistemas assíncronos. Isso foi demonstrado no trabalho de Fischer, Lynch e Paterson [FIS 85], que ficou conhecido como "Impossibilidade FLP", em homenagem aos autores. Essa impossibilidade mostra que não é possível desenvolver um protocolo determinístico que solucione o problema do consenso em sistemas assíncronos, mesmo com uma única falha de *crash*. Isso se deve à impossibilidade de distingüir, com certeza, se um processo está falho ou se apenas está muito lento (semelhante ao problema da determinação dos integrantes da reunião, no exemplo da seção 2.2).

Devido ao impacto inicial da Impossibilidade FLP, pode parecer que não há possibilidade prática de realizar uma operação de consenso (assim como operações mais complexas, como eleições [SAB 95]), em sistemas assíncronos. Guerraoui e Schiper [GUE 97], assim como Birman [BIR 96] mostram que esse temor é infundado. Analisando a Impossibilidade FLP, essa apenas assegura que não há um algoritmo determinístico capaz de solucionar consenso em **todos os casos possíveis**. Com um determinado algoritmo $\mathcal{A}$, pode-se resolver, no máximo, um sub-grupo de todos os casos possíveis. Um algoritmo $\mathcal{A}$', por outro lado, pode resolver um sub-grupo diferente do que $\mathcal{A}$ resolve. Se os casos que $\mathcal{A}$ resolve contêm todos os casos que $\mathcal{A}$' resolve, então $\mathcal{A}$ é mais geral (ou resiliente) que $\mathcal{A}$'.

Dessa forma, a Impossibilidade FLP força a especificação do conjunto de casos onde o algoritmo resolve o consenso. Isso é análogo à especificação do número de falhas toleradas por um algoritmo síncrono. Assim, muitos pesquisadores trabalharam para explorar especificações semelhantes a de um sistema síncrono (número limitado de falhas), para um algoritmo de consenso. Vários modelos de sistemas foram publicados: sincronismo parcial (Dwork, *apud* [GUE 97]), assincronismo temporizado (Cristian, *apud* [GUE 97]), e assincronismo com detectores de defeitos [CHA 96a]. Tais modelos geram algoritmos que não permitem aos processos discordarem do valor de uma decisão (não violando a propriedade de segurança - *safety*).

Tanto o modelo de sincronismo parcial quanto o de assincronismo temporizado assumem um limite δ para o atraso na entrega das mensagens, não limitando o número de falhas de temporização. Para isso, certas restrições são colocadas para os casos de operação:

- o sincronismo parcial assume a existência de um tempo *t* após o qual não ocorrem mais falhas de temporização;

- o modelo de assincronismo temporizado assume que o sistema atravessa períodos "estáveis" e "instáveis". Um período "estável" ocorre quando não é registrada nenhuma falha de temporização.

O modelo assíncrono auxiliado por um detector de defeitos não assume nenhum limite de tempo para os atrasos da comunicação, mas determina os enganos cometidos ao suspeitar de uma falha. Como pode-se supor que um sistema prático apresenta um comportamento razoável na maior parte do tempo, é possível considerar um detector de defeitos $\langle\rangle\mathcal{S}$ como:

- em um sistema assíncrono auxiliado por um detector de defeitos $\langle\rangle\mathcal{S}$, há um tempo *t* após o qual um processo correto não é mais considerado suspeito por nenhum processo correto.

Dessa forma, a Impossibilidade FLP deu origem a diversos modelos que são menos restritivos que o modelo síncrono, mas que permitem a realização de um consenso, ao determinar os casos suportados pelos algoritmos.

Estes não são os únicos meios de resolver o problema do consenso em um sistema distribuído - podem ser utilizadas técnicas de randomização; entretanto essas se baseiam na escolha aleatória de números, sendo consideradas técnicas não determinísticas [RAY 96], que não estão relacionadas no escopo da impossibilidade FLP. Um exemplo de sistema misto (que usa detectores e randomização) será tratado no capítulo 3.

O modelo assíncrono com detectores de defeitos foi proposto por Chandra e Toueg. A detecção de defeitos é feita por mecanismos não confiáveis, ou seja, a suspeita de que um processo está falho pode ser incorreta. Chandra e Toueg demonstraram que esse modelo também resolve o consenso em todos os casos de sincronismo parcial [CHA 96a], de forma que pode ser considerado mais abrangente e simples que os outros modelos [GUE 97].

3. DETECTORES DE DEFEITOS

O modelo de assincronismo auxiliado por um detector de defeitos, introduzido por Chandra e Toueg [CHA 96a], possibilita a resolução de um consenso mesmo em um sistema assíncrono, oferecendo uma alternativa à Impossibilidade FLP, ao definir quais os casos específicos que os algoritmos resolvem.

Para isso, na técnica proposta por Chandra e Toueg, é considerado o uso de detectores de defeitos distribuídos, onde cada processo tem acesso a um módulo detector de defeitos local. Tal módulo monitora um subgrupo dos processos do sistema, e mantém uma lista onde são relacionados os processos suspeitos de falha. Como os módulos detectores são considerados não confiáveis, ou seja, podem suspeitar erroneamente de um processo, um módulo detector pode adicionar um processo *p* à lista de suspeitos mesmo se esse processo está funcionando. Se, em outro instante, este módulo concluir que a suspeita sobre *p* foi um engano, o processo *p* pode ser retirado da lista de suspeitos. Assim, um detector pode adicionar ou remover um processo da sua lista de suspeitos diversas vezes, de acordo com sua percepção momentânea. Além disso, como os módulos são distribuídos e cada processo tem acesso apenas ao módulo local, em um mesmo instante de tempo, dois módulos podem apresentar listas de suspeitos diferentes, de acordo com as suspeitas de cada um. Para tentar manter uma consistência entre as listas de suspeitos de diversos módulos, as implementações normalmente definem que periodicamente a lista de um processo é enviada aos demais, auxiliando na formação de uma lista unificada.

Uma exigência do modelo é que os enganos cometidos por um detector de defeitos não podem impedir que processos corretos se comportem de acordo com sua especificação, mesmo se um processo é erroneamente considerado suspeito por todos os demais processos. Isso difere do sistema implementado pela ferramenta de comunicação de grupo Isis [CHA 96a], onde um processo correto, que foi considerado suspeito pelo detector de defeitos da ferramenta, é obrigado a cometer suicídio, ou seja, ele recebe uma mensagem determinando que se enquadre na visão do sistema.

Para determinar as características de um detector de defeitos, estes foram classificados de acordo com as propriedades de *completeness* (abrangência, completeza) e *accuracy* (precisão). A propriedade *completeness* descreve as exigências para que um detector de defeitos venha a suspeitar de todos os processos que estão realmente falhos. A propriedade *accuracy* descreve as restrições quanto aos enganos que um detector pode cometer.

Chandra e Toueg definiram dois níveis de *completeness* e quatro níveis de *accuracy*. Do relacionamento entre esses níveis das propriedades, originaram-se oito classes de detectores de defeitos, e avaliaram a possibilidade de resolver o problema do consenso usando cada classe de detectores.

Ao longo deste capítulo, seguem as definições do modelo de sistema distribuído e das classes de detectores de defeitos propostas por Chandra [CHA 96a].

3.1 Definição do Modelo

É denominado sistema distribuído assíncrono um sistema onde não há limitações quanto ao atraso das mensagens, para as variações do relógio interno das máquinas, nem no tempo necessário para executar uma tarefa. O sistema consiste de um conjunto de *n* processos, $\Pi = \{p_1, p_2, ..., p_n\}$, onde cada par de processos é conectado por um canal de comunicação confiável.

Também é considerada, para simplificar a percepção do problema, a existência de um relógio global discreto, representado por $\Im$. Esse dispositivo é apenas imaginário, e os processos não têm acesso a ele.

3.2 Defeitos e Padrões de Defeitos

Os processos podem falhar por *crash* (colapso), ou seja, por suspenderem sua operação prematuramente. Um padrão de defeitos F é uma função de $\Im$ em 2^{Π}, onde F(t) indica o conjunto de processos que falharam até o tempo t. Uma vez que um processo falha, ele não se recupera, de forma que $\forall t$:F(t) $\subseteq$ F(t+1). A partir dessa definição das falhas dos processos, são também estabelecidas as funções *crashed* e *correct*, que definem o comportamento dos processos:

crashed(F) = $\cup_{t\in\Im}$ F(t)

correct(F) = Π - *crashed*(F).

Um sistema não pode funcionar se todos os processos integrantes falharem, então são considerados somente padrões de defeito F tais que ao menos um processo é correto, ou seja, *correct*(F)$\neq \emptyset$.

3.3 Detectores de Defeitos

Cada módulo detector de defeitos exibe a lista de processos que atualmente considera suspeitos de estarem falhos. Um histórico H de um detector de defeitos é uma função de $\Pi\times\Im$ em 2^{Π}. Assim, H(p,t) é o conjunto de processos suspeitos pelo detector de defeitos do processo p no instante t. Se um processo $q \in$ H(p,t), é dito que p suspeita de q no instante t, em H.

Por serem módulos distribuídos independentes, os detectores de dois processos diferentes não necessitam concordar sobre a relação dos processos que cada um suspeita, de modo que: se $p \neq q$, é possível que H(p,t)$\neq$ H(q,t).

Com base nesses conceitos, um detector de defeitos $\mathcal{D}$ é uma função que mapeia cada padrão de defeitos F para um grupo de históricos de defeitos $\mathcal{D}$(F), provendo, dessa forma, informações (possivelmente incorretas) sobre o padrão de defeitos F que ocorre na execução.

Chandra e Toueg procuraram demonstrar sua proposta genericamente (facilitando também a prova matemática de suas teorias), e para isso, ao invés de apresentar uma implementação, as diversas classes de detectores de defeitos foram descritas através das propriedades *completeness* e *accuracy*. Tal característica é extremamente importante, pois dá origem a diversas possibilidades de implementações como poder-se-á constatar no capítulo 4 deste trabalho.

3.4 Propriedades de um Detector de Defeitos

Tomando um detector de defeitos $\mathcal{D}$, este pode ser descrito apenas através de suas propriedades *completeness* e *accuracy*. Chandra e Toueg apresentam dois níveis de *completeness* e quatro níveis de *accuracy*, obtidos ao variar as exigências de cada propriedade sobre os elementos

monitorados. Essas variações têm por objetivo aproximar as características exigidas pelas propriedades às possibilidades de um sistema real.

Os dois níveis de *completeness* são:

- *Strong Completeness* - Em um determinado momento (*eventually*), todos processos que falharam serão permanentemente suspeitos por todos processos corretos. Formalmente, $\mathcal{D}$ satisfaz a propriedade de *strong completeness* se:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \Im, \forall p \in crashed(F), \forall q \in correct(F), \forall t' \geq t : p \in H(q,t')$$

- *Weak Completeness* - Em um determinado momento (*eventually*), todos processos que falharam serão permanentemente suspeitos por alguns processos corretos. Formalmente, $\mathcal{D}$ satisfaz a propriedade de *weak completeness* se:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \Im, \forall p \in crashed(F), \exists q \in correct(F), \forall t' \geq t : p \in H(q,t')$$

Isoladamente a propriedade *completeness* não é útil, pois, por exemplo, se um modelo de detector de defeitos originar a suspeita permanente de todos processos, estará respeitando as exigências de *strong completeness*, mas não tem utilidade pois não fornecerá nenhuma informação sobre os defeitos. Para que sejam úteis, as propriedades de *completeness* devem ser utilizadas em conjunto com as propriedades de *accuracy*, que restringem os enganos que um detector de defeitos pode cometer.

Os níveis de *accuracy* são:

- *Strong Accuracy* - Nenhum processo é considerado suspeito antes de ter falhado. Formalmente, $\mathcal{D}$ satisfaz o *strong accuracy* se:

$$\forall F, \forall H \in \mathcal{D}(F), \forall t \in \Im, \forall p,q \in \Pi - F(t) : p \notin H(q,t)$$

Como é difícil (se não impossível) obter *strong accuracy*, na maior parte dos sistemas práticos, também é definido:

- *Weak Accuracy* - Existe pelo menos um processo correto que nunca é suspeito de ter falhado. Formalmente, $\mathcal{D}$ satisfaz a propriedade de *weak accuracy* se:

$$\forall F, \forall H \in \mathcal{D}(F), \exists p \in correct(F), \forall t \in \Im, \forall q \in \Pi - F(t) : p \notin H(q,t)$$

Mesmo o *weak accuracy* garante que ao menos um processo correto nunca é suspeito. Como este tipo de precisão pode ser difícil de obter, são considerados detectores de defeitos que podem suspeitar de todos processos, de vez em quando. Isso significa que a exigência de que o *strong accuracy* ou o *weak accuracy* venham a ser satisfeitos a partir de um instante *t*. As propriedades resultantes desta nova exigência são chamadas *eventual strong accuracy* e *eventual weak accuracy*, respectivamente.

- *Eventual Strong Accuracy* - há um instante de tempo após o qual processos corretos

não são considerados suspeitos por nenhum processo correto. Formalmente, $\mathcal{D}$ satisfaz o *eventual strong accuracy* se:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \Im, \forall t' \geq t, \forall p,q \in correct(F): p \notin H(q,t')$$

- *Eventual Weak Accuracy* - há um instante de tempo após o qual algum processo correto não é considerado suspeito por nenhum processo correto. Formalmente, $\mathcal{D}$ satisfaz o *eventual weak accuracy* se:

$$\forall F, \forall H \in \mathcal{D}(F), \exists t \in \Im, \exists p \in correct(F),$$
$$\forall t' \geq t, \forall q \in correct(F): p \notin H(q,t')$$

3.5 Classes de Detectores de Defeitos

Através da associação entre os dois níveis de *completeness* e os quatro níveis de *accuracy*, podem ser deduzidas oito classes de detectores de defeitos (tab. 3.1). Suas características são ressaltadas pela nomenclatura utilizada para identificar cada classe: um detector que satisfaça as propriedades de *strong completeness* e *strong accuracy* é chamado *Perfect*, e assim por diante.

Algumas dessas classes (as mais exigentes e, portanto, mais confiáveis) são impossíveis de implementar na prática, em um sistema assíncrono. Entretanto, Chandra e Toueg também demonstraram que as classes de detectores de defeitos são relacionáveis através de funções de redutibilidade, ou seja, identificaram as transformações necessárias para que uma classe de detectores comporte as exigências de outra classe. Assim, pode-se construir um algoritmo que se aproxime das classes mais exigentes, através do uso de classes mais simples mas implementáveis, acrescidas de restrições e controles especiais.

TABELA 3.1 - Classe de detectores de defeitos

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

O uso de funções de redutibilidade auxiliam bastante na definição de um detector de defeitos, pois nem sempre as exigências do sistema podem ser atendidas com o uso de detectores "fracos" (no capítulo 4, são dados exemplos de detectores para diversos ambientes de falhas). Até mesmo para a elaboração de um detector com suporte a falhas de *crash,* pode ser interessante descrevê-lo com uma classe de detectores mais refinada, mas de forma que o detector ainda possa ser implementado. Um exemplo dessa preferência, na hora do projeto, refere-se às características *strong completeness* e *weak completeness*. Não é interessante, em um primeiro momento, considerar que "todos processos que falharam serão permanentemente suspeitos por **alguns** processos

corretos", como afirma a *weak completeness*, pois essa é uma idéia muito vaga, difícil de expressar (e imaginar) deterministicamente. Já a propriedade *strong completeness* afirma que "todos processos que falharam serão permanentemente suspeitos por **todos** processos corretos", que é muito mais fácil de adaptar, mas mais difícil de implementar.

Chandra e Toueg [CHA 96a] comprovaram as relações de redutibilidade entre as classes de detectores, ao variar as propriedades de *completeness*, para cada possibilidade de *accuracy*, de modo a estabelecer as seguintes relações entre as classes de detectores:

- $\mathcal{P} \Leftrightarrow Q$
- $\mathcal{S} \Leftrightarrow \mathcal{W}$
- $\Diamond\mathcal{P} \Leftrightarrow \Diamond Q$
- $\Diamond\mathcal{S} \Leftrightarrow \Diamond\mathcal{W}$

De uma forma geral, essas transformações ocorrem através da provisão de informações adicionais sobre a detecção de defeitos em outros processos. Isso é feito através do compartilhamento das listas de suspeitos entre os detectores, de modo a divulgar e estabelecer uma visão mais uniforme sobre os defeitos detectados em todo sistema. O algoritmo geral para a transformação de classes *strong* para *weak*, proposto por Chandra e Toueg, é mostrado na fig. 3.1.

```
Inicialização:
    output_p ← 0

  cobegin

    || Task 1:
    repeat forever
      {p queries its local failure detector module D_p}
      suspects_p ← D_p
      send (p, suspects_p) to all

    || Task 2:
    when receive(q, suspects_q) for some q
      output_p ← (output_p ∪ output_q) - {q}

  coend
```

FIGURA 3.1 - Transformação de *Weak Completeness* em *Strong Completeness*

Através do estudo dessas diversas classes, Chandra [CHA 96b] demonstrou que o detector de defeitos mais fraco (simples) para resolver o problema do consenso que suporte até n-1 processos falhos (ao menos um processo correto), é a classe dos *Weak* ($\mathcal{W}$), enquanto, para um sistema que suporte uma maioria de processos corretos ((n-1)/2 processos falhos), a classe dos *Eventually Weak* ($\Diamond\mathcal{W}$) é suficiente. De fato, um detector chamado $\Diamond\mathcal{W}_0$, que satisfaça apenas as propriedades de $\Diamond\mathcal{W}$ e nenhuma outra, é considerado o detector de defeitos mais fraco que pode resolver um consenso.

Chandra e Toueg também fazem algumas considerações sobre o uso de detectores *Eventually Weak* [CHA 96a, CHA 96b]:

- a qualquer tempo *t*, os processos não podem usar o detector $\langle\rangle\mathcal{W}$ para determinar a identidade de um processo correto. Além disso, os processos não podem determinar se existe um processo correto que não será considerado suspeito após um tempo *t*;

- um detector $\langle\rangle\mathcal{W}$ pode cometer um número infinito de enganos. Assim, um detector $\langle\rangle\mathcal{W}$ pode tanto ficar adicionando e removendo um processo da lista dos suspeitos, como pode acontecer que um processo correto seja considerado suspeito por todos os outros processos, durante todo tempo de execução;

- as propriedades *eventually* de um detector $\langle\rangle\mathcal{W}$ obrigam que, após um tempo *t*, certas condições (um processo correto não é mais considerado suspeito pelos demais processos corretos) se mantenham para sempre. Em um sistema real, isso não pode ser obtido, e, na prática, quando se procura uma solução para uma ação que "termina", como no caso do consenso, basta que as condições sejam mantidas por um tempo suficiente longo para que o algoritmo atinja seu objetivo. Como não é possível determinar quanto é esse tempo "suficientemente longo" em um sistema assíncrono, torna-se conveniente prover as propriedades do $\langle\rangle\mathcal{W}$ com informações adicionais. De fato, a maioria das implementações de detectores de defeitos utiliza limites de tempo e a escolha desses valores é crucial para que o detector de defeitos respeite as propriedades de *completeness* e *accuracy*. *Timeouts* de curta duração permitem que os processos detectem defeitos mais rapidamente, mas também aumentam a probabilidade de suspeitas errôneas, com o risco de violar a propriedade de *accuracy*. Assim, a relação entre latência (*timeouts* curtos) e precisão (*timeouts* longos) deve ser estabelecida de acordo com as características do sistema e da rede de comunicação;

- se um detector comporta-se de forma errônea como, por exemplo, quando ele suspeita incorretamente (e para sempre) de todos processos corretos, a aplicação pode perder seu *liveness* (formalmente, *liveness* é a propriedade que define a "vivacidade", ou o nível de atividade de um sistema), mas não a segurança de suas operações (*safety*). Se, em um consenso, um detector suspeita incorretamente e continuamente dos demais processos corretos, os processos não conseguem decidir em um valor, mas também são impedidos de decidir valores diferentes ou inválidos. Se um algoritmo de Broadcast Atômico (que assegura as propriedades de Validade, Acordo, Integridade e Ordem Total [HAD 93]) apresenta um detector com esse problema, os processos são impedidos de entregar suas mensagens, mas nunca irão entregar mensagens fora de ordem.

4. MODELOS DE DETECTORES DE DEFEITOS

Chandra e Toueg [CHA 96a] descreveram seu modelo de detectores de defeitos apenas através das propriedades de *completeness* e *accuracy*. Isso possibilitou aos desenvolvedores construirem suas implementações baseando-se em características matematicamente comprovadas, sem haver restrições prévias quanto ao modelo de programação ou comunicação empregados.

Diversas questões são elaboradas sobre o desempenho de um sistema distribuído, e muitas dessas dúvidas afetam diretamente o funcionamento de um detector de defeitos. Um detector de defeitos deve atender exigências quanto ao número de mensagens trocadas entre processos, à latência e à precisão sobre a suspeita de falha de um processo, e à necessidade de conhecimento dos processos do sistema. Isto deve-se ao fato que tais características variam de acordo com as características do meio de transmissão de dados, do modelo de falhas considerado e do aumento no número de processos do sistema. Diversas alternativas de implementação foram desenvolvidas para se adequar a certas exigências, de forma que não existe uma proposta melhor que as demais, mas apenas há aquelas que satisfazem melhor as exigências de certos sistemas.

4.1 Classificação quanto ao Fluxo das Informações

Felber [FEL 98] faz uma classificação dos detectores de defeitos baseada na forma como os defeitos dos componentes são propagados no sistema. Sua classificação apresenta as características básicas de dois modelos unidirecionais, *Push* (envio) e *Pull* (recepção), sendo que a análise envolve a verificação do fluxo de informações entre processos monitorados, os detectores de defeitos e os processos clientes (que receberão as suspeitas de defeitos). Também é apresentada uma classe de detectores com um sistema misto de propagação. Essa é uma tentativa de classificar os detectores, mas como pode-se verificar, nos exemplos de detectores apresentados na seção 4.2, as implementações escolhem as soluções de acordo com a conveniência para a solução dos problemas.

4.1.1 Modelo *Push*

No modelo *Push*, o fluxo de controle segue a mesma direção do fluxo das informações, ou seja, dos processos monitorados em direção ao processo cliente. Para isso, em um modelo Push, os processos monitorados necessitam estar ativos (quando se aplica esse conceito sobre objetos Java, isso significa que os objetos *Push* necessitam ter uma *thread* ativa associada). Eles enviam periodicamente mensagens identificadas (conhecidas como "*I'm alive!*") que têm como função principal demonstrar que, se o processo está enviando mensagens, ele ainda está operacional (informação suficiente se for considerado um modelo de falhas de *crash*). Se um detector de defeitos não receber tais mensagens dentro de um limite específico de tempo, este passa a suspeitar do processo monitorado. Este método é consideravelmente eficiente, pois as mensagens enviadas no sistema são *one-way*, e pode ser implementado de forma a utilizar as facilidades de *multicast* providas pela rede de comunicação, otimizando o sistema caso múltiplos detectores de defeitos estejam monitorando os mesmos processos [FEL 98].

A fig. 4.1 demonstra como o modelo *Push* pode ser usado para monitorar os processos de um sistema. As mensagens trocadas entre o detector e o cliente são diferentes das mensagens trocadas entre os processos monitorados e o detector. O detector geralmente notifica o cliente

apenas quando um processo monitorado modifica seu estado (o detector acrescenta o processo à lista de suspeitos, ou então remove-o desta lista), enquanto as mensagens de "*I'm Alive!*" são enviadas continuamente.

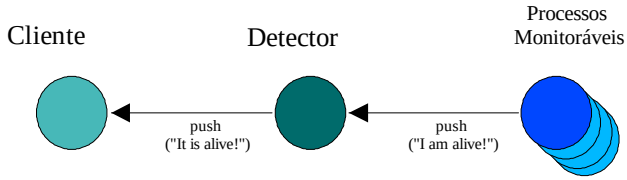

FIGURA 4.1 - Modelo *Push* de monitoramento

A troca de mensagens entre o detector e os processos monitorados com o modelo *Push* é mostrada na fig. 4.2. Nela pode-se identificar o processo monitorado, que periodicamente envia mensagens de "*I'm Alive!*" para o detector de defeitos. Quando o detector recebe uma mensagem, ele reinicializa o *timeout* relativo àquele processo. Se o *timeout* expirar e o detector não tiver recebido nenhuma nova mensagem do processo monitorado, então inicia a suspeita sobre esse processo.

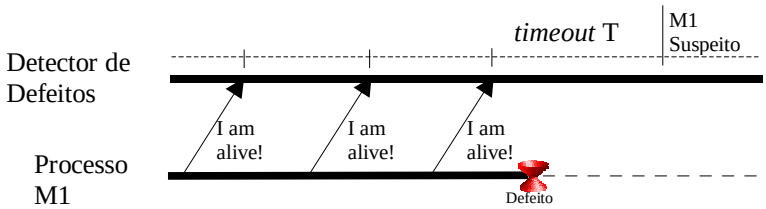

FIGURA 4.2 - Monitorando mensagens no modelo *Push*

4.1.2 Modelo *Pull*

O modelo *Pull* apresenta o fluxo de informações em sentido oposto ao fluxo de controle, ou seja, as informações do detector de defeitos só são obtidas através da requisição dos clientes. Dessa forma, os processos monitorados podem ser passivos (e somente quando questionados pelo detector, enviam uma resposta). O detector de defeitos periodicamente envia mensagens de *liveness request*, ou seja, mensagens que questionam se os processos estão ainda ativos. Se um processo monitorado responde à requisição do detector, significa que ele está operando.

Devido à troca de mensagens entre os processos monitorados e o detector se realizar em duas etapas com bloqueio (*two-ways*), esse modelo tende a ser menos eficiente do que o modelo *Push*, mas apresenta vantagens quanto ao desenvolvimento da aplicação, uma vez que os processos não precisam estar ativos, nem ter conhecimento sobre a temporização do sistema (por exemplo, não há a necessidade de saber qual é a freqüência com que o detector espera receber mensagens). A fig. 4.3 exibe o fluxo das informações de monitoramento.

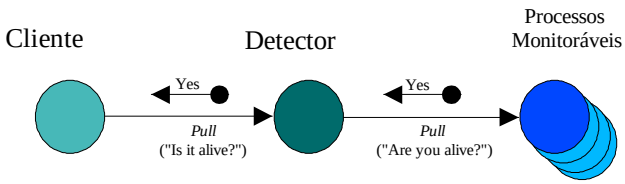

FIGURA 4.3 - O fluxo do modelo *Pull*

A forma com que as mensagens são trocadas entre o detector e um processo monitorado é detalhada na fig. 4.4. Periodicamente, o detector envia uma mensagem de *liveness* para os processos monitorados, e fica esperando uma resposta. Se um processo não envia resposta (ou a mensagem foi perdida pela rede de comunicação), então a expiração de um *timeout* aciona a suspeição deste processo pelo detector.

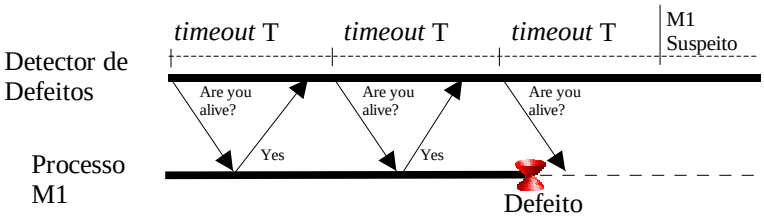

FIGURA 4.4 - Mensagens de monitoramento

4.1.3 Modelo Misto

Uma alternativa para prover maior flexibilidade ao sistema de detecção de defeitos é a de unir os modelos *Push* e *Pull*, para que o detector possa interagir com os processos monitorados usando ambas as formas de acesso. Para isso, são utilizadas apenas mensagens não bloqueantes (*one-way*), mais eficientes, dispostas da seguinte maneira: o detector envia uma mensagem de *liveness request*, usando um envio não bloqueante, para um processo monitorado (modelo *Pull*), e espera uma resposta via uma mensagem *one-way* do processo (modelo *Push*); o processo monitorado também pode enviar uma mensagem indicando que está operando sem que essa tenha sido requisitada. Para melhorar o desempenho do sistema, o detector pode enviar sua requisição de *liveness* apenas se o processo monitorado ainda não enviou sua mensagem de *"I am alive!"* dentro de um limite de tempo. De qualquer forma, com esse modelo misto, o detector pode operar com qualquer tipo de processos, sem a necessidade de um conhecimento prévio sobre qual o modelo suportado pelo processo a ser monitorado. Um exemplo desse tipo de operação pode ser visto na fig. 4.5.

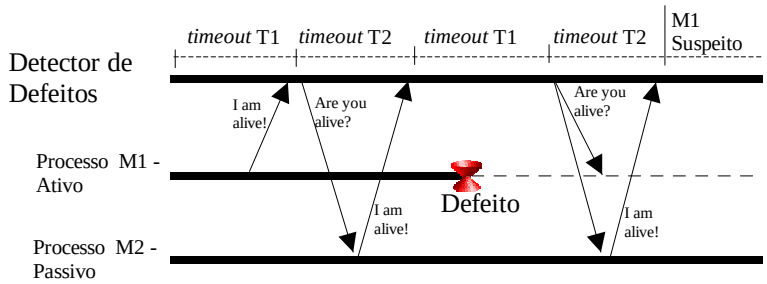

FIGURA 4.5 - Detecção com o modelo Misto

4.1.4 Considerações sobre os modelos

Felber também faz uma breve avaliação das vantagens e desvantagens dos modelos *Push* e *Pull*. Pelo lado do modelo *Push*, as vantagens referem-se à possibilidade de otimização do número de mensagens transmitidas, devido tanto ao uso de mensagens *one-way* quanto ao uso de facilidades de *multicast* da rede. O modelo *Pull*, por outro lado, possibilita que os processos monitorados sejam

passivos, sem a necessidade de conhecimento sobre *timeouts* nem sobre quais são os detectores que os monitoram.

Como desvantagens, o modelo *Push* pode levar a uma inundação de mensagens na rede, enquanto o modelo *Pull* obriga o detector a refazer a lista de processos falhos toda vez que um cliente solicita.

A proporção com que as vantagens e desvantagens influem no sistema somente pode ser verificada caso a caso. Em um sistema que necessita saber imediatamente quando ocorreu uma falha, presta-se melhor o modelo *Push*. Já em um sistema que ocasionalmente requer informações sobre o estado dos processos, o modelo *Pull* pode ser utilizado para minimizar o número de trocas de mensagens.

4.2 Propostas de Implementação

Conforme visto anteriormente, a definição das propriedades de *completeness* e *accuracy* de Chandra e Toueg deixa completamente em aberto os detalhes de implementação dos detectores. Dessa forma, a bibliografia contém diversos exemplos de detectores de defeitos, elaborados para atender os problemas de diversos modelos conceituais de falhas. Isso possibilita aos desenvolvedores escolher qual o algoritmo de detectores que mais se adapta às suas necessidadas e características do sistema.

4.2.1 *"I am alive!"*

O modelo *"I am alive!"* segue os princípios básicos do modelo *Push* apresentado na seção 4.1. Um sistema baseado em mensagens do tipo *"I am alive!"* normalmente considera a existência de um modelo de falhas baseado em *crash*, de forma que um processo é considerado suspeito quando o detector não recebe mensagens de *"I'm alive!"* após um período máximo de espera, sinalizado pela ocorrência de um *timeout*. A mudança no conjunto de elementos considerados suspeitos ocasiona a notificação dos clientes por parte do detector de defeitos, de modo a atualizar suas visões dos processos corretos na rede. O *timeout*, que caracteriza o tempo de amostragem necessário ao julgamento dos suspeitos, tende a ser um valor único para todos os processos, devido à dificuldade em conhecer as características individuais de cada um e do meio de comunicação até ele. Esse *timeout* deve ser uma aproximação de um valor adequado, uma vez que, nos sistemas assíncronos, é impossível determinar os tempos de execução de cada processo. Uma determinação inadequada do *timeout* pode levar a uma constante atualização da lista de processos suspeitos, o que reduz consideravelmente a eficiência da aplicação que o utilize, embora não comprometa a segurança das operações.

Uma possibilidade, levantada por Chandra [CHA 96a], refere-se ao uso de uma atualização dinâmica do *timeout* do sistema. Quando uma mensagem de *"I am alive!"* chega ao detector após o *timeout* ter sido superado, talvez por causa da velocidade da rede, o processo é retirado da lista de suspeitos e seu *timeout* é alongado, para tentar englobar esse processo (evitando romper a propriedade de *accuracy* e a suspeita de um processo correto). De forma semelhante, se suas respostas chegam muito antes do limite estabelecido para o *timeout*, esse valor pode ser diminuído, para melhorar a eficiência da detecção. Todas essas mudanças ocorrem apenas no detector, pois o processo ignora tais limites e apenas comporta-se de acordo com seu relógio interno.

4.2.2 *Liveness Request*

Os detectores que implementam *Liveness Request* utilizam as funcionalidades básicas do modelo *Pull*. De fato, Felber afirma que, nas primeiras versões do seu sistema OGS, o modelo *Liveness Request* era o único implementado, e que somente nas versões mais novas o OGS conta com o modelo Misto [FEL 98].

Quando utiliza-se um sistema com objetos distribuídos, como no caso do OGS, diversos métodos devem ser definidos para prover invocação remota. Ao contrário das invocações locais, as remotas podem estar sujeitas às falhas de comunicação. Assim, uma maneira eficiente de implementar um detector de defeitos usando o modelo *Pull* é fazer com que os objetos monitorados suportem uma operação específica (no caso do OGS, o método `are_you_alive()`), e invocá-la periodicamente. Se o objeto monitorado estiver operando e não ocorrer nenhuma falha de comunicação, a invocação ocorre sem problemas. Entretanto, se a invocação falha, inicia-se o processo de suspeita por parte do detector.

Quando um detector no OGS invoca o método `are_you_alive()` de cada objeto monitorado, este pode ser considerado suspeito ou correto, de acordo com o sucesso da invocação. Esta invocação pode ser realizada sob demanda da aplicação ou periodicamente pelo próprio detector. As informações retornadas aos clientes, indicando o estado de cada objeto monitorado podem ser:

- SUSPECTED - o detector de defeitos considera este objeto suspeito.
- ALIVE - o objeto respondeu corretamente às invocações, estando provavelmente operacional.
- DONT_KNOW - quando o detector não está monitorando o objeto requerido.

4.2.3 *Heartbeat*

O detector *Heartbeat* foi proposto por Aguilera [AGU 97a] como uma forma de resolver problemas de comunicação confiável quiescente em um ambiente com possibilidade de falhas por *crash* e perdas de mensagens.

Um algoritmo é chamado quiescente quando este considera que chegará o momento em que irá parar de enviar mensagens (embora esse instante seja desconhecido). Por exemplo, uma comunicação confiável pode ser implementada de forma que um processo *p* fique enviando uma mensagem *m* para o processo *q*. Se for considerado um cenário sem falha nos processos, mas com possibilidade ilimitada de perdas na rede de comunicação, fica impossível garantir que o processo *q* recebeu a mensagem. Assim, é comum definir um limite para as perdas de uma comunicação, estabelecendo o que se chama conexão *fair* (regular, íntegra). Em uma conexão *fair*, assume-se que, mesmo no caso em que o meio de comunicação perca infinitas vezes as mensagens que transporta, uma mensagem que também fosse reenviada infinitas vezes alcançaria seu destino.

Para solucionar o problema da falta de informação sobre a recepção da mensagem, pode-se modificar o protocolo de comunicação de modo que um processo *p* envie repetidamente a mensagem *m* para *q* até que esse responda com um *ack(m)*. Como o processo *q* responde um

ack(m) para cada mensagem que recebe de *p*, o canal de comunicação pode perder inúmeras mensagens tanto de $p \rightarrow q$ quanto de $q \rightarrow p$. Esse tipo de algoritmo, que um dia pára de enviar mensagens, pode ser considerado um algoritmo quiescente, que se parece muito com um algoritmo com a propriedade Terminação (de fato, todo algoritmo com Terminação é quiescente, embora a reciprocidade não seja verdadeira). Essa característica pode ser bem explorada se for considerado que um sistema computacional freqüentemente está sujeito a falhas de *crash* e da rede, e uma aplicação é executada por um tempo finito, exigindo portanto uma solução em tempo finito.

Quando se considera um cenário com falhas de processos por *crash* e perda de mensagens, a solução acima, envolvendo apenas *acks*, não basta, pois não há informação suficiente para determinar quando um processo falhou ou quando a mensagem foi perdida. Para isso, há a necessidade de um detector de defeitos.

Em um outro trabalho[AGU 97b], Aguilera mostra que para resolver um algoritmo quiescente, um detector de defeitos "tradicional" (que gera uma listagem de processos suspeitos) precisa ser do tipo $\langle\diamond\mathcal{P}$, que não pode ser implementado na prática. Para implementar um detector do tipo $\langle\diamond\mathcal{P}$, seria necessário um consenso sobre as suspeitas, e isso representa um nível de abstração superior ao nível em que os detectores trabalham. Esse mesmo trabalho mostra que se pode construir uma solução para esse problema usando um detector que tem valores de saída ilimitados (diferente de um "restrito" suspeito/não suspeito), não negativos e que nunca decrescem, tornando o detector implementável na prática. Assim, a classe de detectores conhecidos como *Heartbeat* é uma tentativa de implementar tal solução.

Para gerar essa saída de dados diferente do modelo tradicional de detectores, cada processo envia mensagens de *heartbeat* (semelhantes a um "*I am alive!*") para seus vizinhos imediatos (processos conectados diretamente). Quando um processo recebe uma dessas mensagens, incrementa um contador referente a esse vizinho. O detector *Heartbeat* não usa *timeouts*, somente conta o número de mensagens que recebeu. Assim, ele fornece à cada requisição da aplicação apenas um *array* com os contadores, de modo que a própria aplicação compara com os valores da requisição anterior. Quando um processo cessa o envio de mensagens de *heartbeat*, o contador deste processo não é mais incrementado, e ocorre a suspeita de falha. Como esse tipo de saída dos dados tem mais informações que uma simples lista de suspeitos, podem ser avaliados tanto a precisão da detecção (determinação de processos ativos) quanto o estado dos processos monitorados (um incremento maior em um determinado contador pode indicar que um processo é mais rápido que os demais).

Devido ao uso de mensagens *heartbeat* para a sinalização entre os processos, este detector pode ser confundido com os módulos detectores de defeitos de outros sistemas de comunicação de grupo (o módulo da ferramenta Ensemble, por exemplo), que também são denominados *heartbeat*, embora implementem o modelo "*I am alive!*". Apesar do nome em comum, é fácil notar as diferenças, principalmente quanto à forma com que as mensagens são utilizadas para gerar a lista de processos suspeitos (no caso do detector de Aguilera, o *array* de contadores).

Outra diferença refere-se ao número de mensagens enviadas por cada processo. Como dito anteriormente, um processo envia *heartbeats* apenas para seus vizinhos. O algoritmo considera que sempre existe um caminho que interliga os processos da rede, e os processos diretamente conectados são chamados vizinhos. Quando um processo recebe uma mensagem de *heartbeat*, este retransmite para a sua lista de vizinhos, mas antes acrescenta sua própria identificação à mensagem, caracterizando o caminho (*path*) por onde ela circulou. Assim, mesmo que não tenha conexão *full-duplex* entre todos os processos, pode-se ficar sabendo que um processo estava ativo pois retransmitiu a mensagem, e adicionou-se ao *path*. O *path* também é utilizado para evitar que uma

mensagem antiga seja re-inserida na rede. Os processos reenviam as mensagens apenas para os seus vizinhos que não constam no *path*, de forma que, com o tempo, as mensagens não serão mais retransmitidas, cessando sua circulação.

Um detector *Heartbeat* só precisa conhecer a identidade dos seus vizinhos, quando começa a funcionar. Graças à difusão das mensagens, não é necessário transmitir dados a outros processos que não os vizinhos. E o conhecimento dos outros elementos pode ser adquirido a partir do *path* presente nas mensagens recebidas. Isso entretanto pode causar problemas quanto for utilizado um protocolo tradicional (de consenso, por exemplo) junto com o *Heartbeat*, pois tal protocolo espera encontrar informações sobre todos os processos da rede, não apenas sobre os vizinhos. Assim, as implementações do *Heartbeat* que procuram trabalhar com protocolos tradicionais devem coletar informações sobre todos os processos da rede, através do *path* contido nas mensagens.

O algoritmo para a implementação do detector *Heartbeat* é apresentado na fig. 4.6. Conforme este algoritmo, o detector é composto por duas tarefas concorrentes. Na primeira tarefa, um processo periodicamente envia mensagens de *"I am alive!"* para todos os seus vizinhos. Essa mensagem é composta de um campo identificador da operação (HEARTBEAT) e do *path* inicial, que contém apenas o próprio processo. A segunda tarefa é tratar as mensagens recebidas que contenham o formato {HEARTBEAT, *path*}. Ao receber uma mensagem, um processo incrementa o contador de todos os seus vizinhos que estejam relacionados no *path*. Depois, o processo adiciona seu próprio identificador ao fim do *path*, e retransmite a mensagem para os vizinhos que ainda não a receberam.

```
Inicialização:
   for all q ∈ vizinhos(p) do 𝒟_p[q] ← 0

cobegin
   || Task 1:
      repeat periodically
         for all q ∈ vizinhos(p) do send_p,q (HEARTBEAT, p)

   || Task 2:
      upon receive_p,q(HEARTBEAT, path) do
         for all q such that q ∈ vizinhos(p) and q appears in path do
            𝒟_p[q] ← 𝒟_p[q] + 1
         path ← path · p
         for all q such that q ∈ vizinhos(p) and q does not appears in path do
            send_p,q(HEARTBEAT, path)
coend
```

FIGURA 4.6 - Implementação do *Heartbeat*

A saída do detector é o vetor contendo os contadores de cada processo ($\mathcal{D}_p$), sem nenhuma avaliação adicional. Assim, a aplicação (ou o protocolo que usa o detector) compara o vetor com os valores obtidos em uma chamada anterior, de modo a identificar quais são os elementos falhos. A necessidade dessa saída *unbounded* (que neste caso pode-se traduzir por ilimitada, considerando que não há uma saída restrita a "suspeito/não suspeito") é demonstrada por Aguilera [AGU 97a] como a única forma de construir um detector que resolva sistemas quiescentes. Mesmo assim, é fácil imaginar uma camada adicional de *software* para formatar a saída, de acordo com o padrão dos sistemas de detecção. Essa possibilidade deve ser testada, para verificar se não atrapalha os objetivos do sistema.

4.2.4 *Heartbeat* com suporte a partição de rede

A proposta do detector *Heartbeat* [AGU 97a] consiste em um detector de defeitos capaz de resolver algoritmos usando comunicação confiável quiescente, em sistemas sujeitos a falhas de processos através de *crash* (colapso) e perdas de mensagens. Mesmo considerando a possibilidade de falhas nas linhas de comunicação, não era possível obter informações sobre uma rede particionada. O detector *Heartbeat* com Suporte a Partição de Redes [AGU 97c] estendeu as características do primeiro, de modo a suportar também esse cenário de falhas.

Para a definição de particionamento de redes, foram utilizados os seguintes conceitos: uma linha de comunicação que perde mensagens pode apresentar dois tipos de funcionamento, *fair* (regular, desobstruído) e *eventually down* (inoperante). Em uma linha *fair*, mesmo que alguma mensagem seja perdida infinitas vezes, durante a transmissão, se esta for retransmitida também infinitas vezes, há a garantia de que em algum momento ela chegará ao seu destino. Se uma linha é considerada *eventually down*, por outro lado, haverá um momento no qual essa linha cessará completamente o transporte de mensagens. Para aumentar ainda mais a complexidade, considera-se cada linha de comunicação como unidirecional, e a rede pode não ser completamente conectada, ou seja, não existe ligação direta entre todos os pares de nós da rede. A partir dessas definições, é possível atingir um estado de rede particionada, onde dois processos corretos *p* e *q* estão conectados (ou separados) de forma que as mensagens de *p* não chegam a *q* por nenhum caminho, ou seja, não há um caminho *fair* (composto de processos corretos e linhas *fair*) entre *p* e *q*. Essa definição não impede que *p* receba mensagens de *q*, nem que os processos consigam se comunicar com outras partições. Estendendo o conceito da comunicação entre os processos, uma partição é o conjunto máximo de processos que estão mutuamente conectados.

Um exemplo de rede com partição é apresentado na fig. 4.7. Os processos que não falharam serão, em um determinado momento, divididos em quatro partições, A, B, C e D. Cada partição é fortemente interconectada, através de linhas *fair*, de forma que processos da mesma partição podem se comunicar (mesmo que ocorram perdas de mensagens). Nenhuma das partições fica isolada, mas diversas situações podem ocorrer: processos em D podem receber mensagens de C (mas C não consegue receber mensagens de D), não há um caminho *fair* que transmita mensagens de C para A, etc.

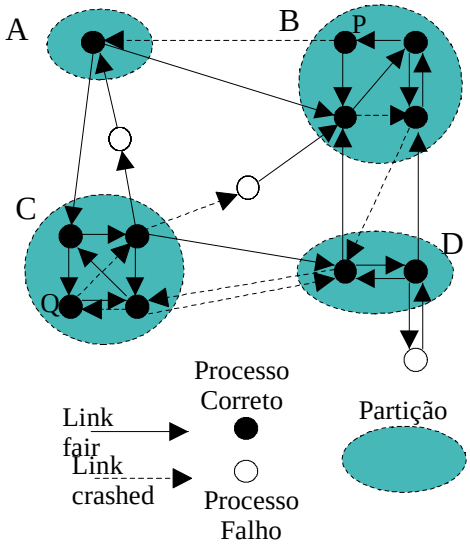

FIGURA 4.7 - Uma rede particionada

A implementação do *Heartbeat* para redes particionáveis (fig. 4.8) é ligeiramente diferente da implementação apresentada na seção 4.2.3. Devido à necessidade de conferir todas as possibilidades de atingir um processo, um detector mantém a relação de todos os processos corretos, obtida a partir das mensagens recebidas. Aguilera não cita explicitamente em seus artigos, mas ao analisar o algoritmo pode-se verificar que as mensagens também circulam por mais tempo na rede para que, em um primeiro momento, elas obtenham os estados dos processos e, em um segundo momento, atualizem todos os processos por onde passaram.

```
Inicialização:
   for all q ∈ Π do 𝒟_p[q] ← 0

cobegin
   || Task 1:
      repeat periodically
         𝒟_p[p] ← 𝒟_p[p] + 1
         for all q ∈ vizinhos(p) do send_p,q (HEARTBEAT, p)

   || Task 2:
      upon receive_p,q(HEARTBEAT, path) do
         for all q ∈ Π such that q appears after p in path do
            𝒟_p[q] ← 𝒟_p[q] + 1
         path ← path · p
         for all q such that q ∈ vizinhos(p) and q appears at most once in path do
            send_p,q(HEARTBEAT, path)
coend
```

FIGURA 4.8 - *Heartbeat* para redes particionáveis

4.2.5 Detector de defeitos para ambientes com falhas por omissão

Dolev *et al.* [DOL 97] propuseram uma extensão aos modelos de detectores de defeitos apresentados por Chandra e Toueg [CHA 96a], para que estes pudessem ser utilizados também em ambientes sujeitos a omissão de mensagens.

De fato, o modelo apresentado considera que um processo pode entrar em *crash* e recuperar-se, mensagens podem ser perdidas ou arbitrariamente retardadas, a rede pode sofrer partição, bem como grupos separados podem se reunificar. Isso é feito através de uma definição diferente sobre processos corretos e falhos (na verdade, também são criados novos estados intermediários), e não através de mudanças nas propriedades dos detectores (os detectores para ambientes com falhas bizantinas na seção 4.2.6 utilizam propriedades específicas). Assim, um detector $\langle\rangle\mathcal{W}(om)$ – *Eventually Weak* para ambientes com omissão – utiliza as mesmas propriedades *eventual weak completeness* e *eventual weak accuracy* que um detector $\langle\rangle\mathcal{W}$, mas a forma como são reconhecidas as falhas é diferente.

O sistema é modelado a partir de um grupo fixo *N* composto de *n* processos, conhecidos por todo sistema. A troca de mensagens é realizada através de datagramas, que por si só

possibilitam diversos tipos de defeitos, entre eles: mensagens perdidas ou entregues fora de ordem, partição da rede em diversos sub-grupos (chamados de "componentes" pelos autores), e a reunificação de partições previamente separadas. Quando um processo falha, ele pode se recuperar, mas para isso ele precisa manter todas as informações sobre seu estado em um armazenamento estável. Dessa forma, como fica difícil diferenciar um processo que falhou e se recuperou de um processo que ficou desconectado dos demais, considera-se um processo em *crash* apenas quando ele não se recupera mais.

Os possíveis estados de um processo são:

- ***alive (vivo)*** - um processo está "vivo" se não apresentou nenhuma falha durante todo o tempo *t.*

- ***crashed (em colapso)*** - um processo é considerado permanentemente *crashed* se falhou durante uma execução σ (que pode ser um intervalo do tempo total *t*). Esse processo não pode se recuperar.

- ***connected (conectado)*** - são considerados *connected* dois processos *p* e *q* que estão *alive*, se *p* recebe, dentro de um período de execução, todas as mensagens enviadas por *q*, e vice-versa. Um conjunto de processos P é *connected* se, para cada par de processos *p* e *q*, ambos estão totalmente interligados.

- ***detached (isolados)*** - dois processos *p* e *q* estão *detached* se, durante todo o tempo *t*, *p* não recebe nenhuma mensagem que *q* enviou, e vice-versa (mesmo durante uma execução). Dois conjuntos de processos P e Q estão *detached* se nenhum de seus processos consegue se comunicar com os processos do outro conjunto.

- ***connected component (componente conectado)*** - Um conjunto de processos *P* é chamado *connected component* em *t* se *P* tem todos seus processos *connected,* e *P* é *detached* dos demais componentes em N.

Tanto no caso dos estados *detached* e *connected component*, se a duração da execução σ se estender ao infinito, estes casos são classificados como *permanent* (permanentes, perenes). Quando ocorre um *permanent connected component*, este define um período estável no qual membros deste conjunto podem trocar mensagens entre si, mas não podem receber mensagens de nenhum outro processo.

Com base nos estados descritos acima, pode-se definir o conceito de processo correto, em um ambiente sujeito a falhas por omissão. Enquanto em um sistema sujeito a falhas de *crash*, os processos corretos são identificados quando se estabelece uma comunicação confiável entre eles, da mesma maneira um sistema sujeito a falhas por omissão considera corretos aqueles processos que venham a se comunicar de forma confiável. A diferença é que, quando se considera a possibilidade de falhas por omissão, é permitido que dois processos corretos sofram perdas de inúmeras (mas não ilimitadas) mensagens, durante um período de transição, para então tornarem-se permanentemente conectados. Dessa forma, este detector caracteriza processos corretos e falhos conforme o enunciado a seguir:

Definição: "Seja P o maior *permanent connected component* durante uma execução σ (se existirem dois ou mais *components* do mesmo tamanho, escolher apenas um). Para cada $p \in P$, p é chamado de correto em σ. Para cada $q \notin P$, q é chamado falho em σ." [DOL 97]

Essa definição engloba todas as possibilidades do ambiente de falhas, uma vez que, quando um sistema não consegue encontrar um período suficientemente estável, ele não consegue delimitar um grupo de processos considerados corretos.

De maneira semelhante, quando ocorre partição na rede, apenas um dos subgrupos pode seguir operando, de forma que as decisões sejam tomadas consistentemente. Pela definição, é possível ter a maior partição do sistema com menos da metade dos elementos, mas isso não é considerado um problema porque os protocolos de acordo costumam restringir as decisões à presença da maioria de processos corretos. A reintegração das partições não é tratada diretamente pelo detector, mas ele está apto a se adaptar à nova visão do grupo, quando os elementos entrarem em um novo período estável.

As propriedades de redutibilidade também são mantidas, de forma que um detector $\Diamond\mathcal{S}(om)$ pode ser construído a partir de um detector $\Diamond\mathcal{W}(om)$. O algoritmo para a transformação é semelhante ao apresentado no capítulo 3, de modo que não será exibido aqui.

Uma análise sobre essa proposta de detectores para ambientes com falhas por omissão sugere que, embora o custo da comunicação entre processos permaneça inalterado ou pouco modificado, com relação ao modelo de detectores apresentado por Chandra e Toueg, a identificação do estado dos processos (e especialmente do estado de um grupo de processos) exige uma carga de processamento bem superior, associada à necessidade de maior troca de informações entre os membros de um conjunto de processos.

Como a determinação das situações estáveis é obrigatória para o estabelecimento de um grupo de processos capazes de realizar um consenso, o módulo de *membership* deve estar intimamente ligado ao detector de defeitos, para compreender e identificar as informações adicionais dos estados dos processos, fugindo então ao conceito de modularidade e generalidade representados pela lista de suspeitos fornecida por um detector de defeitos tradicional.

4.2.6 Detectores de defeitos para falhas bizantinas

Várias construções de detectores de defeitos (inclusive os algoritmos apresentados anteriormente) trabalham apenas com a hipótese de que os processos falham por *crash*, e a complexidade do tratamento aumenta quando se pretende implementar um detector para modelos de falhas mais complexos, como as falhas por omissão. Como um modelo de falhas bizantinas é o mais complexo de todos, sua complexidade deve ser proporcional, já que os detectores devem considerar a possibilidade dos processos comportarem-se tão arbitrariamente que é impossível distinguir apenas através de mensagens recebidas de um processo se há uma falha na rede ou se o próprio processo está falho (por acaso, um problema semelhante ao da detecção de defeitos em um sistema distribuído assíncrono).

Alguns trabalhos tentam especificar detectores de defeitos para ambientes sujeitos a falhas bizantinas. Kihlstrom *et al.* [KIH 97] determinaram tal dispositivo, mostrando como obter detectores $\Diamond\mathcal{S}(Byz)$ e $\Diamond\mathcal{W}(Byz)$, e como alcançar o consenso usando tais detectores. Anteriormente, Malkhi e Reiter (*apud* [KIH 97]) haviam definido uma classe $\Diamond\mathcal{S}(bz)$, mas não uma $\Diamond\mathcal{W}(bz)$. Além disso, tal classe de detectores exigiria uma comunicação confiável causal, gerando um esforço muito grande no mecanismo de comunicação.

Os detectores $\Diamond\mathcal{S}(Byz)$ e $\Diamond\mathcal{W}(Byz)$ assumem que o sistema distribuído para o qual se deseja

implementá-los apresenta uma rede de comunicação não-particionável que não altera o conteúdo das mensagens enviadas, e que a comunicação é confiável de modo a entregar as mensagens, em um certo período após o envio.

Um processo só pode ser considerado correto caso comporte-se (sempre) de acordo com sua especificação. Quando isso não acontece, ele é classificado como falho, exibindo então um comportamento arbitrário. Como um processo bizantino pode agir como se estivesse em *crash*, este tipo de falha também é incluída no modelo Bizantino.

Para otimizar o número de mensagens trocadas para a determinação do comportamento bizantino [JAL 94], todas as mensagens são autenticadas através do uso de uma criptografia de chave pública, fazendo uma assinatura digital. Todos processos conhecem (ou podem obter de forma confiável) as chaves públicas dos outros processos, e verificar a identidade do emissor. Dessa forma, as mensagens carregam informações adicionais que permitem validar a integridade da mensagem e o momento em que ela foi criada [BAL 99].

Nem todos processos que apresentam comportamento arbitrário podem ser identificados. Isso deve-se ao fato de que o detector consegue apenas identificar as chamadas *falhas bizantinas detectáveis*, mas não as *falhas bizantinas não detectáveis.*

As falhas não detectáveis podem ser de dois tipos: aquelas onde os defeitos não podem ser observados nas mensagens recebidas pelos processos, ou aquelas falhas onde é impossível determinar qual o processo defeituoso. Por exemplo, se um processo bizantino envia uma mensagem para todos os processos contendo um valor diferente daquele que ele apresenta internamente, essa falha não é percebida. E se um processo bizantino reenvia uma mensagem que anteriormente foi enviada por um outro processo mas que não continha uma assinatura válida; então, na ausência de mais informações, é impossível determinar que o processo bizantino enviou a mensagem.

As falhas detectáveis são classificadas como falhas por omissão e por co-omissão. Uma falha por omissão ocorre quando um processo se nega a enviar uma mensagem requisitada por um ou mais processos corretos. As falhas por co-omissão podem ser mensagens incorretamente formadas ou mal assinadas, ou pode ser que um processo envie duas ou mais mensagens "mutantes" para outros processos. Mensagens "mutantes" são corretamente formadas e individualmente assinadas, que têm os mesmos tipos de dados, a mesma origem e a mesma identificação, mas contêm dados diferentes. Se um processo recebe mensagens mal formadas ou assinadas, ou então recebe duas mensagens mutantes, poderá detectar o comportamento bizantino do processo que as enviou.

Um detector de defeitos em $\Diamond\mathcal{S}(Byz)$, que reconhece falhas bizantinas detectáveis, deve satisfazer as seguintes propriedades:

- ***Eventual Strong Byzantine Completeness*** - Há um intervalo de tempo após o qual todos processos que exibiram um comportamento bizantino detectável serão permanentemente suspeitos por todos os processos corretos.

- ***Eventual Weak Accuracy*** - Há um intervalo de tempo após o qual alguns processos corretos não serão considerados suspeitos por nenhum processo correto.

Pode-se tentar construir um detector $\Diamond\mathcal{W}(Byz)$ "enfraquecendo" a propriedade de *completeness*, gerando um *Eventual Weak Byzantine Completeness* ("Há um intervalo de tempo após o qual todos processos que exibiram comportamento bizantino detectável serão

permanentemente suspeitos por alguns processos corretos"). Entretanto, com essa propriedade, não é possível criar um algoritmo de transformação similar ao apresentado por Chandra e Toueg, em um ambiente com falhas bizantinas. Um processo que se comporta arbitrariamente pode enviar repetidamente para todos os outros processos uma lista de suspeitos onde considera todos os processos do sistema falhos. Como um processo faz a união das listas que recebeu com a sua própria lista de suspeitos, esse comportamento bizantino pode levar a uma desconfiança mútua entre todos os processos. A solução para isso incorre basicamente na forma como são confirmadas as informações em um sistema bizantino, ou seja, há a necessidade de que a maioria dos processos concorde para que a informação seja considerada confiável. Assim, um processo só notifica sua suspeita para os demais se ele já recebeu comunicações de outros *k+1* processos (sendo *k* o número máximo de processos falhos admitidos no sistema). Uma propriedade de *Completeness* que respeita essa maioria está apta a operar sob comunicações entre processos arbitrários, sem perder a confiabilidade. Um *Eventual Weak Byzantine (k+1) Completeness* é a forma mais fraca de *completeness*, que pode ser criada a partir da transformação de um *Eventual Strong Completeness*, sem perder a confiabilidade.

- ***Eventual Weak Byzantine (k+1) Completeness*** - Há um intervalo de tempo após o qual todos processos que exibiram comportamento bizantino detectável serão permanentemente suspeitos por no mínimo *k+1* processos corretos.

Associando essa classe de *completeness* com o *Eventual Weak Accuracy* obtém-se a classe de detectores de defeitos bizantinos $\langle\rangle\mathcal{W}(Byz)$. Esta classe pode ser transformada em uma classe $\langle\rangle\mathcal{S}(Byz)$ através do algoritmo da fig. 4.9.

```
Inicialização:
  output_p ← 0;
  suspect_p ← 0;
  for each r in S
    suspecting_p[r] ← 0;

cobegin

  || Task 1:
    repeat forever
      suspected_p ← 𝒟_p;
      send (p, suspected_p) to all;

  || Task 2:
    when receive (q, suspected_q) from q
      for each r in S
        if r ∈ suspected_q then
          suspecting_p[r] ← (suspecting_p[r]∪{q});
        else
          suspecting_p[r] ← (suspecting_p[r]−{q});
        if |suspecting_p[r]| ≥ k+1 then
          output_p ←(output_p∪{r});
        else
          output_p ←(output_p −{r});
coend
```

FIGURA 4.9 - Transformação de $\langle\rangle\mathcal{W}(Byz)$ para $\langle\rangle\mathcal{S}(Byz)$

Baldoni *et al.* [BAL 99] propuseram uma extensão ao detector $\langle\rangle\mathcal{S}(bz)$ de Malkhi, onde o detector de defeitos para ambientes sujeitos a falhas bizantinas é dividido em dois módulos: um detector de defeitos para processos *quiet* (processos que após um determinado instante não enviam mais mensagens – um processo em *crash* também é *quiet*), e um detector de comportamento bizantino. Este detector de comportamento bizantino, descrito através de um autômato finito, é definido separadamente para facilitar a comprovação dos algoritmos e para evitar a sobrecarga do sistema de detecção. Assim, um processo que suporte a operação de consenso poderá ser construído modularmente, com os seguintes blocos básicos, como mostra a fig. 4.10: consenso, detecção de comportamento arbitrário, detector de defeitos da classe $\langle\rangle\mathcal{S}(bz)$, certificação e assinatura digital. As setas indicam por quais os módulos passam as mensagens que chegam e saem do detector, caracterizando as operações realizadas sobre elas para a formação das mensagens enviadas e a verificação das mensagens recebidas.

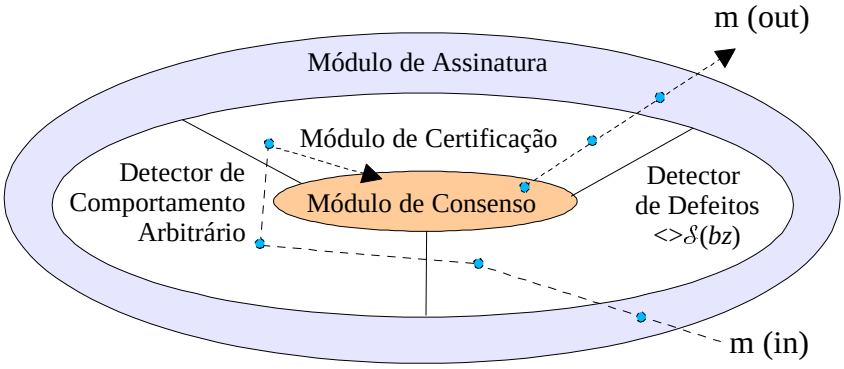

FIGURA 4.10 - Estrutura de um processo com $\langle\rangle\mathcal{S}(bz)$

Os modelos apresentados para os detectores bizantinos são, de uma forma geral, mais independentes (em relação à cooperação com outros módulos de um sistema) do que o modelo apresentado anteriormente para uma detecção em ambientes sujeitos a falhas por omissão. Essa é uma característica desejada, mas que pode acabar significando pouco face à complexidade adicional que o ambiente exige para a construção dos demais protocolos e módulos. Não obstante, a busca pela modularidade (como frisa Kihlstrom [KIH 97], ao defender a separação entre o detector de defeitos e o detector de comportamento bizantino) possibilita uma melhor visualização das relações entre os componentes do sistema e facilita a comprovação das suas funcionalidades.

4.2.7 Detector de defeitos probabilístico Gossip

Como foi apresentado no capítulo 2, a Impossibilidade FLP expressa a impossibilidade de solucionar um problema de consenso de forma determinística, baseando-se apenas nas informações obtidas através da comunicação em um sistema distribuído totalmente assíncrono. De fato, o modelo de detectores de defeitos apresentado por Chandra e Toueg, considerado determinístico, explora apenas uma parte dos casos possíveis em um sistema assíncrono (mas que na prática podem ser considerados como a grande maioria). Além disso, uma implementação prática de tais detectores costuma utilizar *timeouts*, de forma que uma análise mais crítica pode considerar os detectores de Chandra e Toueg apenas como próximos de um determinismo (com uma boa chance de acerto).

Existem outros modelos de detecção de defeitos que não tentam ser tão exigentes quanto ao determinismo, prestando mais atenção a outras características interessantes para um sistema distribuído, como o número de mensagens trocadas, e à iniciativa da tomada de decisões quando as informações são insuficientes para fazer um julgamento da situação.

Guo [GUO 98], ao estudar protocolos de estabilização de mensagens (informalmente, estabilização é a determinação de quais mensagens já foram recebidas por todos os processos, para serem removidas dos *buffers*), identificou que a maioria dos detectores de defeitos costuma tornar-se excessivamente lenta ou imprecisa, quando o número de processos monitorados aumenta muito. Isso se deve sobretudo à grande sobrecarga gerada pelas inúmeras mensagens de monitoramento que trafegam pela rede. Assim, a criação de um modelo de detectores de defeitos que reduzisse o número de mensagens, ao agregar mais informações em um mesmo pacote, e a associação a um modelo probabilístico capaz de atender à maioria das situações enfrentadas pelo sistema distribuído, fizeram com que um protocolo conhecido como *Gossip* (que pode ser traduzido como "fofoca"), descrito e utilizado por diversos pesquisadores desde a década de 70, fosse adaptado para a detecção de defeitos em sistemas distribuídos, servindo também para a disseminação das informações de cada detector.

O mecanismo do protocolo *Gossip* faz com que cada membro transmita novas informações para um outro membro aleatoriamente escolhido, de tempos em tempos. O conjunto dos membros escolhidos aleatoriamente em cada rodada de detecção (chamada de *step interval*) é conhecido como o *gossip subset*. As rodadas de detecção são divididas em fatias de tempo semelhantes.

Cada membro mantém uma lista de processos "L" que indica o grau de "vida" desses elementos, e onde valores mais próximos a zero indicam que o membro está mais ativo. A lista L é inicialmente preenchida com o número zero em cada campo. A cada etapa de detecção, cada membro *i* incrementa em uma unidade o contador de todos os outros elementos na sua lista de "vivos" L, mantendo L[*i*] = 0, e então envia L para um conjunto de processos randomicamente escolhidos.

Quando recebe uma mensagem de dados de outro processo *j*, um membro define L[*j*] = 0. Além disso, quando recebe uma mensagem de detecção contendo uma lista L' de processos, o membro substitui os campos relativos aos outros membros com o valor mínimo obtido da comparação entre sua própria lista L e a lista L' recebida do processo *j*. Um algoritmo baseado no pseudo-código de Guo, e que representa essas operações do protocolo *Gossip*, está representado na fig. 4.11.

```
Inicialização:
  Li ← [0 0 ... 0];

cobegin
   Task 1:
     periodically do
       Li[j] = Li[j] + 1 for all j ≠ i
       send Li to a randomically chosed member

  Task 2:
     when receive(data), do
       Li[j] = 0;
     when receive(L'), do
        Li = ArrayMin (Li, L');
coend
```

FIGURA 4.11 - Detector de defeitos *Gossip*

Como leva algum tempo para que uma lista L de um membro seja propagada através de todo o grupo dos processos, deve-se estabelecer um limite após o qual um processo é considerado falho. Este limite, denominado K_{fail}, representa o valor máximo que o campo correspondente a um processo pode alcançar na lista L, antes que o processo seja considerado falho. A determinação do valor de K_{fail} depende de diversos fatores, principalmente da taxa de propagação (a duração de cada intervalo de detecção) e do número de processos notificados a cada passo. Assim, o valor de K_{fail} é escolhido para que a possibilidade de que algum processo cometa uma detecção errada seja menor do que um limite máximo de erros $P_{mistake}$.

A figura 4.12 mostra um exemplo de operação do detector *Gossip*, considerando que apenas um processo randomicamente escolhido é notificado por vez. A lista L apresentada à esquerda da figura mostra a visão do processo A no decorrer do tempo.

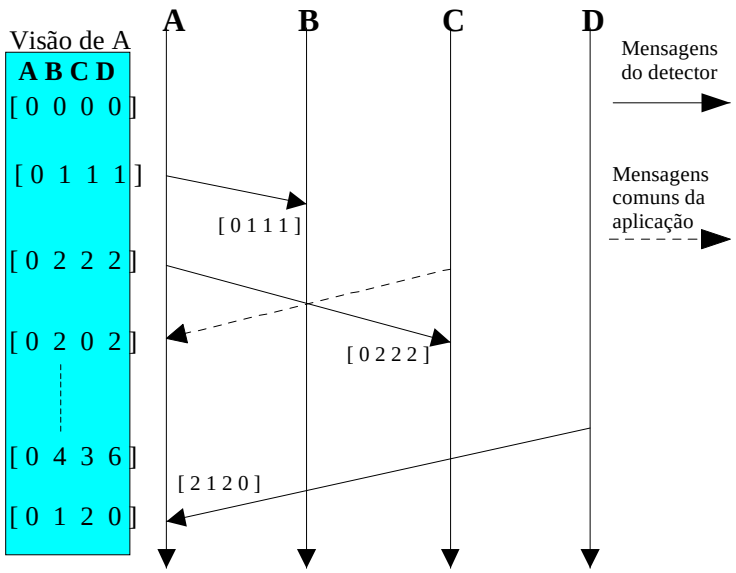

FIGURA 4.12 - Detector *Gossip* em ação

Segundo Guo, uma análise estatística realizada sobre este protocolo por van Renesse *et al.* (*apud* [GUO 98]), onde apenas um membro divulga sua lista a cada passo e apenas um outro membro é escolhido para recebê-la, o crescimento do número de mensagens é logarítmico, e K_{fail} cresce na ordem de n log n. Somente esses valores já representam uma grande economia no número de mensagens trocadas, pois o modelo "todos para todos" do detector de Chandra tem um crescimento exponencial. Mesmo assim, Guo cita, como possibilidade para reduzir o número de mensagens, a adoção de uma estrutura hierárquica.

4.2.8 Detector de defeitos com randomização

Os modelos de detectores de defeitos apresentados por Chandra e Toueg permitem que estes cometam um infinito número de enganos, desde que suas propriedades sejam respeitadas. Acima de tudo, um detector deve preservar a segurança (*safety*) do sistema, e mesmo quando encontra-se totalmente enganado, não pode deixar que operações inconsistentes venham a ocorrer (no caso de um consenso, dois processos não podem decidir valores diferentes).

É comum e razoável esperar que o sistema se comporte de acordo com suas especificações durante a maior parte do tempo, e o detector de defeitos possa ser considerado suficientemente correto. Graças a isso, um consenso auxiliado por um detector de defeitos pode ser finalizado em poucas rodadas assíncronas. Quando o sistema distribuído se depara com períodos instáveis, onde o detector perde sua precisão, o algoritmo de consenso pode interromper seu progresso até que este período ruim seja ultrapassado. Como normalmente os períodos bons são longos, e os maus períodos tendem a ser curtos e raros, esse atraso pode ser tolerado, mas se for considerada a possibilidade de períodos ruins com longa duração, essa espera pode tornar-se crítica para o sistema.

Aguilera e Toueg [AGU 96] propuseram um algoritmo misto para resolver o consenso, de forma que este seja finalizado rapidamente quando o detector de defeitos estiver correto, mas que também possa fazer progressos e finalizar mesmo que o detector esteja passando por um período de imprecisão. Isso é obtido ao utilizar, associado ao detector de defeitos, um módulo de randomização (chamados pelos autores de FD-Oracle e R-Oracle, respectivamente) que assume as tarefas do detector quando este não consegue manter sua precisão. Quando é percebida uma

situação onde o detector de defeitos se enganou (por exemplo, se o número de respostas recebidas não é o suficiente para determinar a maioria necessária à decisão sobre um valor), o módulo de randomização é requisitado e sobre seus resultados são decididos os próximos passos que o algoritmo de consenso dará.

Segundo análise dos autores da proposta, o algoritmo de consenso misto é robusto o suficiente para seguir seu funcionamento mesmo quando o detector de defeitos se encontra impossibilitado de realizar uma previsão confiável. Isso deve-se ao módulo de randomização, que consegue chegar ao consenso com probabilidade 1. Entretanto, tal protocolo é extremamente demorado na prática, e por isso os autores propuseram uma versão otimizada do protocolo de consenso para que somente em casos de falha do detector seja necessário aliar o uso da randomização para resolver o consenso.

4.3 Considerações sobre os Detectores

Um detector de defeitos, como foi apresentado até agora, é um programa praticamente independente, cujo único objetivo é fornecer informações sobre os processos, que poderão ser considerados falhos ou não. Embora essa visão represente a idéia primordial de um detector de defeitos, ela não é suficiente para expressar o seu papel junto a uma ferramenta de comunicação de grupo. De fato, somente pode-se definir quais as características que um detector de defeitos deve conter e fornecer, se for tomada por base a necessidade dos demais módulos do sistema, pois embora tenham sido apresentados modelos de detectores com diversas propriedades, até esse momento não há algoritmo mais ou menos capacitado para realizar uma tarefa. O capítulo a seguir propõe a revisão do papel do detector de defeitos dentro de tais componentes do sistema, sobretudo a forma com que um detector é utilizado para fornecer seus serviços. É com base em tais observações que será formada a estrutura abstrata do protótipo que será implementado.

5. ONDE UTILIZAR OS DETECTORES

Até o presente momento, os detectores de defeitos foram definidos como unidades autônomas, cujo único objetivo é monitorar o estado de outros processos, e fornecer uma lista de suspeitos de falha. Olhando dessa maneira, um detector aproxima-se de outras ferramentas usadas, por exemplo, para a gerência de uma rede, informando quais as máquinas estão com problema.

É justamente na integração ao contexto de uma ferramenta de comunicação de grupo (CG) que os detectores de defeitos se tornam elementos importantes e, de certa forma, inovadores, considerando que a definição do uso de detectores é recente, e que ainda existem ferramentas de comunicação de grupo que não os utilizam.

Os artigos que apresentam os detectores de defeitos não costumam relacioná-los com os demais componentes de uma ferramenta de CG, exceto talvez os algoritmos de consenso, e informações mais detalhadas sobre essa integração são apenas citadas nos trabalhos que apresentam tais ferramentas. De fato, é até possível questionar se os detectores de defeitos realmente existem como um módulo auxiliar ou se estão diluídos pelo código das ferramentas de CG, inseridos nas camadas de comunicação ou de *membership* [NUN 99].

Este capítulo tem por objetivo revisar a bibliografia, em busca dos "pontos de conexão" em que um módulo de detecção de defeitos deve ser encaixado, pois somente com a posse de tais informações será possível estabelecer qual é o modelo mais adequado de detectores de defeitos para os trabalhos do Grupo de Tolerância a Falhas da UFRGS, e quais os serviços que este deverá prover, na possibilidade de participar de uma ferramenta de comunicação de grupo.

5.1 Presença de Detectores nas Propostas Estudadas

Devido à importante relação que une o consenso e um detector de defeitos, tomou-se como ponto inicial desta pesquisa os próprios algoritmos de consenso construídos para utilizar os detectores. A partir desses algoritmos, já se pode começar a formar a idéia de qual a maneira mais comum de inserção dos detectores e, especialmente, qual a freqüência desse uso. Para facilitar a exposição, serão apresentados os algoritmos de consenso construídos com os detectores de defeitos, na mesma ordem em que foram demonstrados os detectores do capítulo 4.

5.1.1 Consenso com detectores $\mathcal{S}$ e $\langle\rangle\mathcal{S}$

Chandra e Toueg, no mesmo trabalho em que propuseram o conceito de detectores de defeitos [CHA 96a], apresentaram dois exemplos de algoritmos de consenso onde são utilizados os detectores. O primeiro modelo utiliza detectores S*trong* ($\mathcal{S}$), conforme a fig. 5.1, e nela pode ser observado que o detector de defeitos é utilizado em duas fases do consenso de modo a evitar que o algoritmo fique esperando por tempo indeterminado uma mensagem vinda de um processo falho. Isso é obtido ao fazer com que todos os processos participantes do consenso, ou tenham enviado a mensagem, ou sejam considerados suspeitos pelo detector.

Every process p execute the following:

procedure *propose*(v_p)
  $V_p \leftarrow \langle \perp, \perp, ..., \perp \rangle$ {p's estimate of the proposed values}
  $V_p[p] \leftarrow v_p$
  $\Delta_p \leftarrow V_p$

  Phase 1: {asyncronous round r_p, $1 \leq r_p \leq n$-1}
    for $r_p \leftarrow 1$ to n-1
      send (r_p, Δ_p , p) to all
      wait until [$\forall q$: *received*(r_p, Δ_q , q) **or** $q \in \mathcal{D}_p$] {query the failure detector}
      $msgs_p[r_p] \leftarrow \{(r_p, \Delta_q , q) \mid received(r_p, \Delta_q , q)\}$
      $\Delta_p \leftarrow \langle \perp, \perp, ..., \perp \rangle$
      for $k \leftarrow 1$ to n
        if $V_p[k] = \perp$ **and** $\exists(r_p, \Delta_q , q) \in msgs_p[r_p]$ with $\Delta_p[k] \neq \perp$ **then**
          $V_p[k] \leftarrow \Delta_q[k]$
          $\Delta_p[k] \leftarrow \Delta_q[k]$

  Phase 2:
    send V_p to all
    wait until [$\forall q$: *received* V_q **or** $q \in \mathcal{D}_p$] {query the failure detector}
    $lastmsgs_p \leftarrow \{V_q \mid received\ V_q\}$
    for $k \leftarrow 1$ to n
      if $\exists\ V_q \in lastmsgs_p$ with $V_q[k] = \perp$ **then** $V_p[k] \leftarrow \perp$

  Phase 3: *decide* (first non $\perp$ component of V_p)

FIGURA 5.1 - Resolvendo consenso com o uso de um detector $\mathcal{S}$

O segundo algoritmo, que usa detectores do tipo *Eventually Strong* ($\langle\rangle\mathcal{S}$) e rotação de coordenadores, trabalha um pouco diferente. Em cada operação de consenso invocada pela aplicação, um coordenador previamente conhecido (através de r_p mod n+1, onde r é o identificador de seqüência da operação atual do algoritmo, chamado "*round*" ou rodada) recebe as mensagens de sugestão dos processos, e quando obtém (n+1)/2 respostas, toma uma decisão, notificando aos demais processos. Os detectores de defeitos são usados então pelos processos que esperam a resposta do coordenador e, havendo uma suspeita de falhas deste, evita-se uma espera por tempo indeterminado. Este algoritmo é apresentado na fig. 5.2.

Every process p executes the following:

procedure *propose*(v_p)
  $estimate_p \leftarrow v_p$ {$estimate_p$ is p's estimate of the decision value}
  $state_p \leftarrow undecided$
  $r_p \leftarrow 0$ {r_p is p's current round number}
  $ts_p \leftarrow 0$ {ts_p is the last round in which p updated $estimate_p$, initially 0}

  {Rotate through coordinators until decision is reached}

  while $state_p = undecided$
  $r_p \leftarrow r_p + 1$
  $c_p \leftarrow (r_p \bmod n) + 1$ {c_p is the current coordinator}

    Phase 1: {All processes p send $estimate_p$ to the current coordinator}
      send (p, r_p, $estimate_p$, ts_p) to c_p

    Phase 2: {The current coordinator gathers (n+1)/2 estimates and proposes a new estimate}
    if $p = c_p$ **then**
      wait until [for $\lceil (n+1)/2 \rceil$ processes q : received (q, r_p, $estimate_q$, ts_p) from q]
      $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received (q, r_p, $estimate_q$, ts_q) from $q\}$
      $t \leftarrow$ largest ts_p such that (q, r_p, $estimate_q$, ts_q) $\in msgs_p[r_p]$
      $estimate_p \leftarrow$ select one $estimate_q$ such that (q, r_p, $estimate_q$, ts_q) $\in msgs_p[r_p]$
      send (p, r_p, $estimate_p$) to all

    Phase 3: {All processes wait for the new estimate proposed by the current coordinator}
      wait until [received (c_p, r_p, $estimate_{Cp}$) from c_p or $c_p \in \mathcal{D}_p$] {query the failure detector}
      if [received (c_p, r_p, $estimate_{Cp}$) from c_p] **then** {p received $estimate_{Cp}$ from c_p}
        $estimate_p \leftarrow estimate_{Cp}$
        $ts_p \leftarrow r_p$
        send (p, r_p, *ack*) to c_p
      else send (p, r_p, *nack*) to c_p {p suspects that c_p crashed}

    Phase 4: {The current coordinator waits for $\lceil (n+1)/2 \rceil$ replies. If they indicate that $\lceil (n+1)/2 \rceil$ processes adopted its estimate, the coordinator R-broadcasts a decide message}
    if $p = c_p$ **then**
    wait until [for $\lceil (n+1)/2 \rceil$ processes q : received (q, r_p, *ack*) **or** (q, r_p, *ack*)]
    if [for $\lceil (n+1)/2 \rceil$ processes q : received (q, r_p, *ack*)] **then**
      R-broadcast (p, r_p, $estimate_p$, *decide*)

{if p R-delivers a decide message, p decides according}

when *R-deliver*(q, r_q, $estimate_q$, *decide*)
  if $state_p = undecided$ **then**
    decide($estimate_q$)
    $state_p \leftarrow decided$

FIGURA 5.2- Resolvendo consenso usando um detector $\langle\rangle\mathcal{S}$

5.1.2 *Heartbeat*

Como a principal preocupação dos autores do *Heartbeat* é prover comunicação quiescente confiável, ao invés de mostrar apenas o uso dos detectores no problema do consenso, são apresentadas soluções para protocolos de comunicação: uma primitiva de *send* e *receive* confiável e um *broadcast* confiável [AGU 97a].

O SEND e o RECEIVE (identificados dessa forma para não haver confusão com as primitivas de *send* e *receive* tradicionais) respeitam a característica da comunicação quiescente e, por causa disso, são classificados como "quase" confiáveis, pois o processo irá retransmitir a mensagem até ela chegar, mas o ambiente de falhas supõe que essa situação pode não se estender para sempre. De fato, a principal propriedade das primitivas SEND e RECEIVE, a ser respeitada além da Integridade, é a *Quasi No Loss* ("quase sem perdas"). Essa propriedade diz que, se há dois processos corretos *r* e *s*, e *s* envia exatamente *k* vezes uma mensagem, *r* irá receber pelo menos *k* vezes a mensagem *m*. Um algoritmo para essas primitivas é mostrado na fig. 5.3.

```
Process s
Inicialization:
  seq ← 0                                  {número de seqüência da mensagem}

  To execute SEND_{s,r}(m):
    seq ← seq + 1
    fork task repeat_send(r, m, seq)
    return

  task repeat_send(r, m, seq):
    prev_hb_r ← -1
    repeat periodically
      hb ← D_s                             {pergunta ao detector Heartbeat}
      if prev_hb_r < hb[r] then
        send_{s,r}(MSG, m, seq)
        prev_hb_r ← hb[r]
    until receive_{s,r}(ACK, seq) from r

Process r

  upon receive_{r,s}(MSG, m, seq) do
    if this is the first time r receive(MSG, m, seq) from s then RECEIVE_{r,s}(m)
    send_{r,s}(ACK,seq)
```

FIGURA 5.3 - *quasi reliable* SEND e RECEIVE

Estas primitivas não são suficientes para os casos de redes parcialmente conectadas ou com caminhos unidirecionais, pois tentam realizar uma comunicação ponto-a-ponto, e somente retornam quando a operação é finalizada. A outra alternativa de comunicação confiável, e que resolve o problema acima, envolve o uso de um *broadcast* confiável, que é capaz de difundir uma mensagem e garantir que todos os processos conectados a recebam (fig. 5.4).

Em ambos algoritmos (SEND/RECEIVE e *broadcast*/*deliver*), os detectores de defeitos são utilizados para determinar se um processo *p* deve continuar sendo contactado, o que reduz consideravelmente o número de mensagens enviadas. Além disso, um processo não precisa ficar controlando se todos receberam sua mensagem, pois a característica do algoritmo (todos para

todos), associada à informação contida no campo *path* da mensagem, indica que todos algoritmos não falhos irão receber as mensagens de algum processo vizinho.

Quanto à solução para o consenso, os trabalhos de Aguilera que apresentam o detector *Heartbeat*, só mostram um algoritmo que usa detectores $\langle\rangle\mathcal{S}$, semelhante ao algoritmo de Chandra e Toueg. Tal algoritmo é apresentado junto ao detector *Heartbeat* para redes particionáveis [AGU 97c], mas parece ser completamente adaptável ao modelo mais simples do detector. Nele, as chamadas ao detector só são realizadas quando o algoritmo se encontra esperando mensagens de resposta dos processos, de modo a evitar uma espera indeterminada. Além disso, o algoritmo usa as primitivas de comunicação confiável apresentadas, construindo assim todo um relacionamento entre esses "blocos básicos".

```
To execute broadcast(m):
  deliver(m)
  got[m] ← {p}
  fork task diffuse(m)
  return

task diffuse(m):
  for all q ∈ neighbor(p) do prev_hb[q] ← -1
  repeat periodically
    hb ← D_s                                  {pergunta ao detector Heartbeat}
    if for some q ∈ neighbor(p), q ∉ got[m] and prev_hb[q] < hb[q] then
      for all q ∈ neighbor(p) such that prev_hb[q] < hb[q] do send_p,q(m, got[m],p)
      prev_hb ← hb
  until neighbor(p) ⊆ got[m]


upon receive_p,q(m, got_msg, path) do
  if p has not previously executed deliver(m) then
    deliver(m)
    got[m] ← {p}
    fork task diffuse(m)
  got[m] ← got[m] ∪ got_msg
  path ← path ·p
  for all q such that q ∈ neighbor(p) and q appears at most once in path do
    send_p,q(m, got[m], path)
```

FIGURA 5.4 - *broadcast* e *deliver* usando o detector *Heartbeat*

5.1.3 Detectores para ambientes com omissão

Dolev *et al.* [DOL 97] propuseram dois algoritmos de consenso que usam os detectores $\langle\rangle\mathcal{S}(om)$ sugeridos por eles. O primeiro algoritmo é apenas uma adaptação do algoritmo de Chandra e Toueg, onde a capacidade de suportar omissões no envio e na recepção de mensagens é adicionada através das seguintes ações:

- todos processos mantêm o registro de todas as mensagens que enviaram, e reenviam-nas de "carona" (*piggyback*) nas novas mensagens a serem expedidas;

- todos processos que ainda não decidiram, reenviam periodicamente a última mensagem para todos os outros processos, não importando em qual passo do algoritmo se encontrem. Assim, eles retransmitem as últimas mensagens geradas

mesmo que se encontrem em um período de "pausa", sem operações de consenso no sistema;

- quando um processo recebe uma mensagem repetida, ou mensagens relacionadas a uma rodada anterior do protocolo, descarta essa mensagem;

- quando um processo finalmente chega a uma decisão, ele cessa o envio periódico de mensagens. Assim, quando recebe uma mensagem diferente de *decide*, ele responde enviando um *decide* junto com o valor da sua decisão. Desse modo, membros que ainda não tinham decidido e que se reconectam aos membros que já decidiram recebem uma mensagem *decide*, e o protocolo cessa o envio de mensagens apenas se todos processos corretos chegaram a uma decisão.

Esta solução é considerada pelos próprios autores como ingênua, uma vez que o tamanho das mensagens (e a necessidade de armazená-las) cresce linearmente com o número de processos no sistema, tornando o uso do protocolo proibitivo na prática.

A solução prática para esse problema envolve a adoção de rotação do coordenador e pode ser descrita a partir da fig. 5.5, que representa o diagrama de estados dos processos. O coordenador de uma rodada *r* é previamente conhecido em função do cálculo de *coordenador* = (*r* mod *n*) + 1. O coordenador informa aos processos quando inicia uma nova rodada, e espera até reunir (*n* + 1)/2 propostas com o mesmo valor. Do lado dos subordinados ou *slaves*, estes tentarão responder às mensagens do coordenador, e somente dois eventos irão alterar sua rotina: o recebimento de uma mensagem com o identificador da rodada superior ao seu valor atual, e a suspeita de que o coordenador se encontra falho. Ambos eventos irão determinar que estes processos reiniciem suas operações, através da escolha de um novo coordenador. No caso da suspeita do coordenador atual, o protocolo de consenso baseia-se na definição dos detectores $\langle\rangle\mathcal{S}(om)$, pela qual em um determinado momento haverá um processo correto *u* que não será considerado suspeito por nenhum outro processos correto, e que portanto será escolhido o coordenador.

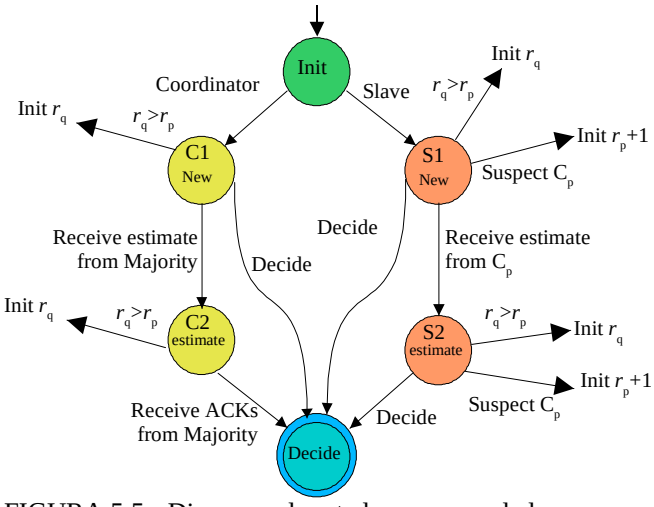

FIGURA 5.5 - Diagrama de estados para a rodada r_p

Nesse algoritmo, portanto, a função do detector de defeitos é informar aos *slaves* quando um coordenador pode estar apresentando falhas por omissão, o qual é um comportamento indesejado se os elementos desejam chegar a um consenso sem que cada processo, *slave* ou coordenador, tenha que retransmitir infinitamente todas as mensagens para os demais membros.

5.1.4 Detectores bizantinos

O uso de detectores de defeitos junto a algoritmos de consenso, em sistemas sujeitos a falhas bizantinas, segue a metodologia proposta por Chandra e Toueg: ou seja, utilizar um detector para "completar" o conjunto de processos do sistema, evitando uma espera indeterminada pelo contato de algum processo falho. Entretanto, como o próprio ambiente de falhas, os algoritmos de consenso apresentam características adicionais com relação ao conjunto de processos do sistema e, sobretudo, com relação à própria detecção dos defeitos.

Kihlstrom *et al.* [KIH 97] apresentam uma implementação de um algoritmo de consenso que usa os mesmos princípios do detector apresentado na seção 4.2.6, ou seja, autenticação das mensagens e detecção do comportamento bizantino. Todas as operações realizadas sobre o conjunto de processos que realizam o consenso passam pela verificação da integridade e da autenticação das mensagens, descartando mensagens mal-formadas, que acarretariam imprecisões à própria operação do consenso. Além disso, como o modelo de falhas bizantinas com autenticação utilizado permite no máximo (n-1)/3 processos falhos no sistema sem afetar o resultado do consenso, sempre é feito o controle do número de processos corretos, como forma de acelerar o algoritmo, evitando esperar a chegada de informações não mais necessárias à tomada de decisões.

Todo esse comportamento especial com relação às mensagens e ao conjunto de processos também é expresso na hora de controlar a "reunião" das mensagens do consenso. O coordenador espera respostas íntegras e autenticadas de no mínimo $(2n+1)/3$ processos corretos. Se existem mais de $(n-1)/3$ propostas semelhantes, ou seja, não há possibilidade de que todas as mensagens sejam emitidas por processos falhos, que são no máximo $(n-1)/3$, então o coordenador pode escolher esse como o valor de consenso, e notifica sua decisão para todos os processos.

Cada processo, ao receber a mensagem do coordenador, envia uma confirmação para todos os processos. Se reunir $(2n+1)/3$ respostas de confirmação dos seus pares, então os processos decidem aquele valor.

O sistema de detectores de falhas bizantinas proposto por Baldoni *et al.* [BAL 99] apresenta como exemplo de algoritmo de consenso, o protocolo conhecido como Hurfin-Raynal. Este protocolo difere dos demais em alguns aspectos, como, por exemplo, o número de mensagens necessárias para a certificação de uma decisão, (n+1)/2, mas usa rotação de coordenador e autenticação das mensagens como o protocolo de Kihlstrom. Os autores afirmam também que este algoritmo é mais eficiente do que os demais nos casos em que o detector de defeitos não comete enganos (não importando se existem ou não falhas no sistema). Entretanto, o mecanismo de interação do detector de defeitos com o protocolo de consenso é semelhante aos demais, ou seja, o detector é utilizado para verificar se o coordenador da rodada não está falho, ocasionando a escolha de um novo coordenador.

5.1.5 Detectores não-determinísticos

O detector *Gossip,* proposto por Guo [GUO 98], apresentado na seção 4.2.7 e utilizado para a construção de algoritmos de estabilidade de mensagens, não contém nenhum algoritmo de consenso, pois esse não é seu objetivo principal, embora sugira que certas estruturas de dados mantidas pelo protocolo de estabilização de mensagens podem ser utilizadas como uma fonte adicional de informações para o sistema, integrando-se assim ao detector de defeitos. Essa estrutura de dados contém, em cada membro do grupo, um mapa que guarda toda a trajetória das mensagens até aquele processo, podendo-se assim identificar processos ativos, de forma indireta, semelhantemente ao registro do *path* utilizado pelo detector *Heartbeat.*

O detector de defeitos híbrido, apresentado por Aguilera e Toueg utiliza tanto o detector de defeitos como o módulo de randomização, conforme a fig. 5.6 [AGU 96]. A função principal do módulo de randomização é a de sugerir um valor novo e aleatório (entre 0 e 1) quando dois ou mais processos propõem valores diferentes — se um processo recebe uma proposta diferente da sua, envia uma mensagem de indecisão ("?"), em vez de sua decisão. As mensagens são identificadas com as letras R, P, S ou E, que significam respectivamente, *reports* (notificações), *proposals* (propostas), *suggestions* (sugestões, dos processos para o coordenador) e *estimates* (estimativas, do coordenador para os processos).

Every process p executes the following:

procedure *consensus*(v_p) {v_p é o valor inicial do processo p}
  $x \leftarrow v_p$ {x is p's current estimate of the decision value}
  $k \leftarrow 0$

  while true **do**
    $k \leftarrow k + 1$ {k is the current phase number}
    $c \leftarrow p_k \bmod n$ {c is the current coordinator}
    send (R, k, x) to all processes

  wait for messages of the form (R, k, *) **from** n-f processos {"*" pode ser 0 ou 1}
  if received more than n/2 (R, k, v) with the same v
  then send (P, k, v) to all processes
  else send (P, k, *?*) to all processes

  wait for messages of the form (P, k, *) from n-f processes {"*" pode ser 0, 1 ou ?}
  if received at leat f+1 (P, k, v) com o mesmo valor $v \neq ?$
    then decide v
  if at least one (P, k, v) with $v \neq ?$ **then** $x \leftarrow v$ **else** $x \leftarrow ?$
  send (S, k, x) to c

  if $p = c$ **then**
    wait for messages of the form (S, k, *) from n-f processes
    if received at least one (S, k, v) with $v \neq ?$
      then send(E, k, v) to all processes
      else
        random_bit $\leftarrow$ R-Oracle {query R-Oracle}
        send (E, k, *random_bit*) to all processes

  wait until receive (E, k, *v_coord*) from c **or** $c \in$ FD-Oracle {query FD-Oracle}
  if received (E, k, *v_coord*)
    then $x \leftarrow$ *v_coord*
    else if x = ? **then** $x \leftarrow$ R-Oracle {query R-Oracle}

FIGURA 5.6 - Algoritmo híbrido para consenso

5.2 Detectores de Defeitos na Estrutura de um CG

A seção 5.1 tratou sobre a maneira pela qual os detectores de defeitos contribuem para os algoritmos de consenso. Essa é uma boa forma de compreender quais são as exigências para com um detector, uma vez que a freqüência e a forma com que seus dados são obtidos podem determinar a escolha de uma ou outra implementação. Para atingir esse objetivo, entretanto, resta

analisar ainda as relações dos detectores de defeitos com os outros módulos de uma ferramenta de Comunicação de Grupo (CG), estabelecendo uma visão sobre a contribuição dos detectores sobre todo o sistema.

A descrição das relações entre os componentes freqüentemente é esquecida, sendo que muitas vezes apenas um modelo conceitual é apresentado, sem compromisso com a definição correta das dependências entre esses módulos. Um exemplo dessa forma de apresentação é o "Lego®" que caracteriza a ferramenta de Comunicação de Grupo Horus. Ela consegue apresentar claramente a noção de módulos encaixáveis segundo a necessidade, mas não fornece mais nenhuma informação. De fato, a falta de detalhamento nessas especificações freqüentemente leva a um questionamento sobre a real modularidade dos componentes, uma vez que o modelo abstrato pode servir apenas para classificar melhor as idéias, mas os componentes podem estar distribuídos pela aplicação para melhorar o desempenho [NUN 99].

Uma exceção a esse comportamento é encontrada nos trabalhos do grupo da *Ecóle Polytechnique Fédérale de Lausanne*, na Suíça, orientados por Guerraoui e Schiper. Seus trabalhos costumam incluir uma descrição sobre a interação dos diversos módulos que compõem o sistema, e essas conclusões serão tratadas aqui.

5.2.1 Relacionamento entre os módulos

O trabalho mais completo encontrado, no sentido dessa descrição do papel de um detector de defeitos em uma ferramenta de CG, é integrante da tese de Garbinato [GAR 98], que apresenta uma proposta de estruturação através do uso de *design patterns* e a abstração possibilitada pela orientação a objetos, de modo a construir sistemas distribuídos confiáveis. Uma das primeiras considerações de Garbinato refere-se ao relacionamento entre os diversos componentes, identificando as relações de Problema→Solução. A fig. 5.7 demonstra essas ligações, no caso de uma ferramenta de CG.

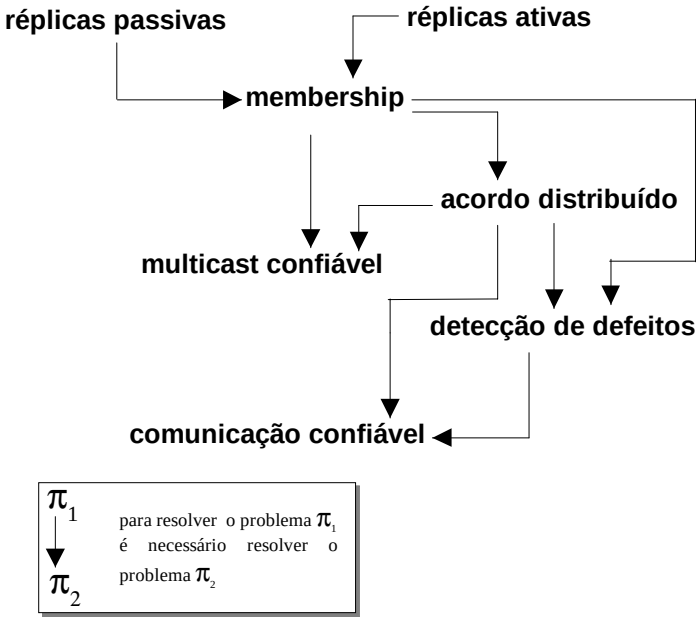

FIGURA 5.7 - Relação entre pares Problema-Solução

Felber, também do mesmo grupo da Suíça, representou as dependências entre os componentes da sua ferramenta de CG em CORBA [FEL 98] como uma pilha de serviços

construídos sobre o nível de transmissão representado pelo *Object Request Broker* (ORB) do CORBA, conforme a fig. 5.8. Os serviços necessários à ferramenta de CG foram agrupados em módulos que representam funcionalidades semelhantes, conforme a relação abaixo:

- ***Messaging Service*** - comunicação confiável ponto-a-ponto e comunicação por *multicast.*
- ***Monitoring Service*** - usa comunicação confiável para detectar defeitos nos objetos.
- ***Consensus Service*** - usa detecção de defeitos e comunicação confiável para resolver um consenso distribuído.
- ***Group Membership*** - utiliza detectores de defeitos para monitorar os membros dos grupos, e um protocolo de consenso para realizar os acordos sobre as novas visões.
- ***Group Multicast*** - usa comunicação confiável, consenso e *membership* para garantir a entrega atômica para todos os membros do grupo.

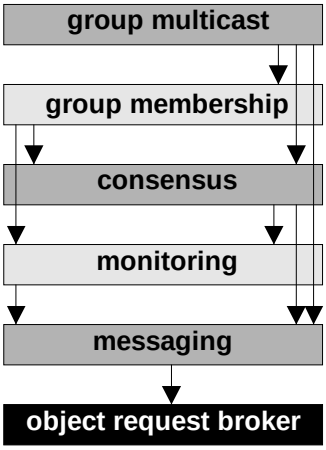

FIGURA 5.8 - Dependência entre os módulos do OGS

Um fato importante a notar nessas duas abordagens é que elas citam a necessidade da comunicação confiável para a detecção de defeitos. No conceito dos detectores de defeitos de Chandra e Toueg, a comunicação é implicitamente confiável devido à definição das linhas de comunicação, mas é questionável o uso de uma camada adicional de comunicação confiável, pois, como no caso do OGS, o protocolo TCP sobre o qual funciona o ORB já provê garantias de entrega. De fato, como mostra a seção 4.1, Felber especifica (no mesmo trabalho, [FEL 98]) os detectores de defeitos sem a necessidade de uma camada adicional de comunicação confiável, e Aguilera consegue determinar um detector de defeitos onde diversas mensagens podem ser perdidas sem prejuízos para a detecção. A polêmica em torno desse assunto foi levantada em uma discussão interna ao Grupo de Tolerância a Falhas da UFRGS [EST 99], que concluiu ser desnecessária a comunicação confiável como suporte à detecção de defeitos.

Garbinato também representou a pilha de protocolos do sistema Bast, e nela pode ser identificado também o nível de interação do módulo detector de defeitos com o nível de comunicação de rede (fig. 5.9). Esse relacionamento é importante, pois comumente há dúvidas quanto à essa integração, sobretudo pela possibilidade de integrar um detector de defeitos à própria camada de comunicação, obtendo dessa maneira informações sobre os processos a partir de qualquer mensagem recebida.

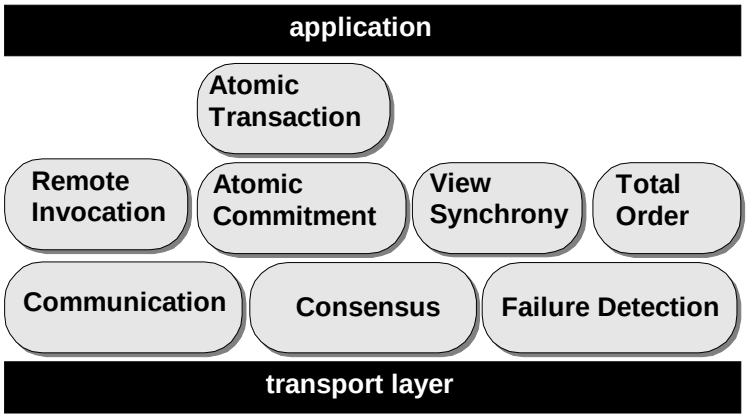

FIGURA 5.9 - Visão parcial da arquitetura Bast

5.2.2 Acesso à lista de suspeitos

Outro aspecto, já bem mais interno ao projeto de um detector de defeitos, refere-se à forma com que os outros módulos irão obter a lista de suspeitos fornecida pelo detector.

Quando se imagina um detector de defeitos apenas como uma entidade independente, é suficiente supor que a aplicação que necessita das informações sobre os processos irá invocar um método que retorna a lista de suspeitos. Uma análise superficial dos protocolos de consenso da seção 5.1 também parece reforçar essa possibilidade, uma vez que o consenso pode pedir ao detector de defeitos quais são os processos suspeitos de haverem falhado, continuando seu processamento. Entretanto, isso pode não ser correto na prática, pois quando o protocolo de consenso espera a resposta de todos os processos, possivelmente terá sua linha de execução (assumindo que sejam utilizadas *threads* para obter uma programação concorrente mais eficaz) suspensa até que um evento significativo a acorde. Esses eventos ou podem ser como o recebimento das mensagens que estão sendo esperadas, ou também podem ser como a notificação dos processos considerados falhos. De fato, imaginando um cenário onde o algoritmo de consenso entra em *wait* após receber uma mensagem (que era a última mensagem de processos corretos, embora o algoritmo não possa saber disso) somente o detector de defeitos, quando perceber a falha dos outros processos, poderá acordar o protocolo de consenso e evitar que este fique para sempre na espera.

Outra situação que comprova tal necessidade de eventos gerados pelo módulo detector de defeitos é o próprio protocolo de *membership*. Não é interessante, por ser custoso e pouco preciso, deixar que o *membership* pergunte periodicamente quais são os processos ativos dentro do grupo. A maneira mais econômica e precisa para isso é a implementação de *callbacks*, que são ativados apenas quando há a necessidade de informar o módulo superior da mudança de estado na lista dos processos suspeitos. Essa informação fornecida assincronamente aos módulos superiores é bastante utilizada pelo sistema Bast. Conforme a fig. 5.10 demonstra, diversas classes apresentam operações de *callback*, ou seja, a estrutura da ferramenta de CG possibilita um caminho bidirecional para a tomada de ações, fazendo com que independentemente da origem dos eventos, o sistema reaja a esses estímulos.

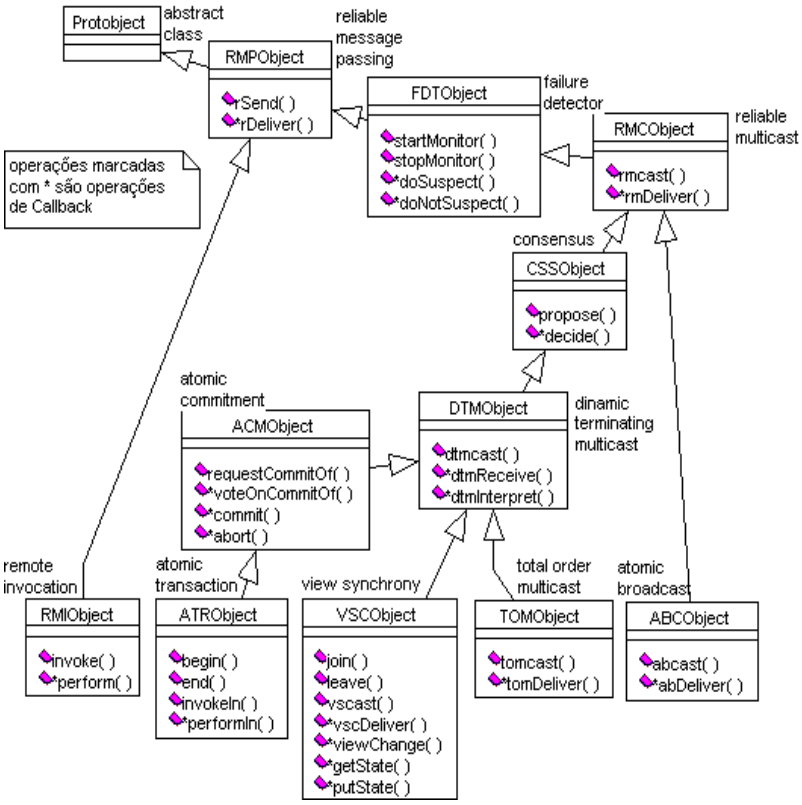

FIGURA 5.10 - Relacionamento entre algumas classes do Bast

5.2.3 Granularidade da detecção de defeitos

Essa questão refere-se ao nível de detecção que o detector de defeitos irá conduzir. Freqüentemente, a própria restrição do modelo de falhas do sistema é suficiente para restringir as possibilidades, ou seja, se for utilizado um modelo *fail-stop*, por exemplo, considera-se que todos os processos de uma determinada máquina irão falhar quando a máquina parar de funcionar (incluindo o detector de defeitos daquela máquina). Com isso, pode-se montar um sistema distribuído que monitore as máquinas da rede, colocando-se um detector de defeitos em cada máquina, que se comunica com os detectores das demais máquinas. Essa estratégia permite também aumentar a transparência do sistema, já que os demais processos que não precisam saber a lista de suspeitos, não necessitam sequer saber da existência do detector de defeitos.

Entretanto, com o uso cada vez mais difundido da orientação a objetos, é tentadora a possibilidade de monitorar cada um dos objetos do sistema, mantendo informações detalhadas sobre ele. Fazendo uma analogia com a área de gerência de redes, onde surgem diversas ferramentas capazes inclusive de avisar ao administrador qual indivíduo está executando um jogo em horário de trabalho, o conhecimento apenas de que a máquina está ligada ou desligada parece insuficiente .

Essa granularidade "extremamente fina" na detecção de defeitos, ou seja, o monitoramento sobre os próprios objetos da aplicação, pode oferecer uma quantidade incrivelmente superior de informações, mas também gera inúmeros problemas a serem resolvidos. A ferramenta de comunicação de grupo OGS [FEL 98] implementa esse tipo de monitoramento, e será utilizada para auxiliar a explicação dos casos a resolver.

Em primeiro lugar, surge a questão sobre como será feita a troca de informações entre os

detectores de defeitos. Não parece viável acrescentar um detector de defeitos "completo" para cada objeto a ser monitorado, tanto pelo custo do processamento, quanto mais pelo custo em comunicação. Se um detector de defeitos tradicional fosse utilizado para cada objeto, este deveria também se comunicar com todos os detectores de defeitos da sua e das outras máquinas do sistema, e, a não ser que fosse feita uma grande otimização no envio das mensagens, estas cresceriam exponencialmente, inundando a rede com seu trafégo. Os protocolos que utilizam as listas de suspeitos também seriam duramente afetados, pois iriam receber informações (possivelmente divergentes) de vários detectores na sua própria máquina, o que exigiria modificar tais protocolos também, para conseguir comportar esse novo ambiente.

O sistema OGS resolve esse problema, ao separar a detecção de defeitos em vários componentes. A fig. 5.11 demonstra a organização das classes, como mostra Felber, e a fig. 5.12 demonstra um exemplo de situação de interação entre estes elementos. Assim, há um objeto `Monitor`, responsável pela suspeita de defeitos (e provavelmente, o único detector na máquina). Cada objeto monitorado implementa apenas uns poucos métodos do objeto `Monitorable`, de forma que ele apenas se comunica com o `Monitor`, enviando periodicamente mensagens de "*I'm alive!*" ou aceitando mensagens de *liveness request* (e por estarem na mesma máquina, podem utilizar formas mais eficientes e econômicas de comunicação do que troca de mensagens). Por fim, há um objeto `Notifiable`, que recebe as suspeitas do `Monitor`, e que deve ser implementado pelos protocolos de consenso, *membership*, ou quaisquer aplicações que utilizarem os detectores. Como o detector de defeitos implementado pelo OGS é do tipo $\langle\rangle\mathcal{S}$, a lista de suspeitos é periodicamente enviada a todos os demais detectores, de forma que mesmo uma máquina remota tenha conhecimento sobre os estados dos objetos locais.

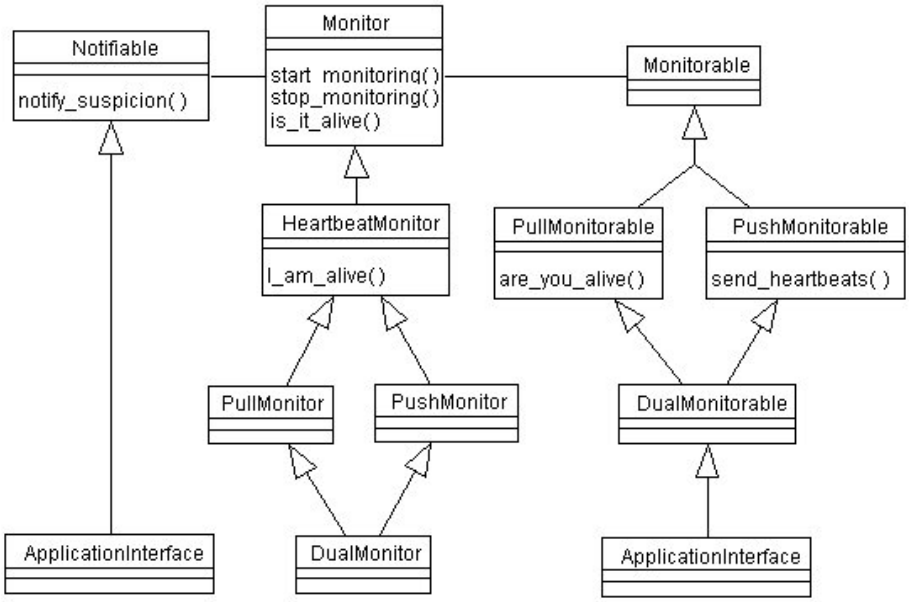

FIGURA 5.11 - Estrutura de detecção do OGS

A presença do detector de defeitos passa a ser percebida pelos processos monitorados, de forma que, mesmo com a redução do número de mensagens que trafegam pela rede, não é vantajoso adicionar objetos `Monitorable` a cada parte do sistema. Assim, somente os elementos estratégicos que realmente necessitem a detecção devem ser monitorados (que possivelmente já vêm com um `Monitorable` integrado) .

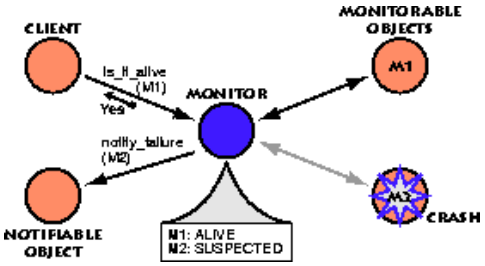

FIGURA 5.12 - Exemplo de detecção de defeitos no OGS

Como as abordagens da detecção tradicional e em objetos são muito diferentes, no que se refere à implementação, fica praticamente impossível adaptar um sistema de detecção tradicional, tanto nos aspectos do custo da comunicação, quanto na perda abrupta da transparência para a aplicação, e a escolha de qual é o modelo a ser utilizado deve também fazer parte da estratégia do projeto do sistema, assim como é o modelo de falhas, por exemplo.

5.3 Intervalo de Amostragem

A construção de detectores de defeitos comumente considera, como solução prática, o uso de *timeouts* para a obtenção de amostras do estado dos processos. Esse intervalo de tempo representado pelo *timeout* é apenas uma aproximação, que tenta encontrar a melhor relação entre a latência e a precisão no julgamento da situação dos processos: se for usado um intervalo de amostragem muito grande, a precisão será boa, mas pode passar muito tempo até que o defeito seja detectado e notificado. Se for utilizada uma detecção excessivamente agressiva, há o risco de que muitos processos sejam erroneamente julgados, e as operações no *membership* necessárias para corrigir esses enganos podem acabar custando muito ao desempenho do sistema.

Mesmo se for utilizado um detector com atualização dinâmica de *timeouts* (tentando adequar a amostragem à velocidade de cada processo), onde os tempos de amostragem tendem a atingir um valor estável, há a necessidade de estabelecer um valor adequado desde o início.

Em um debate realizado na lista de discussões HORUS-L, da Universidade de Cornell [HOR 99] (os *e-mails* estão incluídos no Anexo 2), foi questionada a falta de agressividade na detecção de defeitos, mesmo com as altas velocidades de transmissão que as redes atuais suportam. Mais especificamente, um dos participantes questionava o motivo pelo qual ainda se utilizavam *timeouts* da ordem de 10ms, quando os tempos médios de latência (usando o protocolo VIA) eram de cerca de 50µs. As respostas para essa questão consideraram como o principal fator limitante desse intervalo de amostragem o desempenho reduzido (comparado às novas tecnologias em rede de comunicação) que as placas de rede apresentam, o que, no caso de um *timeout* muito agressivo, iria saturar o *buffer* de armazenamento, gerando perdas de mensagens. Além disso, as operações de renovação da visão e balanceamento de carga entre os processos restantes são muito mais caras ao sistema, e devem ser realizadas apenas quando há uma forte garantia do estado dos processos. De fato, cita-se que os fabricantes de sistemas distribuídos costumam definir intervalos de até sete segundos, devido ao seu conhecimento das características de todos os componentes do sistema e da experiência prática obtida.

Quando essas questões são aplicadas sobre sistemas distribuídos de larga escala (mais de 100 máquinas), é essencial ter garantias quase totais na hora de refazer um *membership*, pois nem mesmo através da separação do *membership* em camadas hierárquicas, o controle dos grupos

costuma ser suficientemente leve.

Infelizmente, tais parâmetros não são fáceis de estabelecer, e somente a experiência da utilização de um detector de defeitos sobre o sistema desejado poderá ajudar a caracterizar tais pontos. No próximo capítulo, será apresentado o protótipo de um detector de defeitos, contruído para que pudesse ser feita uma avaliação mais completa do comportamento de tais sistemas em uma situação prática. Não obstante, como o protótipo foi testado como um elemento único na aplicação, não foi possível prever seu comportamento quando em operação integrado a diversos outras camadas e protocolos da aplicação.

6. PROTÓTIPO DE UM DETECTOR

Os capítulos anteriores demonstraram que os detectores de defeitos são ferramentas de grande importância para a construção de sistemas de comunicação de grupo (CG), e sobre sua capacidade de julgar o estado de outros processos do sistema recai a responsabilidade de fornecer informações corretas e precisas. Embora um detector de defeitos possa cometer enganos, a bibliografia frisa que se deve evitar tais situações, pois suas suspeitas influenciam diretamente vários outros módulos e protocolos do sistema, e o resultado de enganos freqüentes é a degradação do desempenho do sistema. Além disso, também há uma preocupação freqüente com relação à escalabilidade dos sistemas, colocando a transparência e o tráfego de mensagens entre os principais pontos a serem melhorados.

Apesar de toda essa preocupação por parte dos pesquisadores, não há realmente uma publicação, em toda bibliografia consultada, que discorra sobre resultados práticos de um detector de defeitos, nem sobre os obstáculos enfrentados em sua implementação. Este capítulo irá apresentar as lições aprendidas durante a experiência da implementação de um destes detectores, as escolhas que precisaram ser feitas e as observações feitas sobre o funcionamento de uma aplicação de testes.

6.1 Definição do Algoritmo

O primeiro passo para a implementação de um detector de defeitos refere-se à escolha de qual o algoritmo, entre as propostas coletadas no capítulo 4, que oferece as melhores condições de observação e as características mais interessantes. Como não há uma definição sobre o melhor sistema, e de fato, freqüentemente os detectores são escolhidos de acordo com as necessidades da aplicação em que ele será utilizado, foi preciso analisar diversos aspectos e necessidades, a fim de determinar o modelo a ser implementado.

O primeiro fator de escolha considerado, refere-se ao modelo de falhas a ser observado. Pode-se inicialmente supor que um modelo de falhas mais abrangente é o mais indicado, mas isso deve ser avaliado com extrema cautela. De fato, quanto mais abrangente o modelo de falhas, mais complexo é o sistema a ser desenvolvido, e toda a aplicação onde o detector será inserido sofrerá o impacto desta escolha. Assim, para escolher qual o modelo de falhas a ser utilizado, foram verificados se existem outros trabalhos no grupo de Tolerância a Falhas, que poderiam utilizar um detector de defeitos. Embora exista a idéia de construir diversos módulos componentes de um sistema de comunicação de grupo, atualmente há apenas um conjunto de componentes, para comunicação *unicast* e *multicast* confiáveis, criado por Amaral [AMA 00]. Este sistema utiliza como modelo de falhas o formato *crash*, e a partir dele está sendo elaborado um protocolo de consenso, cujo desenvolvimento ainda está na fase inicial. Além deste sistema, existem apenas estudos introdutórios, mas a tendência indica a manutenção dos modelos de *crash* ou *fail-stop*, concentrando-se nos aspectos do comportamento dos sistemas de comunicação de grupo em redes sujeitas a partições.

De acordo com as propostas de detectores de defeitos apresentadas no capítulo 4, somente os modelos "*I am Alive!*" e *Liveness Request* atendem puramente ao modelo de falhas de *crash*. Essas opções, por serem as propostas mais simples e antigas, não trazem nenhuma situação inédita a explorar. Além disso, tais métodos consideram implicitamente a necessidade de um meio de comunicação sem falhas, ou então o emprego de uma comunicação confiável. Isso também não é

interessante, pois segundo as considerações sobre comunicação confiável para detectores de defeitos defendidas pelo grupo de Tolerância a Falhas, um detector de defeitos não necessita comunicação confiável [EST 99], e a comprovação prática desse ponto de vista é um dos objetivos deste trabalho.

A alternativa mais atraente fica, portanto, com o detector *Heartbeat* de Aguilera [AGU 97a]. Este detector considera que os processos podem falhar por *crash*, e que a rede de comunicação pode perder algumas mensagens, sem que isso implique na perda de precisão do detector. Além disso, não há nenhuma referência na bibliografia sobre o comportamento de suas implementações. Já a variação do detector *Heartbeat* para redes particionáveis [AGU 97c] é um pouco mais complexa, e o modelo de testes a ser aplicado a ele é muito mais árduo, dificultando a implementação e teste em tempo hábil. Ainda assim, ambos modelos de detectores são muito similares, e a extensão de um detector para operar em redes particionáveis pode ser uma alternativa para trabalhos futuros.

Outro ponto importante para a escolha de um detector de defeitos é a capacidade de integração com os outros módulos de uma ferramenta de comunicação de grupo. Como apresentado nas seções 4.2 e 5.1, o detector *Heartbeat* tem características próprias, como a maneira de determinar os suspeitos, as informações enviadas em cada mensagem e a redução do número de mensagens, que o torna um pouco diferente do modelo tradicional de detecção de defeitos, e que em um primeiro instante tornam-no incompatível com a saída dos outros detectores, impendindo uma utilização transparente. Este trabalho sugere que uma versão levemente modificada do detector *Heartbeat* é implementável e possibilita essa compatibilidade com os outros módulos, restando a um trabalho futuro comprovar matematicamente que tais modificações no algoritmo são válidas.

6.2 Modificações no detector *Heartbeat*

As modificações realizadas sobre o modelo de Aguilera referem-se ao tratamento com relação aos nós vizinhos, ao processo de suspeita e à maneira com que são notificadas estas suspeitas. Tais modificações aparentemente não alteram as características de funcionamento para o qual foi desenvolvido o *Heartbeat*, e incluem facilidades que permitem uma integração mais transparente e rápida a outros módulos de uma ferramenta de comunicação de grupo. O objetivo principal destas mudanças é prover ao detector uma interface de manipulação bem mais semelhante aos modelos de detectores tradicionais, facilitando a operação e a integração do detector com os outros módulos e protocolos.

6.2.1 Vizinhos

O detector *Heartbeat* proposto por Aguilera considera como vizinhos apenas as máquinas diretamente conectadas, comunicando-se apenas com estas. As demais máquinas da rede podem ser monitoradas através do *path* associado às mensagens trocadas entre os detectores. Entretanto, o modelo do *Heartbeat* não faz isso, e apenas os vizinhos são observados. Tal comportamento só é possível porque a descrição dos detectores *Heartbeat* levou em consideração algoritmos de consenso e difusão confiável construídos especificamente para as facilidades de difusão desse ambiente. A não ser que sejam utilizados esses algoritmos especialmente desenvolvidos, não é possível operar um consenso "tradicional" com tal estrutura, havendo a necessidade de modificar tal comportamento do detector para poder operar com estes protocolos.

As modificações introduzidas mantém a proposta do conjunto de vizinhos, mas realiza a

verificação de estado não somente nestes, mas em todos os processos do sistema, através das informações transmitidas na variável *path* do próprio *Heartbeat*. Essa alteração não modifica o modo de funcionamento do detector, mas como apresenta dados sobre todos os processos do sistema, facilita a operação do detector junto a um módulo de *membership*.

Quanto à granularidade do monitoramento, embora o ambiente de objetos distribuídos possibilitasse um monitoramento individual sobre cada objeto, optou-se por implementar um sistema de monitoramento menos abrangente que o apresentado por Felber para o OGS [FEL 98]. Isso se deve ao fato de que no OGS há perda de transparência, e como o modelo de falhas adotado restringe-se ao *crash* de toda a máquina, ficaria redundante o controle individual de cada objeto. Já a notificação da lista de suspeitos foi feita de forma semelhante ao OGS, uma vez que não há perda da transparência, e um único detector por máquina pode fornecer suas suspeitas a diversos receptores. Assim, o modelo de monitoramento-notificação assemelha-se ao exposto na fig. 6.1, onde os detectores de defeitos trocam mensagens entre si, notificando os objetos interessados nas suas listas de suspeitos.

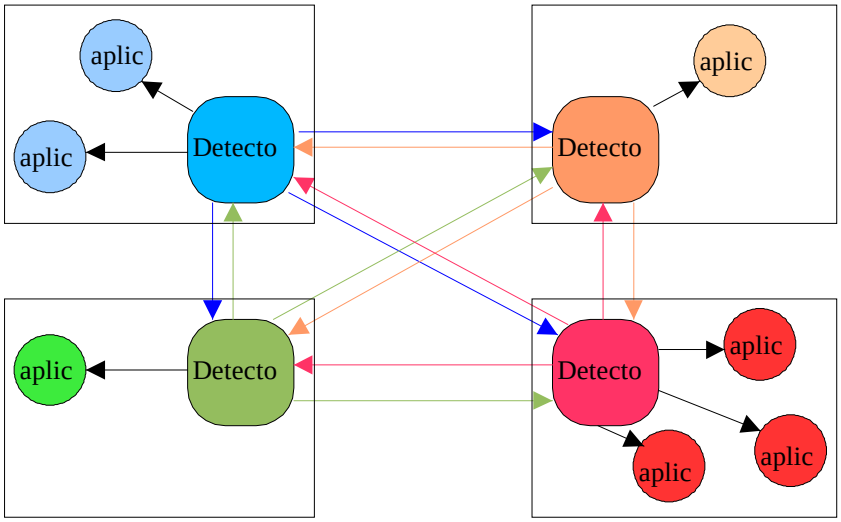

FIGURA 6.1 - Modelo de detecção-notificação

6.2.2 Tempos de amostragem

O detector *Heartbeat* tem como característica a ausência de *timeouts* para o recebimento de mensagens dos demais nós da rede. Ao invés de acusar suas suspeitas quando uma mensagem não chega em um tempo limite, o detector *Heartbeat* fornece apenas o número de mensagens que recebeu de cada nó, cabendo à aplicação (ou à qualquer camada de *software* imediatamente superior) comparar os contadores obtidos em diferentes instantes, à procura de nós que não tiveram seus contadores incrementados. Isso torna a detecção pouco transparente, pois a aplicação também fica encarregada de realizar o processo de suspeita das máquinas.

O algoritmo modificado inclui uma camada adicional específica para esse fim, ou seja, um complemento ao detector *Heartbeat* que realiza periodicamente a verificação de quais nós podem ser considerados suspeitos. Dessa forma, a aplicação apenas recebe uma lista de suspeitos, de maneira bem semelhante ao formato entregue pelos demais detectores de defeitos apresentados no capítulo 4.

6.2.3 Notificação da suspeita

Como o detector *Heartbeat* repassa a responsabilidade da detecção de suspeitos para a aplicação, não há nenhuma especificação quanto à forma com que a aplicação é notificada sobre os processos suspeitos. Conforme demonstrado na seção 5.1, o procedimento mais comum entre os outros detectores é a notificação da aplicação apenas quando há uma nova suspeita. Essa notificação é executada sem uma requisição proveniente da aplicação, exigindo então a utilização de um mecanismo de *callback*, que no caso do protótipo deste trabalho, foi implementado em conjunto com a camada de amostragem.

Não obstante, para evitar restringir as formas de interação com o detector de defeitos, foram incluídos outros mecanismos para a obtenção da lista de suspeitos, além do *callback*. Um destes métodos corresponde à requisição ao detector, por parte de um módulo específico, da lista de suspeitos. Somente o objeto que realizou esta chamada irá receber a lista de suspeitos atualizada, enquanto os demais objetos deverão aguardar até uma nova amostragem, realizada periodicamente. Esse método pode ser utilizado também por elementos que não façam parte do conjunto de "notificáveis", e que apenas ocasionalmente necessitam obter a lista de suspeitos.

Outro método disponível permite que seja disparada uma nova amostragem, fora do momento pré-definido, sob demanda. Este mecanismo pode ser utilizado por elementos que desconfiam de algum processo externo, como por exemplo o módulo de comunicação, e que desejam acelerar a troca da visão do sistema.

6.3 A Estrutura do Detector

O protótipo do detector de defeitos implementado neste trabalho foi estruturado de forma a aproveitar as facilidades oferecidas pelas orientação a objetos e pelo suporte da linguagem Java ao processamento concorrente. Além disso, procurou-se montar uma estrutura adequada não somente ao algoritmo do detector *Heartbeat*, mas capaz de ser reutilizada em outras implementações.

A descrição da estrutura implementada será feita através da escala crescente de abstração, de forma a mostrar como os diversos componentes são integrados para fornecer um detector de defeitos que se conecte de forma transparente à aplicação.

6.3.1 O Nível de Comunicação

Como o modelo de defeitos do *Heartbeat* depende diretamente do nível de comunicação de rede que o suporta, é interessante começar a apresentação do protótipo a partir das operações de *send* e *receive*.

Como afirmado anteriormente, considera-se que o meio de comunicação que interliga os detectores *Heartbeat* pode perder mensagens, embora exista a garantia de que se a mensagem for retransmitida infinitas vezes, chegará ao seu destino. Além disso, não há necessidade de ordenamento de mensagens, pois a detecção é baseada apenas no número de mensagens recebidas. Assim, a escolha do protocolo de transporte de mensagens é muito importante para a correta expressão desse ambiente, e o protocolo UDP, que apresenta características de transporte não

confiável de datagramas serve bem a este propósito, além de encontrar-se disponível em praticamente todos os ambientes de rede.

O envio de mensagens, desempenhado pelo objeto `SendServer`, foi criado seguindo as especificações normais do Java, e não apresentou nenhum problema adicional. As mensagens são constituídas pelo *path* (conjunto de identificadores dos detectores) por onde a mensagem já trafegou, de forma que foi considerado interessante utilizar as operações de serialização de objetos do Java, a fim de adaptar estruturas de dados de alto nível ao envio pela rede. Devido às características da transmissão UDP, foi também necessário definir um tamanho máximo para as mensagens que serão transportadas. Conforme testes realizados, uma mensagem que contenha cerca de 25 a 30 máquinas em seu *path* terá aproximadamente 1024 bytes, sendo esse um tamanho considerado suficiente para o protótipo. Caso necessário, este valor pode ser mudado, bastando alterar a variável `BUFFERSIZE` da classe `RecvServer`, sem nenhum outro impacto no sistema.

A recepção das mensagens envolve a definição do tratamento das mensagens que chegam, e considerou-se uma das seguintes possibilidades:

- a primeira alternativa é o tratamento bloqueante das mensagens, uma a uma, como pode ser visto na fig. 6.2. Essa alternativa é a menos atrativa, pois impede que outras mensagens sejam recebidas enquanto uma está sendo processada. Isso pode levar à perda das demais mensagens que chegam, a não ser, é claro, que os demais emissores fiquem retransmitindo até obter sucesso;

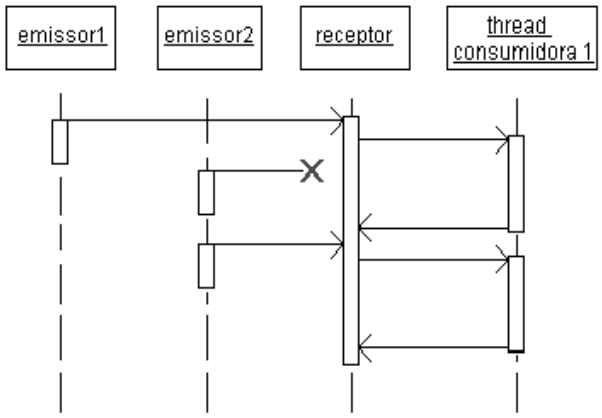

FIGURA 6.2 - Recepção bloqueante

- a segunda alternativa envolve a utilização de um *buffer* de recebimento, onde as mensagens recebidas são tratadas por uma ou mais *threads* paralelas à recepção das mensagens, conforme a fig. 6.3. Neste sistema é minimizada á perda de mensagens devido ao bloqueio do receptor, uma vez que o receptor apenas as armazena em um *buffer*, sem a tarefa de processá-las. O acesso ao *buffer* é disputado pelo receptor e pelas *threads* consumidoras, e para manter a integridade do *buffer*, deve-se sincronizar este acesso, reduzindo o paralelismo no tratamento das mensagens. Ainda assim, é uma alternativa a ser considerada;

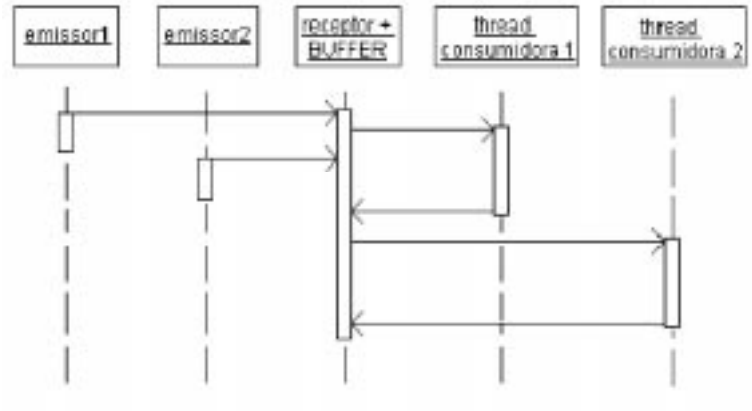

FIGURA 6.3 - Recepção com buffer

- a última alternativa considerada neste trabalho envolve a criação de uma nova *thread* para cada mensagem recebida, como mostra a fig. 6.4. Neste modelo não há perda de mensagens nem a necessidade de um acesso sincronizado a um *buffer*, mas pode haver degradação na memória e no desempenho do sistema, caso o número de *threads* geradas for muito superior ao número de *threads* finalizadas.

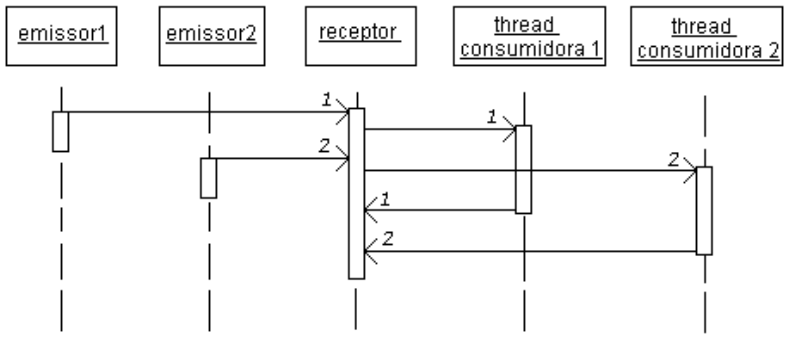

FIGURA 6.4 - Recepção com múltiplas threads

Tanto a segunda quanto a terceira alternativa foram consideradas satisfatórias para as exigências do sistema. Como os testes em uma versão preliminar do detector indicaram que a última alternativa não onerou o sistema de forma perceptível, esta foi selecionada para a construção do detector, sem que a segunda opção fosse implementada. As mensagens recebidas pelo objeto `RecvServer` são repassadas para uma *thread* do tipo `DeliverBoy`. Cada *thread* `DeliverBoy` gerada tem como função desserializar as mensagens, recompondo as estruturas do *path*, e informar seu conteúdo ao mecanismo principal do detector de defeitos, a classe `HBEngine`, através do *callback* `deliver()`. Como podem existir diversas *threads* tentando acessar o mesmo objeto e notificá-lo, haverá uma situação de concorrência onde a ordem de entrega é indeterminada. Isso não causa nenhum problema ao sistema, uma vez que o detector *Heartbeat* não exige ordenamento das mensagens.

6.3.2 O mecanismo principal do detector *Heartbeat*

A classe que implementa as funcionalidades do detector *Heartbeat* é chamada `HBEngine`. Ela é a responsável pela definição do conteúdo das mensagens a serem enviadas, e pelo tratamento

das mensagens recebidas, incluindo o incremento dos contadores relativos a cada objeto monitorado.

Cada objeto monitorado é representado por um objeto do tipo `Member`, e é acompanhado de uma variável que contabiliza o número de mensagens recebidas daquele detector. Um objeto `Member` contém apenas os campos referentes ao nome da máquina, endereço IP e porta de comunicação, e pertence ao conjunto de classes elaboradas por Amaral [AMA 00]. Entretanto, este objeto pode ser facilmente substituído, caso necessário, eliminando assim a ligação entre os dois trabalhos.

Conforme a definição do *Heartbeat*, uma das tarefas da classe `HBEngine` é o envio periódico de mensagens para os demais detectores. Cada mensagem enviada por essa tarefa contém apenas o endereço do próprio detector no *path*. O *path* é representado por uma estrutura `Hashtable` pertencente ao próprio Java, que contém os objetos `Member` relativos a cada detector, permitindo incluir os identificadores à medida em que a mensagem trafega entre os nós. O `Hashtable` foi escolhido por indexar os objetos de acordo com uma chave (o nome da máquina onde estão os objetos monitoráveis), reduzindo os custos de processamento no momento de procurar se determinada máquina está no *path*, na hora do recebimento da mensagem. Como a comunicação é feita utilizando somente o endereço IP e a porta, e não o nome da máquina, pode-se colocar mais de um detector em uma mesma máquina, bastando cadastrá-los com nomes e portas diferentes.

A segunda tarefa da classe `HBEngine` envolve o tratamento das mensagens recebidas. Esta tarefa, ao contrário da anterior, não é executada periodicamente, mas a cada nova mensagem que chega, através de uma interface de *callback* `deliver()`. Como já foram desserializadas pelas *threads* da recepção `DeliverBoy`, as mensagens já estão prontas para serem utilizadas. Elas contêm o *path* por onde trafegou a mensagem, e o detector de defeitos realiza as seguintes operações:

1. incrementar os contadores dos processos que estão presentes no *path* (se um processo retransmitiu a mensagem, significa que ele está funcionando);

2. adicionar seu próprio identificador ao *path*;

3. retransmitir a mensagem, com o novo *path*, para todos os seus vizinhos que não constavam do *path* recebido;

É o `HBEngine` que gerencia o conjunto dos objetos monitoráveis do sistema, relacionando os contadores aos objetos, e conversa com os seus vizinhos. Como a definição do *Heartbeat* não faz nenhuma operação de suspeita, apenas informa os contadores, a classe `HBEngine` apenas contém uma lista de identificadores dos objetos monitorados e o número de mensagens recebidas de cada um deles, acessível através do método `getCounters()`.

6.3.3 Amostragem

O detector *Heartbeat* (no caso deste trabalho, a classe `HBEngine`) não realiza nenhum tratamento sobre os contadores de mensagens, repassando essa responsabilidade para os níveis superiores da aplicação. Para evitar que isso interfira na transparência com que são usados os detectores, optou-se por adicionar uma camada de *software* que faz esse tratamento, a classe `HBDetector`, de forma a entregar à aplicação uma lista de suspeitos já elaborada, semelhante às

listas fornecidas pelos demais modelos de detectores.

A cada vez que ocorre uma amostragem, é realizada a comparação entre o número de mensagens recebidas, por máquina, obtidos em dois instantes. Quando não há incremento no contador, isso pode significar que o detector monitorado falhou, e por conseqüinte, a máquina onde se encontrava é adicionada na lista de suspeitos.

O período de amostragem, definido em milissegundos, é inicialmente determinado na criação do detector, mas pode ser alterado posteriormente através da chamada ao método `setSamplingRate()`. Quando este intervalo é definido como zero, o detector cessa a amostragem periódica, bastando redefinir um outro valor para que volte à atividade.

A cada amostragem é gerada uma lista de suspeitos, e os objetos cadastrados para receber as notificações são avisados (mesmo nos casos onde não há suspeita). O cadastro dos objetos interessados é feito através dos métodos `addNotifiable()` e `removeNotifiable()`. Todos os objetos registrados para receber as notificações de suspeitos são avisados, através da interface de *callback* `notifySuspicion()`.

Outra opção de amostragem, que gera a notificação para todos os processos interessados, é obtida através da requisição `asyncSampling()`. Este método dispara a amostragem a qualquer instante, independente do instante da amostragem periódica.

Por fim, pode-se obter uma lista de suspeitos invocando diretamente o método `getSuspects()`. Esse método retorna a lista de suspeitos apenas para quem pediu, não enviando para os processos cadastrados.

Quando a aplicação deseja modificar sua visão, ou seja, mudar o conjunto das máquinas a serem monitoradas, deve-se utilizar os métodos `startMonitoring()` e `stopMonitoring()` para adicionar ou remover elementos da lista de objetos monitoráveis.

6.3.4 Interface FailureDetector

O detector *Heartbeat* implementado neste trabalho é somente um dos diversos modelos de detectores disponíveis. Através do estudo da bibliografia [FEL 98, GAR 98, EUG 98] foi determinada uma nomenclatura comum para os principais métodos de interação entre uma aplicação e um detector de defeitos. Como forma de aumentar a transparência e facilitar a definição de outros detectores que porventura sejam implementados, foi criada a interface `FailureDetector`, que define estas chamadas comuns aos detectores. A vantagem de desenvolver novos detectores utilizando esta interface é a transparência implícita, ou seja, a aplicação não necessitará ser modificada em toda sua extensão para poder utilizar um novo detector, pois todos apresentam os mesmos métodos usuais de interação com a aplicação.

A fig. 6.5 mostra o relacionamento entre as classes do detector e entre o detector e a aplicação. Pode ser observado que, através do uso das interfaces `FailureDetector` e `SuspicionHandler`, tanto a aplicação quanto o detector de defeitos não precisam conhecer exatamente qual o tipo de objetos com que se relacionam, mantendo assim a transparência na utilização do detector.

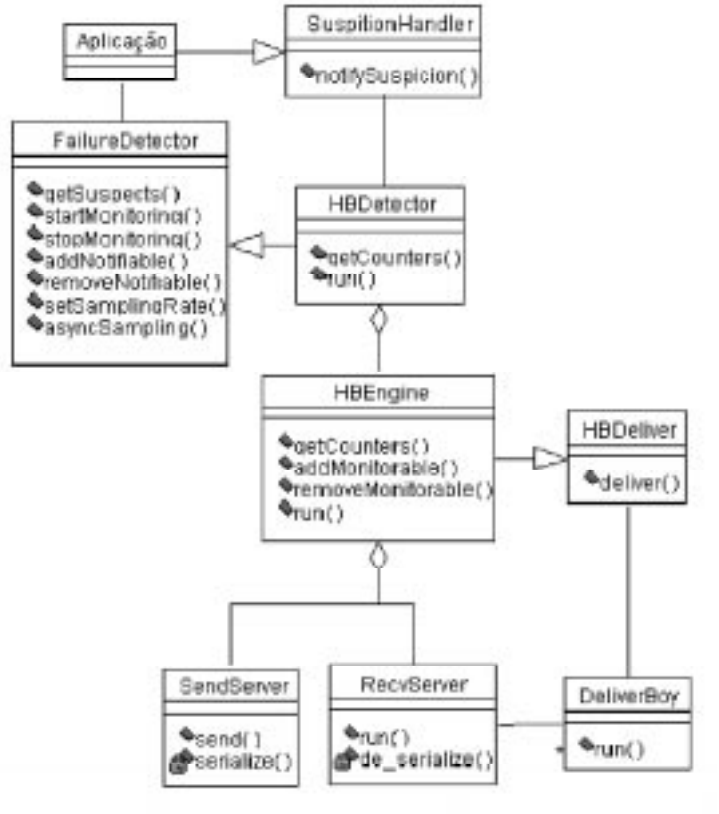

FIGURA 6.5 - Relacionamento entre as classes do detector

6.4 Considerações sobre o Protótipo

A implementação de um protótipo de um algoritmo tem como principal objetivo analisar sua real funcionalidade, suas fraquezas e as possíveis modificações que podem torná-lo mais eficiente. O protótipo do detector de defeitos *Heartbeat* não se omite quanto a esse objetivo, tendo sido analisado sob vários aspectos durante a implementação e sobretudo nos testes de funcionamento.

Como não há nenhuma referência sobre quais aspectos a analisar, optou-se por verificar questões normalmente sugeridas quando este tema é apresentado, entre elas a necessidade de comunicação confiável, o número de mensagens enviadas e a integração com outras fontes de suspeitas. Estas questões apenas comprovam que o desenvolvimento de ferramentas de comunicação de grupo, especialmente as que utilizam detectores de defeitos, ainda está experimentando as possíveis variações de estrutura, a fim de encontrar uma configuração eficiente e barata.

Muitas opções de aprimoramento são sugeridas neste trabalho, mas que de forma alguma engloba todas as possibilidades, ficando sim como um ponto de partida para trabalhos futuros.

6.4.1 Comunicação não confiável

A utilização de um protocolo de transporte que não garante a entrega, como no caso do UDP, não teve nenhum efeito negativo sobre a precisão do detector. De fato, o monitoramento das mensagens enviadas entre máquinas de uma rede local revelou que não ocorreram perdas de mensagens em número suficientemente alto para comprometer a detecção. O modelo do detector *Heartbeat* ainda contribui para minimizar a influência dessas perdas, pois cada mensagem que consegue chegar ao seu destino carrega informações sobre diversas máquinas por onde passou antes. O número de mensagens trocadas varia de acordo com o número de máquinas adicionadas conjunto de vizinhos, mas quanto mais interligados forem os processos, tende a ser mais alto o número de mensagens. Assim, é pouco provável que não ocorra a coleta de informações sobre todas as máquinas monitoradas, com algumas poucas mensagens recebidas.

Como o número de mensagens que trafegam tende a ser grande, a utilização de um protocolo como o UDP reduz o custo do estabelecimento de cada transmissão, e as possíveis perdas de mensagens são suficientemente compensadas pelo grande número de informações enviadas.

6.4.2 Quantidade de mensagens

Um detector de defeitos, apesar de ser um módulo dedicado, não deve representar um grande peso no desempenho do sistema em que será integrado, nem é desejável que a precisão de suas suspeitas seja condicionada à super-utilização da rede de comunicação.

Quando foi apresentado o detector *Heartbeat*, no capítulo 4, foi citada como uma de suas vantagens a minimização do número de mensagens, pois todas as transmissões eram baseadas no conjunto de vizinhos. Esse modelo, entretanto, não permite a integração com outros protocolos, pois considera que somente os contadores dos vizinhos serão incrementados.

A implementação do detector estendeu o modelo do *Heartbeat* ao utilizar as informações contidas na própria estrutura *path* das mensagens, para monitorar todos os processos da rede.

A determinação do número ótimo de processos as ser considerado no conjunto de vizinhos é essencial na operação dos detectores, e para auxiliar nesse julgamento foram realizados alguns testes com o detector.

Os testes realizados utlizaram cinco detectores se comunicando em computadores diferentes, de forma a representar o monitoramento de cinco máquinas. Além disso, os mesmos testes foram executados entre cinco detectores rodando em uma mesma máquina, a fim de verificar a carga imposta ao sistema. A simulação das falhas dos processos e de suas recuperações foi criada através da finalização e reinício manual das aplicações que rodavam os detectores.

Em um primeiro momento, foi analisada a comunicação entre os detectores quando se utiliza todos os processos do sistema dentro do conjunto de vizinhos. Embora a detecção de defeitos não apresentou nenhum problema, o número excessivo do número de mensagens trocadas representou um possível problema quando forem utilizadas redes com mais processos. A razão desse aumento excessivo é que quando todos os objetos monitorados pertencem ao conjunto de vizinhos, ocorre que cada mensagem recebida será retransmitida para todos os outros detectores, até que não existam mais objetos vizinhos ausentes do *path*. O número de mensagens, embora não tenha causado nenhum problema ao funcionamento dos sistemas, apresentou uma curva de crescimento muito acentuada, o que pode exigir a definição de um limite prático ao número de

objetos monitorados, a fim de manter os sistemas em boas condições de operação.

Uma possível alternativa para manter a associação entre o conjunto de elementos monitoráveis e o conjunto de vizinhos, sem onerar tanto o sistema, implica na utilização de retardos na retransmissão das mensagens. Isso reduz bastante o número de mensagens, pois como os detectores retransmitem as mensagens para todos os vizinhos que não estejam no *path* imediatamente após o recebimento delas, muitas mensagens com informações semelhantes são recebidas por cada processo. A definição de um detector preguiçoso (*lazy*) poderia definir intervalos de tempo onde as informações recebidas até aquele instante seriam retransmitidas, sem inundar a rede com informações redundantes.

No outro extremo, quando considera-se apenas um dos outros processos como vizinho, a transmissão correta de informações somente ocorre quando todos são dispostos em um ciclo fechado. Qualquer falha em um dos processos do ciclo irá interromper a circulação das mensagens, causando suspeitas incorretas nos demais processos do sistema, como demonstra a fig. 6.6.

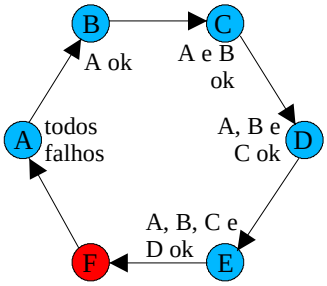

FIGURA 6.6 - Quebra do ciclo

Avaliando-se as situações médias, onde existem mais de um processo dentro do conjunto de vizinhos, mas sem se aproximar do número total de processos do sistema, verifica-se que o número de mensagens é suficientemente controlado, não impondo sobrecarga ao sistema. Entretanto, mesmo nesses casos deve-se tomar extrema precaução quanto à topologia com que são interligados os processos. Deve-se evitar processos que somente enviem ou recebam mensagens, pois em tais casos estes não funcionarão ou serão considerados adequadamente. Outro problema interessante refere-se à possibilidade de falhas em todos os vizinhos de um processo, como demonstra a fig. 6.7. Nestes casos, um detector é isolado dos demais através de uma partição lógica, e a visão do sistema será afetada pelas diferentes suspeitas observadas entre os detectores.

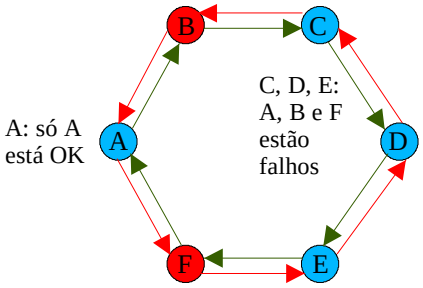

FIGURA 6.7 - Ocorrência de partição lógica incorreta

Recomenda-se assim que o número de processos diretamente conectados e a topologia dessas conexões sejam tomados de acordo com um objetivo de tolerância a falhas pré-estabelecido mínimo para o suporte ao sistema.

6.4.3 Outras formas de coletar informações

Embora um detector de defeitos seja conceitualmente um módulo separado dos demais módulos de uma aplicação, freqüentemente há o questionamento sobre uma possibilidade de integração das funcionalidades do detector dentro de outros módulos. Embora a principal causa desse questionamento seja com relação ao desempenho do sistema, há também uma dúvida sobre a independência do detector quanto à coleta de informações.

Por ser um elemento com funcionamento dedicado, um detector de defeitos possivelmente é o mais indicado para a coleta de suas informações. Entretanto, outros módulos, como o de comunicação confiável, também podem identificar defeitos nas outras máquinas, e poderia haver uma maior integração entre esses elementos para ampliar a coleta de informações. De fato, quando o modelo de falhas é mais complexo que o *crash*, há a possibilidade de certos serviços, como o detector, estarem funcionando normalmente, mas outros estarem falhos. Wogels (apud [BIR 96]) propôs a utilização de outros serviços para a obtenção de mais informações, como por exemplo as bases de dados MIB para a gerência de redes.

De qualquer forma, é questionável a maneira como será feita a integração dessas informações, pois deve-se levar em consideração o tempo de validade de uma informação externa ao detector, e principalmente, a possibilidade de divergência entre o detector e os outros módulos.

7. CONCLUSÃO

Ao longo deste trabalho pôde-se examinar mais de perto os problemas que envolvem a coordenação de atividades em sistemas distribuídos assíncronos, quando sujeitos a falhas. Tais sistemas procuram normalmente manter suas características operacionais nos ambientes instáveis e indeterminísticos representados pelos sistemas computacionais reais, e por isso as ferramentas desenvolvidas nesse ambiente tendem a ser confiáveis e robustas.

Entre as operações que dão suporte à coordenação nos sistemas distribuídos destaca-se o consenso. Além de ser largamente utilizado, o consenso apresenta-se como uma das mais básicas operações, de modo que outras operações também comuns, como a difusão atômica e a eleição, são tão ou mais complexas que o consenso. Isso representa uma questão muito séria, pois segundo a chamada Impossibilidade FLP [FIS 85], um consenso não pode ser executado de forma determinística em sistemas assíncronos sujeitos a falhas, pois é impossível determinar com exatidão se os processos estão falhos ou apenas muito lentos.

Através do uso de detectores de defeitos, propostos por Chandra e Toueg [CHA 96a], é possível contornar tal impossibilidade. O presente trabalho revisou a definição dos detectores, e apresentou algumas das propostas presentes na bibliografia. Tais propostas apresentavam diversas variações, tanto na forma de operação quanto nos modelos de falhas para o qual foram planejadas, e além dos detectores determinísticos foram apresentadas dois modelos de detectores não-determinísticos, com o objetivo de avaliar as limitações e vantagens dos modelos não-determinísticos com relação aos detectores tradicionais.

Uma das principais contribuições do presente trabalho, apresentada no capítulo 5, foi a avaliação das possíveis maneiras de integrar um detector de defeitos a outros protocolos e serviços que fazem parte de uma ferramenta de comunicação de grupo. Normalmente, as publicações que apresentam os detectores de defeitos não se detêm de forma esclarecedora nesses aspectos, nem apresentam os resultados de testes práticos de seus modelos.

A avaliação realizada identificou em primeiro lugar a forma com que são obtidas as listas de suspeitos, onde a ocorrência mais freqüente é o uso de notificações assíncronas através de *callbacks*. Essa constatação leva à necessidade de uma maior integração entre o projeto dos detectores e das aplicações que os utilizam, pois ambos devem se preocupar com sua integração. Essa relação próxima favorece a utilização de estruturas padronizadas, de modo a evitar a redefinição dos sistemas no caso da troca do modelo de detector.

Também foi levantada a questão sobre os tempos de amostragem que costumam ser utilizados em situações práticas. Tal quantificação é essencial, pois interfere diretamente na relação entre a precisão e a agilidade dos sistemas. As respostas obtidas indicam que é mais valorizada a precisão da detecção, pois ações baseadas em detecções errôneas costumam custar mais caro aos sistemas, comparado à perda de agilidade.

A útlima etapa deste trabalho envolveu a construção de um protótipo, baseado no modelo *Heartbeat* [AGU 97a]. Algumas modificações sobre o modelo original foram propostas e testadas com sucesso, objetivando principalmente a adaptação da saída de dados do *Heartbeat* ao formato normalmente utilizado pelos demais detectores.

Apesar de todos trabalhos já desenvolvidos sobre detectores, ainda pode-se considerar que

apenas a fase inicial do desenvolvimento foi alcançada. Novas propostas de detectores podem ser oferecidas, pois não há restrição quanto à forma com que são construídos, e cada vez mais é importante reduzir o número de mensagens trocadas, e também observar a inserção dos detectores nos sistemas orientados a objetos. Além de todos os aspectos relativos exclusivamente aos detectores de defeitos, mostrou-se muito interessante a possibilidade de complementar suas funcionalidades com a associação a outros mecanismos não-determinísticos, como forma de ampliar a sua atuação e evitar a situações que levam a uma espera prolongada.

Todos os estudos e melhorias feitos sobre os detectores de defeitos contribuem para aprimorar a eficiência e a confiabilidade das aplicações distribuídas, aumentando a possibilidade da adoção do paradigma da comunicação de grupo em um futuro próximo, quando sistemas distribuídos com alta confiabilidade e disponibilidade tornarem-se presentes e necessários.

Bibliografia

[AGU 96] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **Randomization and Failure Detection: A Hybrid Approach to Solve Consensus**. Technical Report, Cornell University, Junho 1996. Disponível por WWW em http://cs-tr.cs.cornell.edu:80/ Dienst/UI/1.0/Display/ncstrl.cornell/TR96-1592 (11 Janeiro 2000)

[AGU 97a] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In: 11TH INTERNATIONAL WORKSHOP ON DISTRIBUTED ALGORITHMS, **Proceedings...**, Setembro 1997. Também publicado como Technical Report, Cornell University, Maio 1997. Disponível por WWW em *http://cs-tr.cs.cornell .edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR97-1631* (11 Janeiro 2000)

[AGU 97b] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **On the Weakest Failure Detector for Quiescent Reliable Communication**, Technical Report, Cornell University, Julho 1997. Disponível por WWW em *http://cs-tr.cs.cornell.edu:80/ Dienst/UI/1.0/Display/ncstrl.cornell/TR97-1640* (11 Janeiro 2000)

[AGU 97c] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **Quiescent Reliable Communication and Quiescent Consensus in Partitionable Networks**, Technical Report, Cornell University, Julho 1997. Disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR97-1632* (11 Janeiro 2000)

[AGU 98] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **On Quiescent Reliable Communication**, Technical Report, Cornell University, Dezembro 1998. Disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR98-1692* (11 Janeiro 2000)

[AMA 00] AMARAL, Jeferson Botelho do. **Definição de Classes para Comunicação Multicast e Unicast**, Dissertação de Mestrado a ser defendida, Instituto de Informática, Universidade Federal do Rio Grande do Sul, 2000.

[BAL 99] BALDONI, Roberto; HÉLARY, Jean-Michel; RAYNAL, Michel; TANGUY, Lénaïck. **Consensus in Byzantine Asynchronous Systems**, Technical Report, INRIA, França, Abril 1999. Disponível por WWW em *http://www.inria.fr/RRRT/RR-3665.html* (12 Janeiro 2000)

[BIR 96] BIRMAN, Kenneth P. **Building Secure and Reliable Network Applications**. pg. 210;244-249, Manning Publications, Greenwich, 1996.

[CHA 96a] CHANDRA, Tushar Deepak.; TOUEG, SAM. Unreliable Failure Detectors for Reliable Distributed Systems. **Journal of the ACM**, v. 43, n. 2, pg 225-267, Março 1996. Também disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display /ncstrl.cornell/TR95-1535* (12 Janeiro 2000)

[CHA 96b] CHANDRA, Tushar Deepak.; HADZILACOS, Vassos; TOUEG, SAM. The Weakest Failure Detector for Solving Consensus. **Journal of the ACM**, v. 43,

n. 4, pg 685-722, Julho 1996. Também disponível por WWW em *http://cs-tr.cs.cornell.edu:80/ Dienst/UI/1.0/Display/ncstrl.cornell/TR94-1426* (12 Janeiro 2000)

[DOL 97] DOLEV, Danny; FRIEDMAN, Roy; KEIDAR, Idit; MALKHI, Dahlia. Failure Detectors in Omission Failure Environments, In: 16TH ANNUAL ACM SYMPOSIUM ON PRINCIPLES OF DISTRIBUTED COMPUTING, Agosto 1997, Santa Bárbara, USA. **Proceedings...** Também disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR96-1608* (12 Janeiro 2000)

[EUG 98] EUGSTER, Patrick. **OGS Java Implementation**, *École Polytechnique Fédérale de Lausanne*, Suíça, Dezembro 1998. Disponível por WWW em *http://lsewww.epfl.ch/OGS/OGS_Java_ Implementation.pdf* (12 Janeiro 2000)

[FEL 98] FELBER, Pascal. **The CORBA Object Group Service**, Tese de Doutorado, *École Polytechnique Fédérale de Lausanne*, Suíça, 1998. Disponível por WWW em *http://lsewww.epfl.ch/OGS/thesis/* (12 Janeiro 2000)

[FIS 85] FISCHER, Michael J.; LYNCH, Nancy A.; PATERSON, Michael S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM**, v. 32, n. 2, pg 374-382, 1985.

[GAR 98] GARBINATO, Benoît. **Protocol Objects and Patterns for Structuring Reliable Distributed Systems**, Tese de Doutorado, *École Polytechnique Fédérale de Lausanne*, Suíça, 1998. Disponível por WWW em *http://lsewww.epfl.ch/garbinato/PhD/* (11 Janeiro 2000).

[GUE 97] GUERRAOUI, Rachid; SCHIPER, André. Consensus: The Big Misunderstanding. In: IEEE INTERNATIONAL WORKSHOP ON FUTURE TRENDS IN DISTRIBUTED COMPUTING SYSTEMS (FTDCS'97), Outubro 1997. **Proceedings....** Também disponível por WWW em *http://lsewww.epfl.ch/~rachid/papers/ftdcs2-97.ps* (12 Janeiro 2000)

[GUE 98] GUERRAOUI, Rachid; FELBER, Pascal; GARBINATO, Benoît; MAZOUNI, Karim. System Support for Object Groups. In: ACM CONFERENCE ON OBJECT ORIENTED PROGRAMMING SYSTEMS, LANGUAGES AND APPLICATIONS, 1998, Vancouver, Canadá. **Proceedings....** Também disponível por WWW em *http://lsewww.epfl.ch/~rachid/papers/experience.ps.gz* (12 Janeiro 2000)

[GUO 98] GUO, Katherine Hua. **Scalable Message Stability Detection Protocols**, Tese de Doutorado, Cornell University, Maio 1998. Disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/ 1.0/Display/ncstrl.cornell/TR98-1684* (11 Janeiro 2000)

[HAD 93] HADZILACOS, Vassos; TOUEG, Sam. Fault-Tolerant Broadcasts and Related Problems. In: MULLENDER, Sape. **Distributed Systems**, pg. 97-146. Addison-Wesley/ACM Press, 2a. Edição, 1993.

[HOR 99] van RENESSE, Robert; BIRMAN, Kenneth; VOGELS, Werner; THAKER, Gautam. Mensagens postadas na lista de discussão sobre as ferramentas de comunicação de grupo Isis, Horus e Ensemble. Disponível por e-mail em *HORUS-L@cornell.edu* (20 Outubro 1999). Ver Anexo 2.

[JAL 94] JALOTE, Pankaj. **Fault Tolerance in Distributed Systems**, Englewood Cliffs:Prentice Hall, 1994.

[KIH 97] KIHLSTROM, Kim Potter; MOSER, Louise. E.; MELLIAR-SMITH, P. M. Solving Consensus in a Byzantine Environment Using an Unreliable Fault Detector, In: INTERNATIONAL CONFERENCE ON PRINCIPLES OF DISTRIBUTED SYSTEMS, Dezembro 1997. Chantilly, França. **Proceedings...** Também disponível por WWW em *http://alpha.ece.ucsb.edu/pub/ publications.html* (12 Janeiro 2000)

[LAP 98] LAPRIE, Jean-Claude. Dependability of Computer Systems: From Concepts to Limits. In: IFIP INTERNATIONAL WORKSHOP ON DEPENDABLE COMPUTING AND ITS APPLICATIONS, 1998, Johannesburg, África do Sul. **Proceedings...** Disponível por WWW em *http://www.cs.wits.ac.za/research/workshop/programme.html* (11 Janeiro 2000)

[NEI 93] NEIGER, Gil. **Distributed Consensus Revisited**, 1993, Georgia Institute of Technology, USA, 1993.

[NUN 98] NUNES, Raul Ceretta. **Programação Orientada a Grupos: o Ponto de Vista das Aplicações**, Trabalho Individual, Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1998. Disponível por WWW em *http://www.inf.ufrgs.br/gpesquisa/tf/ portugues/producao/ticeretta.zip* (12 Janeiro 2000)

[POW 96] POWELL, David. Group Communication. **Communications of the ACM**, v. 39, n. 4, pg 50-53, Abril 1996.

[RAY 96] RAYNAL, Michel. **Fault-Tolerant Distributed Systems: a Modular Approach to the Non-Blocking Atomic Commitment Problem**, Technical Report, INRIA, França, Setembro 1996. Disponível por WWW em *http://www.inria.fr/RRRT/RR-2973.html* (12 Janeiro 2000)

[SAB 95] SABEL, Laura S.; MARZULLO, Keith. **Election Vs. Consensus in Asynchronous Systems**, Cornell University, Fevereiro 1995. Disponível por WWW em *http://cs-tr.cs.cornell.edu:80/Dienst/UI/ 1.0/Display/ncstrl.cornell/TR95-1488* (11 Janeiro 2000)

[SCH 93] SCHROEDER, Michael D. A State-of-the-Art Distributed System: Computing with BOB. In: MULLENDER, Sape. **Distributed Systems**, pg. 1-16. Addison-Wesley/ACM Press, 2a. Edição, 1993.

[VOG 96] VOGELS, Werner. World Wide Failures, In: 7TH ACM SIGOPS EUROPEAN WORKSHOP, Setembro 1996. Connemara, Irlanda. **Proceedings...** Também disponível por WWW em *http://mosquitonet.Stanford.EDU/sigops96/papers/vogels.ps* (12 Janeiro 2000)

Outras Referências

[EST 99] ESTEFANEL, Luiz Angelo Barchet. **Necessidade de comunicação confiável para detectores de defeitos**, Instituto de Informática,Universidade Federal do Rio Grande do Sul, Setembro 1999. Discussão interna do grupo de Tolerância a Falhas. A ser disponibilizado por WWW em *http://www.inf.ufrgs.br/~angelo.*

[NUN 99] NUNES, Raul Ceretta. **Relacionamento entre detectores de falhas e os protocolos de consenso e *membership***. Instituto de Informática, Universidade Federal do Rio Grande do Sul, 1999. Comunicação verbal.

ANEXO 1 - Classes do Detector de Defeitos

Interface FD.FailureDetector

```
package FD;

import COM.*;
import java.util.Vector;

public interface FailureDetector {
   public Vector getSuspects();
   public boolean startMonitoring (Vector objs);
   public boolean stopMonitoring (Vector objs);
   public boolean addNotifiable (SuspicionHandler nreader);
   public boolean removeNotifiable (SuspicionHandler nreader);
   public void setSamplingRate (long milisecs);
   public void asyncSampling ();
}
```

Interface FD.SuspicionHandler

```
package FD;

import java.util.Vector;

public interface SuspicionHandler
{
    public void notifySuspicion (Vector objs);
}
```

Interface FD.HBDeliver

```
package FD;

import java.util.Hashtable;

public interface HBDeliver
{
   public void deliver (Hashtable path);
}
```

Classe FD.HBDetector

```
package FD;

import COM.*;
import java.util.*;

public class HBDetector extends Thread implements FailureDetector
{
  private boolean notifyHigherLevel;
  private Vector lastCounter[];
  private Vector notifiables;
  private Member myId;
  private HBEngine hbgear;
  private long sampleRate;

  public HBDetector (Member myId, long stime)
  {
    notifyHigherLevel = false;
    notifiables = new Vector();
    this.myId = myId;
    sampleRate = stime;
    hbgear = new HBEngine (myId);
    hbgear.start();
    lastCounter = new Vector[2];
    lastCounter[0] = new Vector();
    lastCounter[1] = new Vector();
  }

  public void run()
  {
    while (this.isAlive())
    {
      if (!notifiables.isEmpty())
      {
        Vector suspects = getSuspects();

        SuspicionHandler newsReader;

        for (int i=0;i<notifiables.size();++i)
        {
          newsReader = (SuspicionHandler)notifiables.elementAt(i);
          newsReader.notifySuspicion(suspects);
        }
      }
      try {
        sleep (sampleRate);
      } catch (InterruptedException ex) {
        System.out.println("Something interrupted the program.");
        System.exit(1);
      }
    }
  }

  public synchronized boolean addNotifiable (SuspicionHandler nhandler)
  {
    if (!notifiables.contains(nhandler))
    {
      if (notifiables.isEmpty())
      {
```

```
    notifyHigherLevel = true;
      }
      notifiables.addElement(nhandler);
      return true;
    }
    else
      return false;
}

public synchronized boolean removeNotifiable (SuspicionHandler nhandler)
{
    if (notifiables.removeElement(nhandler))
    {
      if (notifiables.isEmpty())
      {
        notifyHigherLevel = false;
      }
      return true;
    }
    else
      return false;
}

public synchronized boolean startMonitoring (Vector objs)
{
    if (hbgear.addMonitorable(objs))
    {
      return true;
    }
    else
return false;
}

public synchronized boolean stopMonitoring (Vector objs)
{
    if (hbgear.removeMonitorable(objs))
      return true;
    else
      return false;
}

public Vector[] getCounters ()
{
    return hbgear.getCounters();
}

public Vector getSuspects()
{
    Vector suspects;
    suspects = new Vector();
    Vector counter[] = hbgear.getCounters();
    int j;
    Long oldOne, newOne;
    for (int i=0;i<counter[0].size();++i)
    {
      j = lastCounter[0].indexOf(counter[0].elementAt(i));
      if (j > -1)
      {
        oldOne = (Long)lastCounter[1].elementAt(j);
        newOne = (Long)counter[1].elementAt(i);
        if (oldOne.longValue()==newOne.longValue())
        {
```

```
            suspects.addElement(counter[0].elementAt(i));
          }
        }
      }
      lastCounter[0] = (Vector)counter[0].clone();
      lastCounter[1] = (Vector)counter[1].clone();
      return suspects;
    }

    public synchronized void asyncSampling ()
    {
      if (!notifiables.isEmpty())
      {
        Vector suspects = getSuspects();
        SuspicionHandler newsReader;
        for (int i=0;i<notifiables.size();++i)
        {
          newsReader = (SuspicionHandler)notifiables.elementAt(i);
          newsReader.notifySuspicion(suspects);
        }
      }
    }

    public void setSamplingRate (long msecs)
    {
      if (msecs == 0 && sampleRate > 0)
        this.suspend();
      if (msecs > 0 && sampleRate == 0)
        this.resume();
      this.sampleRate = msecs;
    }
  }
```

Classe FD.HBEngine

```
package FD;

import COM.*;
import java.util.*;

public class HBEngine extends Thread implements HBDeliver
{
  private Vector neighbor[];
  private Vector processes[];
  private Member myId;
  private RecvServer recv;
  private SendServer send;
  private long SAMPLE_TIME = 1000;

  public HBEngine (Member myId)
  {
    this.myId = myId;
    neighbor = new Vector[2];
    neighbor[0] = new Vector();
    neighbor[1] = new Vector();
    processes = new Vector[2];
    processes[0] = new Vector();
    processes[1] = new Vector();
    recv = new RecvServer (this, myId.UDPport);
    send = new SendServer ();
  }

  public void run ()
  {
    recv.start();
    while (true)
    {
      Hashtable path = new Hashtable();
      path.put(myId.name,myId);
      for (int i=0;i<neighbor[0].size();++i)
      {
        Member destination = (Member)neighbor[0].elementAt(i);
        send.send(destination, path);
      }
      try {
        sleep (SAMPLE_TIME);
      } catch (InterruptedException ex) {
        System.out.println("Something interrupted the program.");
        System.exit(1);
      }
    }
  }

  public synchronized void deliver (Hashtable path)
  {
    Hashtable path2 = (Hashtable)path.clone();
    Enumeration pathList;
    String nextEl;
    Long iterator;
    long value;
    int i,j;
    boolean exists = false;
    Member localBase,remoteBase;
```

```
  if (!path.containsKey(myId.name))
  {
    path2.put(myId.name,myId);
    for (i=0;i<neighbor[0].size();++i)
    {
      localBase = (Member)neighbor[0].elementAt(i);
      if (!path.containsKey(localBase.name))
        send.send(localBase, path2);
    }
    for (i=0;i<processes[0].size();i++)
    {
      localBase = (Member)processes[0].elementAt(i);
      if (path.containsKey(localBase.name))
      {
        iterator = (Long)processes[1].elementAt(i);
        value = iterator.longValue() + 1;
        iterator = new Long(value);
        processes[1].setElementAt(iterator,i);
      }
    }
    if (path.size()!=processes[0].size())
    {
   i = processes[0].size();
      for (pathList=path.keys(); pathList.hasMoreElements() ;)
      {
        nextEl = (String)pathList.nextElement();
        for (j=0; j<i;++j)
        {
          remoteBase = (Member)processes[0].elementAt(j);
          if ((remoteBase.name).equals(nextEl))
          {
            exists = true;
            j=i;
          }
        }
        if (!exists)
        {
          processes[0].addElement(path.get(nextEl));
          processes[1].addElement(new Long(1));
        }
        exists = false;
      }
    }
  }
}

public Vector[] getCounters ()
{
  return processes;
}

public boolean addMonitorable (Vector objs)
{
  try {
    Member machine;
    for (int i=0;i<objs.size();++i)
    {
      machine = (Member)objs.elementAt(i);
      if (!neighbor[0].contains(machine))
      {
        neighbor[0].addElement(machine);
```

```
            neighbor[1].addElement(new Long(0));
            processes[0].addElement(machine);
            processes[1].addElement(new Long(0));
          }
        }
        return true;
      } catch (Exception ex) {
        return false;
      }
    }

    public boolean removeMonitorable (Vector objs)
    {
      try {
        Member machine;
        Member partner;
        int position;
        for (int i=0;i<objs.size();++i)
        {
          machine = (Member)objs.elementAt(i);
          position = -1;
          for (int j=0;j<neighbor[0].size();++j)
          {
            partner = (Member)neighbor[0].elementAt(j);
            if ((partner.name).equals(machine.name))
            {
              if ((partner.IP).equals(machine.IP))
              {
                if (partner.UDPport == machine.UDPport)
                {
                  position = j;
                  if (position > -1)
                  {
                    neighbor[0].removeElementAt(position);
                    neighbor[1].removeElementAt(position);
                  }
                }
              }
            }
          }
        }
        return true;
      } catch (Exception ex) {
        return false;
      }
    }
  }
```

Classe FD.SendServer

```
package FD;

import java.net.*;
import java.io.*;
import java.util.Hashtable;
import COM.Member;

public class SendServer
{
  private DatagramSocket sendWay;

  public SendServer ()
  {
    try {
      sendWay = new DatagramSocket();
    } catch (java.net.SocketException ex) {
      System.out.println("Could not open a transmition port.");
      System.exit(1);
    }
  }

  public synchronized void send (Member destination, Hashtable path)
  {
    boolean messageSent = false;
    try {
      byte[] buf = serialize(path);
      try {
        InetAddress address = InetAddress.getByName(destination.IP);
        DatagramPacket packet = new DatagramPacket(buf, buf.length, address, destination.UDPport);
        while (!messageSent)
        {
          try {
            sendWay.send(packet);
            messageSent = true;
          } catch (java.io.IOException ex) {
            //System.out.print(ex.getMessage());
          }
        }
      } catch (UnknownHostException ex) {
        System.out.println("Invalid address.");
      }
    } catch (java.io.IOException ex) {
      System.out.println("I/O error.");
    }
  }

  private byte[] serialize (Hashtable v) throws IOException {
    ByteArrayOutputStream bout = new ByteArrayOutputStream();
      ObjectOutputStream out = new ObjectOutputStream(bout);
      out.writeObject(v);
      byte arraya[] = bout.toByteArray();
   return arraya;
  }
}
```

Classe FD.RecvServer

```
package FD;

import java.util.Hashtable;
import java.net.*;
import COM.Member;

public class RecvServer extends Thread
{
  private DatagramSocket recvWay;
  private HBDeliver deliverClient;
  private int localPort;
  private String HBmessage;
  private int BUFFERSIZE = 1024;

  public RecvServer (HBDeliver deliverClient, int localPort)
  {
    this.deliverClient = deliverClient;
    this.localPort = localPort;
    try {
      recvWay = new DatagramSocket(localPort);
    } catch (java.net.SocketException ex) {
      System.out.println("The program wan not able to open a receive port.");
      System.exit(1);
    }
  }

  public void run()
  {
    byte[] buf;
    byte[] recvData;
    DatagramPacket packet;
    DeliverBoy deliverBoy;
    Runtime r = Runtime.getRuntime();

    while (true)
    {
            buf = new byte[BUFFERSIZE];
            packet = new DatagramPacket(buf, buf.length);
            try {
        recvWay.receive(packet);
        recvData = packet.getData();
        deliverBoy = new DeliverBoy(deliverClient,recvData);
        deliverBoy.start();
        r.gc();
      } catch (java.io.IOException ex) {
        System.out.println("Could not receive the messages.");
        System.err.println(ex);
      }
    }
  }
}
```

Classe FD.DeliverBoy

```
package FD;

import java.util.Hashtable;
import java.io.*;

public class DeliverBoy extends Thread
{
  private HBDeliver deliverClient;
  private byte[] HBmessage;

  public DeliverBoy (HBDeliver client, byte[]  message)
  {
    deliverClient = client;
    HBmessage = message;
  }

  public void run ()
  {

    Hashtable path = de_serialize(HBmessage);
    if (path != null) {
      deliverClient.deliver (path);
    }
  }

  private Hashtable de_serialize (byte source[]) {
    try {
      ByteArrayInputStream bin = new ByteArrayInputStream(source);
      ObjectInputStream in = new ObjectInputStream(bin);
      Hashtable vet2 = (Hashtable)in.readObject();
      return vet2;
    } catch (StreamCorruptedException ex) {
      System.out.println("The message was corrupted.");
      return null;
    }
    catch (java.io.OptionalDataException ex) {
      System.out.println("Invalid content.");
      return null;
    }
    catch (ClassNotFoundException ex) {
      System.out.println("Class java.lang.Vector not found. Verify the CLASSPATH");
      return null;
    }
    catch (java.io.IOException ex) {
      System.out.println("I/O error " + ex.getMessage());
       ex.printStackTrace(System.out);
       return null;
    }
  }
}
```

Classe COM.Member

```
// Member.java
// Jeferson B. do Amaral - 1999

package COM;

public class Member implements java.io.Serializable {
  public String name=null;
  public String IP=null;
  public int UDPport;

  public Member(String name,String IP,int UDPport) {
    this.name = name;
    this.IP = IP;
    this.UDPport = UDPport;
  }

}//Member Class
```

Classe de teste Initdetec e Testedetec

```
// Just start the "application" thread
import FD.*;

public class initdetec {

   static public void main(String args[])
   {
      testedetec abc = new testedetec();
      abc.start();
   }
}


import COM.Member;
import FD.*;
import java.util.Vector;

public class testedetec extends Thread implements SuspicionHandler {

  public void run()
  {
    try {
      // creating my own identificator
      Member myId = new Member ("detector1","127.0.0.1",4011);
      // creating a failure detector with my Id and 1000 milliseconds of sampling rate
      FailureDetector hb = new HBDetector(myId,1000);
      // insert myself as suspects receiver
      hb.addNotifiable(this);
      // include two other detectors in a Vector
      Vector susp = new Vector();
      Member pair = new Member ("detector2","127.0.0.1",4021);
      susp.addElement(pair);
      pair = new Member ("detector4","127.0.0.1",4041);
      susp.addElement(pair);
      // include the members in the Vector on my neighbor list
      hb.startMonitoring(susp);
      // starts the Failure detector operation
      ((HBDetector)hb).start();
      System.out.println("Start Monitoring 1 and 2");

      /* Some other usefull commands
       *
       * Remove element from neighbor - hb.stopMonitoring(susp);
       * Modify sampling rate - hb.setSamplingRate(0);
       * print the Suspect list -  System.out.println(hb.getSuspects());
       * Making an asynchronous sampling - hb.asyncSampling();
       */

    } catch (Exception ex) {
      System.out.println("Something wrong happened.");
      ex.printStackTrace(System.out);
    }
  }

  public void notifySuspicion (Vector objs)
  {
    Member suspeito;
    if (objs.size()>0)
```

```
      {
        System.out.println("\nSuspects:");
        for (int i=0;i<objs.size();++i)
        {
          suspeito = (Member)objs.elementAt(i);
          System.out.println(suspeito.name + " (" + suspeito.IP + ":" + suspeito.UDPport+ ")");
        }
      }
      else
      {
        System.out.println("\nThere's no suspects.");
      }
    }
}
```

ANEXO 2 - Mensagens da Lista HORUS-L

Date: Mon, 18 Oct 1999 17:07:59 -0400
From: Gautam H Thaker <gthaker@atl.lmco.com>
To: tclark@cs.cornell.edu
Cc: kemme@inf.ethz.ch, HORUS-L@cornell.edu
Subject: Re: Wrong view changes

Tim,

Years go by and we are not able to become very aggresive in using timeout values. This is a shame. *Average* message latency numbers are now very low. Attached chart shows round trip message latecies when using VIA (non-IP protocol) under Linux. At low end average message latencies are 50 usec (round trip) or about 20,000 messages per second. Since these have UDP like semantics we put in a simple 10 msec timeout for retransmissions. (Only had 3 timeouts at 32k msg size).

Certainly one can go down to about ~50 msec for suspect_sweep (which is 3 orders of magniute over the mean in these graphs. I think that would be agressive enough to start with). On a RT OS (we are working with several) priority inversions are generally pretty well bounded and indeed one should be able to use Ensemble for some very fast failovers. ANyway, my $0.02 worth.

Gautam H. Thaker
Distributed Processing Lab; Lockheed Martin Adv. Tech. Labs
A&E 3W; 1 Federal Street; Camden, NJ 08102
856-338-3907, fax 856-338-4144 email: gthaker@atl.lmco.com
Note: Old area code, 609, will work thru approximately Oct. 99

Date: Mon, 18 Oct 1999 20:02:10 -0400
From: Werner Vogels <vogels@cs.cornell.edu>
To: HORUS-L@cornell.edu
Subject: RE: Wrong view changes

Gautham, I understand your frustration, but I am afraid I cannot agree with your point about using more aggressive time-outs over lower latency paths. I have been building protocols over VIA and VIA like architectures for years now and indeed they behave very well in laboratory conditions. But unfortunately roundtrip times are not the dominant issue in determining the time until you can detect a failure.

What dominates in failure detection is not how fast you can raise your first suspicion, but how long you have to wait until you can decide, with high level of certainty, than the member is no longer able to participate. For a long time we thought that the lack of low-latency paths was what prevented us from doing really-fast failure detection, but unfortunately we now know that this expectation was wrong. What dominates failure detection (among many things)is the load on the machine, the concurrently ongoing I/O and the quality of the machine hardware.

For example, you should run your communication response tests again after you have started Oracle Parallel Server on the machine and requested a sort plus indexing of database which is several times larger than your physical memory. You will see that your nice graph suddenly is not so nice anymore, and the responsiveness of your application will be all over the place (believe me, in the "seconds" range on a big-fat multiprocessor sparc, and the same for a Intel Quad running NT4). Of course you can suggest that these machines should run a real-time OS to ensure the predictability, but reality is that there is not RT support for the type of real-world computing.

Another very, very important aspect is the hardware in the machines, there is so much really bad combinations of hardware out there, causing response times of networking hardware to be very unpredictable. Some very well-know raid controllers are able to saturate the PCI-bus for such long periods (several seconds), that it is almost impossible to draw any conclusions about communication silence of a node.

So whether you run over VIA, with 50 usec latency or over ethernet with 200-500 usec, you have to be very conservative in your failure detection timing. The cost of reconfiguration triggered by an incorrect failure suspicion is quite often larger than the advantages of cutting edge failure detection. For example if you run a cluster service on 32 nodes and a failure detection will cause all your databases to reconfigure, clients to shuffle around, load-balancing to be reconfigured, etc. you want to be 99.9999% sure of the failure before you take such action. A large vendor has its failure detection timeout set to 7 seconds based on field experiences, with hardware, heavy load, etc.

BTW failure detection using VIA is a completely different beast all together, RTT's play no role there in the initial detection, only predictable scheduling of the application under scrutiny is of importance. VIA notifies you of all error conditions, except a stalled application which you can try to determine by observing the application's communication pattern.

my 5 cents.
--
Werner

Date: Tue, 19 Oct 1999 09:18:39 -0400
From: Ken Birman <ken@cs.cornell.edu>
To: vogels@cs.cornell.edu, gthaker@atl.lmco.com
Cc: HORUS-L@cornell.edu
Subject: Re: Wrong view changes

I just want to add a small observation. When we scale systems up, effects of scale are hard to avoid. With virtual synchrony, two effects of scale dominate:
1) Congestion, caused by the ultimately bounded size of buffers. As a system gets larger the time needed to achieve stability grows and the likelihood of slow acks rises (no matter how we do acks), hence the senders buffers seem to shrink with scale, causing flow control to kick in.
2) Overhead due to false failure detections and process rejoining.

One tends to set the detection parameters aggressively to combat the first problem, but now the second problem grows in cost and importance.

Overall, I am convinced that this limits conventional implementations of virtual synchrony to about 100 members. For very large groups, we need to focus on other methods -- simple hierarchical structures, for example.

Within small groups, though, I think that rapid failure detection is one way to go. The trick, as Werner points out, is to have the application rapidly detect failures using methods of its own. The faster we notice them, the faster we can react. For systems like Ensemble that detect failures using very indirect methods, like severe congestion, we'll never be able to be as aggressive as one would wish...

Ken

Date: Tue, 19 Oct 1999 09:56:15 -0400
From: Robbert van Renesse <rvr@cs.cornell.edu>
To: Werner Vogels <vogels@cs.cornell.edu>,
"'gthaker@atl.lmco.com' '" <gthaker@atl.lmco.com>
Cc: "'HORUS-L@cornell.edu '" <HORUS-L@cornell.edu>
Subject: RE: Wrong view changes

I want to put in my 2cts worth in as well!

Failure detection is a difficult beast. With just two processes you already can see the difficulty: p may think that q is faulty because q is faulty, or because q is (temporarily) slow, or because the link is (temporarily) slow, or because p is (temporarily) slow. With more than two, things get worse. First, the rate of failures grows linearly with the number of members. Second, the rate of these temporary disturbances also grows at least linearly. Third, if everybody checks everybody else the load on the network grows quadratically. If you use a coordinator based approach, the coordinator gets overloaded and starts emitting false reports. If you use a tree of coordinators, failure detection can become quite slow when one of the coordinators fail. And even using the gossip-based failure detection (see list of papers on my home page), latency is quite high.

For fast failure detection, you can use a two-level scheme. A slow failure detector reports relatively accurate information about failures. The resulting membership is used by the second-level to divide the membership into little islands of processors that monitor each other using an agressive scheme. In case of a detected failure in such an island a representative of the island broadcasts a failure. This approach is similar to Werner's world-wide failure approach if you use locality to form the islands. I've used this scheme successfully in a recent commercial development of group membership.

Dealing with the linear growth of failures is harder. The problem is that view changes also become more expensive with the size of the membership. So you have a pretty bad situation: the more members you have, the more failures there are, but the fewer membership changes you can perform. A potential solution to this is to divide the membership into a hierarchy of domains. In such a scheme, you know less about processes that are far away. You can read about some initial work on that in a paper that is also accessible from my home page, called Scalable and Secure Resource Location.

Oh, by the way, my home page is http://www.cs.cornell.edu/home/rvr.

Cheers,

Robbert