Avaliação dos Detectores de Defeitos e sua Influência nas Operações de Consenso

Luiz Angelo Steffenel

▶ **To cite this version:**

Luiz Angelo Steffenel. Avaliação dos Detectores de Defeitos e sua Influência nas Operações de Consenso. 2001. hal-00091931

HAL Id: hal-00091931
https://hal.science/hal-00091931v1

Preprint submitted on 7 Sep 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

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

Avaliação dos Detectores de Defeitos e sua Influência nas Operações de Consenso

por

LUIZ ANGELO BARCHET ESTEFANEL

Dissertação submetida à avaliação,
como requisito parcial para a obtenção do grau de Mestre
em Ciência da Computação

Profa. Dra. Ingrid Jansch-Pôrto
Orientadora

Porto Alegre, fevereiro de 2001.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Estefanel, Luiz Angelo Barchet
Avaliação dos detectores de defeitos e sua influência nas operações de consenso / por Luiz Angelo Barchet Estefanel. - Porto Alegre:PPGC da UFRGS, 2001.
125 p. : il.
Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientadora: Jansch-Pôrto, Ingrid.
1. Detectores de Defeitos. 2. Terminação do Consenso. 3. Sistemas Distribuídos Assíncronos. I. Jansch-Pôrto, Ingrid. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profa. Wrana Maria Panizzi

Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann

Superintendente de Pós-Graduação: Prof. Philippe Olivier Alexandre Navaux

Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux

Coordenador do PPGC: Prof. Carlos Alberto Heuser

Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Agradecimentos

Gostaria de agradecer à minha esposa Manuele, que esteve sempre presente, dando força, sugestões e até mesmo "puxões de orelha" quando me distraía das tarefas que tinham que ser cumpridas. Por tua causa eu sou o homem mais feliz do mundo.

Dedico este trabalho também aos meus pais e ao meu irmão, que apesar da distância, sempre me apoiaram e ajudaram a me manter em Porto Alegre.

À professora Ingrid, a melhor orientadora que alguém poderia desejar: solícita, interessada e motivadora. Gostaria de me tornar um professor e um orientador tão bom quanto você.

Ao professor e colega Raul Ceretta Nunes, por ter me indicado para o grupo de Tolerância a Falhas, bem como pela disponibilidade e pelos bons conselhos durante o mestrado. Foi a melhor recomendação que poderia ter recebido.

Aos demais colegas do grupo de Tolerância a Falhas, pelas calorosas e produtivas reuniões, onde foi possível delinear as bases do presente trabalho.

Ao colega Roberto Drebes, que foi o grande responsável pelo bom estado do laboratório onde foram realizados os experimentos.

Às outras pessoas que contribuiram para este momento, em especial à amiga e "cunhada" Andrea.

Ao CNPq, pelo suporte financeiro recebido.

Sumário

Lista de Abreviaturas e Siglas

API	*Application Program Interface*
CORBA	*Common Object Request Broker Architechture*
CPU	*Central Processing Unity*
IP	*Internet Protocol*
RMI	*Remote Method Invocation*
TCP	*Transport Control Protocol*
UDP	*User Datagram Protocol*
2PC	*Two phase commit*
3PC	*Three phase commit*
ack	*acknowledge* (reconhecimento)
nack	*negative acknowledge* (reconhecimento negativo)
mod	resto da divisão inteira

Lista de Símbolos

Δ_i	intervalo de envio
Δ_{to}	*timeout*
Δ_S	intervalo de amostragem
$\Diamond$	*eventually*
$\mathcal{W}$	detector *Weak*
$\Diamond\mathcal{W}$	detector *eventually Weak*
$\mathcal{S}$	detector *Strong*
$\Diamond\mathcal{S}$	detector *eventually Strong*
$\mathcal{P}$	detector *Perfect*
$\Diamond\mathcal{P}$	detector *eventually Perfect*

Lista de Figuras

Lista de Tabelas

Resumo

Este trabalho relata observações e análises sobre como os detectores de defeitos influenciam as operação de consenso. O conceito dos detectores de defeitos é essencial para as operações de consenso em sistemas distribuídos assíncronos, uma vez que esses representam uma das únicas formas de sobrepujar as limitações impostas pela chamada Impossibilidade FLP (impossibilidade de diferenciar um processo falho de um processo mais lento). Enquanto os detectores de defeitos têm seu funcionamento bem definido através de duas propriedades, *completeness* e *accuracy*, não há nenhuma restrição quanto à forma de implementá-los. Na literatura são encontrados vários modelos de detectores de defeitos, construídos com as mais variadas estratégias, mecanismos de comunicação e de detecção. No entanto, estes modelos não costumam ser acompanhados de uma comparação com os detectores já existentes; os autores limitam-se a apresentar as inovações dos mecanismos sugeridos. De toda literatura pesquisada, apenas um trabalho procurou comparar diferentes modelos de detectores de defeitos, e através de simulações, avaliou o impacto destes detectores sobre o tempo de terminação das operações de consenso. Entretanto, aquele trabalho era bem limitado, tanto nos modelos de detectores analisados quanto nos objetivos das observações. O presente trabalho procurou estender aquele experimento, incluindo mais modelos de detectores, e transportando-os para um ambiente prático de execução. As observações realizadas não ficaram limitadas às avaliações já realizadas por aquele trabalho, de tal forma que os modelos de detectores testados foram analisados sob diversas métricas, situações e parâmetros de operação. Essas avaliações possibilitaram verificar o comportamento dos detectores frente aos padrões de falhas mais significativos, avaliar o impacto de cada detector sobre as operações de consenso e a sua interação com os elementos do ambiente de execução. Essas avaliações permitiram fazer uma comparação dos detectores, possibilitando a identificação de suas limitações, suas situações de melhor desempenho e possíveis otimizações para serem realizadas em trabalhos futuros.

Palavras-chave: Detectores de defeitos, terminação do consenso, sistemas distribuídos assíncronos

Abstract

This work presents our observations and analysis on the influence of the failure detectors over the consensus algorithm. Failure detectors are essential to the consensus in an asynchronous distributed system, as they represent one of the few techniques that are able to circumvent the limitation imposed by the FLP Impossibility (the impossibility to distinguish a crashed process from a slow one, in asynchronous systems). While failure detectors are well defined through two properties, completeness and accuracy, there's no rule about their implementation. Thus, in the literature there are many models of failure detectors, each one implemented using different approaches to the communication and detection strategies. However, these detectors seldom compare themselves to the existing ones; their authors usually present only the advantages and innovations of the new model. Indeed, we only found one work that tried to compare different failure detectors. Using simulation techniques, that work evaluated the impact of the failure detectors on the consensus termination time. However, that research was very limited in the number of detectors analyzed and in the evaluation goals. The present work extended that experience, including more detectors in the analysis and evaluating them in a practical environment. Also, the observations were not restricted to those from the original paper, and the detectors were analyzed with more metrics, failure patterns and operational parameters. The evaluation allowed us to identify the behavior from the detectors in face of the most significant failure patterns, their influence on the consensus operation and their interaction with the execution environment. These evaluation also enabled us to compare the detectors, identifying their limitations, their best performance situations and possible optimizations to future developments.

Key-words**:** *Failure detectors, consensus termination, asynchronous distributed systems*

1 Introdução

"A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable."
Leslie Lamport, 1987

Os sistemas distribuídos, devido às suas características inerentes, representam uma maneira de aumentar a confiabilidade e disponibilidade do sistema nos casos de falhas, pois permitem que a operação continue mesmo durante o colapso ou falha de alguns componentes. O consenso, uma das mais importantes operações de acordo entre processos, é conhecido por sua incapacidade de garantir o fechamento de uma operação, em ambientes assíncronos sujeitos a falhas. Isso se deve à impossibilidade de determinar precisamente se um processo encontra-se falho ou apenas muito mais lento que os demais [FIS 85]. Essa restrição, conhecida como Impossibilidade FLP[1], ocasiona que, devido à falta de conhecimento sobre o sistema, o algoritmo de consenso deve esperar a resposta de um processo, sob risco de tornar as decisões inconsistentes. No caso da falha de um componente, o algoritmo irá esperar indefinidamente. Essa restrição não afeta somente o consenso; outras operações freqüentemente utilizadas, como a votação e a difusão atômica, são tão ou mais complexas que o consenso [SAB 95], e ficam sujeitas à mesma restrição de operação.

Existem algumas técnicas que objetivam contornar tais restrições, como por exemplo o sincronismo parcial (Dwork *apud* [GUE 97]) e o assincronismo temporizado (Cristian *apud* [GUE 97]). Tais propostas, entretanto, servem apenas a situações especiais, não contemplando as diversas possibilidades do sistema. Chandra e Toueg [CHA 96a] propuseram um modelo de assincronismo auxiliado por um detector de defeitos, que demonstrou ser mais abrangente e adaptável que os demais [GUE 97]. Tais detectores funcionam através da manutenção de uma lista suspeitos, construída a partir de mensagens trocadas entre os detectores e os processos. Embora os detectores de defeitos tampouco possam determinar com exatidão o estado dos processos do sistema, sua utilização aumenta a quantidade de conhecimento sobre os demais elementos, auxiliando na finalização das operações de consenso em tempo hábil.

Chandra e Toueg definiram os detectores de defeitos através de duas propriedades a serem respeitadas: *completeness* (abrangência, completeza) e *accuracy* (precisão). A propriedade *completeness* refere-se à capacidade do detector identificar todos os processos que estão falhos, enquanto *accuracy* determina a precisão desta suspeita, a fim de evitar a inclusão de processos corretos nas listas de suspeitos. Ao respeitar essas duas propriedades, um detector de defeitos garante que os algoritmos que o utilizem não perderão a integridade nem ficarão bloqueados indefinidamente (as chamadas propriedades *safety* e *liveness*, respectivamente). Esta definição, no entanto, não vincula nenhuma técnica específica para a implementação dos detectores, pois a essência de seu funcionamento baseia-se na manutenção das características de *completeness* e *accuracy*. Dessa forma, ao longo dos anos, surgiram diversas propostas de algoritmos detectores

1 Em homenagem aos autores do artigo: Fischer, Lynch e Paterson.

de defeitos, que respeitavam as propriedades de *completeness* e *accuracy* mas diferiam sob diversos aspectos, tanto nas características de construção e comunicação quanto nos modelos de falhas a que se propunham. Esta diversidade, conforme constatado em trabalho anterior [EST 00a], não foi devidamente acompanhada por uma avaliação de desempenho ou confiabilidade, de forma que a escolha sobre qual modelo de detector utilizar baseia-se apenas em considerações lógicas sobre o funcionamento dos próprios detectores. Por exemplo, Felber [FEL 98] avaliou os detectores de defeitos segundo estimativas de tráfego de mensagens. Essas considerações, entretanto, não passam de especulações, pois não há nenhuma verificação prática do impactos dos modelos sobre os sistemas. De fato, como mostra Guerraoui [GUE 97], nem sempre o algoritmo que parece mais eficiente corresponde às expectativas, pois em um sistema prático existem diversos fatores que influenciam no desempenho dos algoritmos.

Enquanto a literatura dispõe vários modelos de detectores de defeitos, há uma grande ausência de trabalhos que objetivam a avaliação destes modelos, caracterizando-os quanto às suas limitações e seus comportamentos. Um passo em direção ao preenchimento desta lacuna foi realizado por Sergent *et al.* [SER 99], onde através de simulações foram analisados alguns modelos de detectores de defeitos, verificando o impacto de seus mecanismos de comunicação sobre o desempenho dos algoritmos de consenso. Aquele trabalho entretanto foi muito limitado, pois como seu objetivo era comprovar que detectores de defeitos podem atender os sistemas distribuídos com eficiência, foi considerado um conjunto muito restrito de detectores, cuja análise restringiu-se aos pontos de melhor desempenho.

De fato, quando se considera um sistema real, existem muitos fatores que podem influenciar o processo de detecção. Enquanto a ocorrência de suspeitas incorretas afeta diretamente o tempo de terminação do consenso, existem outros aspectos resultantes da interação entre o detector e o ambiente (sistema operacional, rede de comunicação) que são responsáveis pela perda de desempenho. Por exemplo, o modelo de comunicação utilizado pelo detector tem impacto direto no tráfego na rede; além disso, o intervalo entre as mensagens também pode acarretar a sobrecarga dos recursos do sistema, pois tanto o meio de comunicação quanto o processamento das mensagens são afetados. Como cada detector reage de forma diferente aos diversos cenários de falhas, a melhor forma de manter o desempenho do consenso envolve a escolha do detector de defeitos e do conjunto de parâmetros que melhor se adaptem a cada situação.

Este trabalho procurou estender as análises de Sergent *et al.*, realizando uma comparação teórica e prática dos algoritmos. Utilizando como ponto de partida os experimentos de Sergent *et al.*, foram elaborados experimentos práticos para avaliar implementações de detectores, representantes dos mais conhecidos modelos presentes na literatura. Estes detectores foram exaustivamente avaliados, e a descrição de suas reações frente às diversas situações de teste são de grande importância para o entendimento das limitações de cada modelo, bem como suas melhores aplicações.

A divisão deste trabalho segue a seguinte estrutura: no capítulo 2 será revisado o problema do consenso e a importância dos detectores de defeitos para sobrepujar essas limitações. Também serão analisados alguns algoritmos de consenso, incluindo o algoritmo original de Chandra e Toueg e propostas de otimização.

O capítulo 3 apresentará a definição dos detectores de defeitos, segundo Chandra e Toueg, sendo apresentadas as propriedades que definem o funcionamento dos detectores de defeitos, e a importância destas propriedades para uma correta expressão

do mecanismo de suspeitas. Também serão apresentados os modelos de detectores de defeitos avaliados neste trabalho, e será feita uma breve análise sobre a influência de seus mecanismos de comunicação sobre o desempenho da detecção.

No capítulo 4 serão descritos o ambiente de testes e as métricas utilizadas nas avaliações, bem como as situações práticas nas quais serão submetidos os detectores. Neste capítulo também serão detalhados aspectos dos parâmetros utilizados para a comparação, e a forma como serão aplicados em cada modelo de detector testado.

O capítulo 5 apresentará as especificações das implementações dos detectores de defeitos e do algoritmo de consenso utilizados, incluindo o protocolo de comunicação utilizado pelo algoritmo de consenso. Também serão expostas as modificações realizadas nos algoritmos, necessárias para a condução dos testes em cada situação avaliada.

O capítulo 6 contém os resultados obtidos nos testes, sendo que em um primeiro momento os detectores serão analisados individualmente, para ressaltar suas características e comportamento. Em uma segunda etapa, os resultados dos detectores são comparados, permitindo a identificação dos problemas mais comuns e das situações onde cada detector melhor se destaca.

Por fim, no capítulo 7 serão apresentadas as conclusões resultantes da observação dos detectores nos experimentos realizados.

2 O Problema do Consenso

2.1 O Que é um Consenso

O consenso pode ser visto como uma forma geral de resolver problemas de acordo em sistemas distribuídos. No problema do consenso, um conjunto de processos ou agentes deve unanimamente concordar em uma decisão obtida a partir de seus estados iniciais. Tipicamente, essa decisão é feita apenas sobre dois valores, 0 e 1. No entanto, isso não impede que o protocolo seja estendido para decisões sobre valores mais complexos, desde que sejam mantidas as propriedades do consenso [TUR 94].

Formalmente, um protocolo de consenso está correto quando as seguintes condições são respeitadas [CHA 96a]:

- **Terminação (*Termination*)**: todo processo correto (não falho[2]) decide um valor em um número finito de passos.
- **Integridade Uniforme (*Uniform Integrity*)**: todo processo decide no máximo uma vez.
- **Acordo (*Agreement*)**: dois processos corretos não decidem diferentemente.
- **Validade Uniforme (*Uniform Validity*)**: se um processo decidir *v*, então *v* foi proposto anteriormente por algum processo.

Alternativamente, Turek e Shasha [TUR 94] definem o consenso através de propriedades um pouco diferentes:

- **Consistência (*Consistency*)**: todos processos decidem sobre um único valor, e todas decisões são finais.
- **Validade**: o valor acordado pelos processos deve ter sido proposto previamente por um deles.
- **Terminação**: todos processos decidem em um valor dentro de um número finito de passos.

Embora estas duas definições sejam um pouco diferentes, seus objetivos são semelhantes, notadamente no que se refere à irrevogabilidade das decisões (propriedades Consistência, segundo Turek e Shasha [TUR 94], e Acordo e Validade segundo Chandra [CHA 96a]). A propriedade Validade impede que seja escolhido um valor que não pertence ao conjunto proposto (por exemplo, uma falha de rede poderia criar um sinal inválido). A propriedade Terminação não permite que o consenso se prolongue indefinidamente, uma vez que a violação da Terminação permitiria que um estado de indecisão permanente fosse considerado válido.

A manutenção das propriedades acima tem efeito direto sobre duas propriedades essenciais ao protocolo de consenso. A primeira dessas propriedades, *safety*, diz

2 Chandra e Toueg consideram apenas falhas por colapso, assim um processo falho é aquele que suspendeu sua atividade prematuramente (antes do fim do processamento).

respeito à validade e à consistência das decisões. Quando *safety* é violada, dois processos podem chegar a decisões sobre valores diferentes, contrariando a função do consenso. Já a propriedade *liveness* refere-se à capacidade do algoritmo progredir uma vez que o consenso não pode ficar bloqueado indefinidamente, e segundo a Terminação, estados de indecisão permanente não são válidos.

É claro, existem diversos outros protocolos de acordo em sistemas distribuídos, e muitos deles são essenciais a diversos sistemas comumente usados. Um exemplo marcante é o dos protocolos de *commit* distribuído, largamente usados em sistemas de bancos de dados. Nestes protocolos, todos servidores devem concordar se irão decidir fazer o fechamento em acordo (*commit*) sobre as operações ou se vão abortá-las ; se qualquer servidor sugerir que a operação deve ser abortada, todos os servidores deverão abortá-la. Como a presença de um único valor de aborto leva ao cancelamento da operação em todos os servidores, observa-se que o problema do *commit* distribuído é um pouco mais complexo do que o próprio problema do consenso. De fato, essa relação pode ser estendida, e qualquer impossibilidade que se aplique ao consenso pode ser traduzida como uma impossibilidade ao problema do *commit*.

Outro exemplo comum refere-se aos protocolos de difusão atômica com ordenamento de mensagens (também conhecidos como difusão ou *broadcast* atômico). Estes protocolos tentam garantir que, se duas mensagens m e m' forem enviadas, então m será entregue antes de m' em todos processos, ou m' será entregue antes de m em todos os processos. Assim, um sistema que implemente a difusão atômica pode atingir um consenso da mesma forma que, quando o consenso for impossibilitado de executar, a difusão atômica também o será.

De fato, existem diversas operações em sistemas distribuídos cuja complexidade é igual ou superior à do problema de consenso [SAB 95] (sincronização de relógios, eleição, etc), de forma que todas elas estão sujeitas às mesmas restrições do consenso.

Assim, mesmo que seja um problema simples, o consenso é um elemento essencial para a elaboração de protocolos de acordo mais complexos, e a determinação de suas restrições aplica-se diretamente a tais algoritmos.

2.2 Tipos de Consenso

Devido às suas características e aplicações, o problema do consenso está intimamente ligado à elaboração de sistemas tolerantes a falhas. Assim, a determinação do ambiente de falhas é essencial para a correta determinação das exigências e comportamentos que o consenso deverá exibir.

Por exemplo, um sistema pode ser classificado entre os limites de comportamento síncrono ou assíncrono. No modo síncrono, há um conhecimento temporal prévio do comportamento do sistema e de seus componentes (processador, rede, etc), de forma que é possível predizer suas velocidades e tempos máximos de resposta. No caso de sistemas tolerantes a falhas, isso significa que é possível determinar com exatidão se um componente falhou, pois ele não enviou resposta dentro do prazo máximo conhecido pelo sistema. No outro extremo, estão os sistemas assíncronos, que podem ser definidos pela ausência de conhecimento ou limitações temporais de qualquer ordem, dentre as quais pode-se citar os tempos de processamento ou de atraso das mensagens.

Por diversos fatores, a exploração do modelo assíncrono oferece muitas vantagens para o desenvolvimento dos sistemas. A mais simples talvez refere-se à liberdade que os sistemas assíncronos oferecem para a escolha dos componentes onde operam; como não há conhecimento nem limite para os tempos de processamento ou de atraso das mensagens, o ambiente pode ser composto por componentes heterogeneamente distribuídos. Essa liberdade tem um apelo ainda maior quando se considera a complexidade que a determinação dos parâmetros máximos teria em um sistema síncrono; toda vez que o ambiente é modificado, mesmo que em apenas um único componente, a possibilidade de que os parâmetros anteriormente conhecidos tenham que ser reavaliados é muito grande.

Somando-se ao grau de sincronismo, estão os tipos de falhas que o sistema deverá suportar: pode-se considerar que o sistema apresente apenas falhas por colapso (*crash*), onde um componente pára sua operação prematuramente [CHA 96a], ou pode-se considerar modelos de falhas mais complexos, até o caso das falhas arbitrárias ou bizantinas (Lamport *apud* [JAL 94]), onde um componente falho não necessariamente demonstre seu problema, mas poderá apresentar um comportamento não esperado a qualquer instante.

2.3 O Problema do Consenso em Sistemas Distribuídos Assíncronos

Enquanto a construção de sistemas distribuídos usando um modelo assíncrono é muito atraente, existem diversos problemas que devem ser resolvidos. Especialmente no caso das operações de acordo, existe a chamada Impossibilidade FLP. Segundo seus autores (Fischer, Lynch e Patterson [FIS 85]), não há como garantir o cumprimento das propriedades de um protocolo de consenso em um sistema distribuído assíncrono que esteja sujeito até mesmo a uma única falha por colapso. Isso se deve ao fato de que, na ausência de resposta de um dos processos, os demais participantes do consenso não têm como identificar se houve falha ou se o processo apenas está mais lento que os demais. Como em um ambiente puramente assíncrono não é possível impor nenhum limite para a recepção de mensagens, os participantes do consenso irão esperar indefinidamente pela resposta do processo falho. Como mostrado anteriormente, esta espera é uma violação da propriedade Terminação, e representa um sério obstáculo para a implementação do consenso (e dos protocolos que dependem dele) em um sistema distribuído assíncrono.

Face a essa restrição, diversos autores procuraram examinar as limitações que o consenso sofre em outros modelos de falhas ou de sincronismo. Por exemplo, Fischer (*apud* [TUR 94]) demonstrou que o consenso também pode não ser solucionado em ambientes síncronos mesmo que apenas um terço dos processos tenha comportamento bizantino (a exemplo do acordo bizantino de Lamport, citado por [JAL 94], onde o número máximo de processos falhos não poderia ultrapassar (n-1)/3 processos).

Uma opção natural seria a de procurar adaptar algoritmos desenvolvidos inicialmente para sistemas síncronos, aumentando seu nível de assincronismo em alguns aspectos, e respeitando os limites máximos do número de falhas. No entanto, Guerraoui [GUE 97] ressalta que esta prática costuma ser seguida de um inconveniente: quando o limite de componentes falhos f é ultrapassado, estes algoritmos costumam permitir que dois processos decidam diferentemente, violando a propriedade *safety*. Isso não é devido ao ambiente síncrono em si, mas às suposições que os algoritmos fazem para ele: como o próprio ambiente síncrono permite identificar processos falhos dos

processos corretos, os algoritmos não se preocupam com as situações onde apenas alguns processos conseguiram detectar a falha dos componentes.

Frente a essas limitações, alguns autores procuraram especificar sistemas diretamente para situações intermediárias entre o sincronismo e o assincronismo puro. Entre esses modelos pode-se destacar o consenso para sistemas parcialmente síncronos (*partially synchronous*), apresentado por Dolev (*apud* [GUE 97]) e para o assincronismo temporizado (*timed asynchronous*), apresentado por Cristian (*apud* [GUE 97]). Ambos procuram trabalhar com falhas de temporização, ou seja, falhas onde uma mensagem apresente atrasos incomuns (a falha de um processo pode ser considerada como uma falha de temporização que se prolonga indefinidamente).

Os modelos de consenso para sistemas parcialmente síncronos e com assincronismo temporizado geram algoritmos que nunca violam a propriedade *safety*. Ambos assumem que há um limite de tempo δ para o atraso das mensagens, mas não impõem limites ao número de falhas de temporização. No entanto, são necessárias as seguintes restrições para os ambientes de execução:

- o modelo parcialmente síncrono assume a existência de um instante de tempo *t* após o qual não ocorrem mais falhas de temporização;
- o modelo assincronismo temporizado assume que o sistema atravessa períodos estáveis e instáveis. Um período estável é caracterizado pela ausência de falhas de temporização.

Um estudo sobre quais as condições em que se pode obter informações sobre os demais processos, de modo a não comprometer as propriedades do consenso, foi mostrado por Dolev *et al.* (*apud* [TUR 94]). Os autores analisaram um sistemas através de quatro parâmetros:

1. **Processadores**: síncronos ou assíncronos
2. **Comunicação**: atrasos podem ser limitados (*bounded*) ou ilimitados (*unbounded*)
3. **Ordenação**: mensagens podem ser entregues em ordem ou fora de ordem
4. **Transmissão**: o mecanismo de transmissão pode ser ponto a ponto ou através de difusão (*broadcast*)

Com base nesses parâmetros, foi montada a tabela 2.1, que indica se o consenso pode ser resolvido ou não. Por exemplo, em um sistema síncrono com atrasos ilimitados na comunicação, o consenso exige mensagens ordenadas para oferecer garantia de execução.

TABELA 2.1 - Possibilidade de executar o consenso (Dolev *et al. apud* [TUR 94])

	Message Order				
Processors	*Unordered*		*Ordered*		**Communication**
Asynchronous	No	No	Yes	No	*Unbounded*
	No	No	Yes	No	*Bounded*
Synchronous	Yes	Yes	Yes	Yes	
	No	No	Yes	Yes	*Unbounded*
	Point-to-Point	*Broadcast*		*Point-to-Point*	
	Transmission Mechanism				

De modo geral, os casos onde o consenso é solúvel podem ser agrupados em três grupos de comportamento:

- **Caso 1**: processadores síncronos e comunicação limitada.
- **Caso 2**: mensagens ordenadas e transmissão por difusão.
- **Caso 3**: processadores síncronos e mensagens ordenadas.

Em nenhuma destas situações está a possibilidade de operação do consenso em ambientes puramente assíncronos. Isso se deve à impossibilidade de um processo identificar o colapso de outro processo sem usar *timeouts* ou mecanismos similares, que por sua vez, impõem limites aos atrasos que as mensagens podem apresentar.

Para resolver o consenso em ambientes puramente assíncronos, Chandra e Toueg [CHA 96a] propuseram um modelo de consenso auxiliado por detectores de defeitos. De fato, o mérito desta solução foi considerar as informações de componentes que podem cometer enganos, como no caso dos detectores de defeitos, aumentando o conhecimento sobre os demais elementos e estruturando os algoritmos de forma que as detecções incorretas não acarretem inconsistências entre os participantes do consenso.

O funcionamento dos detectores de defeitos será mais bem detalhado no próximo capítulo, mas basicamente, são "oráculos" [AGU 96] que, a partir de mensagens trocadas entre as máquinas ou processos, são capazes de indicar se alguém é suspeito de estar falho. Como os detectores podem cometer enganos, suas suspeitas são utilizadas apenas para evitar que o consenso não fique esperando indeterminadamente pela resposta de um processo falho, de forma que não são violadas nem as propriedades *safety,* nem *liveness*. Assim, os algoritmos de consenso são estruturados para que quando uma suspeita ocorre, os processos passam para a próxima "rodada" do consenso, numa nova tentativa de finalizar o consenso com todos os processos corretos.

2.4 Consenso de Chandra e Toueg

Além da definição dos detectores de defeitos, os quais são classificados em oito categorias diferentes de acordo com suas propriedades, Chandra e Toueg apresentaram dois algoritmos de consenso. Enquanto o algoritmo para detectores *Strong* ($\mathcal{S}$) [CHA 96a] possibilita a detecção com mais processos falhos, o mais conhecido é o algoritmo para detectores *Eventually Strong* ($\Diamond\mathcal{S}$), de modo que este será o modelo

estudado neste trabalho. Mais informações sobre os algoritmos para detectores $\mathcal{S}$ podem ser encontradas nos trabalhos de Chandra e Toueg [CHA 96a] e de Greve [GRE 00].

O detector $\Diamond\mathcal{S}$ tem as seguintes propriedades:

- ***Strong Completeness***: em um tempo finito (*eventually*), todos os processos que falharam serão permanentemente suspeitos por todos os processos corretos.
- ***Eventual Weak Accuracy***: há um instante de tempo após o qual algum processo correto não é mais considerado suspeito por nenhum processo correto.

Estas propriedades garantem que, após um determinado intervalo de tempo (intervalo finito, mas desconhecido), todos processos falhos serão considerados suspeitos, mas ao menos um processo correto não será suspeito por nenhum detector. Com isso, haverá um instante no qual o consenso pode ser finalizado, pois todos processos falhos serão suspeitos. Dessa forma são respeitadas as propriedades do consenso, pois a construção do algoritmo impede que os processos decidam antes desse momento (ou que dois processos decidam de forma diferente).

O algoritmo para detectores do tipo $\Diamond\mathcal{S}$ utiliza o paradigma da rotação de coordenadores; 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 os 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 figura 2.1.

Abstraindo os passos do algoritmo, cada rodada do consenso para detectores $\Diamond\mathcal{S}$ pode ser descrita através das seguintes fases [SER 99]:

- na primeira fase, os processos enviam suas propostas (*estimates*) para o coordenador;
- na segunda fase, o coordenador espera pelas propostas dos processos; quando recebe a maioria das mensagens, ele escolhe um destes valores e propõe aos processos;
- na terceira fase, os processos esperam pela proposta do coordenador. Quando a proposta do coordenador chega, os processos adotam este valor, enviam sua confirmação (mensagem de *ack*) e passam para a próxima rodada. Entretanto, se um dos processos suspeitar do coordenador antes de receber o valor proposto, este irá enviar ao coordenador uma mensagem de confirmação negativa (mensagem *nack*), e irá passar à próxima rodada;
- na quarta fase, o coordenador espera pelas mensagens *ack* ou *nack* da maioria dos processos. Se todas as mensagens forem *ack*, então a proposta torna-se decisão, e o coodenador executa uma difusão confiável para todos os processos. Quando um processo recebe a mensagem correspondente gerada pela difusão confiável, também adota esse valor como decisão, e retorna à aplicação.

Pode-se verificar, portanto, que os processos progridem através de rodadas assíncronas, ou seja, quando um processo envia seu *ack* ou *nack*, ele não espera até que os outros também passem para a próxima rodada. O que impede que os processos se distanciem é a necessidade do coordenador de cada rodada obter uma maioria de respostas. Isso também indica que independentemente da rodada atual dos processos, quando estes recebem a difusão confiável de algum coordenador, pode-se ter certeza que, em alguma rodada o coordenador conseguiu chegar à decisão final.

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 $q\}$
      $ts_p \leftarrow$ 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 2.1 - Algoritmo de consenso para detectores $\Diamond\mathcal{S}$ [CHA 96a]

Pode-se observar também na terceira parte (*Phase 3*) do algoritmo de Chandra (figura 2.1) um mecanismo que impede os coordenadores de rodadas diferentes proporem valores diferentes: se um processo recebe a proposta de um coordenador, ele adota esse valor como sua proposta (não decisão), independentemente se o coordenador irá conseguir maioria de *acks* ou não. Dessa forma, se o algoritmo passar para a próxima rodada, o novo coordenador irá receber o valor proposto pelo coordenador da primeira rodada da maioria dos processos; apenas os processos que mandaram *nack* não terão atualizado seus valores, mas estes também não terão incrementado seu *timestamp ts*, e portanto suas sugestões não serão consideradas pelo coordenador.

2.5 Outros Algoritmos de Consenso

Os algoritmos propostos por Chandra e Toueg são os mais conhecidos para o uso com detectores de defeitos em ambientes com falhas por colapso. Não obstante, alguns pesquisadores dedicaram-se a otimizar estes algoritmos, procurando, por exemplo, reduzir o número de rodadas ou o número de mensagens enviadas em cada rodada. Esta seção apresentará alguns exemplos encontrados na literatura.

2.5.1 *Early Consensus*

A observação do algoritmo de Chandra e Toueg (figura 2.1) para detectores $\diamond\mathcal{S}$ leva à constatação de que nele existem passos de comunicação em excesso; como o tempo de transmissão das mensagens é um dos principais fatores que reduzem o desempenho das aplicações distribuídas, a redução do número de passos permite que o tempo de terminação do consenso seja menor. Além disso, pode-se observar que todos processos são dependentes da difusão confiável do coordenador e, sem ele, não podem decidir sobre um valor.

Pela avaliação de Schiper [SCH 97], o algoritmo de Chandra exige no mínimo quatro passos de comunicação para atingir o consenso (considerando, é claro, que não ocorra nenhuma suspeita). Uma possível otimização é apresentada na figura 2.2. Ao invés de coletar as propostas dos processos num primeiro momento, e só então escolher qual o valor que deverá ser decidido, o coordenador da rodada pode tentar impor sua proposta, eliminando a primeira fase do algoritmo de Chandra e reduzindo o número de passos de comunicação para três. No entanto, mesmo com esta otimização, os processos ainda continuam dependentes da difusão confiável do coordenador para retornarem a decisão à aplicação.

O algoritmo de Schiper intitulado "*Early Consensus*" (consenso antecipado) procura solucionar ambos problemas: primeiro, reduz o número de passos de comunicação (o algoritmo usa dois passos no melhor caso), e também permite que os processos cheguem à decisão assim que receberem uma maioria de mensagens de confirmação, ao invés de esperar a contabilização de um coordenador e sua posterior difusão confiável.

Assim como no algoritmo de Chandra, a proposta de Schiper considera o paradigma da rotação de coordenadores, e cada processo controla uma variável local *estimatei*. Em uma rodada *r*, o coordenador c_r tenta impor sua proposta, e para isso envia uma mensagem contendo o valor de sua variável *estimate* para todos os processos.

Entretanto, os processos não respondem com um *ack*, a exemplo do algoritmo de Chandra. Quando o processo recebe $estimate_C$, ele repassa esse valor para todos os processos do consenso. Quando um processo recebe mensagens da maioria dos processos do sistema, ele pode decidir o valor $estimate_C$.

```
Every process p executes the following:

procedure propose(v_p)
  estimate_p ← v_p                          {estimate_p is p's estimate of the decision value}
  state_p ← undecided
  r_p ← 0                                   {r_p is p's current round number}
  ts_p ← 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 ← r_p + 1
  c_p ← (r_p mod n) + 1                     {c_p is the current coordinator}

    Phase 1: {The current coordinator sends estimate_Cp to all processes}
      send (c_p, r_p, estimate_Cp, ts_p) to all

    Phase 2: {All processes wait for the estimate proposed by the current coordinator}
      wait until [received (c_p, r_p, estimate_Cp) from c_p or c_p ∈ 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 ← estimate_Cp
        ts_p ← 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 3: {The current coordinator waits for ⌈(n+1)/2⌉ replies. If they indicate that
     ⌈(n+1)/2⌉ processes adopted its estimate, the coordinator R-broadcasts a decide message}
    if p = c_p then
    wait until [ for ⌈(n+1)/2⌉ processes q : received (q, r_p, ack) or (q, r_p, ack)]
    if [for ⌈(n+1)/2⌉ 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 ← decided
```

FIGURA 2.2 - Otimização para o algoritmo de Chandra e Toueg

Se, no entanto, o coordenador da rodada *r* falha ou é considerado suspeito pela maioria dos processos, o consenso avança para a próxima rodada. Antes disso, porém, todos os processos devem atualizar suas variáveis *estimatei*, para satisfazer a propriedade Acordo Uniforme. Segundo ela, se um processo decidiu $estimate_C$ na rodada *r*, a decisão das próximas rodadas não poderá ser diferente de $estimate_C$. A figura 2.3 mostra o algoritmo *Early Consensus* de Schiper.

```
function early-consensus(vi);              /* algorithm for a process pi */
  ri ← 0                                   /* current round */
  estimatei ← (i,vi)      /* the two fields are written "estimatei.first", respt "estimatei.second" */
  phasei                  /* each round has two phases, numbered 1 and 2 */
  coordi                  /* coordinator from round ri */
  currentRoundTerminatedi /* boolean variable */
  msgCounteri             /* integer variable */
  coordSuspectedi         /* "true" if and only if coordi is suspected */
  nbSuspicionsi           /* integer variable */

  cobegin
  || upon reception of (pj, rj, vj, decide) from pj :
    send (pi, rj, vj, decide) to all;
    return vj;

  || loop
    phasei ← 1; currentRoundTerminatedi ← false;
    coordSuspectedi ← false; nbSuspicionsi ← 0;
    coordi ← (ri mod n) + 1;

    if i = coordi then                     /* pi is the coordinator for the current round */
      send (pi, ri, 1, estimatei) to all;

    while not currentRoundTerminatedi
      select                               /* select one of the braches below */

      upon reception of (pj, ri, 1, estimatej) from pj when phasei = 1:
        /* estimatej is the estimate of the coordinator of round ri */
        first reception: msgCounteri ← 1;
                         if i ≠ coordi then estimatei ← estimatej;
                                            send (pi, ri, 1, estimatei) to all;
        other receptions: msgCounteri ← msgCounteri + 1;
        if msgCounteri > n/2 then
           send (pi, ri, estimatei.second, decide) to all;
           return estimatei.second;

      upon coordi ∈ ◇S when not coordSuspectedi :
        send (pi, ri, suspicion) to all;
        coordSuspectedi ← true;

      upon reception of (pj, ri, suspicion) from process pj :
        nbSuspicionsi ← nbSuspicionsi + 1;
        if nbSuspicionsi > n/2 then  phasei ← 2;
                                     send (pi, ri, 2,estimatei,) to all;

       upon reception of (pj, ri, 2,estimatej) from pj :
        first reception: msgCounteri ← 1;
                         if phasei = 1 then phasei ← 2;
                                            send (pi, ri, 2,estimatei,) to all;
        other receptions: msgCounteri ← 1;
        if estimatei.first = coordi then estimatei ← estimatej;
        if msgCounteri > n/2 then
          currentRoundTerminatedi ← true;
          ri ← ri + 1;

      end select
    end while
  end loop
coend
```

FIGURA 2.3 - Algoritmo *Early Consensus* [SCH 97]

O protocolo de consenso inicia quando cada processo chama a função `early-consensus()`, passando seu valor inicial v_i como parâmetro. A chamada à função termina quando um processo p_i executa a instrução **return**; neste momento, diz-se que p_i decidiu o valor v.

A função `early-consensus()` é composta por duas tarefas (*tasks*) concorrentes. A primeira tarefa é reponsável pela recepção da mensagem de decisão (p_j, v_j, *decide*), e pela retransmissão desta mensagem para todos os processos. Isto assegura que, se um processo correto decide, então todos os processos corretos irão decidir sobre este valor também.

A segunda tarefa é o mecanismo central do algoritmo. A variável r_i representa a rodada atual em que o processo p_i se encontra. Esta variável é incluída em todas as mensagens enviadas por p_i; com isso, um processo apenas aceita mensagens que foram enviadas em uma rodada igual à sua rodada atual. Cada rodada no algoritmo de *early consensus* é composto por duas fases, chamadas 1 e 2:

- na fase 1 de cada rodada, o algoritmo tenta impor o valor *estimate* do coordenador da rodada;
- se o coordenador da rodada for suspeito pela maioria dos processos, então a fase 2 do algoritmo é usada para definir um novo consenso, para ser resolvido na próxima rodada. O valor inicial proposto pelo coordenador da próxima rodada é a variável *estimate* que aquele processo tinha no final da fase 2 da rodada anterior.

2.5.2 *Sliding Round Window*

Os algoritmos para $\langle\rangle\mathcal{S}$ (por exemplo, o algoritmo de Chandra e o *Early Consensus* de Schiper) costumam seguir a mesma estrutura de controle: os processos executam rodadas assíncronas, de forma que seja possível convergir para o mesmo valor em algum instante (e então decidir sobre este valor). Cada rodada é gerenciada por um coordenador predeterminado, que tenta impor seu valor *estimate* atual como o valor da decisão final. Entretanto, em todos protocolos, cada um dos n processos executa as rodadas seqüêncialmente, até que este decida ou falhe. Uma das principais diferenças entre estes algoritmos está no padrão de troca de mensagens utilizado em cada rodada [HUR 00]:

- cada rodada do algoritmo de Chandra utiliza um padrão de centralização das mensagens. As mensagens são originárias do ou dirigem-se para o coordenador da rodada. Um processo passa de uma rodada r para uma rodada r +1 quando este enviou um ACK para o coordenador, ou quando suspeita deste coordenador (enviando assim um NACK). Quando um coordenador suspeita de todos os outros processos, ele pode ir diretamente para a próxima rodada que deveria coordenar. Este salto não requer sincronização, mas o coordenador de uma nova rodada tem que esperar pelas mensagens dos outros processos. Isso assegura que, se um valor foi decidido pelos outros

processos, a nova rodada vai ter que ser iniciado com um valor de *estimate* igual ao valor decidido pelos outros processos.

- O protocolo de Schiper utiliza um padrão de troca de mensagens descentralizado. O fim de cada rodada é controlado por uma barreira de sincronização que previne os processos de passarem para a próxima rodada com um valor se algum dos outros processos decidiu diferentemente. Esta barreira requer que um processo receba um número "suficiente" de mensagens dos outros processos antes de poder ir para a próxima rodada.

Ainda assim, os protocolos de Chandra e de Schiper, quando envolvidos em uma rodada r, executam apenas as funções referentes a essa rodada e não participam mais de nenhuma rodada $r' < r$. Conceitualmente, poder-se-ia dizer que os protocolos tradicionais executam com um *sliding round window*[3] de tamanho 1. O algoritmo de consenso proposto por Hurfin *et al.* [HUR 00] permite que os processos operem dentro de um *sliding round window* com tamanho n, ou seja, continuem a executar a rodada atual e as n-1 rodadas anteriores. De fato, a limitação imposta pela sincronização dos processos define que um processo só precisa estar fortemente sincronizado com os demais a cada n rodadas, sem ser impedido de participar simultaneamente em diversas rodadas. Assim, por exemplo, o algoritmo de Chandra só faz essa sincronização a cada n rodadas, quando o processo volta a ser o coordenador, enquanto o algoritmo de Schiper obriga os processos a sincronizarem-se no fim de todos as rodadas.

Além desse controle mais fraco sobre a sincronização dos processos, o algoritmo proposto por Hurfin adiciona um mecanismo que permite controlar dinamicamente o número de mensagens trocadas entre os processos em uma determinada rodada, de acordo com a sua análise da carga da rede. O protocolo permite que cada processo defina dinamicamente um conjunto de processos X_i; em cada rodada, um processo p_i deve enviar mensagens que auxiliem a tomada da decisão naquela rodada; e ao controlar o conjunto de processos X_i que irão receber essas mensagens, o protocolo pode adaptar seu comportamento ao tráfego na rede.

Para controlar o consenso em diversas rodadas, o protocolo de Hurfin utiliza três tipos de variáveis:

- variáveis que abrangem toda computação:
  - r_i: número da última rodada que p_i entrou;
  - est_i: valor atual do *estimate*;
  - ts_i: *timestamp* usado em conjunto com est_i.
- variáveis relacionadas à última rodada:
  - *pos_ack*$_i$: variável booleana que indica que os processos já enviaram uma mensagem de reconhecimento positivo (ACK);
  - neg_ack_i: variável booleana que indica que os processos enviaram um reconhecimento negativo (NACK).

3 "Janela deslizante de rodadas", uma forma de manter o controle sobre um conjunto de rodadas ativas.

- variáveis usadas para o controle da janela de rodadas:
  - $prevc_i$: número da última rodada coordenado por p_i;
  - $bnextc_i$: número da rodada que precede a próxima rodada que p_i irá coordenar ($bnextc_i = prevc_i + n - 1$);
  - tot_prevc_i: número total de mensagens do tipo *estimate* recebidas quando $r = prevc_i$;
  - tot_bnextc_i: número total de mensagens do tipo *estimate* recebidas quando $r = bnextc_i$;
  - $pos_i[r_i - (n - 1):r_i]$: um *array* que representa a janela deslizante das rodadas, e onde cada entrada contém o número de ACKs recebidos em uma determinada rodada.

O protocolo de consenso de Hurfin é composto de três tarefas concorrentes: a primeira gerencia o lançamento das rodadas de forma seqüencial, enquanto as outras duas tarefas são responsáveis pelo tratamento das mensagens recebidas.

A tarefa T1 inicia novas rodadas até que seja atingida uma decisão. No entanto, o lançamento de uma nova rodada não significa que o processo p_i pare de participar das rodadas anteriores. De fato, o comportamento da tarefa T1 depende do processo que a executa:

- se p_i é o coordenador ds rodada r (sendo r a rodada atual), a tarefa T1 reinicia suas variáveis de controle da janela, incrementa o *timestamp* ts_i para o valor r e faz uma difusão da mensagem EST(r, est_i, r), anunciando assim para os demais processos que iniciou a rodada r;
- se p_i não é o coordenador da rodada r, a tarefa T1 espera até que ocorra a suspeita do coordenador da rodada, ou até que receba uma mensagem do tipo *estimate* do coordenador. Além disso, como o algoritmo só pode atingir a decisão após coletar um número suficiente de mensagens de reconhecimento, o processo só poderá iniciar uma nova rodada após ter enviado um ACK ou NACK para o coordenador. Assim, compete à tarefa T1 o envio das mensagens de reconhecimento negativo (EST(r_i,est_i,ts_i) com $r_i \neq ts_i$) quando ocorre uma suspeita sobre o coordenador da rodada e o processo p_i ainda não tenha enviado um ACK; a mensagem de reconhecimento positivo (EST(r_i,est_i,ts_i) com $r_i = ts_i$) é enviada pela tarefa T2. Para evitar que o algoritmo bloqueie, a mensagem de NACK é enviada para o coordenador da rodada atual (p_{cc}) e para o coordenador da próxima rodada (p_{nc}); não é necessário enviar essa mensagem para os demais processos, pois o NACK não auxilia na decisão.
- adicionalmente, se o processo p_i será o coordenador da próxima rodada, o protocolo força o processo a esperar até que não seja mais possível que a rodada controlada anteriormente por p_i seja decidida (prevenindo o lançamento excessivo de rodadas) ou até que a variável est_i seja atualizada (evitando que a propriedade Acordo seja violada).

A tarefa T2 gerencia o recebimento das mensagens EST(*r*, *est*, *ts*). A primeira providência de T2 diz respeito às mensagens de rodadas fora da janela; as mensagens antigas ($r \leq r_i - n$) são descartadas, enquanto as mensagens de rodadas que ainda não estão sendo tratadas ($r > r_i$) são momentaneamente atrasadas.

Se as mensagens forem do tipo ACK (EST(r_i,est_i,ts_i) com $r_i = ts_i$) o contador de ACKs ($pos_i[r]$) é incrementado; se o número de ACKs recebidos for suficiente, o processo aceita o valor enviado pelo coordenador da rodada. Além disso, se a rodada da mensagem é a mais recente rodada em que o processo p_i participa, esta mensagem é retransmitida para indicar que foi aceita a proposta do coordenador. O conjunto de processos que receberão essas mensagens (X_i) é ajustado dinamicamente a cada rodada, mas deve incluir no mínimo os coordenadores da rodada atual e da próxima rodada.

A tarefa T3 implementa uma difusão confiável, para evitar que um processo fique indefinidamente bloqueado esperando por um valor que já foi decidido.

O algoritmo *Sliding Round Window* de Hurfin pode ser visto na figura 2.4:

Function Consensus (v_i)

$r_i \leftarrow 0$; $est_i \leftarrow v_i$; $ts_i \leftarrow 0$; $nc \leftarrow 0$; $prevc_i \leftarrow 0$; $tot\_prevc_i \leftarrow n$; $bnextc_i \leftarrow (i - 1)$; $tot\_bnextc_i \leftarrow 0$;

Task T1:
while *true* **do**
$r_i \leftarrow r_i + 1$; $cc \leftarrow nc$; $nc \leftarrow (r_i \bmod n) + 1$; $pos\_ack_i \leftarrow false$; $neg\_ack_i \leftarrow false$; $pos_i[r_i] \leftarrow 0$;
if ($i = cc$)
then $prevc_i \leftarrow r_i$; $tot\_prevc_i \leftarrow 0$; $bnextc_i \leftarrow prevc_i + n - 1$; $tot\_bnextc_i \leftarrow 0$; $ts_i \leftarrow r_i$;
broadcast EST(r_i, est_i, ts_i); $pos\_ack_i \leftarrow true$
else wait until (($cc \in suspected_i$) $\vee$ pos_ack_i);
if ($\neg pos\_ack_i$) **then** send EST(r_i, est_i, ts_i) to $\{p_{cc}, p_{nc}\}$; $neg\_ack_i \leftarrow true$ **endif**;
if ($i = nc$) **then**
wait until (($tot\_prevc_i \geq n - f$) $\wedge$ ((pos_ack_i) $\vee$ ($tit\_bnextc_i \geq n - f$))) **endif**;
endif

Task T2:
upon *receipt* **of** EST(r_i, est_i, ts_i) **such that** $r_i - n < r \leq r_i$;
% If $r \leq r_i - n$: the message is discarded. If $r_i < r$: the message is delayed %
if ($ts > ts_i$) **then** $ts_i \leftarrow ts$; $est_i \leftarrow est$ **endif;**
if ($r = ts$) **then**
$pos_i[r] = pos_i[r] + 1$;
if ($pos_i[r] \geq f + 1$) **then** $\forall j \neq i$: send DECISION(est_i) to p_j; return (est_i); **endif;**
if (($r = r_i$) $\wedge$ $\neg pos\_ack_i$ $\wedge$ $\neg neg\_ack_i$) **then**
let X_i **be** a locally defined set including $\{p_{cc}, p_{nc}\}$;
send EST(r, est, ts) to X_i; $pos\_ack_i \leftarrow true$; **endif;**
endif;
if ($r = bnextc_i$) **then** $tot\_bnextc_i \leftarrow tot\_bnextc_i + 1$; **endif;**
if ($r = prevc_i$) **then** $tot\_prevc_i \leftarrow tot\_prevc_i + 1$; **endif**

Task T3:
upon receipt **of** DECISION (est) **from** p_k: $\forall j \neq i,k$: send DECISION(est) to p_j; return (est)

FIGURA 2.4 - O algoritmo de consenso *Sliding Round Window* [HUR 00]

2.6 Análise de Custos

A análise estática do custo de um algoritmo pode ser feita de diversas maneiras. No que se refere aos sistemas distribuídos, existem duas métricas que são muito eficazes, o número de mensagens trocadas e o grau de latência. Os objetos de análise dessas métricas têm resultado direto sobre as principais limitações de desempenho dos algoritmos distribuídos, sendo portanto fatores essenciais para a definição do algoritmo mais adequado à situação. Nas próximas seções, serão apresentadas as bases dessas métricas, que serão utilizadas para a análise dos algoritmos de consenso descritos. Posteriormente, essas métricas serão utilizadas também para a avaliação dos modelos de detectores de defeitos.

2.6.1 Número de mensagens trocadas

O número de mensagens que os algoritmos trocam reflete diretamente no tráfego da rede: essa normalmente é uma das principais restrições à escalabilidade dos sistemas distribuídos (pequenas variações podem ocorrer devido ao tamanho de cada mensagem trocada). Como o consenso é uma operação que pode se estender por diversas rodadas, caso ocorram sucessivas suspeitas, são consideradas apenas as mensagens trocadas no melhor caso, ou seja, quando o consenso é finalizado em uma única rodada. Schiper [SCH 97] considera dois possíveis parâmetros: o número de mensagens necessárias para atingir o consenso e o número total de mensagens trocadas. O problema com a primeira hipótese é que ela subentende um grau de confiabilidade e conhecimento que não são comuns nos sistemas assíncronos, de forma que normalmente os algoritmos fazem uma difusão confiável após atingirem sua decisão, a fim de garantir que os demais processos fiquem consistentes. Assim, será analisado aqui o número de mensagens (ponto a ponto) trocadas pelos algoritmos, até que seja atingido o consenso. É claro, se a rede que interliga os processos tivesse primitivas de difusão, o custo seria muito menor, uma vez que quase todas as operações são disseminações das mensagens, mas este não é o caso do nosso ambiente de execução, que utiliza apenas mensagens ponto-a-ponto.

- **Algoritmo de Chandra e Toueg [CHA 96a]**: como toda comunicação é centralizada no coordenador (exceto a difusão confiável da decisão), o algoritmo de Chandra e Toueg requer (n-1) mensagens para cada uma das quatro primeiras etapas (*estimate*, *propose*, ACK/NACK e *decision*), e n(n-1) mensagens para a difusão confiável. Assim, o algoritmo de Chandra gera (4+n)(n-1) mensagens para finalizar o consenso. Isto representa um custo de $O(n^2)$ mensagens.
- **Algoritmo otimizado de Chandra e Toueg [SCH 97]**: por não ter a primeira etapa do algoritmo tradicional, este gera (n-1) mensagens a menos. Assim, o número total de mensagens geradas pelo algoritmo é de (3+n)(n-1) mensagens, embora o custo continue sendo de $O(n^2)$, pois difusão confiável no final do algoritmo obriga a troca de mensagens entre todos os processos.
- ***Early Consensus* de Schiper [SCH 97]**: o algoritmo apresenta duas etapas. Na primeira, o coordenador envia sua proposta para os processos, e estes repassam a mensagem para os demais (total de n(n-1) mensagens). Na

segunda etapa, quando um processo recebe a maioria das retransmissões, ele envia a sua decisão para todos os processos; se estes ainda não tinam um valor de decisão, adotam o valor recebido e repassam a mensagem de decisão para os demais (totalizando n(n-1) mensagens). Assim, se não ocorrem suspeitas (e conseqüentemente, novas rodadas), o algoritmo de Schiper envia um total de 2n(n-1) mensagens, a um custo de $O(n^2)$ mensagens.

- ***Sliding Round Window* [HUR 00]**: o algoritmo de Hurfin segue o mesmo modelo de comunicação do *Early Consensus* de Schiper. Na verdade, a diferença entre os dois algoritmos está na possibilidade de participar de múltiplas rodadas ao mesmo tempo e na forma como são montadas as mensagens de NACK (no algoritmo de Schiper é gerada uma mensagem *suspicion*, enquanto no algoritmo de Hurfin os componentes *r* e *ts* da mensagem são comparados). Em uma rodada sem suspeitas (melhor caso), o algoritmo de Hurfin também gera um total de 2n(n-1) mensagens, com um custo equivalente a $O(n^2)$.

O que se pode observar nestes exemplos é que a complexidade do modelo de comunicação é a mesma ($O(n^2)$). No entanto, o algoritmo de Chandra (e sua otimização) apresentam um número de mensagens geradas levemente inferior ao dos demais protocolos. Dessa forma, se for considerado apenas o tráfego da rede, a escolha do algoritmo mais eficiente poderia recair sobre o consenso de Chandra e Toueg; porém, os demais algoritmos foram propostos com o objetivo de melhorar a eficiência do consenso, então deve-se tomar uma segunda métrica que auxilie na avaliação de outro ponto crucial no desempenho dos sistemas distribuídos.

2.6.2 Graus de latência

É comum comparar o custo de algoritmos distribuídos em termos de número de fases. No entanto essa medida é um pouco ambígua, uma vez que o número de fases não expressa de maneira uniforme o número de passos de comunicação. Por exemplo, um algoritmo com uma fase tem dois passos de comunicação (mensagem e ACK), enquanto um de duas fases tem três passos de comunicação (por exemplo, o algoritmo *two phase commit* – 2PC) [SCH 97].

Schiper [SCH 97] introduziu o conceito de graus de latência, que podem ser descritos como uma pequena variação do relógio lógico de Lamport [LAM 78]:

- eventos locais e de *send* em um processo p_i não modificam o relógio lógico de p_i;
- seja *ts*(*send*(*m*)) o *timestamp* do evento de *send*, e *ts*(*m*) o *timestamp* carregado pela mensagem *m*. Define-se então que *ts*(*m*) = *ts*(*send*(*m*))+1;
- o *timestamp* de um evento *receive*(*m*) no processo p_i é o maior entre os valores do *timestamp* da mensagem (*ts*(*m*)) e o *timestamp* do evento de p_i imediatamente anterior ao evento de *receive*(*m*).

Quando se analisa algoritmos de acordo, o evento mais importante é o da decisão, então a determinação do *timestamp* deste evento pode dar mais informações sobre o algoritmo. Assim, para um determinado algoritmo de acordo, define-se o grau de latência como o maior *timestamp* até atingir o evento da decisão. Por exemplo, em um algoritmo onde 1) um processo p_i envia uma mensagem para todos os processos de seu grupo e 2) cada processo $p_j \neq p_i$, após receber a mensagem de p_i, envia um *ack*(m) para p_i. O processo p_i decide tão logo receber as mensagens de *ack*(m) da maioria dos processos, e nenhum outro processo decide mais. Este algoritmo comumente chamado de protocolo de uma fase tem então um grau de latência igual a 2.

É claro, um algoritmo de acordo pode chegar às decisões com diferentes graus de latência. No exemplo do consenso de Chandra e Toueg, o número de rodadas pode estender-se muito, se ocorrerem falhas ou suspeitas; assim, o grau de latência é definido como a menor latência que um algoritmo A pode produzir em todas as possíveis situações. A latência mínima é normalmente obtida em um caso onde não ocorram suspeitas, que é a situação mais freqüente [GUE 97].

Com esses parâmetros, pode-se classificar os algoritmos segundo a tabela 2.2:

TABELA 2.2 - Graus de latência de alguns protocolos de acordo

Algoritmo	***Graus de Latência***
Chandra e Toueg	4
Chandra e Toueg otimizado	3
Early Consensus	2
Sliding Round Window	2
Two phase commit (2PC)	3
Three phase commit (3PC)	5

2.7 Considerações Finais

Neste capítulo foram apresentados o problema do consenso e a proposta de Chandra e Toueg para resolver esse problema, utilizando detectores de defeitos. Além do algoritmo de Chandra e Toueg, também foram exibidos alguns algoritmos para detectores $\langle\rangle\mathcal{S}$ destinados a otimizar o desempenho do consenso: estes modelos foram comparados ao algoritmo original, com relação ao número de mensagens geradas e ao grau de latência.

No próximo capítulo serão apresentadas a definição dos detectores de defeitos, as características de alguns modelos encontrados na bibliografia e também uma análise do custo destes detectores para os sistemas.

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 [FIS 85], 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. Dessa forma, um detector tem como referência apenas 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, 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 [CHA 96a].

Para determinar as características dos detectores 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.

Neste capítulo, inicialmente serão revisadas as definições do modelo de sistema distribuído e das classes de detectores de defeitos propostas por Chandra [CHA 96a]. Após, serão conceituados os modelos de detectores de defeitos estudados neste trabalho. Por fim, serão apresentadas as características de implementação que foram mais marcantes, para cada detector desenvolvido.

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 colapso (*crash*), 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:

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

$$correct(F) = \Pi - crashed(F).$$

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 no momento 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 será possível constatar no final deste capítulo.

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 tempo finito (*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 tempo finito (*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 terá utilidade pois não fornecerá nenhuma informação sobre os defeitos. Para que seja útil, a propriedade de *completeness*, independente do nível escolhido, deve ser utilizada em conjunto com um dos níveis da propriedade de *accuracy*, que restringe 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:

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

Como é difícil (se não impossível) obter *strong accuracy* na maior parte dos sistemas práticos, também são definidos outros níveis mais flexíveis:

- ***Weak Accuracy***: existe pelo menos um processo correto que jamais tornar-se suspeito de falha. Formalmente, $\mathcal{D}$ satisfaz a propriedade de *weak accuracy* se:

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

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, mas por um tempo finito. Isso significa que a exigência de que um dos níveis, *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 a propriedade *eventual strong accuracy* se:

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

- ***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:

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

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 (tabela 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.

TABELA 3.1 - Classe de detectores de defeitos [CHA 96a]

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* $\Diamond\mathcal{W}$

Algumas dessas classes (as mais exigentes e, portanto, mais confiáveis) não encontram implementação correspondente 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, 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, acrescidas de restrições e controles especiais.

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". Até mesmo para a elaboração de um detector com suporte a falhas de colapso, 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 á opção entre os níveis de completeza: *strong completeness* ou *weak completeness*. A exigência da propriedade *weak completeness* de que "todos processos que falharam serão permanentemente suspeitos por **alguns** processos corretos" é 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", é muito mais fácil de compreender, mas mais difícil de implementar. Dessa forma, a especificação dos algoritmos pode ser feita partindo do comportamento caracterizado como *strong completeness*, mas a implementação pode ser feita com um detector que satisfaça apenas *weak completeness*, acrescida de algumas restrições (por exemplo, de que todos os detectores compartilhem suas listas de suspeitos).

Chandra e Toueg [CHA 96a] comprovaram que as relações de redutibilidade entre os detectores podem ser descritas ao variar os níveis relacionados à propriedade *completeness*, mantendo fixo um determinado grau de *accuracy*. 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 figura 3.1.

Enquanto a proposta de Chandra e Toueg para o algoritmo de redução utiliza a unificação das listas de suspeitos, é possível imaginar algoritmos de redução que utilizem o mesmo mecanismo para unificar as listas de processos corretos. Outra

possibilidade é a de periodicamente utilizar o consenso para unificar estas listas, como sugere Lung [LUN 00].

```
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*

Esses algoritmos permitem que sejam estabelecidas relações entre as seguintes classes de detectores:

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

Com isso, Chandra *et al.* [CHA 96b] puderam demonstrar que o detector de defeitos mais fraco (simples) capaz de resolver o problema do consenso pertence à classe dos *Eventually Weak* ($\Diamond\mathcal{W}$), desde que o sistema assegure uma maioria de processos corretos ((n-1)/2 processos falhos, no máximo). Se há a necessidade de garantir o consenso com até n-1 processos falhos (ao menos um processo correto), deve-se utilizar a classe dos *Weak* ($\mathcal{W}$). 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* - $\Diamond\mathcal{W}$ [CHA 96a][CHA 96b]:

- a qualquer tempo *t*, os processos não podem usar o detector $\Diamond\mathcal{W}$ 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 $\diamond\mathcal{W}$ pode cometer um número infinito de enganos. Assim, um detector $\diamond\mathcal{W}$ 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;
- a modificação das propriedades através do uso do nível *eventually* de um detector $\diamond\mathcal{W}$ 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 $\diamond\mathcal{W}$ com informações adicionais. Na prática, 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, este pode obrigar que a aplicação não mantenha a propriedade de *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 de forma inconsistente ou com valores inválidos. Se um algoritmo de difusão ou *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 entregá-las fora de ordem.

3.6 Modelos de Detectores de Defeitos

As propriedades *accuracy* e *completeness* propostas por Chandra e Toueg têm como principal característica descrever o comportamento de detectores de defeitos capazes de dar suporte às operações de consenso, sem no entanto limitar o desenvolvimento dos detectores a alguma técnica específica. Não obstante, eles fizeram algumas análises sobre como poderia ser implementado um detector de defeitos utilizando *timeouts*, e poucas variações sobre esta idéia são apresentadas na literatura. A influência do trabalho de Chandra e Toueg também pode ser sentida na definição de detectores para outros modelos de falhas. As propriedades *accuracy* e *completeness* de Chandra e Toueg foram definidas para ambientes com falhas por colapso, e mesmo os

detectores para falhas mais "complexas" procuram seguir a estrutura do trabalho original, mesmo que suas propriedades contenham definições específicas para estes ambientes. Esta seção irá mostrar os modelos mais tradicionais de detectores de defeitos e algumas variações encontradas na literatura, analisando suas características.

O grande problema encontrado para a elaboração desta seção refere-se à nomenclatura utilizada para definir os detectores de defeitos. De fato, pelos exemplos encontrados na literatura, não parece existir nenhuma padronização ou consenso sobre o nome dos detectores mais comuns. Além disso, muitos autores dão nomes específicos aos seus detectores, de forma que a confusão persiste. Por exemplo, um detector que envia mensagens do tipo *"I am alive"* é comumente referenciado por *heartbeat* ou *Push*. Entretanto, existem outros modelos de detectores que são chamados pelos seus autores *Heartbeat* [AGU 97a] e *ad-hoc "heart-beat"* [SER 99], apesar de apresentarem características diferentes daquele. Para evitar essa confusão, optou-se por utilizar a nomenclatura proposta por Felber [FEL 98] para os detectores tradicionalmente conhecidos como *"I am alive"* e *"Are you alive?"*, uma vez que estes não têm um autor conhecido, enquanto para os demais detectores optou-se por utilizar os nomes definidos por seus autores.

3.6.1 Detector *Push*

No modelo *Push* [FEL 98], o fluxo de controle segue a mesma direção do fluxo das informações, ou seja, dos processos monitorados em direção ao detector. Para isso, em um detector *Push*, os processos monitorados necessitam estar ativos (quando se aplica esse conceito sobre objetos Java, isso significa que os objetos monitorados 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 colapso). Se um detector de defeitos não receber tais mensagens dentro de um limite específico de tempo, elee passa a suspeitar do processo monitorado. Este método é consideravelmente eficiente, pois as mensagens enviadas no sistema são unidirecionais (*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 figura 3.2 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 notifica o cliente quando um processo monitorado é considerado suspeito, enquanto as mensagens de "*I'm Alive!*" são enviadas continuamente.

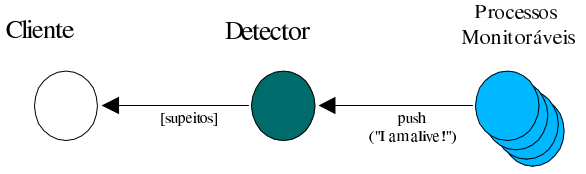

FIGURA 3.2 - Modelo *Push* de monitoramento [FEL 98]

A troca de mensagens entre o detector e os processos monitorados com o modelo *Push* é mostrada na figura 3.3. 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 mensagem nova do processo monitorado, então inicia a suspeita sobre esse processo.

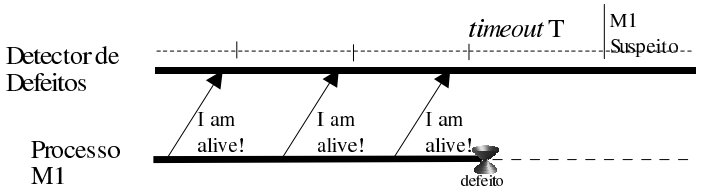

FIGURA 3.3 - Monitorando mensagens no modelo *Push*

3.6.2 Detector *Pull*

O modelo *Pull* [FEL 98] 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 junto aos processos monitorados. 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 ser bidirecional (*two-ways*), esse modelo tende a ser menos eficiente do que o modelo *Push*[4], 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 figura 3.4 exibe o fluxo das informações de monitoramento.

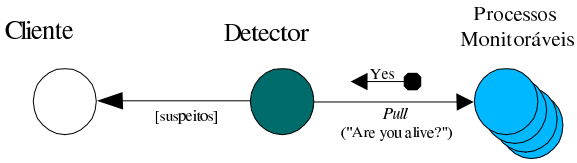

FIGURA 3.4 - O fluxo do modelo *Pull* [FEL 98]

A forma com que as mensagens são trocadas entre o detector e um processo monitorado é detalhada na figura 3.5. Periodicamente, o detector envia uma mensagem

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

de questionamento (*liveness-request*) 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.

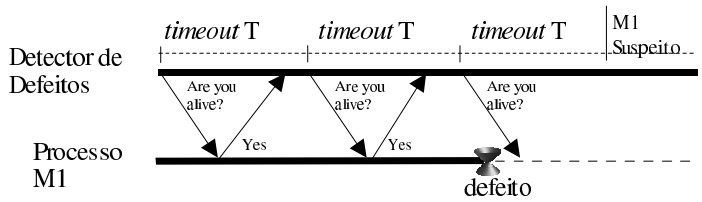

FIGURA 3.5 - Mensagens de monitoramento

3.6.3 Detectores adaptativos

Os detectores adaptativos, de modo geral, não representam um modelo diferente de realizar a detecção de defeitos. Estes detectores na realidade são providos de um mecanismo que procura melhorar a precisão das suspeitas dos detectores, que é especialmente utilizado junto aos detectores *Push* e *Pull*, realizando adaptações dinâmicas sobre o *timeout* dos processos. No modo tradicional de implementar os detectores, onde se utilizam *timeouts* com valor fixo, é comum verificar que a detecção sofre distorções devido às diferentes velocidades de processamento e de comunicação dos processos monitorados. Processos mais lentos tendem a ser incorretamente considerados suspeitos pois estes têm mais possibilidade de exceder o tempo limite. Simplesmente aumentar o valor do *timeout*, a fim de englobar estes processos mais lentos também pode trazer prejuízos à detecção, pois esta tem a sua freqüência de atualização diminuída, aumentando a latência da detecção. Dessa forma, a escolha de um valor fixo que seja eficaz para ambas situações torna-se uma operação delicada, e que está sujeita a todo tipo de influência do ambiente. Se, ao invés de usar um valor fixo, o detector puder corrigir o *timeout* de forma dinâmica, esta detecção possivelmente será mais precisa, e cada processo monitorado, independente de sua velocidade, será tratado de forma adequada.

A idéia dos detectores adaptativos foi proposta por Chandra [CHA 96a] em uma observação sobre a detecção de defeitos em ambientes com sincronismo parcial (figura 3.6), e busca atingir, de forma incremental, um valor de *timeout* no qual todos os processos corretos, mesmo os mais lentos, possam responder à detecção ainda em tempo hábil. É claro que, enquanto este *timeout* não é alcançado, os processos que excederem o tempo limite são considerados suspeitos, mas após o transcorrer de um certo número de ciclos de detecção, aumenta a probabilidade de que os processos que ainda não responderam estejam definitivamente falhos. Além do modelo adaptativo sugerido por Chandra e Toueg, onde o próprio detector decide quando aumentar os *timeouts*, existem trabalhos que propõem o uso de um módulo especializado para o monitoramento da rede e dos recursos do sistema, de forma que a detecção seja adaptada com base em diversos fatores. Entre esses trabalhos, pode-se destacar o modelo *Connectivity Time Indicator*, de Macêdo [MAC 00] e as observações realizadas por Chen *et al.* [CHE 00].

```
Every process p executes the following:

output_p ← 0
for all q ∈ Π                    { Δ_p(q) denotes the duration of p's time-out interval for q}
        Δ_p(q) ← default time-out interval

cobegin
|| Task 1: repeat periodically
        send "p-is-alive" to all

|| Task 2: repeat periodically
        for all q ∈ Π
            if q ∉ output_p and
              p did not received "q-is-alive" during the last Δ_p(q) ticks of p's clock
              output_p ← output_p ∪ {q}
                                        { p times-out on q: it now suspects q has crashed }

|| Task 3: when receive "q-is-alive" for some q
        if q ∉ output_p                 { p knows that it prematurely timed-out on q }
            output_p ← output_p - {q}       { 1. p repents on q, and }
            Δ_p(q) ← Δ_p(q) + 1             { 2.  p increases its time-out period for q}
coend
```

FIGURA 3.6 - Detector adaptativo para sincronismo parcial [CHA 96a]

3.6.4 Detector *Heartbeat*

O detector *Heartbeat* faz parte do conjunto que se propõe a diminuir os custos da comunicação entre os detectores através de uma hierarquização da comunicação (ao contrário do modelo "todos-para-todos" tradicionalmente empregado). Esses esforços podem ocorrer tanto no modelo de detecção quanto na disposição dos detectores e dos processos monitorados. O detector *Heartbeat* situa-se na primeira categoria, enquanto exemplos da segunda categoria, normalmente utilizados para diagnóstico dos sistemas, podem ser encontrados em Brawerman [BRA 00].

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 colapso e perdas de mensagens.

Um algoritmo é chamado quiescente quando considera que em um determinado momento irá parar de enviar mensagens (embora esse instante seja desconhecido). Por exemplo, uma comunicação confiável pode ser implementada de forma que um processo p envie repetidamente 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. 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)*. Assim, o processo q responde um *ack(m)* para cada mensagem que recebe de p. Como o canal de comunicação pode perder inúmeras mensagens tanto de $p \rightarrow q$ quanto de $q \rightarrow p$, no pior caso p jamais conseguirá parar de enviar mensagens, pois estas se perderam antes de

chegar a *q* ou os *ack(m)* correspondentes não chegaram. Neste caso, apesar da comunicação ser considerada confiável, ela não pode ser considerada quiescente, pois podem ocorrer casos nos quais ela nunca termina. 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 numerosas (até mesmo infinitas) vezes as mensagens que transporta, uma mensagem que também fosse reenviada muitas vezes vezes alcançaria seu destino.

Algoritmos que garantem esta terminação são considerados algoritmos quiescentes. De fato, todo algoritmo que garante a propriedade Terminação é quiescente, embora a reciprocidade não seja verdadeira[5]. Essa característica pode ser bem explorada se for considerado que um sistema computacional freqüentemente está sujeito a falhas de colapso e da rede, e como uma aplicação é executada por um tempo finito, ela exige uma solução em tempo finito.

Em relação à detecção de defeitos, quando se considera um cenário com falhas de processos por colapso e perda de mensagens, a solução abordada 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 com características especiais.

Aguilera também mostra que, para resolver um algoritmo quiescente, um detector de defeitos "tradicional" (que gera uma listagem de processos suspeitos baseada em *timeouts*) precisa ser do tipo $\Diamond\mathcal{P}$, que não pode ser implementado na prática [AGU 97b]. 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), e utilizando contadores cujo incremento é sempre positivo e unitário. 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 *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 uma matriz 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).

O algoritmo para a implementação do detector *Heartbeat* é apresentado na figura 3.7. Conforme este algoritmo, o detector é composto por duas tarefas concorrentes. Na

---

5 segundo Aguilera, sendo dois processos *p* e *q*, onde *p* envia diversas vezes uma mensagem e *q* envia um *ack*, se *p* cessa o envio de mensagens, caracteriza o algoritmo com a propriedade terminação; no entanto, não é um algoritmo quiescente, pois *q* continua esperando para responder as mensagens de *p* (ou seja, só uma algoritmo com terminação não basta, tem que permitir ganho de conhecimento para ser quiescente).

primeira tarefa, um processo periodicamente envia mensagens *"I am alive!"* para todos os seus vizinhos. Essa mensagem é composta de um campo identificador da operação (HEARTBEAT) e do caminho (*path*) inicial, que contém apenas o próprio processo. A segunda tarefa 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 caminho. Depois, o processo adiciona seu próprio identificador ao fim do caminho, 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 3.7 - Implementação do *Heartbeat* [AGU 97b]

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.

Outra diferença do detector de Aguilera com relação aos demais modelos de detectores de defeitos refere-se ao número de mensagens enviadas por cada processo. Como exposto 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 a retransmite para seus vizinhos, acrescentando antes sua própria identificação à mensagem, o que irá compor o caminho por onde aquela mensagem 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 caminho. O caminho também é utilizado para evitar que uma mensagem antiga seja reinserida 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 caminho presente nas mensagens recebidas.

Um dos problemas principais, ao utilizar um detector construído com base no *Heartbeat*, é a determinação do número ótimo de processos a ser considerado no conjunto de vizinhos. Em um trabalho anterior [EST 00c], foi feita a determinação do número de mensagens que cada processo envia, em um ciclo de transmissão, ao variar o número de processos no seu grupo de vizinhos. Os testes realizados utilizaram cinco detectores comunicando-se em computadores diferentes, de forma a representar o monitoramento de cinco máquinas. 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 seja amplamente favorecida pela quantidade de informações e pelo reduzido grau de latência (o detector atinge todos os processos com apenas um passo de comunicação), um número excessivo de mensagens trocadas representa um problema quando forem utilizadas redes com muitos processos. A razão desse aumento excessivo é que, quando todos os objetos monitorados pertencem ao conjunto de vizinhos, cada mensagem recebida será retransmitida para todos os outros detectores, até que não existam mais objetos vizinhos ausentes do caminho. O número de mensagens apresentou uma curva de crescimento muito acentuada (figura 3.8), resultante principalmente das mensagens que são recebidas e repassadas aos vizinhos, o que pode obrigar a definição de um limite prático ao número de objetos monitorados, para manter os sistemas em boas condições de operação.

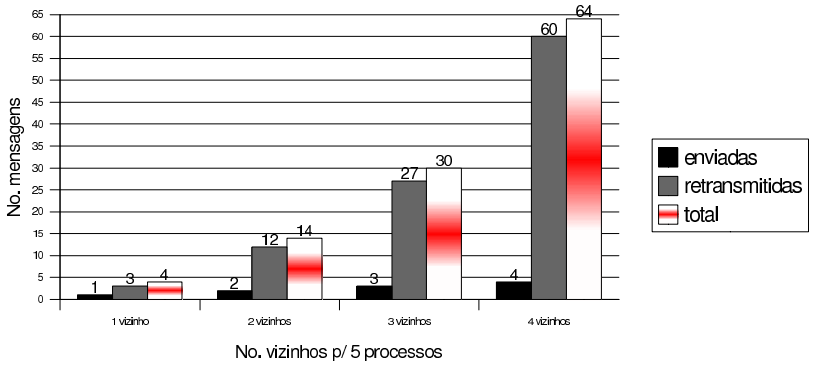

FIGURA 3.8 - Número de mensagens transmitidas por um processo, em uma rodada

No outro extremo, quando se considera apenas um dos outros processos como vizinho, a transmissão correta de informações somente ocorre quando todos são dispostos em um ciclo fechado[6]. 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 figura 3.9.

Outro problema interessante refere-se à possibilidade de falhas em todos os vizinhos de um processo, como demonstra a figura 3.10. Nestes casos, um detector é isolado dos demais através de uma partição lógica incorreta, e a visão do sistema será afetada pelas diferentes suspeitas observadas entre os detectores, mesmo quando os protocolos superiores puderem se comunicar diretamente com os demais processos.

6 Uma disposição lógica dos processos em estrela, por exemplo, obrigaria que o detector central tivesse um conjunto de vizinhos maior do que o dos demais processos, tornando heterogênea a distribuição das mensagens, sem no entanto evitar a ocorrência de um *single point of failure*.

Ao se avaliar as situações médias, onde exista um número de processos vizinhos razoável, verifica-se que o número de mensagens é suficientemente controlado, não impondo sobrecarga excessiva ao sistema. Entretanto, mesmo nesses casos deve-se tomar extrema precaução quanto à topologia com que são interligados os processos.

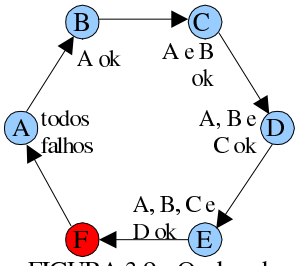

FIGURA 3.9 - Quebra do ciclo [EST 00c]

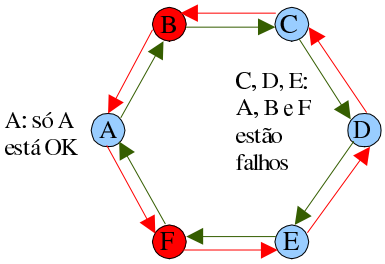

FIGURA 3.10 - Partição lógica incorreta [EST 00c]

Com base nos casos apresentados, tanto o número de processos vizinhos quanto a topologia dessas conexões devem ser definidos de acordo com um objetivo mínimo de tolerância a falhas para o suporte ao sistema (por exemplo, número máximo de falhas suportadas).

3.6.5 Detectores especializados (*ad-hoc*)

Os detectores especializados, ou *ad-hoc*, como chama Sergent *et al.* [SER 99], originaram-se da observação que a detecção de defeitos é necessária apenas em pontos específicos da operação dos protocolos de acordo. Ao construir o serviço de detecção de defeitos unido aos demais protocolos do sistema, é possível obter um desempenho melhor, uma vez que a detecção deixa de ser feita por um módulo independente, com mensagens e processamento próprio. Este tipo de implementação apresenta certas desvantagens, pois o detector de defeitos torna-se dependente da sua interação com o protocolo a que está ligado, o que impede tanto a reutilização de módulos quanto a avaliação de diferentes modelos de detectores. Além desta falta de flexibilidade, diversos trabalhos indicam que os detectores deverão assumir cada vez mais um papel de "serviço" dentro dos sistemas [VOG 96][FEL 99], de forma que, em maior ou menor grau, todas as aplicações distribuídas (e de monitoramento de redes) utilizem os serviços deste detector dedicado. Ao especializar os detectores em excesso, torna-se difícil o compartilhamento das informações, obrigando cada aplicação a realizar sua própria detecção.

No caso dos detectores *ad-hoc* estudados neste trabalho, suas otimizações se referem exclusivamente ao protocolo de consenso. Do ponto de vista do desempenho do sistema, este é um aspecto muito importante, pois um algoritmo de consenso necessita da detecção de defeitos apenas em alguns momentos de sua operação, o que leva ao questionamento da necessidade de um detector dedicado e da sobrecarga causada pelas suas mensagens de detecção.

Embora utilize também o algoritmo de Chandra para detectores $\langle\rangle\mathcal{S}$ [CHA 96a], o trabalho de Sergent *et al.* [SER 99] procurou evitar o comprometimento com uma implementação de consenso específica, de forma que foi utilizada uma abstração destes

detalhes: a noção de mensagem crítica e de resposta crítica pode ser estendida a todos os algoritmos de consenso, bem como a outros algoritmos com funções semelhantes.

Mensagens críticas, segundo esta abstração, são caracterizadas pelo fato de que após enviar uma mensagem m_p para um processo q, o processo p irá esperar pela resposta m_q de q antes de prosseguir seu processamento. Se q apresentar uma falha , p deve ser capaz de suspeitar de q e parar de esperar pela mensagem m_q, quando transcorreu um tempo máximo de espera.

A forma mais simples de implementar este tipo de detecção envolve apenas o uso de mecanismos de *timeout*, sem nenhuma mensagem de detecção de defeitos adicional. Se p não recebe m_q após uma espera Δ_{to}, então p suspeita de q (figura 3.11). Esta implementação é extremamente econômica em termos de mensagens, pois não gera nenhuma mensagem de detecção, uma vez que utiliza as próprias mensagens do algoritmo a que serve. No contexto do algoritmo de consenso considerado neste trabalho [CHA 96a], a situação de mensagem crítica/resposta crítica ocorre quando q é o coordenador de uma rodada r, m_p é a mensagem *estimate* enviada por um processo participante p ao coordenador q, e m_q é a mensagem *propose* que o coordenador manda para os participantes.

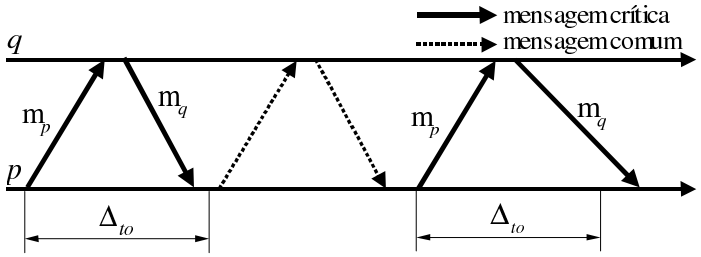

FIGURA 3.11 - Detecção de defeitos no modelo "*no message*" [SER 99]

A desvantagem desta estratégia é a grande possibilidade de efetuar um suspeita incorreta. Por exemplo, na figura 3.12, o processo q espera pela mensagem m_p do processo p e pela mensagem m_r do processo r antes de enviar sua resposta crítica m_q. Se o processo r for muito lento, o processo p pode atingir o *timeout* e suspeitar incorretamente do processo q.

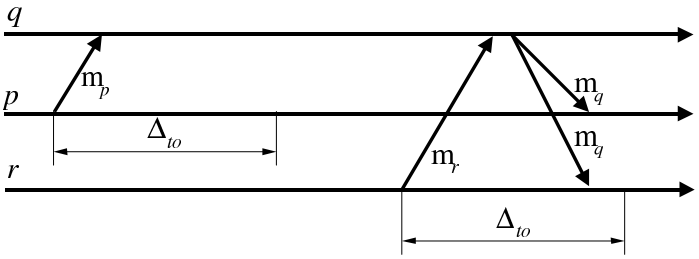

FIGURA 3.12 - Suspeitas incorretas com o detector "*no message*" [SER 99]

Já o detector *ad-hoc* "*heart-beat*" procura diminuir essa possibilidade de suspeitas incorretas. Para isso, toda vez que um processo q recebe uma mensagem crítica de um processo p, ocorre o envio o envio de mensagens "*I am alive!*" para p. A emissão destas

mensagens cessa somente após q ter enviado sua resposta crítica m_q. Deste modo, o processo p é periodicamente informado que q não falhou, e que apenas ainda não enviou sua resposta crítica (figura 3.13).

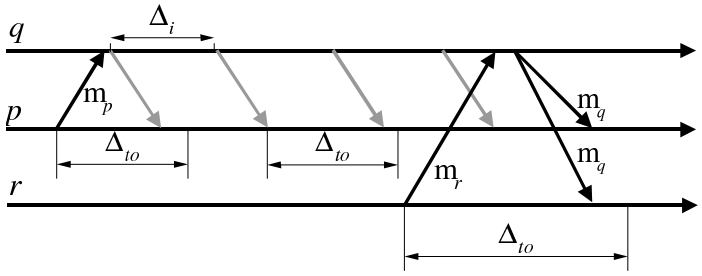

FIGURA 3.13 - detector *ad hoc "heart-beat"* [SER 99]

3.7 Utilização de comunicação não confiável

Os detectores de defeitos foram definidos através de duas propriedades: *completeness* (abrangência, completeza) e *accuracy* (precisão). A propriedade *completeness* refere-se à capacidade do detector identificar todos os processos que estão falhos, enquanto *accuracy* determina a precisão desta suspeita, a fim de evitar a inclusão de processos corretos nas listas de suspeitos. Ao respeitar essas duas propriedades, um detector de defeitos garante que os algoritmos que o utilizem não perderão a consistência das decisões, nem ficarão indefinidamente bloqueados (preservando as propriedades *safety* e *liveness*, respectivamente).

Segundo a própria definição destas propriedades, um detector que continuamente inclua e remova os processos da sua lista de suspeitos não é capaz de garantir as condições mínimas ao consenso, pois tal comportamento constitui uma violação da propriedade de *accuracy* dos detectores (por exemplo, a propriedade *accuracy* do detector *Eventually Weak* determina que "há um instante de tempo após o qual um processo correto não é considerado suspeito por nenhum processo correto").

A indeterminação que acompanha um ambiente assíncrono impediria a definição de um *timeout* adequado ao sistema, uma vez que sempre poderia haver um processo mais lento. Chandra e Toueg [CHA 96a] afirmam, entretanto, que em situações práticas tais detectores poderão considerar *timeouts* "suficientemente longos", de modo que todos os processos corretos consigam responder a tempo. Outra opção, quando não é possível determinar o máximo *timeout* suportável, é o uso de *timeouts* adaptativos. Ainda assim, a observação dos modelos de detectores mais utilizados (*Push, Pull*) indica que comumente são utilizados *timeouts* fixos, ao invés de adaptativos.

A opção majoritária por soluções que utilizam *timeout* fixo leva a crer que mesmo nesses casos as suspeitas incorretas são "aceitáveis" nos sistemas. De fato, tais situações devem-se ao próprio comportamento dos detectores e ao ambiente de utilização mais comum deles. Quando um detector comete um engano, suas características fazem com que um processo correto seja suspeito de falhas, e não o contrário, onde um processo falho é considerado correto. Como o exemplo mais usado do emprego dos detectores envolve os algoritmos de consenso, onde os detectores são usados para acusar quando o coordenador está falho, uma detecção incorreta leva ao cancelamento da rodada, preservando as propriedades de *safety* e *liveness* da aplicação.

No entanto, essas propriedades oferecem restrições ao modelo de comunicação que se prentende utilizar. É bem verdade que o algoritmo de consenso exige uma comunicação confiável [CHA 96a][GAR 98], pois há a necessidade de garantir a consistência entre os processos, mas no caso dos detectores, essa não é uma opção obrigatória.

Em um trabalho anterior [EST 00b], observou-se que os detectores de defeitos não necessitam garantir a consistência com outros detectores durante sua operação. De fato, Chandra e Toueg já definiam que os detectores trabalham apenas sobre suas percepções locais, podendo ter listas de suspeitos completamente diferentes em um mesmo instante de tempo. Assim, a formação desta "visão" local de cada detector deve ocorrer de modo instantâneo, ou seja, a cada momento a lista de suspeitos pode mudar, expressando a percepção das novas mensagens recebidas.

Embora a possibilidade de corrigir os enganos tantas vezes quanto necessáriocontradiga a definição das propriedades *completeness* e *accuracy*, na prática pode-se admitir que os detectores mudem suas listas de suspeitos tantas vezes quantas forem necessárias[7].

Quando a implementação dos detectores baseia-se em *timeouts* para o processo de detecção dos defeitos, esses períodos de tempo devem ser estabelecidos de forma que a detecção ocorra com o menor atraso possível, porém sem induzir um número excessivo de falsas suspeitas (como, por exemplo, na presença de processos mais lentos). Entretanto, somente *timeouts* sobre as mensagens não bastam para distinguir os casos de processos lentos ou falhos dos casos de mensagens extraviadas, de forma que a capacidade de corrigir seus enganos torna-se ainda mais essencial para o funcionamento dos detectores.

Dessa forma, os detectores de defeitos reagem diferentemente para o uso de comunicação confiável ou não confiável. Enquanto o algoritmo de consenso necessita utilizar comunicação confiável (normalmente implementada através da retransmissão de mensagens), o efeito desta retransmissão de mensagens para os detectores terá o mesmo impacto do recebimento de novas mensagens. Na verdade, o uso de retransmissão pode vir a sobrecarregar a rede sem que o recebimento de todas mensagens influenciem positivamente na detecção. Além disso, do ponto de vista temporal, somente as mensagens novas poderão efetivamente auxiliar na precisão da detecção, pois estas trazem informações atualizadas sobre o estado dos processos.

Se além do uso de retransmissão de mensagens for realizado um ordenamento, comum em protocolos de entrega confiáveis (o protocolo TCP, por exemplo), pode-se impossibilitar a operação dos detectores quando coexistem processos de velocidades relativas muito diferentes. Um exemplo claro desta inadequação pode ser encontrado na implementação do detector *Pull*. Neste modelo, um processo p inicialmente questiona um processo q sobre seu estado, enviando mensagens do tipo "*Are you alive?*". Quando q recebe tais mensagens e não está falho, envia uma mensagem com a resposta "*Yes, I am!*". Supondo um cenário onde um dos processos (no caso q) é muito lento, verifica-se que a lista de processos suspeitos é corrigida apenas quando chegam as primeiras respostas, no momento t_n (figura 3.14).

---

7 Segundo Chandra e Toueg, na prática pode-se considerar que um detector mantém suas suspeitas por um tempo suficientemente longo para a terminação das operações de acordo que estão sendo realizadas.

Se for considerado que o processo *q*, a partir deste momento, responderá todas as mensagens subseqüentes com um intervalo aceitável, ele será retirado da lista de suspeitos, pois estará respondendo dentro do tempo esperado. Se fosse utilizado um controle sobre o número de seqüência, à resposta da primeira mensagem seria atribuído um atraso muito superior ao limite de tempo considerado, o que levaria à suspeita do processo *q* mesmo que todas as suas mensagens chegassem em ordem a partir desse momento.

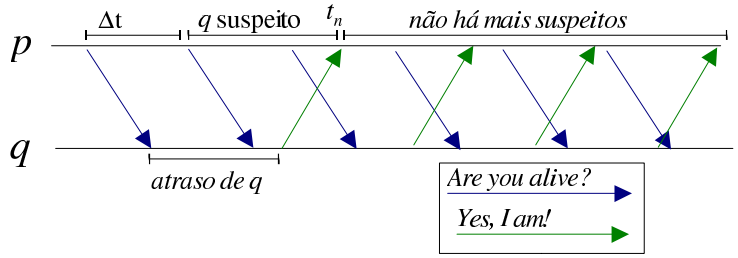

FIGURA 3.14 - Detecção com mensagens não identificadas [EST 00b]

Ainda assim, sempre haverá o risco de que as mensagens respondidas não consigam chegar a tempo, pois obstáculos na transmissão inseriram um atraso superior ao tolerado. A solução mais prática para esse problema envolve a modificação da relação entre o intervalo de envio das mensagens e o *timeout* máximo tolerado para a resposta. Dessa forma, é possível aumentar a probabilidade de que ao menos uma mensagem sempre esteja nos *buffers* de entrada do detector, diminuindo o risco de detecções incorretas. A contrapartida desta técnica é que além do aumento do tráfego da rede, aumenta a latência da detecção, pois todas as mensagens do *buffer* são processadas, não importa há quanto tempo foram enviadas. É por isso que escolher uma relação adequada entre o envio e o *timeout* deve ser sempre uma preocupação na hora de operar um detector. Nos capítulos seguintes serão analisadas mais profundamente a importância e o impacto das relações de envio/*timeout* sobre o funcionamento dos detectores.

3.8 Análise de Custos

3.8.1 Número de mensagens trocadas

O número de mensagens que os detectores trocam influencia diretamente o tráfego da rede, aumentando a probabilidade de mensagens chegarem atrasadas ao seu destino, bem como aumentando a sobrecarga dos sistemas com o processamento destas mensagens. Assim, como na análise feita sobre os algoritmos de consenso, será considerado aqui o número de mensagens ponto a ponto trocadas entre os detectores, a fim de garantir a difusão das informações por todos os processos. Essa análise foi feita com base na utilização de comunicação ponto-a-ponto. O uso de difusão de mensagens pode reduzir os custos para os algoritmos, mas depende da disponibilidade da rede e das ferramentas utilizadas.

- **Detectores *Push***: em cada ciclo de detecção, cada processo envia mensagens "*I am alive*" para todos os demais, o que corresponde a n-1 mensagens por processo. Entretanto, cada processo deve enviar essas mensagens, de forma

que ao final de um ciclo de detecção deverão ter sido transmitidas n(n-1) mensagens, ao total. Isto representa um custo de $O(n^2)$ mensagens.

- **Detectores *Pull***: cada ciclo de detecção envolve a troca de duas mensagens, uma "*Are you alive?*" e outra "*Yes, I am!*". Assim, cada processo gera 2(n-1) mensagens, mas o número total de mensagens geradas é de 2n(n-1), que é levemente superior ao dos detectores *Push*. O custo do algoritmo continua sendo de $O(n^2)$ mensagens.
- **Detector *ad-hoc "no message"* [SER 99]**: o detector *ad-hoc "no message"* não gera mensagens de detecção, não representando nenhuma carga para o tráfego na rede.
- **Detector *ad-hoc "heart-beat"* [SER 99]**: O detector *ad-hoc "heart-beat"* gera mensagens do tipo "*I am alive*", assim como o detector *Push*. Entretanto, apenas o coordenador da rodada envia essas mensagens para os demais processos, de forma que a quantidade total de mensagens trocadas em um ciclo de detecção é equivalente a n-1 mensagens. Dessa forma, o custo deste algoritmo é igual a O(n).
- **Detector *Heartbeat* [AGU 97a]**: embora o conjunto de vizinhos do detector seja um fator que auxilia na diminuição do número de mensagens enviadas, o mecanismo de disseminação das informações acaba gerando um número excessivo de mensagens, em uma taxa que cresce exponencialmente, de acordo com o número de processos e o número de vizinhos, conforme pode-se observar na figura 3.8.

3.8.2 Graus de latência

Os graus de latência, a princípio, indicam o número mínimo de passos de comunicação que um determinado algoritmo necessita para terminar sua operação ou tarefa. Um exemplo claro deste uso são os algoritmos de consenso, onde podem ser contabilizados os passos de comunicação entre os processos necessários para que o acordo seja alcançado.

Entretanto, esse não é o caso dos detectores de defeitos, que em sua grande maioria têm como característica o funcionamento ininterrupto. Assim, ao invés de contabilizar os passos de comunicação necessários para a terminação de uma determinada operação, uma medida possível envolve o número de etapas necessárias para que um ciclo de detecção seja realizado. Outra métrica necessária é a verificação do número mínimo de passos de comunicação para a disseminação das informações para todos os processos (de acordo com a implementação, pode haver transmissões adicionais que não contribuem para a disseminação, apenas para o aumento do número de mensagens). Embora os detectores utilizem poucas variações do modelo de disseminação, a verificação dos graus de latência que cada modelo possui é necessária para identificar quais são os limites operacionais de tais detectores.

- Detectores tipo *Push*

Os detectores *Push* (incluindo o detector adaptativo) contabilizam o intervalo de recebimento entre duas mensagens do processo monitorado. Assim, como o *timeout*

começa a ser contado no instante de recepção da primeira mensagem, basta apenas uma nova mensagem para fechar o ciclo de detecção. Além disso, normalmente esses detectores utilizam como modelo de comunicação a difusão para todos os processos. Assim, pode-se considerar que um detector *Push* tem um grau de latência equivalente a 1 para a identificação dos defeitos, e que a disseminação das mensagens também necessita apenas um passo de comunicação.

- Detectores tipo *Pull*

Os detectores tipo *Pull* requerem duas etapas de comunicação para realizar o julgamento dos processos monitorados. Na primeira etapa, o detector envia uma mensagem questionando o estado dos processos, e na segunda etapa, recebe mensagens daqueles processos confirmando seu funcionamento. Com isso, o ciclo de detecção envolve um grau de latência igual a 2. Isso é extremamente importante, pois indica que o limite mínimo de tempo necessário para uma detecção correta deve ser maior ou igual ao tempo que a primeira mensagem leva para ser transmitida, processada e respondida.

Quanto à difusão das mensagens, estes detectores apresentam uma condição diferente dos detectores *Push*. Para aqueles detectores, quando um processo enviava suas mensagens para todos os demais, resultava que em um único passo de comunicação todos processos ficavam sabendo das condições de um único processo. No caso dos detectores *Pull*, um detector pede informações para todos os processos, ou seja, ao fim dos dois passos de comunicação um detector terá conhecimento sobre todos processos. Embora notadamente diferentes, pode-se considerar que para fins práticos, em ambas situações todos processos realizam a detecção, ou seja, o conhecimento sobre todos os processos monitorados pode ser adquirido em um único ciclo de detecção. Dessa forma, pode-se unificar os conceitos e concluir que os detectores *Pull* têm um grau de latência equivalente a 2 também para a disseminação das mensagens.

- Detectores *ad-hoc "no message"*

Apesar de não utilizarem mensagens próprias de detecção, os detectores *ad-hoc "no message"* fazem uso das chamadas mensagens críticas, que no caso específico do algoritmo de consenso, são as mensagens *estimate* e *propose*. O mecanismo de suspeita, neste caso, é idêntico ao utilizado pelos detectores *Pull*, e assim como o mecanismo de disseminação, apresenta um grau de latência igual a 2.

Quanto à disseminação das mensagens, há uma diferença substancial quanto aos detectores *Pull*, mas que não reduz o número de passos de comunicação necessários. Os detectores *ad-hoc* centralizam sua detecção no coordenador da rodada corrente, ou seja, só necessitam obter informações deste único processo, ao invés dos detectores modulares (que por não serem interligados ao consenso, não têm como saber qual é o processo coordenador). Assim, mesmo que o número de passos de comunicação seja equivalente ao dos detectores *Pull*, a quantidade de mensagens geradas é bem menor.

- Detectores *ad-hoc "heart-beat"*

Em um primeiro momento, o detector *ad-hoc "heart-beat"* parece comportar-se assim como os detectores *Push*, ou seja, basta um único passo de comunicação para que o processo de suspeita seja finalizado. Apesar deste comportamento possivelmente predominar durante a operação deste detector, há um detalhe que impede a atribuição de um grau de latência unitário. Este detalhe refere-se à primeira mensagem *"I am alive"* recebida, pois o envio desta é dependente da mensagem *estimate*, ou seja, no início da operação, este detector comporta-se como um detector *Pull*. Essa única operação obriga a atribuição de um grau de latência equivalente a 2 para este detector. Se por acaso fosse considerada apenas uma etapa de comunicação, a definição dos *timeouts* possivelmente utilizaria o tempo mínimo para o transporte de uma mensagem, ao invés do tempo de *roundtrip*. Com isso, mesmo que a rede conseguisse realizar a transmissão dentro do intervalo definido, este detector iria atingir o limite do *timeout* ainda antes de receber a primeira mensagem *heart-beat*.

Isso obriga também a atribuição de um grau de latência 2 para a difusão das mensagens, independente da centralização da detecção (como no detector *ad-hoc "no message"*).

- Detector *Heartbeat*

Este detector possivelmente é o que apresenta as maiores variações, em relação aos demais modelos de detectores. Devido ao mecanismo de disseminação *gossip*, a entrega das mensagens não é feita para todos os processos ao mesmo tempo, e embora seu mecanismo de detecção funcione de forma semelhante ao dos detectores *Push*, seu grau de latência depende de muitos outros fatores.

O mecanismo de disseminação *gossip* objetiva reduzir o número de mensagens enviadas em um único instante por um processo, de forma a reduzir o tráfego na rede. Para isso, ao invés de enviar mensagens para todos processos, o detector envia apenas para um sub-grupo, também conhecido como "vizinhos". Na proposta de um detector *gossip* apresentada por Guo [GUO 98] (figura 3.15), cada detector mantém uma tabela onde contabiliza o número de ciclos desde a última mensagem recebida de cada processo, e periodicamente transmite essa tabela para os seus vizinhos.

Cada processo que recebe estas tabelas atualiza seus contadores se as mensagens contiverem um valor menor do que o seu (figura 3.16). Com isso, a probabilidade de que todos recebam informações sobre um processo cresce à medida em que os processos disseminam suas tabelas. De fato, o número mínimo de retransmissões necessárias para atingir todos os processos pode ser considerado correspondente ao grau de latência da disseminação, e depende diretamente da quantidade de processos monitorados e do número de vizinhos com que cada processo se comunica por vez.

O número mínimo de passos de comunicação necessários para disseminar as informações pode ser obtido através da relação $passos = (processos - vizinhos) + 1$, que considera a melhor probabilidade de difusão (para o detector *Heartbeat*, o conjunto de vizinhos é fixo; mas para o detector de Guo, é escolhido randomicamente a cada envio, então a difusão pode demorar um pouco mais).

```
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 3.15 - Algoritmo do detector *gossip* [GUO 98]

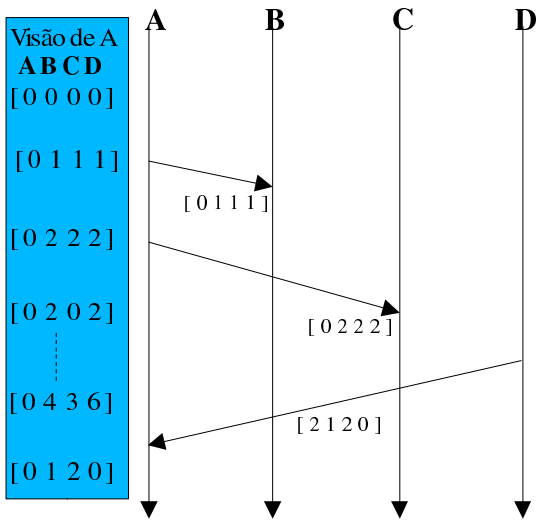

FIGURA 3.16 - Atualização dos contadores em um detector *gossip* (baseado em [GUO 98])

Enquanto o número mínimo de passos de comunicação depende apenas da quantidade de vizinhos, o grau de latência para o procedimento de suspeita no detector *Heartbeat* mostrou-se muito dependente da forma como é implementada a coleta das informações. Se no modelo de Guo os processos podem identificar mensagens repetidas ou velhas através dos contadores relacionados com cada processo, no detector *Heartbeat* não há esse conhecimento, e a suspeita ocorre apenas após terem sido retiradas de circulação todas mensagens que carregam o identificador do processo falho.

Isso significa que, enquanto no modelo de Guo a detecção de um defeito pode ser realizada com um grau de latência idêntico ao da disseminação das informações, no detector *Heartbeat* esse grau de latência será sempre igual ao número total de processos.

3.9 Considerações Finais

Este capítulo apresentou as bases do funcionamento dos detectores de defeitos, dando ênfase às características que devem ser respeitadas, a fim de manter tanto a consistência dos processos como a evolução das suas atividades (*safety* e *liveness*, respectivamente). Já na análise dos modelos encontrados na literatura, foi feita uma avaliação das suas versatilidades e possíveis deficiências. Além disso, foi feita uma análise sobre o custo de cada modelo de detector, através do número de mensagens geradas e do número de passos de comunicação necessários para a realização das detecções.

No próximo capítulo, serão apresentadas a motivação e objetivos deste trabalho, além da descrição detalhada do ambiente e dos testes propostos para a comparação dos modelos de detectores de defeitos.

4 Comparação entre os Detectores

4.1 Análise do Artigo de Sergent *et al.*

O trabalho de Sergent, Défago e Schiper [SER 99] analisa principalmente o impacto que os diversos modelos de comunicação empregados pelos detectores de defeitos têm sobre o tempo de terminação do consenso. Esta análise é importante, pois cada modelo tem vantagens e desvantagens, e encontrar o detector que melhor se adapte a uma determinada situação ainda é uma atividade baseada na experiência pessoal e em suposições empíricas.

A escolha do tempo de terminação do consenso, como métrica de comparação, mostrou-se uma escolha bem adequada, uma vez que o consenso é o componente que mais necessita de uma detecção ágil e precisa. Se esta relação entre a latência da detecção e a precisão não for bem definida, pode ocorrer que um sistema que privilegie a agilidade incorra em enganos excessivos, enquanto a super-valorização da precisão pode acarretar atrasos longos demais para o consenso. Em ambos caso, o tempo de terminação do consenso sai prejudicado, pois um detector muito ágil tende a sobrecarregar a comunicação e o processamento das mensagens, enquanto um detector muito lento tende a estender demasiadamente o tempo de espera do consenso pela informação do estado dos processos.

4.1.1 Ambiente de simulação

O ambiente de testes simulado por Sergent *et al.* é bem semelhante ao encontrado em redes locais, contendo por exemplo, uma rede Ethernet 10 Mbits com *buffers* FIFO, máquinas homogêneas, etc. Para a representação destas características, os autores utilizaram parâmetros obtidos em medições práticas, de forma que a simulação não se distancia da realidade, nestes aspectos. Entretanto, ainda existem algumas diferenças entre a simulação e uma operação real, e que residem basicamente nas simplificações impostas pelos autores do artigo. As simplificações mais simples referem-se ao próprio ambiente de testes, onde, por exemplo, as máquinas envolvidas rodam apenas a aplicação que está sendo testada, e não há tráfego externo.

Outra simplificação, que pode trazer diferenças mais significativas, é a de que o tempo de processamento, por ser desprezível se comparado ao tempo de comunicação, foi considerado igual a zero. Deve-se ressaltar que, apesar desta diferença relativa entre o tempo de processamento e o tempo de comunicação ser grande, este não é um parâmetro que possa ser desconsiderado nas situações práticas, pois a disputa pelos recursos das máquinas é um agravante que tende a aumentar o impacto deste tempo de processamento, especialmente quando o próprio detector está operando em situações que geram sobrecarga (número excessivo de mensagens, por exemplo). Além disso, por menor que seja, há uma diferença de carga computacional entre o coordenador e os processos participantes. Mesmo que essa diferença seja pequena, ou que se dilua na média geral (afinal, é feita a média entre o tempo do coordenador e o de outros quatro processos, neste caso), há a tendência clara de aumentar o tempo de terminação devido

a essa sobrecarga. Se dois detectores apresentarem médias muito próximas, este é um item de comparação que não pode ser descartado.

4.1.2 Situações de falhas

Os ambientes de falhas considerados na simulação foram chamados de "*failure free*" e "*worst case*". No ambiente *failure free* todos os processos considerados na simulação (cinco) estão ativos e trocando mensagens. Assim, os fatores que mais influenciam no desempenho do consenso será o processamento das mensagens trocadas entre os detectores. Aumentando o intervalo entre as mensagens de detecção é possível reduzir este esforço de processamento, mas deve-se observar que um maior intervalo entre as mensagens de detecção aumenta a latência da detecção dos defeitos.

Já no ambiente *worst case* é explorada a situação do pior caso (em termos do tempo de terminação) que um algoritmo de consenso pode estar submetido em uma rodada: logo após receber o número de mensagens *estimate* necessárias para prosseguir o algoritmo e mandar sua proposta (mensagem *propose*), o coordenador falha. Nesse caso, os demais processos dependem da agilidade do detector para identificar esta falha o mais depressa possível, e passar para a próxima rodada onde o consenso será resolvido. É claro que, em uma situação normal, o consenso pode se prolongar por diversas rodada (devido a falhas ou detecções incorretas) antes que o consenso seja finalizado, mas como essa situação não pode ser determinada com precisão (não há como garantir um número máximo de consensos antes da finalização), a análise da ocorrência de um único "pior caso" permite identificar a tendência do comportamento dos detectores.

4.1.3 Modelos de detectores

Os detectores estudados por Sergent *et al.* correspondem, basicamente, a dois tipos: "módulos" detectores de defeitos e detectores integrados ao algoritmo de consenso. No primeiro grupo encontram-se aqueles detectores comumente referenciados pela literatura, os detectores que enviam mensagens "*I am alive!*" e "*Are you alive?*". Seguindo a nomenclatura de Felber [FEL 98], eles vêm sendo classificados, respectivamente, como detectores *Push* e *Pull*. Estes detectores utilizam métodos de comunicação diferentes, mas têm em comum o fato de mandarem mensagens próprias para a detecção e também o seu estado de funcionamento ininterrupto. Os detectores do segundo grupo, no entanto, são detectores específicos para a aplicação (no caso, o consenso), e incluem diversas otimizações com o intuito de reduzir a carga imposta pelos detectores tradicionais. No caso do consenso, a principal otimização advém da observação de que a detecção só é necessária ao algoritmo em momentos bem delimitados, e especificamente no caso do algoritmo de Chandra para detectores $\diamond\mathcal{S}$, entre o envio do *estimate* ao coordenador e o recebimento do seu *propose*.

Os detectores *ad-hoc* (especializados) estudados por Sergent *et al.* são o *ad-hoc "no message"* e o *ad-hoc "heart-beat"*. Ambas implementações realizam suas detecções apenas durante o intervalo específico do consenso, entre o *estimate* e o *propose*. Elas diferem, no entanto, na quantidade de mensagens utilizadas. Enquanto o

ad-hoc "no message" utiliza exclusivamente as mensagens *estimate* e *propose* para julgar o estado do outro processo (por *timeout*), o detector *ad-hoc* "*heart-beat"* fica enviando, entre estas duas mensagens críticas, mensagens de "*I am alive!*", a fim de evitar suspeitas incorretas.

4.1.4 Detectores testados

Do conjunto de quatro detectores estudados por Sergent *et al.*, apenas três foram simulados e comparados. A alegação para a exclusão do detector *Push* é a de que o detector *ad-hoc "heart-beat"* é uma versão mais otimizada daquele.

Embora o autor do presente trabalho discorde um pouco desta escolha, uma vez que pode haver um instante onde o funcionamento constante do detector *Push* ofereça uma detecção mais rápida e mais precisa do que o *ad-hoc "heart-beat"* (afinal, o detector *Push* consegue ter uma visão mais global do comportamento dos processos), a decisão baseada na sobrecarga de comunicação causada pelos dois detectores é uma justificativa aceitável. Ainda assim, teria sido interessante contar com a avaliação de todos os detectores.

Mesmo que os detectores tenham sido testados em situações idênticas e com parâmetros semelhantes, Sergent *et al.* avaliaram cada detector com uma maneira diferente:

- O detector *Pull* foi testado usando alguns valores fixos para o intervalo de envio (Δ_i), variando apenas o *timeout* para cada valor de Δ_i (figura 4.1).

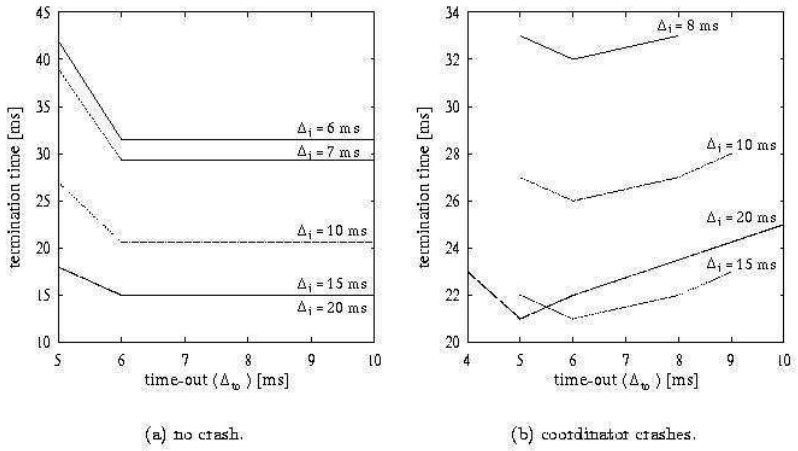

FIGURA 4.1 - Desempenho do detector *Pull* [SER 99]

- O detector *ad-hoc "heart-beat"*, como apresenta a característica comum aos detectores *Push* de que o envio e o *timeout* das mensagens são realizados ininterruptamente, utilizou um intervalo de envio ligeiramente inferior ao valor do *timeout*, numa proporção de $\Delta_i = 98\% \Delta_{to}$ (figura 4.2). Este valor menor aumenta a probabilidade de que o *timeout* não seja ultrapassado antes que uma nova mensagem seja enviada por um processo correto. Se isto

ocorresse (o envio posterior da mensagem), o detector iria alternadamente inserir e remover o processo monitorado da sua lista de suspeitos, à medida que o *timeout* e o recebimento de uma nova mensagem se alternassem.

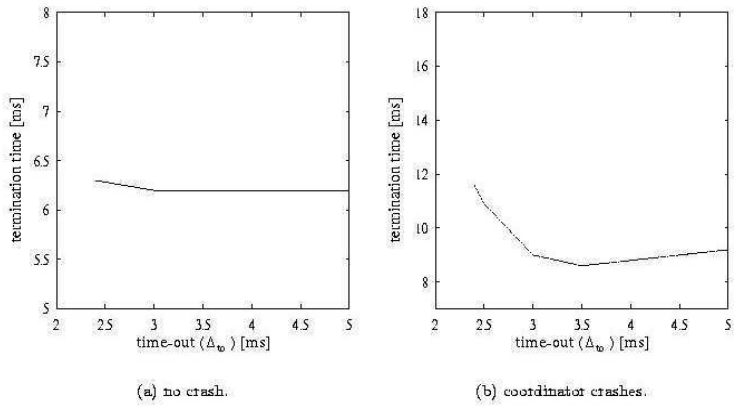

FIGURA 4.2 - Desempenho do detector *ad-hoc "heart-beat"* [SER 99]

- O detector *ad-hoc "no message"*, por não apresentar envio de mensagens de detecção, foi testado variando apenas o *timeout* (figura 4.3).

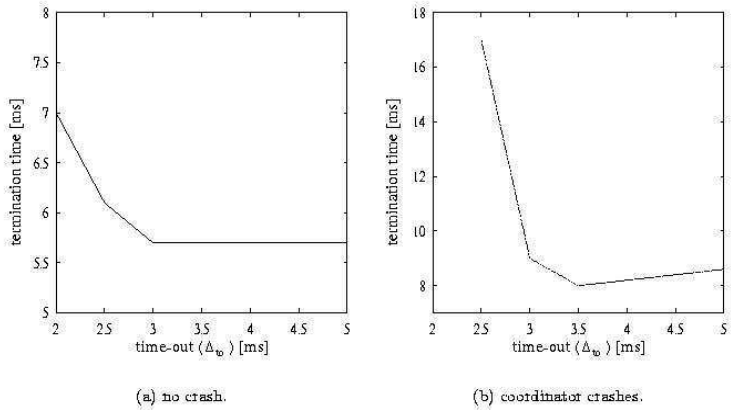

FIGURA 4.3 - Desempenho do detector *ad-hoc "no message"* [SER 99]

Sendo o tempo de terminação do consenso a única métrica considerada por Sergent *et al.* (o trabalho atual usa outras métricas extras, para a validação dos resultados), os detectores são comparados apenas pelos seus pontos de melhor desempenho, em ambos ambientes de falhas dos processos (*failure free* e *worst case*), conforme pode ser visto na tabela 4.1. Esta comparação, pela forma como foi executada, deixa muitas dúvidas sobre a real vantagem ou desvantagem dos modelos de detectores, pois não parece ter sido feita nenhuma análise mais profunda sobre o comportamento dos detectores sob os diversos parâmetros de entrada fornecidos.

TABELA 4.1 - Tempos de terminação do consenso (simulação com 5 processos) [SER 99]

failure detection	***parameters***	***failure free execution***	***coordinator crashes (worst case)***
interrogation (*Pull*)	Δ_i = 15 ms, Δ_{to} = 6 ms	15 ms	21.7 ms
ad-hoc "no message"	Δ_{to} = 3.5 ms	5.7 ms	8 ms
ad-hoc "heart-beat"	Δ_i = 3.4 ms, Δ_{to} = 3.5 ms	6.2 ms	8.6 ms

Pode-se observar também que Sergent *et al.* somente puderam indicar um detector "mais eficiente" graças à concidência de valores mais baixos para o detector *ad-hoc "no message"*. Se este não fosse o caso, a simples análise dos melhores pontos não seria capaz de trazer nenhuma conclusão sobre o assunto.

4.1.5 Contribuições do artigo

Talvez a conclusão mais importante do artigo de Sergent *et al.* não seja a comparação dos detectores, nem a afirmação de que os modelos *ad-hoc* são mais eficientes que os detectores modulares. Variando as implementações, podem ocorrer semelhanças qualitativas, e este é um dos principais pontos que o presente trabalho pretende avaliar. No entanto, o trabalho de Sergent *et al.* serve para reforçar dois princípios que devem nortear o desenvolvimento e a utilização dos detectores. Primeiro, que os objetivos lógicos (prova de completeza, *safety*, *liveness*, etc.) e os objetivos da implementação (mais especificamente, a determinação dos parâmetros que levam à maior eficiência) são ortogonais e podem simplificar a construção e a validação do *software*. O segundo princípio diz respeito à própria determinação dos intevalos onde os detectores apresentam uma maior eficiência, pois o fato de um detector ser modular, e não especificamente elaborado para um algoritmo, não significa que este é isento de considerar objetivos temporais quando o desempenho é um fator requerido: em outras palavras, "modularidade e eficiência não são objetivos antagônicos" [SER 99].

4.2 Prerrogativas de Operação deste Trabalho

Enquanto o trabalho desenvolvido por Sergent *et. al.* esclarece várias questões sobre a influência do modelo de comunicação e da possibilidade de construir algoritmos assíncronos que ofereçam um bom desempenho, certos resultados carecem de uma comprovação prática.

Este trabalho procurou utilizar os experimentos conduzidos por Sergent *et al.* como base para uma nova comparação entre os detectores. Além de incluir mais modelos de detectores na comparação, procurou-se avaliar não só a influência dos detectores no tempo de finalização do consenso, como também identificar quais as limitações que o ambiente (sistema operacional, rede de comunicação) impõe e qual a tendência de comportamento dos detectores sob diversas combinações de parâmetros.

4.2.1 Modelos de falhas

Os detectores de defeitos definidos por Chandra foram projetados para operar em ambientes sujeitos a falhas por colapso, ou seja, a falha de um processo faz com que suas atividades cessem completamente, e o estado de suas informações torna-se incerto para os demais processos. Em [EST 00a] foram apresentados alguns modelos de detectores de defeitos presentes na literatura, e alguns destes eram extensões do modelo de Chandra ou novos modelos capazes de suportar também falhas mais abrangentes, como por exemplo as falhas bizantinas. Sergent *et al.* consideraram para seus experimentos apenas detectores para colapso: esta escolha é importante, uma vez que estes detectores apresentam a maior variedade de implementações, são conceitualmente mais simples e provavelmente são os mais utilizados. No presente trabalho também serão considerados apenas detectores para colapso, e as situações de falhas serão representadas pela paralisação das atividades dos processos.

4.2.2 Modelos de detectores avaliados

No trabalho de Sergent *et al.*, apenas três modelos de detectores foram comparados, e essa quantidade demonstrou-se muito inferior à identificada na literatura estudada. Desse modo, o conjunto de detectores aqui avaliados foi enriquecido com implementações que representam diferentes possibilidades de realizar a detecção dos defeitos. Procurou-se identificar representantes de modelos de comunicação diversos, desde os modelos mais tradicionais até aqueles que objetivam otimizações no número de mensagens. Além disso, estes detectores permitem confrontar diferentes abordagens de integração ao algoritmo de consenso, e a análise das vantagens e desvantagens destas propostas tem relação direta com a atual tendência de oferecer a detecção de defeitos como um serviço do próprio sistema operacional ([FEL 99][VOG 96]).

Assim, neste trabalho foram considerados sete modelos diferentes de detectores, que são apresentados na tabela 4.2.

TABELA 4.2 – Características de construção dos detectores de defeitos

Detector	**Comportamento**	**Construção**
Push	por *timeout* fixo	modular
Pull	por *timeout* fixo	modular
AdaptivePush	*timeout* adaptativo	modular
AdaptivePull	*timeout* adaptativo	modular
Heartbeat	sem *timeout*	modular
ad-hoc "no message"	por *timeout* fixo	integrado ao consenso
ad-hoc "heart-beat"	por *timeout* fixo	integrado ao consenso

Estes detectores representam algumas das implementações de detectores de defeitos encontradas na bibliografia e que funcionam em ambientes com falhas por colapso. Basicamente, a partir do comportamento apresentado, podem ser classificados em três grupos: no primeiro estão os detectores que utilizam *timeout* fixo para o

julgamento das mensagens (incluindo os detectores *ad-hoc*); o segundo engloba os detectores adaptativos, que são variações dos modelos de *timeout* fixo e que procuram evitar detecções incorretas utilizando *timeouts* que se adaptam dinamicamente à capacidade de cada processo monitorado. No terceiro grupo, está o detector *Heartbeat* que foi proposto por Aguilera; ele é construído de forma que o controle de *timeout* não é feito diretamente sobre as mensagens, além de utilizar um modelo de disseminação de mensagens diferente dos outros detectores.

Quanto ao estilo de construção, os detectores representam as duas tendências encontradas: ou os detectores são construídos como "módulos", onde o detector é um componente independente que fornece serviços às aplicações, ou é construído integrado ao algoritmo de consenso, de forma que seu desempenho é otimizado mas perde a independência e flexibilidade.

4.2.3 Ambiente de testes

Outro ponto essencial para a realização das experiências é a descrição do ambiente onde serão executadas as medições. Assim como no experimento de Sergent *et al.*, as observações deste trabalho serão realizadas sobre a interação entre cinco processos participantes de um consenso, sendo que cada um estará alocado em uma máquina distinta da rede.

Os experimentos foram realizados nas dependências Laboratório de Tolerância a Falhas do Instituto de Informática, cujas máquinas estão interligadas por uma rede Ethernet de 10MBits/s. As máquinas empregadas na experiência executam o sistema operacional Linux (kernel 2.2.16), e têm as características mostradas na tabela 4.3. Como as máquinas não são idênticas, foram adotadas algumas soluções na implementação do consenso para que essas diferenças fossem diluídas e não induzissem desvios do comportamento dos sistemas testados (Sergent utilizou como parâmetros valores obtidos em uma rede heterogênea de máquinas Sun SPARCStation-20). Essas soluções serão tratadas com mais detalhes na seção 5.1, que trata especificamente da implementação deste algoritmo.

TABELA 4.3 - Características das máquinas utilizadas

Máquina	***Processador***	***Memória***
chevy	Pentium II 233MHz	64 MB RAM
mercedes	Pentium MMX 200MHz	64 MB RAM
buick	Pentium MMX 233MHz	64 MB RAM
jaguar	Pentium MMX 233MHz	64 MB RAM
porsche	Pentium Pro 200MHz Dual	64 MB RAM

Como a rede não é isolada nem os sistemas testados são os únicos processos presentes nas máquinas, foram tomadas algumas precauções para minimizar os efeitos que a interação entre esses elementos pudesse causar nos resultados. Mais especificamente, procurou-se executar todos os testes em horários de baixa utilização, principalmente à noite e durante finais de semana.

4.2.4 Linguagem de implementação

Um ponto crucial dos testes realizados (e por vezes, polêmico) refere-se à linguagem com que foram implementados os detectores e o consenso. Ao invés de escolher uma linguagem mais tradicional, como C por exemplo, optou-se por implementar os sistemas em Java.

A linguagem Java tem a seu favor a simplicidade de expressão, portabilidade e suporte à programação concorrente. Por ter sido desenvolvida objetivando dar suporte a aplicações em uma rede, torna mais fácil a distribuição e a intercomunicação das aplicações. Além disso, devido à orientação a objetos, permite a reutilização de classes desenvolvidas, e provê estruturas de alto nível que facilitam o entendimento e agilizam a construção das aplicações.

Apesar da utilização de uma máquina virtual e da interpretação de código reduzirem o desempenho dos sistemas desenvolvidos, existem outros fatores que pesaram a favor na escolha desta linguagem para a implementação dos componentes testados:

- disposição do grupo de Tolerância a Falhas da UFRGS para implementar componentes de uma ferramenta de Comunicação de Grupo em Java;
- experiência obtida anteriormente na implementação de um detector de defeitos [EST 00a]. A reutilização da estrutura de desenvolvimento minimizou o esforço desprendido na construção dos diversos detectores;
- tipo de comparação entre os detectores: como a avaliação é sobretudo qualitativa, o desempenho absoluto não é um ítem essencial.

De fato, ao comparar qualitativamente os detectores, as implementações foram liberadas da necessidade de otimizar o desempenho: foram então utilizadas estruturas de dados de alto nível, como `Vector` e `Hashtable`[8] do Java, que possibilitam uma melhor compreensão das operações realizadas.

No entanto, as facilidades da API do Java foram preteridas em alguns momentos, principalmente no que se refere à comunicação. Ao invés de utilizar uma biblioteca de comunicação de alto nível, como Java RMI, foram utilizados *sockets* UDP. Esta escolha deve-se tanto à tentativa de equiparar o sistema de comunicação utilizado por Sergent *et al.* [SER 99] (datagramas UDP/IP com comunicação ponto-a-ponto) como pela facilidade de traduzir *sockets* em outras bibliotecas de comunicação, se algum trabalho futuro necessitar.

4.2.5 Limitações do ambiente de operação

A contrapartida para a escolha da linguagem de implementação e do mecanismo de comunicação está na determinação das limitações do ambiente Java, para saber quais as restrições do próprio sistema que podem interferir na análise do desempenho dos detectores.

Para tanto, foram realizadas medições com os detectores de defeitos para indicar quais as taxas de envio e recebimento de mensagens que a aplicação e o ambiente Java

8 Estas estruturas permitem a armazenagem de objetos heterogêneos e agilizam o acesso a eles através de buscas seqüenciais e por palavras-chave.

suportam. Estes testes foram conduzidos com os detectores dispostos na mesma configuração dos experimentos (cinco detectores, cada um em uma máquina diferente), e os dados foram obtidos junto a um destes detectores.

Na figura 4.4 pode-se observar que a máxima capacidade de envio de um detector é atingida por volta dos 20 ms de intervalo de envio, quando são enviadas aproximadamente 90 mensagens por segundo. Como os testes dos detectores utilizam intervalos de envio de no mínimo 75 ms, pode-se assumir inicialmente que não há restrição por parte do ambiente Java para as taxas de envio das mensagens.

Na figura 4.5 é mostrada a capacidade de recebimento das mensagens dos detectores. Como os cinco detectores estão enviando mensagens uns para os outros, pode-se observar que existe um limite prático, onde os detectores atingem sua capacidade máxima de recepção quando cada processo envia mensagens a intervalos de 100 ms. Intervalos menores apenas sobrecarregam os recursos da rede, sem que as mensagens sejam processadas na mesma velocidade. Com base nos dados obtidos, verifica-se que a taxa máxima de processamento de cada detector é de cerca de 40 mensagens por segundo. Esse valor é muito importante, pois essa limitação pode criar diversos problemas para os detectores que geram muitas mensagens.

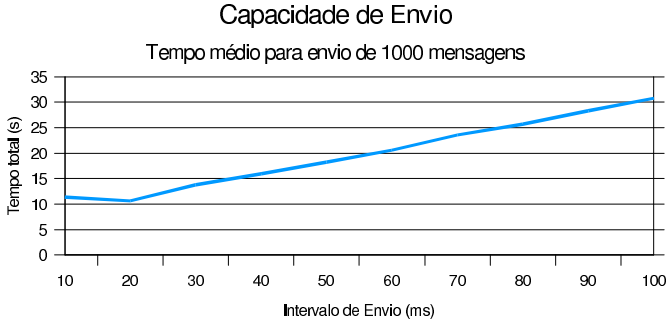

FIGURA 4.4 - Capacidade de envio

FIGURA 4.5 - Capacidade de recebimento de mensagens

Embora estas experiências apontem as limitações do ambiente de execução para enviar e receber as mensagens dos detectores, não são feitas considerações sobre a capacidade dos detectores processarem suas próprias mensagens. Como os detectores e o consenso trabalham com dados e operações mais complexas, suas capacidades de processamento devem ser bem menores do que as taxas de envio e recepção indicam.

Dessa forma, uma nova experiência foi executada, a fim de obter o tempo médio que um detector gasta para receber, processar e enviar mensagens.

O teste realizado pode ser considerado como a avaliação do *roundtrip* médio (tempo que uma mensagem leva para ir e voltar entre duas aplicações, incluindo o processamento das mensagens), e foi executada sobre um detector *Pull*. A escolha desse modelo de detector deve-se à sua própria construção interna, que conta o tempo decorrido entre o envio de uma mensagem de interrogação e o recebimento de sua resposta. Se outro modelo fosse escolhido, um detector *Push*, por exemplo, a correta definição das medidas obrigaria o uso de relógios sincronizados. Como são computadas também as operações de empacotamento e extração das mensagens, o *roundtrip* medido representa satisfatoriamente o comportamento que os detectores implementados compartilham.

O *roundtrip* médio de uma mensagem, obtido com esta experiência, foi de cerca de 133,43 ms. Do ponto de vista da detecção de defeitos, isso significa que um detector do tipo *Pull* corre o sério risco de realizar uma detecção incorreta se escolher um *timeout* com um valor inferior ou muito próximo a 133,43 ms. Por outro lado, os detectores que não dependem da resposta às suas mensagens estão menos sujeitos a esse limite, pois o tempo necessário para que uma mensagem seja enviada a um processo deve ser menor. Se este tempo de envio for considerado como a metade do *roundtrip*, o limite mínimo de tais detectores deve estar próximo a 65 ms.

4.3 Situações de Teste Propostas

Enquanto o artigo de Sergent *et al.* explora o comportamento dos diversos detectores através de uma simulação, o presente trabalho pretende analisá-los em situações práticas, onde a interação com os demais componentes do sistema influencia o desempenho final, tanto dos detectores quanto do consenso. Dessa forma, procura-se executar testes semelhantes como forma de comparar as avaliações, sem no entanto desprezar as peculiaridades encontradas na execução real.

Sergent *et al.* [SER 99] utilizaram basicamente duas situações de testes: operação com e sem a presença de falhas nos processos. Mais especificamente, o modelo de testes sem falhas é executado de forma que, na simulação, nenhum dos processos participantes do consenso fique defeituoso. Ainda assim os detectores podem cometer enganos, sobretudo quando há sobrecarga devido a um número excessivo de mensagens.

Já o modelo com falhas explora o pior caso que o algoritmo de consenso pode encontrar, em uma rodada: após o coordenador receber um número de mensagens do tipo *estimate* suficiente para emitir sua decisão, este falha sem ter enviado a mensagem *propose* para os processos. Neste ponto, os demais participantes do consenso necessitam obter um sinal do detector que indique este defeito. Como o tempo necessário para obter essa informação depende sobretudo da agilidade com que o detector realiza a sua detecção, este é um teste adequado para analisar a relação entre a latência da detecção, a precisão e o desempenho.

Embora a métrica de comparação adotada por Sergent *et al.* (o tempo de terminação da operação de consenso) seja eficiente para medir o impacto dos detectores, os critérios utilizados para a geração das medidas variaram de acordo com as características de cada detector: em certos casos, Sergent avaliou o comportamento do

detector quando é variado o *timeout*, mantendo o intervalo de envio fixo; em outros casos, o intervalo de envio variou proporcionalmente ao *timeout*. Essa diversidade de medições torna difícil a comparação do comportamento dos detectores, uma vez que as medidas são obtidas para combinações diferentes de parâmetros. Dessa forma, Sergent *et al.* preferiram fazer análises pontuais, comparando os detectores apenas nos pontos onde apresentaram os melhores desempenhos (tabela 4.1).

Quando se pretende avaliar os detectores em um ambiente de operação normal, a metodologia dos testes (e os próprios testes) devem ser repensados. Estes itens envolvem as métricas utilizadas, os modelos de operação, a maneira como serão obtidas as medidas e como serão injetadas as falhas (quando necessário), entre outros [JAI 91].

4.3.1 Métricas principais e auxiliares

O tempo de terminação do consenso foi adotado por Sergent *et al.* para indicar a influência dos modelos de comunicação e da sobrecarga que eles induzem no sistema. De fato, essa é uma das melhores formas de relacionar esses impactos, e foi também escolhida como métrica principal deste trabalho.

O registro do tempo de terminação do consenso foi realizado através da comparação, em valores absolutos, do instante onde o algoritmo de consenso inicia o diálogo com o coordenador e os demais processos e do instante onde finalmente o algoritmo obteve uma decisão unânime. Como a operação prática envolve a interação de diversos componentes dos sistemas, o tempo de computação assume um papel mais importante dentro do tempo de terminação do consenso, e o papel que um processo assume em uma rodada do consenso pode significar um maior ou menor esforço de processamento.

A avaliação exclusiva do tempo de terminação do consenso, foi considerada insuficiente, pois quando a aplicação de testes é executada em um ambiente real, o tempo de terminação do consenso pode ser influenciado por fatores externos aos algoritmos, que poderiam causar distorções cuja origem é difícil determinar. Por exemplo, os casos onde a sobrecarga no processamento induz a uma detecção incorreta podem ter influências externas que não são distingüíveis da sobrecarga causada especificamente pelo modelo de comunicação. Por isso, a validação das observações deve empregar outras métricas [JAI 91].

Infelizmente, dentro do algoritmo de consenso não há outra métrica que possa ser relacionada com tal facilidade ao impacto das detecções incorretas, da sobrecarga e da agilidade do detector sobre o desempenho da aplicação. Assim, esta validação deve ser feita a partir de um observador externo ao processo.

O ambiente Java não dispõe de nenhum método que permita expressar de maneira satisfatória a sobrecarga do sistema. Existem apenas dois métodos, `Runtime.totalMemory()` e `Runtime.freeMemory()` que retornam, respectivamente, a quantidade total de memória reservada à máquina virtual Java e a quantidade de memória disponível para os objetos. Entretanto, esses dados não são suficientes para uma boa análise.

A escolha final recaiu então sobre o sistema operacional. Existem diversas ferramentas que permitem capturar informações sobre o desempenho dos processos e a carga de processamento no sistema operacional Linux. Entre elas, a ferramenta `top` mostrou-se bem adequada, pois exibe diversas informações sobre os processos, e

apresenta duas métricas que foram consideradas muito úteis à validação do tempo de terminação do consenso: o tempo de CPU e a utilização de memória.

As métricas escolhidas têm relação direta com a medida do desempenho da aplicação. O tempo de CPU indica a quantidade de processamento que realmente foi utilizada pelo programa, de forma que é possível distingüir quando o tempo de terminação do consenso é influenciado pela sobrecarga do sistema ou quando este é influenciado pelo tempo em espera do algoritmo. A utilização da memória pelo processo indica a quantidade de memória que teve que ser alocada, e esta medida é especialmente representativa quando se considera que a sobrecarga dos processos induz a um aumento dos *buffers* de envio e recepção de mensagens, bem como a criação de novos objetos ou *threads* para o tratamento das mensagens.

4.3.2 Modelos de operação

Uma das principais modificações realizadas, em relação ao trabalho de Sergent *et al.*, diz respeito aos modelos de operação utilizados para obter as medidas. Naquele trabalho [SER 99], o tempo de terminação do consenso era analisado em dois casos: execução livre de defeitos (*failure free*) e pior caso (*worst case*) com o colapso de um processo. Em ambos modelos de simulação, o desempenho do sistema era avaliado através do tempo de resposta e da sobrecarga gerada na disputa pelo meio de comunicação. Dessa forma, as curvas de desempenho apresentavam tanto o impacto causado pelo tempo de espera pela detecção quanto o impacto das detecções incorretas (causado pela sobrecarga imposta pelo modelo de comunicação).

Quando levados para uma aplicação real, estes modelos que consideram ambas possibilidades de degradação do desempenho se mostraram muito suscetíveis à interação com os componentes do sistema (*buffers*, acesso à rede, outros processos, velocidade de processamento da máquina), de forma que a curva que mostra a tendência de operação dos detectores pode ser atrapalhada por essas interferências, e amostras obtidas em momentos distintos podem ser significativamente diferentes. Um exemplo dessas variações ocorre no caso da falha do coordenador: o consenso pode ser finalizado tanto na rodada seguinte como se estender por diversas rodadas, até que um coordenador não seja considerado falho pelos processos. Como essas manifestações são praticamente impossíveis de controlar, optou-se por modificar os modelos de operação, de modo que estes apresentassem as curvas de tendências de forma mais estável, e que pudessem ser comparadas às dos demais detectores.

Além disso, a análise do comportamento dos detectores em toda a faixa de testes, ao contrário da análise do ponto de melhor desempenho, permite que os detectores sejam comparados independente das características da implementação, tornando-se assim uma comparação qualitativa dos detectores, ao invés de uma comparação quantitativa.

Optou-se então pela análise de três casos de operação distintos: um caso "normal", e dois casos hipotéticos. Os casos hipotéticos representam o melhor e o pior cenário de falhas que o algoritmo de consenso pode encontrar em uma rodada. A grande importância destes casos hipotéticos é que suas construções tentam remover os efeitos da imprevisibilidade dos detectores, de forma que o desempenho do consenso seja avaliado apenas pelo modelo de comunicação ou pela latência da detecção.

Mais especificamente, no caso hipotético da melhor situação, chamado que representa a situação *best case*, não ocorrem suspeitas por parte dos detectores, de forma que o consenso pode ser terminado em uma única rodada. Isso faz com que a curva de desempenho represente unicamente o impacto causado pela sobrecarga do tratamento das mensagens, e a ausência das suspeitas permite que sejam avaliadas situações de grande sobrecarga sem que o tempo de terminação seja influenciado pelas detecções incorretas. Como os detectores de defeitos podem gerar ocasionalmente suspeitas incorretas, para construir esta situação hipotética foi necessário fazer o algoritmo de consenso ignorar as suspeitas geradas pelos detectores. Esta adaptação, no entanto, não significa o mesmo que "desligar" os detectores e realizar o consenso em uma situação forçosamente sem falhas; é necessário a presença dos detectores e de seu funcionamento, caso contrário não seria possível comparar o impacto dos diversos modelos de detectores.

A segunda situação hipotética, chamada *worst case*, torna possível medir a latência de detecção dos diversos modelos de detectores testados. Esta situação representa os cenários de falhas onde o coordenador entra em colapso imediatamente após receber mensagens da maioria dos processos do consenso (mais especificamente, o coordenador pára antes de enviar sua mensagem do tipo *propose*). Quando identifica a falha do coordenador, um processo inicia uma nova rodada, para tentar finalizar o consenso. No entanto, devido às falsas suspeitas, o consenso pode se estender por diversas rodadas até que um coordenador não seja suspeito por nenhum detector. Essa possível variação no número de rodadas necessárias para atingir o consenso impossibilitaria a análise dos dados, de forma que foram necessárias algumas medidas para restringir o número de rodadas necessárias. Assim, a situação *worst* case é construída de forma que a cada nova invocação do consenso, o coordenador da primeira rodada paralisa suas atividades (mais especificamente, paralisa seu detector) imediatamente antes de enviar uma mensagem *propose*, de forma que o consenso seja obrigatoriamente finalizado apenas na próxima rodada. Na segunda rodada também é feito um controle sobre as detecções incorretas, para que o consenso seja resolvido na segunda rodada. Por ser representativo de uma situação bem definida, o modelo hipotético *worst case* implementado permite que seja destacado o impacto da latência da detecção na identificação do defeito do primeiro coordenador, a exemplo do que faz o modelo hipotético *best case* com relação à sobrecarga do sistema.

De nada adiantaria, entretanto, as duas curvas de comportamento se não fosse possível identificar a tendência dos detectores em um determinado instante. É por isso que o caso de operação "normal" é necessário, uma vez que, com ele, se pode avaliar não só a probabilidade do consenso ser mais suscetível à sobrecarga ou às detecções incorretas, mas também avaliar o comportamento dos detectores quanto à precisão de sua detecção. Na figura 4.6, por exemplo, estão representadas as curvas dos três casos de operação para um detector *Push*. Nela, pode-se observar que na situação *best case* o tempo de terminação do consenso somente é afetado quando se utilizam *timeouts* menores do que 400 ms, pois o detector *Push* obriga o envio de mensagens a intervalos mais curtos do que o *timeout*, aumentando a sobrecarga do sistema. Já o modelo *worst case* demonstra que quanto maior for o *timeout*, maior será a latência do detector, e por conseqüência, mais tempo levará para finalizar o consenso. A análise do caso normal (*normal case*) demonstra que, para *timeouts* maiores do que 400 ms, o tempo de terminação do consenso se aproxima do modelo *best case*, o que significa que são muito poucas as suspeitas incorretas. No entanto, para valores de *timeout* menores do que 400

ms, essa relação se inverte, chegando ao ponto em que o tempo médio de terminação do consenso ultrapassa o do modelo *worst case*. Isto evidencia que não somente existem muitas detecções incorretas, como o consenso se estende por diversas rodadas até que consiga encontrar um coordenador que não é considerado suspeito.

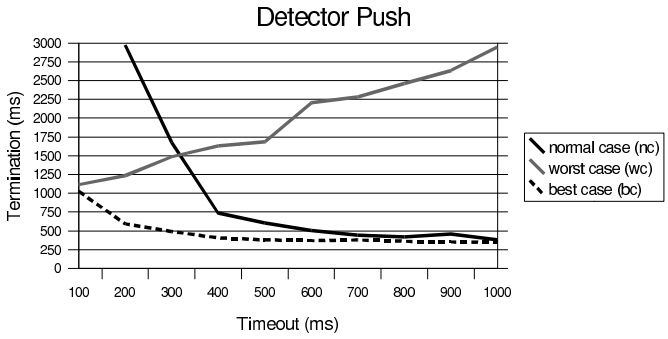

FIGURA 4.6 - Exemplo dos três modelos de operação

Assim, é possível comparar detectores diferentes através dos modelos hipotéticos, e analisar suas tendências de operação através do modelo de execução normal.

4.3.3 Parâmetros de inicialização

De forma geral, existem dois parâmetros de inicialização que os detectores estudados utilizam. Os parâmetros principais são o intervalo de envio (Δ_i) entre as mensagens e o *timeout* (Δ_{to}) para a suspeita do defeito [SER 99]. Somente o detector *Heartbeat* utiliza um conjunto diferente de parâmetros, entre eles o intervalo de envio (Δ_i), o tempo entre as amostragens (Δ_s) e o número de vizinhos [EST 00a].

Cada detector combina esses parâmetros conforme as suas exigências e características, e a tabela 4.4 mostra quais parâmetros cada detector utiliza.

TABELA 4.4 - Parâmetros de inicialização dos detectores

Detector	**Δ_i**	**Δ_{to}**	**Δ_s**	**Nº. vizinhos**
Push	✓	✓		
Pull	✓	✓		
AdaptivePush	✓			
AdaptivePull	✓			
ad-hoc "no message"		✓		
ad-hoc "heart-beat"	✓	✓		
Heartbeat	✓		✓	✓

Os detectores *Push*, *Pull* e *ad-hoc "heart-beat"* são construídos de forma que as mensagens de detecção são enviadas periodicamente e julgadas através de um *timeout*. Os detectores adaptativos funcionam de maneira similar, mas como o *timeout* é modificado dinamicamente, este não é fornecido entre os parâmetros de inicialização. O

detector *Heartbeat* faz uma amostragem periódica do número de mensagens que recebeu no último período, e não através de um *timeout* vinculado à recepção de uma mensagem. Por fim, o detector *ad-hoc "no message"*, utilizando as próprias mensagens *estimate* e *propose* do consenso, observa o tempo entre elas e julga através de um *timeout*.

O trabalho de Sergent *et al.* escolheu como parâmetro variável o *timeout* (Δ_{to}). No entanto, para cada detector, as medições eram feitas de formas diferentes, adaptadas a cada detector. Assim, por exemplo, o detector *Pull* era avaliado com valores fixos de Δ_i, enquanto o detector *ad-hoc "heart-beat"* era avaliado com um $\Delta_i = 98\% \Delta_{to}$. Para evitar essas diferenças na hora de tomar as medidas, procurou-se relacionar proporcionalmente as variáveis dos detectores sempre que possível, tentando exprimir o comportamento dos detectores nessa relação.

Detector *Push*: no caso do detector *Push*, cada detector controla o *timeout* a partir do recebimento das mensagens enviadas pelos outros detectores. Isso pode representar um problema se as máquinas envolvidas no processamento tiverem seus relógios internos com velocidades diferentes. Assim, o detector que envia as mensagens deve tentar garantir que o intervalo de envio seja menor do que o *timeout* de quem recebe, para que o tempo limite não seja ultrapassado antes que uma nova mensagem seja recebida (é comum, na prática, observar *timeouts* equivalentes a duas vezes o intervalo de envio). A partir dessa interrelação, é fácil deduzir uma relação proporcional entre o intervalo de envio (Δ_i) e o *timeout* (Δ_{to}).

Detector *Pull*: já nos detectores do tipo *Pull*, o controle de *timeout* é feito entre o envio da mensagem "*Are you alive?*" e a sua resposta. Dessa forma, um detector não precisa se preocupar com os valores dos *timeouts* dos outros detectores, mas deve respeitar o tempo mínimo necessário para que a mensagem de interrogação seja enviada, processada e respondida. Esse tempo mínimo é o *roundtrip*, que já foi determinado para o ambiente de testes, na seção 4.2.5. No entanto, o intervalo entre os envios pode ser maior, o que reduz a sobrecarga da rede e do sistema, mas aumenta o tempo de espera pela detecção dos defeitos. O problema da definição dos parâmetros, aqui, refere-se então à escolha do melhor intervalo de envio. Se fosse considerado apenas o desempenho final dos detectores, talvez a utilização de longos períodos entre as transmissões levaria a desempenhos ótimos: isso faria com que o modelo de comunicação tivesse um efeito mínimo, dificultando a análise de seu impacto.

Assim, para a avaliação dos detectores *Pull*, optou-se por trabalhar com relações mais próximas entre o intervalo de envio e o *timeout*. Nestes casos, o impacto do modelo de comunicação é maior sobre o desempenho do sistema.

Um fato a ressaltar, na análise dos detectores *Pull*, é que se decidiu utilizar relações proporcionais entre seus parâmetros, ao contrário dos experimentos de Sergent *et al.* Isso se deve à análise preliminar dos dados, onde foi constatado que o uso de valores fixos para o Δ_i acarreta um custo de mapeamento muito maior (para cada valor de Δ_i seria necessário aplicar todos os testes), enquanto os resultados obtidos com uma relação proporcional podem ser abstraídos de modo a representar a situação dos parâmetros fixos. Na figura 4.7, observa-se o comportamento dos experimentos com Δ_i fixo. Como cada parâmetro exibe um comportamento constante, pode-se extrair esse comportamento a partir da curva proporcional: se for utilizado um detector *Pull* com uma relação entre intervalo de envio e *timeout* equivalente a 100%, ou seja, com um

timeout igual ao intervalo de envio, o valor obtido para um *timeout* de 400 ms será equivalente ao exibido pelo detector que usa um intervalo de envio fixo em 400 ms.

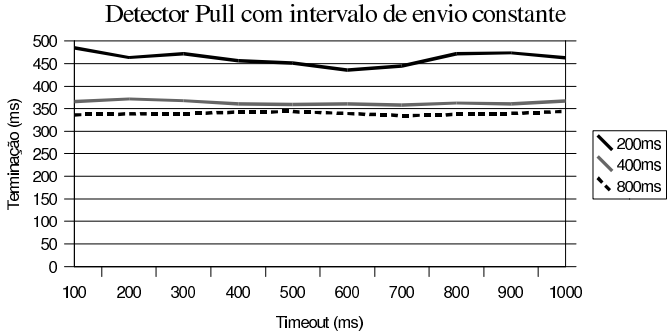

FIGURA 4.7 - Detector *Pull* usando Δ_i constante

Detectores Adaptativos: os detectores adaptativos, por suas características de definir dinamicamente valores de *timeout* adequados a cada processo monitorado, não necessitam nenhum parâmetro fixo, e a variação é feita sobre o intervalo de envio. Foi necessário apenas definir o valor inicial de *timeout* que os detectores utilizarão, e qual é a taxa de incremento que será aplicada sobre este *timeout*.

Detector *Heartbeat*: o detector *Heartbeat* é conhecido por ser "isento" de *timeout* [AGU 97a], isto é, ele não faz uso de *timeouts* diretamente sobre as mensagens, como outros detectores. No entanto, isto não o desobriga de definir instantes onde as suspeitas serão julgadas (o que acaba configurando uma espécie de *timeout*). Enquanto os outros detectores utilizam o próprio evento gerado pela violação do *timeout*, o *Heartbeat* necessita um outro mecanismo capaz de controlar esses intervalos. Esse temporizador, no entanto, está ausente da definição original do detector (na definição, o detector apenas contabiliza as mensagens, mas quem é responsável pela detecção é a aplicação), e a implementação anexou-o ao detector, para padronizar a relação entre o consenso e os detectores. Desse modo, o parâmetro que indica o intervalo entre os julgamentos das suspeitas foi chamado intervalo de amostragem (Δ_S). Assim como nos detectores *Push*, é interessante que o intervalo entre os envios seja menor do que o tempo de amostragem.

O *Heartbeat* ainda conta com mais um parâmetro que pode influenciar no seu desempenho: o número de vizinhos. Embora seja uma técnica desenvolvida para diminuir o número de mensagens na rede, quando o detector foi implementado e adaptado para a operação com o algoritmo de consenso de Chandra, este parâmetro passou a ser um dos grandes responsáveis pelo desempenho do detector [EST 00c]. Como mostra a figura 3.8, a escolha do número de vizinhos que um detector utiliza influencia diretamente a quantidade de mensagens que este irá transmitir, e sobretudo, repassar aos demais detectores. Quando o número de mensagens cresce muito, o algoritmo ultrapassa a capacidade de processamento que o ambiente permite, ficando muito fácil acumular mensagens nos *buffers* e realizar detecções incorretas.

Desse modo (e como comprovaram posteriormente os experimentos) o detector *Heartbeat* implementado apresenta um desempenho inferior ao dos demais detectores, desqualificando-o para os casos onde seja necessário uma maior agilidade de detecção. Para constar entre os detectores analisados, o *Heartbeat* foi testado com dois processos

na sua lista de vizinhos (caso que gera o menor número de mensagens), e a faixa de testes do detector foi diferente dos demais detectores, pois seu processamento é mais demorado e gera uma sobrecarga bem maior. Não foi considerado o caso de operação com um único vizinho, pois este impediria a propagação das mensagens no caso da falha do coordenador (*worst case*).

Detector *ad-hoc "no message"*: um detector *ad-hoc* é uma implementação otimizada para uma determinada aplicação, e no caso, o algoritmo de consenso. O detector *ad-hoc "no message"* utiliza somente a troca de mensagens do consenso para fazer a detecção dos defeitos, através de um *timeout*. Embora este detector possa apresentar um grande número de detecções incorretas quando o *timeout* é muito pequeno, em relação ao avanço do algoritmo de consenso, ele não influi no tráfego da rede nem na carga de processamento, tornando muito rápida a detecção em situações de poucas falhas. O detector *"no message"* tem o *timeout* como único parâmetro que pode ser variado, uma vez que ele não emite nenhum tipo de mensagem de detecção adicional.

Detector *ad-hoc "heart-beat"*: outra variante dos detectores *ad-hoc*, o "*heart-beat"* trabalha sobre a mesma base do detector *ad-hoc* "*no message"*, mas procura evitar os problemas daquele detector ao enviar mensagens "*I am alive*" no intervalo em que é necessária a detecção. Assim, um processo não considera erroneamente suspeito o outro, que está demorando para enviar sua resposta. O *ad-hoc "heart-beat"* funciona de forma semelhante a um detector do tipo *Push*, embora apenas envie mensagens de detecção durante um intervalo bem específico do consenso. Para os experimentos foram inicialmente escolhidos os mesmos parâmetros de teste empregados no detector *Push:* no entanto, a experiência demonstrou que a relação $\Delta i = 98\% \Delta to$ não permitia a realização do consenso na situação *normal case* (a ocorrência de suspeitas incorretas em excesso impedia a finalização do consenso). Assim, os resultados apresentados consideram apenas uma relação de $\Delta i = 75\% \Delta to$.

A tabela 4.5 reúne os valores dos parâmetros e variáveis empregados nos experimentos.

TABELA 4.5 - Parâmetros utilizados nos experimentos

Detector	Parâmetros "fixos"	Parâmetro variável	Intervalo de testes
Push	$\Delta i = 98\% \Delta to$ e $\Delta i = 75\% \Delta to$	Δto	100-1000 ms
Pull	$\Delta i = 150\% \Delta to$ e $\Delta i = 100\% \Delta to$	Δto	100-1000 ms
AdaptivePush	Δto inicial = 100 ms, incremento = 50 ms	Δi	100-1000 ms
AdaptivePull	Δto inicial = 100 ms, incremento = 50 ms	Δi	100-1000 ms
Heartbeat	$\Delta i = 75\% \Delta s$ e vizinhos = 2	Δs	200-2000 ms
ad-hoc "no message"	-	Δto	100-1000 ms
ad-hoc "heart-beat"	$\Delta i = 75\% \Delta to$	Δto	100-1000 ms

4.3.4 Número de iterações

A escolha do número de iterações com cada parâmetro de teste é essencial para uma correta expressão da curva de comportamento, uma vez que a média dos valores tende a corrigir eventuais desvios provocados pelo indeterminismo do consenso. Como objetivo de cada execução (considera-se uma execução cada avaliação de um detector, para um conjunto de parâmetros e situações específicas), foi determinado como um valor aceitável a realização de cerca de 200 operações de consenso, uma vez que, quando for feita a compilação dos tempos de terminação nas cinco máquinas, haverá uma média de 1000 medidas de tempo. A escolha de um número maior de iterações teria um custo de tempo elevado, considerando que este valor deveria ser aplicado a cada parâmetro testado nos diversos detectores.

Ocorre, no entanto, que os detectores de defeitos, quando geram uma excessiva quantidade de suspeitas, tendem a bloquear a operação de consenso - suas características dizem que é preferível bloquear o consenso quando não conseguem executar corretamente suas tarefas do que permitir a violação da consistência do consenso. Assim, nestes casos não é possível finalizar todos os consensos estabelecidos. A aceitação dos valores registrados (quando um detector permite que alguns poucos consensos sejam realizados antes de bloquear) deixa de ser automática e leva em consideração a observação *in loco* do autor: se os poucos valores registrados ainda são representativos, estes serão aceitos; senão, um novo *run* é executado com aqueles parâmetros, tentando atingir um número de registros adequados. Apenas em casos onde ambas tentativas falham na avaliação é que os valores são ignorados e aquele conjunto de testes é retirado da curva de comportamento dos detectores analisados.

Outra escolha que teve que ser feita envolve a maneira como seria rodada a aplicação de testes. No caso da simulação, possivelmente é indiferente rodar duzentas vezes uma aplicação que faz um único consenso, ou rodar uma única vez uma aplicação que realiza duzentos consensos. Já na prática esta escolha é importante, pois ela diz respeito ao custo e à precisão das medidas. A escolha, neste caso, recaiu sobre uma aplicação que realiza diversos consensos. Isso se deve sobretudo à constatação de que, entre os detectores a serem avaliados, alguns exigem um "tempo de estabilização", antes de operarem eficientemente (no caso, os detectores adaptativos), e outros detectores não iriam sentir o impacto do acúmulo de mensagens de detecção se esta fosse repetidamente interrompida.

4.4 Considerações Finais

Neste capítulo foi feita uma revisão sobre o artigo de Sergent *et al.*, que serviu de base para a proposta deste trabalho. A partir daquele artigo, foram especificadas as situações de testes dos detectores, bem como o ambiente e as métricas utilizadas para a avaliação dos dados.

No próximo capítulo serão exploradas as peculiaridades da implementação do algoritmo de consenso e dos detectores.

5 Implementação dos Sistemas

Uma parte significativa deste trabalho envolveu a implementação dos detectores, do algoritmo de consenso e a adaptação do ambiente para os testes. A tradução dos algoritmos em componentes da aplicação deve ser feita de forma cuidadosa, de modo que tanto os objetivos da especificação sejam seguidos quanto seja evitada a utilização de técnicas e estruturas que notadamente possam interferir na análise de desempenho.

Como os componentes desenvolvidos podem vir a fazer parte de um projeto maior, que objetiva a construção de uma ferramenta de comunicação de grupo por parte dos pesquisadores do grupo de Tolerância a Falhas da UFRGS, a definição da funcionalidade dos componentes e de suas interligações procurou ser o mais genérica e flexível possível, a fim de adaptar-se aos seus possíveis usos.

Neste capítulo serão enfocadas as técnicas e soluções utilizadas para a construção do algoritmo de consenso e dos detectores, da aplicação de testes utilizada, e as modificações necessárias para a obtenção dos resultados da análise.

5.1 Implementação do Consenso

Apesar das vantagens dos algoritmos de consenso apresentados no capítulo 2, a escolha do algoritmo a ser utilizado nos testes foi limitada principalmente devido ao uso do algoritmo de Chandra e Toueg [CHA 96a] no trabalho de Sergent *et al.* [SER 99]. Como as simulações basearam-se sobretudo no impacto do número de mensagens trocadas e na saturação dos *buffers* de recepção e transmissão, a escolha de um algoritmo com um número diferente de passos de comunicação poderia influenciar profundamente os resultados obtidos.

Mesmo recaindo a escolha sobre o algoritmo mais conhecido para a realização do consenso, diversos aspectos de implementação tiveram que ser analisados. Um dos principais desafios, e que influenciou muito as escolhas do projeto, foi a de implementar o algoritmo de modo que este fosse um componente que contivesse todas as funcionalidades básicas do consenso, e que pudesse ser facilmente anexado a outros componentes que foram ou serão criados.

Outra escolha, inspirada na exigência de flexibilidade do componente de consenso, é a de não restringir o tipo de dado submetido ao acordo. Enquanto a maioria dos algoritmos trabalha com valores simples, como 0 e 1, as necessidades de uma aplicação real obrigam a extensão desses tipos de dados. Por outro lado, o processamento do algoritmo não é alterado substancialmente, e apenas influencia o tamanho das mensagens, e conseqüentemente, o desempenho final das operações.

5.1.1 Estrutura de classes

O algoritmo de Chandra tem como característica o fato de impedir o avanço independente dos processos que estão participando do consenso. No entanto, em sistemas distribuídos complexos (como, por exemplo, uma ferramenta de comunicação

de grupo) existem diversas situações onde não há dependência de dados, e a limitação do avanço do consenso não deveria impedir essas operações.

A solução encontrada envolveu uma modificação na forma de utilização do algoritmo de consenso. Ao invés de ser construído como uma função ou procedimento, chamados seqüencialmente pela aplicação, determinou-se que a aplicação pode se relacionar com diversos objetos `Consensus`, sendo que cada um deles é auto-suficiente para realizar a operação de consenso. Além disso, esses objetos não são descartáveis, podendo ser reutilizados pela aplicação durante todo o período de atividade. Assim, se uma determinada operação necessita realizar diversos consensos e manter a consistência dos dados, deverá obrigatoriamente realizar todas as operações usando o mesmo objeto `Consensus`. Por outro lado, se puder realizar operações em paralelo, deverá utilizar dois ou mais desses objetos.

Como um objeto `Consensus` pode ser utilizado para a execução de diversos acordos entre os processos, foi necessário redefinir a durabilidade de muitas variáveis. Um exemplo claro refere-se ao identificador da rodada. Se fosse mantida a mesma funcionalidade do algoritmo de Chandra, cada novo consenso faria com que a contagem das rodadas fosse reiniciada. Isso possibilitaria que mensagens atrasadas interferissem em diferentes operações de consenso. Por isso, foi adotada uma contagem contínua, de forma que mensagens atrasadas fossem barradas pelas próprias restrições do algoritmo de consenso.

Essa contagem contínua das rodadas mostrou-se muito importante também na obtenção das medidas deste trabalho. Se fosse utilizada uma nova contagem para cada consenso realizado, invariavelmente o coordenador da primeira rodada seria o mesmo processo, pois o algoritmo determina uma ordem fixa e pré-determinada para a escolha dos coordenadores: $coord = (round \bmod n) + 1$. Como, em grande parte dos casos, o consenso é resolvido em apenas uma rodada [GUE 97], a análise de desempenho estaria firmemente ligada a uma única máquina e às suas características específicas. Além disso, na situação onde é simulada a falha do coordenador, sempre o mesmo processo iria ser considerado falho. Como as máquinas utilizadas apresentam características diversas (suas especificações foram citadas na tabela 4.3), as medidas poderiam ser tendenciosas. Com a contagem ininterrupta das rodadas, os processos se alternam no papel de coordenador da primeira rodada, de forma que o desempenho medido corresponde à média destes processos.

O controle sobre o objeto destinatário das mensagens é feito através de uma variável `ConsensusID`, que faz parte do cabeçalho de cada mensagem. O elemento responsável pela verificação do `ConsensusID` é o objeto `ConsensusFactory`. Na verdade, o `ConsensusFactory` é um dos mais importantes elementos do sistema, embora não seja ele o responsável direto pela realização dos consensos. Através do `ConsensusFactory`, são realizadas tanto as tarefas de criação e eliminação dinâmica de objetos `Consensus`, quanto a centralização do envio e da recepção das mensagens (figura 5.1).

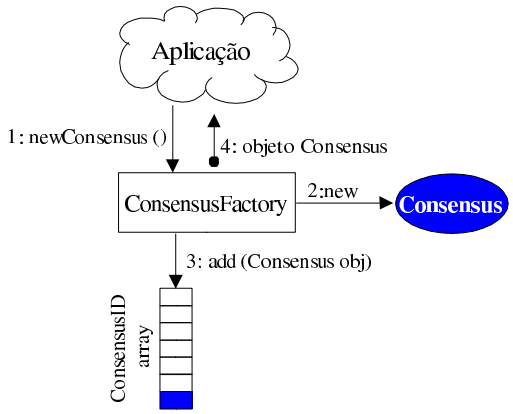

FIGURA 5.1 - Instanciação de um objeto Consensus

Essa centralização das mensagens é de extrema importância, pois não só permite que diversos objetos `Consensus` coexistam e operem simultaneamente, como evita a alocação de uma porta de comunicação para cada novo objeto `Consensus` criado.

Uma das escolhas do projeto foi a de utilizar comunicação por *sockets* UDP/IP, ao invés de protocolos mais sofisticados como, por exemplo, RMI ou CORBA. Como o consenso e a detecção de defeitos fazem parte dos primeiros componentes desenvolvidos e ainda não é possível ter uma visão geral da funcionalidade e das necessidades de uma aplicação mais global, optou-se pelo uso de *sockets* porque estes podem ser facilmente substituídos por outros protocolos de mais alto nível, enquanto a operação inversa nem sempre é possível. Assim, a partir de um único binômio endereço IP/porta é possível acessar todos os objetos `Consensus` com um conhecimento prévio mínimo (o endereço do objeto `ConsensusFactory`), conforme mostra a figura 5.2. Para auxiliar o `ConsensusFactory` na tarefa de entregar as mensagens para os objetos `Consensus`, são criadas *threads* do tipo `Postman`, específicas para realizar a entrega de forma concorrente, liberando assim o `ConsensusFactory` para outras atividades e eventos.

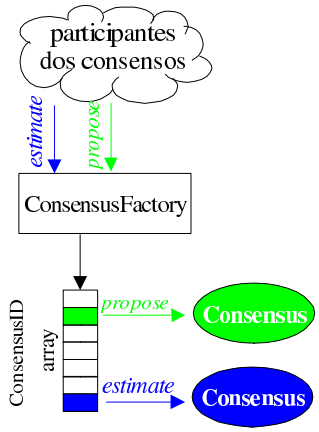

FIGURA 5.2 - Recepção e redirecionamento das mensagens através do ConsensusID

Outra decisão de projeto refere-se à localização dos detectores de defeitos. A granulosidade de detecção empregada possibilita que um detector esteja operando com cada processo (embora o caso mais comum seja um detector por máquina). Assim, a criação e ativação de um detector de defeitos para cada objeto `Consensus` não condiz com esta especificação, nem com o comportamento presumível dos detectores: embora não seja o mais indicado, uma aplicação pode criar um novo objeto `Consensus` para cada operação que deseja realizar, e a ativação de um novo detector impede o aproveitamento do conhecimento adquirido anteriormente (e vital para alguns tipos de detectores), assim como gera mais sobrecarga para o sistema.

A localização escolhida para os detectores ficou definida então como junto ao objeto `ConsensusFactory`. Para cada novo objeto `Consensus` criado, é passada uma referência a esse detector já existente, e o mecanismo de *call-back* empregado pelos detectores permite que os objetos `Consensus` também sejam dinamicamente incluídos na lista de notificáveis dos detectores (figura 5.3). Quem controla a inclusão e a remoção dos objetos `Consensus` da lista de notificáveis do detector é o objeto `ConsensusFactory`, quando da criação ou remoção dos objetos.

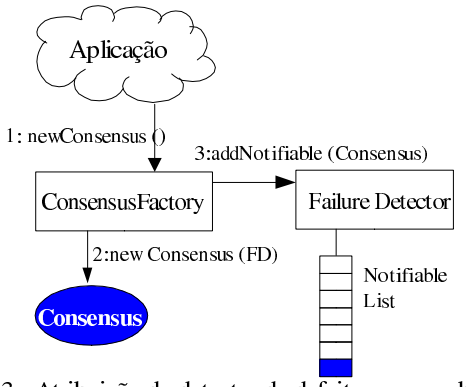

FIGURA 5.3 - Atribuição do detector de defeitos para o objeto Consensus

A partir destas especificações foi implementado o algoritmo de consenso, cujas classes em Java estão estruturadas conforme a figura 5.4.

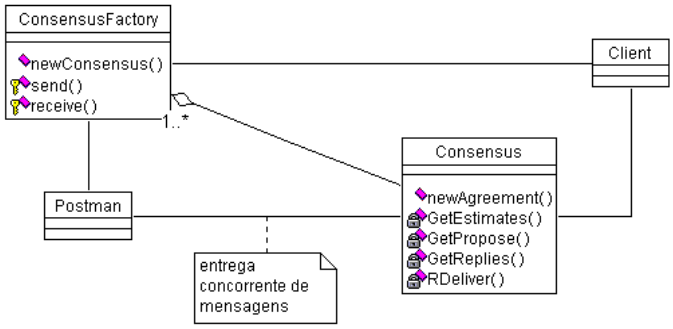

FIGURA 5.4 - Diagrama de classes do algoritmo de consenso implementado

5.1.2 Protocolo de troca de mensagens

A troca de mensagens tem um papel indispensável no algoritmo de consenso, pois é a única forma de contato entre os processos distribuídos. A construção do protocolo utilizado na aplicação, no entanto, teve que adicionar certos elementos não existentes no algoritmo original tais como, por exemplo, a variável `ConsensusID`, que identifica qual o objeto `Consensus` destinatário da mensagem. Além disso, certas mensagens de controle tiveram que ser incorporadas ao protocolo como, por exemplo, para prover a sincronização dos processos antes do consenso (garantir que todos processos estão prontos para começar a operação). Todas as mensagens foram compostas utilizando estruturas de alto nível da linguagem Java, principalmente objetos *Vector*. Os objetos Java do tipo *Vector* funcionam como recipientes para outros objetos, inclusive de forma recursiva, oferecendo assim a flexibilidade de conteúdo que é desejada para o protocolo de consenso desenvolvido.

Além disso, devido à estrutura de redirecionamento de mensagens do objeto `ConsensusFactory`, as mensagens foram compostas em duas camadas, permitindo que em um primeiro momento estejam disponíveis as informações essenciais ao analisador do `ConsensusFactory`, e que este possa repassar aos objetos `Consensus` apenas os parâmetros que necessita, sem no entanto precisar modificar a estrutura da mensagem.

- mensagens de sincronização

Embora o mecanismo do consenso seja projetado para trabalhar assincronamente, há uma necessidade prática de estabelecer um mínimo de disponibilidade entre os processos que participam do consenso. Isso significa que é necessário garantir que os processos que participam de um determinado consenso estão prontos para a operação. Se não houvesse esse controle, um processo poderia iniciar sua participação em um consenso até mesmo depois que os demais processos terminaram, obrigando o sistema a prover retransmissão de mensagens e outros mecanismos, para manter a consistência entre os participantes do consenso.

Embora a responsabilidade de sincronizar previamente os processos possa ser repassada à aplicação, optou-se por fazer isso dentro do próprio algoritmo de consenso. Dessa forma, toda vez que um novo consenso é chamado, há uma sincronização entre os processos através de um *handshaking*, de modo que apenas quando todos os processos estão prontos a operação é iniciada. O controle desta sincronização é centralizado no primeiro processo da lista de participantes. Essa sincronização é necessária somente no início da operação de consenso, pois se fosse executada a cada rodada obrigaria que todos os processos estivessem corretos, o que nem sempre corresponde ao padrão de falhas.

Inicialmente, cada processo envia a primeira mensagem para o primeiro processo da lista dos participantes do consenso (a lista é fixa e de conhecimento geral, para facilitar a definição do coordenador). Este processo contabiliza as mensagens que chegam (as mensagens são identificadas, e para evitar problemas de conexão, são retransmitidas periodicamente). Quando todos processos respondem, o "coordenador" da sincronização envia uma mensagem indicando o início do consenso (a composição das mensagens está na figura 5.5).

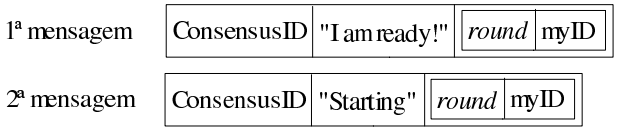

FIGURA 5.5 - Estrutura das mensagens de sincronização

- mensagem *estimate*

A mensagem *estimate* carrega as sugestões dos processos para a análise do coordenador, que escolhe uma entre elas para ser a sua sugestão de decisão. A primeira camada da mensagem contém o campo relativo ao `ConsensusID` do objeto `Consensus` destinatário, uma seqüência de caracteres que identifica o tipo de mensagem (neste caso, este campo contém o valor "Estimate") e o *Vector* que contém a segunda camada da mensagem. Quando o objeto `ConsensusFactory` identifica o destinatário, repassa este *Vector* para o método `GetEstimates()` do objeto destino.

O método `GetEstimates()` confere o campo *round*, para ver se a mensagem é válida para aquele consenso, e extrai os campos *estimate* e *timestamp*. O campo *estimate* contém o valor proposto, e na verdade também é um objeto do tipo *Vector* (cabe à aplicação saber qual o tipo do dado, uma vez que este aspecto não é importante para o consenso), enquanto o campo *timestamp* é utilizado pelo coordenador para escolher a sugestão do processo mais atualizado. A figura 5.6 mostra a estrutura da mensagem *estimate*.

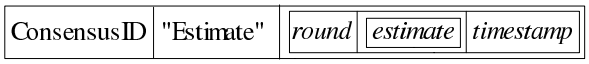

FIGURA 5.6 - Composição da mensagem *estimate*

- mensagem *propose*

A mensagem *propose* é utilizada pelo coordenador para enviar sua decisão aos processos. A primeira camada da mensagem contém, além do identificador `ConsensusID`, a seqüência de caracteres "Propose" que identifica o tipo de mensagem e o *Vector* que contém a segunda camada da mensagem. Quando o objeto `ConsensusFactory` identifica o destinatário, repassa este *Vector* para o método `GetPropose()` do objeto destino.

O método `GetPropose()` confere o campo *round*, para ver se a mensagem é válida para aquele consenso, e extrai o campo *decision*. O valor de *decision* é aceito como o valor proposto pelo coordenador (embora ainda não validado pelo consenso), e o *timestamp* é atualizado para o valor da variável *round* que veio do coordenador (que possibilita aos processos atrasados alcançarem o coordenador). A figura 5.7 mostra a estrutura da mensagem *propose*.

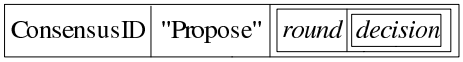

FIGURA 5.7 - Composição da mensagem *propose*

- mensagem *ACK/NACK*

A mensagem *ACK/NACK* tem função especial dentro do consenso, pois é ela quem indica ao coordenador se algum dos processos considerou-o suspeito (invalidando assim a rodada atual). A primeira camada da mensagem contém, além do identificador `ConsensusID`, a seqüência de caracteres "ACK/NACK" que identifica o tipo de mensagem e o *Vector* que contém a segunda camada da mensagem. Quando o objeto `ConsensusFactory` identifica o destinatário, repassa este *Vector* para o método `GetReplies()` do objeto destino.

O método `GetReplies()` confere o campo *round*, para ver se a mensagem é valida para aquele consenso, e extrai o conteúdo do último campo, que pode ser uma *string* "ACK" ou "NACK". Se for um "ACK", é incrementado o contador de ACKs do coordenador (quando receber ACKs da maioria dos processos, sem NACKs, o coordenador pode fazer a difusão confiável da decisão). Se for um NACK, é ativada a variável booleana NACK, que impede o coordenador de fazer a difusão confiável. A figura 5.8 mostra a estrutura da mensagem *ACK/NACK*.

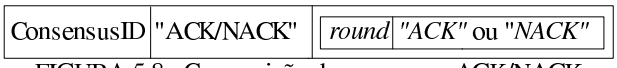

FIGURA 5.8 - Composição da mensagem ACK/NACK

- mensagem *RBCast*

A mensagem *RBCast* é utilizada para que todos os processos saibam que o consenso foi aceito pela maioria dos processos. Esta mensagem é difundida através da técnica chamada "melhor esforço", onde os processos que ainda não tinham recebido esta mensagem repassam-na para todos os demais processos. A primeira camada da mensagem contém, além do identificador `ConsensusID`, a seqüência de caracteres "RBCast" que identifica o tipo de mensagem e o *Vector* que contém a segunda camada da mensagem. Quando o objeto `ConsensusFactory` identifica o destinatário, repassa este *Vector* para o método `Rdeliver()` do objeto destino.

O método `Rdeliver()` confere o campo *round*, para ver se a mensagem é válida para aquele consenso, e extrai os campos *myID* e *decision*. O valor de *myID* é usado pelo processo para verificar se não foi ele mesmo que enviou essa mensagem anteriormente. O valor *decision* é transmitido novamente para que algum processo mais lento que perdeu a mensagem *propose* possa também finalizar o consenso. A figura 5.9 mostra a estrutura da mensagem *RBCast*.

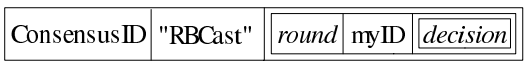

FIGURA 5.9 - Composição da mensagem *RBCast*

5.1.3 Seqüência de operação

A implementação do consenso foi estruturada como uma máquina de estados, onde os eventos de transição são o recebimento de mensagens ou a suspeita do detector de defeitos. Além disso, a implementação permite que as tarefas normais de um

participante e as tarefas de coordenação possam ser executadas de forma independentemente, de modo que um processo participe do consenso e seja o coordenador de uma rodada sem obter nenhuma vantagem ou influência devido à sua função.

A figura 5.10 mostra as duas trilhas de processamento e suas respectivas ações. Para a implementação, a linha de execução principal do objeto (*thread* principal) foi designada para a trilha "Participante", uma vez que ela sempre está ativa. A trilha "Coordenador" é ativada apenas pelo recebimento de mensagens (*estimate* e *ACK/NACK*), de modo que não necessita realizar nenhuma atividade prévia (nem mesmo para saber se é o coordenador da rodada, uma vez que o recebimento das mensagens *estimate* indica isso).

Além da inter-relação entre os estados dos participantes e do coordenador, um participante ainda pode "saltar" estados caso receba mensagens de etapas posteriores. Isso permite que um processo lento não fique demasiadamente atrasado em relação aos demais.

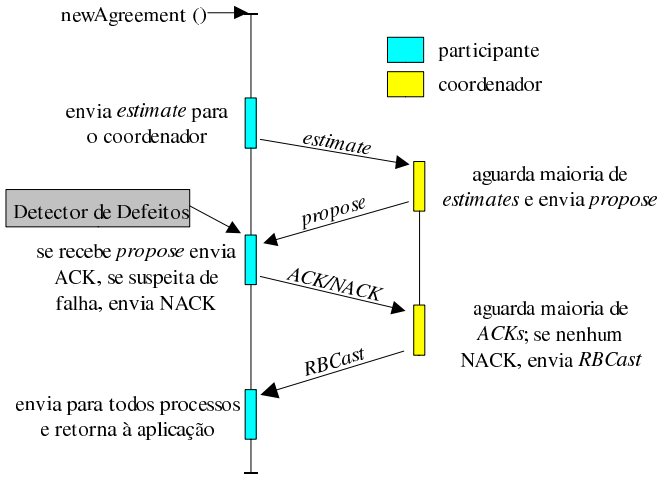

FIGURA 5.10 - Relação entre as atividades do coordenador e dos participantes

5.2 Implementação dos Detectores de Defeitos

5.2.1 Estrutura geral

Este trabalho considerou, para fins de implementação, a estrutura básica de um detector de defeitos utilizada na implementação do detector *Heartbeat*, para o Trabalho Individual [EST 00a]. Esta estrutura, que provou se adaptar bem às características dos diversos modelos de detectores considerados neste trabalho (excetuando-se os detectores *ad-hoc*), apresenta uma construção modular que é facilmente adaptada ao mecanismo interno de cada detector, e cujo diagrama de classes pode ser vista na figura 5.11. De fato, apenas as classes do mecanismo de detecção necessitam ser modificadas para expressar as funcionalidades de cada detector; as demais classes de suporte (comunicação, por exemplo) são comuns a todos os detectores. Esta estrutura também provê uma interface de interação com os demais componentes de uma

ferramenta de comunicação de grupo, de forma que as chamadas e controles mais comuns estão representados nesta interface. Além disso, a utilização de *call-backs* possibilita, juntamente com a interface de interação, abstrair o modelo de detector de defeitos que está implementado, tornando transparente a utilização dos diversos detectores testados.

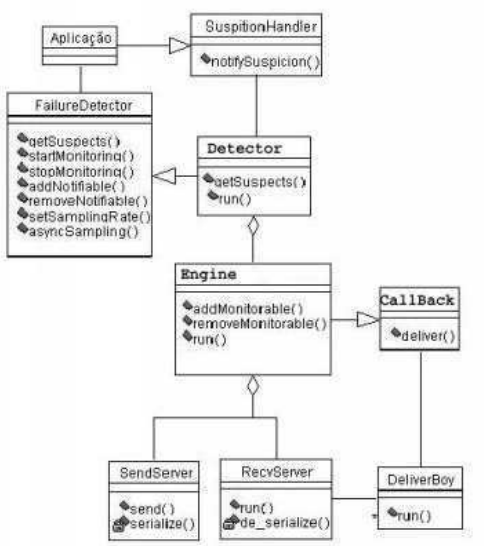

FIGURA 5.11 - Estrutura genérica para os detectores modulares

Como o trabalho visa principalmente uma comparação relativa entre os detectores, e como também não é objetivo deste trabalho otimizar os mecanismos de detecção, todos os algoritmos e mensagens foram construídos com estruturas de alto nível, de forma que a compreensão das implementações seja fácil. Além disso, a utilização de estruturas bem conhecidas permitiu agilizar o desenvolvimento dos detectores, aumentando o número de detectores disponíveis para as avaliações.

A especificação dos detectores de defeitos normalmente indica que cada processo monitorado deve ser julgado separadamente. Isso é necessário pois cada processo trabalha sob situações específicas, então o controle do *timeout* das suas mensagens não pode se misturar ao julgamento dos demais processos.

A maneira mais fiel de transcrever esse comportamento envolve a designação de *threads* para controlar cada processo monitorado. Cada *thread* é responsável pela verificação do *timeout* das mensagens, bem como pela atualização da lista de suspeitos. Entretanto, essa técnica tende a ser mais cara e pode diminuir consideravelmente o desempenho dos detectores, porque o controle da concorrência dessas *threads* obriga a manipulação de estruturas mais complexas.

A solução adotada na construção dos detectores usados neste trabalho mantém a individualidade de tratamento para cada processo, mas reduz o número de *threads* necessárias para a monitoração dos processos. Esta solução envolve a simplificação do mecanismo de controle do *timeout*, e na realidade é a continuação do modelo de julgamento utilizado na implementação do detector *Heartbeat* [EST 00a]. Ao invés de uma *thread* para cada processo controlado, há apenas uma única *thread* que é ajustada

para realizar os julgamentos em períodos pré-determinados, que normalmente correspondem ao *timeout*. Ao ser ativada periodicamente, esta *thread* verifica o momento de recebimento da última mensagem de cada processo. Se a mensagem foi recebida dentro do último *timeout*, isto é indicado quando a diferença entre o instante de recebimento da mensagem e o instante do julgamento é menor do que o intervalo máximo permitido. Se esta diferença for maior do que o *timeout*, nenhuma mensagem nova chegou após o julgamento anterior, de forma que o processo pode ser considerado suspeito.

Essa solução pode ser adaptada para todos os detectores implementados: um detector *Heartbeat*, ao invés de comparar o instante de recebimento da última mensagem, verifica se houve incremento nos contadores. Já os detectores adaptativos, por não utilizarem um *timeout* comum, configuram o intervalo de julgamento mais curto do que a maioria dos *timeout*, e a comparação é feita utilizando os *timeouts* máximos para cada processo.

Com essa estrutura de julgamento, o mecanismo dos detectores de defeitos pôde ser separado em duas camadas: uma responsável pela contabilização das mensagens enviadas e recebidas, e outra camada responsável pelo julgamento dos processos suspeitos (figura 5.12). Enquanto a camada de envio e recepção armazena o instante do recebimento das mensagens (e também do envio, no caso dos detectores tipo *Pull*), a camada de julgamento apenas compara esses valores com relação ao *timeout* global ou individual dos processos. Isso também permite a reutilização de certas estruturas, quando os detectores apresentam funcionalidades semelhantes.

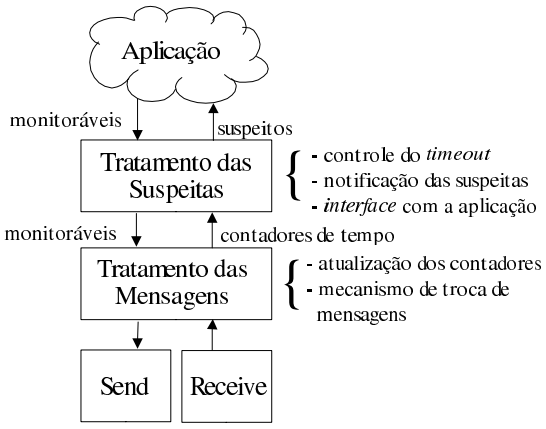

FIGURA 5.12 - Funções das camadas do detector

5.2.2 Implementação de um detector *Push*

Um detector *Push* é basicamente um mecanismo que controla o intervalo de recebimento das mensagens dos demais processos. Este controle é feito a partir de seu relógio interno, o que significa que nenhum detector sabe como os demais detectores vão julgar suas mensagens. Assim, todos os detectores devem ter parâmetros idênticos (ou muito próximos) para o intervalo de envio das mensagens e o *timeout* que será julgado sobre elas. Se um detector apresentar parâmetros muito diferentes, os demais detectores não serão capazes de julgar tal fato, e o detector divergente poderá ser

considerado suspeito caso essa diferença seja muito grande, ou então julgar erroneamente os demais.

Aliada à essa necessidade de "conhecimento mútuo", é visível a necessidade de definir um intervalo de envio inferior ao tempo máximo do *timeout*. Se um detector envia mensagens a intervalos maiores, este irá ser alternadamente inserido e removido da lista de processos suspeitos, pois em um momento suas mensagens não irão chegar, ultrapassando o *timeout*, mas em um segundo instante este engano deverá ser corrigido ao receber as mensagens atrasadas. A melhor forma de controlar esta ligação entre o intervalo de envio e o *timeout* é fazer com que assumam uma relação proporcional, pois do contrário não será possível garantir nem a precisão nem a agilidade da detecção.

Como um detector *Push* necessita estar sempre enviando mensagens, do contrário poderá ser considerado suspeito, é altamente provável que este comportamento induza a uma sobrecarga no sistema devido ao próprio nível de atividade do detector (sem contar a sobrecarga na rede, que uma taxa de comunicação muito freqüente pode acarretar). Um detector *Push* necessita estar ativo, e conhecer o conjunto de processos (detectores) que irão receber suas mensagens.

Na implementação deste detector, o julgamento da lista de suspeitos é realizado a partir da comparação entre um *timestamp* que registra o momento de recebimento da última mensagem de cada objeto monitorado e o momento atual: se este intervalo ultrapassou o limite de tempo pré-estabelecido, o processo é considerado suspeito.

5.2.3 Implementação de um detector *Pull*

Enquanto os detectores *Push* não têm controle sobre o julgamento de suas mensagens pelos demais processos, uma vez que estes baseiam-se em seus relógios locais, os detectores *Pull* são os únicos responsáveis pelo controle do *timeout*, uma vez que este sabe exatamente o momento em que enviou mensagem de interrogação ("*Are you alive?*"), e deve apenas tomar cuidado para não desconsiderar o *roundtrip* (tempo de ida e volta) das mensagens, pois isto resultaria em uma detecção incorreta ocasionada pelos próprios limites da rede. O processo interrogado não precisa estar ativo, pois apenas responde a um evento específico (o recebimento da mensagem "*Are you alive?*"), nem precisa, no momento de responder à pergunta, ter conhecimento sobre todos os detectores do sistema. De fato, quanto trata-se de detectores *Pull*, pode-se diferenciar um detector de um processo do sistema, visto que um detector na verdade é o componente que faz as perguntas e julga as respostas. Um detector inativo, que apenas responde a eventos específicos, pode ser desassociado dos detectores e tratado como apenas um método para a obtenção de informações para o detector. Ainda assim, no contexto deste trabalho, é mais interessante trata-los todos como detectores *Pull* ativos.

A partir destas definições, pode-se caracterizar um detector *Pull* como um detector que tem um papel passivo com relação ao sistema, de tal forma que apenas os detectores que necessitam fornecer listas de suspeitos à aplicação devem saber quais são os elementos a que devem dirigir suas mensagens de interrogação. Isso o diferencia do detector *Push* apresentado anteriormente, que precisa manter sua atividade e necessita conhecer quais são os detectores que esperam suas mensagens.

Como não existe relação direta entre o intervalo de envio e o *timeout*, um detector *Pull* pode funcionar com intervalos de envio longos, e *timeouts* mais curtos. Como

afirmado anteriormente, desde que seja respeitado o tempo de *roundtrip*, a detecção se dará com uma probabilidade mínima de falsas suspeitas, e o intervalo de envio longo evita a sobrecarga da rede com as mensagens de detecção. Não obstante, este cenário traz algumas desvantagens, sobretudo relacionadas à latência da detecção. Assim, para fins de testes, este trabalho considerará cenários onde o intervalo de envio e o *timeout* são mais próximos, de forma a avaliar a sobrecarga que estes induzem no meio de comunicação nos casos onde esta sobrecarga tende a ser mais crítica.

Para a implementação destes detectores, foram utilizadas listas de processos monitorados, onde cada um tem associados dois *timestamps* que registram o instante do último envio da mensagem *"Are you alive?"* e o instante da recepção da última resposta. A cada período de julgamento é avaliada a diferença entre esses instantes. Se o momento da resposta é posterior ao momento do último envio, considera-se que a requisição foi respondida ainda dentro do *timeout*. Em contrapartida, se o instante de recepção for anterior ao momento do envio da última requisição, é necessário verificar se o processo monitorado já não ultrapassou o limite de tempo sem ter fornecido uma resposta.

5.2.4 Implementação dos detectores adaptativos

Os detectores adaptativos foram implementados com base nos detectores *Push* e *Pull* descritos anteriormente. A característica mais marcante destes detectores, independente do modelo de comunicação, é o tratamento personalizado que cada processo monitorado recebe. Quando um detector não recebe a mensagem de um processo dentro do *timeout*, além de inserí-lo na lista de suspeitos, aumenta o *timeout* relativo ao processo suspeito, na tentativa de adaptar-se aos casos onde as mensagens chegam atrasadas.

Quanto à implementação destes detectores, a maior diferença para os detectores *Push* e *Pull* refere-se à manipulação dos registros de cada processo. Estes registros contêm, além dos instantes de envio e recepção, o *timeout* de cada processo. Enquanto a camada de envio e recepção é responsável pelo registro dos instantes de envio e recepção das mensagens, é a camada de julgamento que determina se um processo terá seu *timeout* incrementado. Como a manutenção desses dados em estruturas separadas comprometeria seriamente as tarefas de inclusão e remoção de processos monitorados, optou-se por manter os dados em uma única estrutura, e adicionar um método específico para este incremento. É claro, esta operação retira a independência entre as camadas de julgamento e de envio/recepção, mas por ser aplicada apenas a um grupo especial de detectores, não chega a prejudicar a estrutura geral dos detectores.

5.2.5 Implementação do detector *Heartbeat*

O detector *Heartbeat* utilizado nos testes é similar ao detector implementado em um trabalho anterior [EST 00a]. A única modificação realizada refere-se ao tamanho das mensagens. Enquanto no trabalho anterior a preocupação era permitir que uma mensagem carregasse informações de um número grande de processos, as experiências deste trabalho objetivam um conjunto bem definido de processos. Assim, ao invés de mensagens de 1024 bytes, que poderiam conter aproximadamente 30 identificadores (segundo projeções contidas naquela monografia), reduziu-se o tamanho das mensagens para 128 bytes, suficientes para conter a identificação dos cinco processos da

experiência. Isso permitiu também uma melhor parametrização do desempenho dos detectores, uma vez que o tamanho das mensagens influencia muito o desempenho da comunicação.

O funcionamento do detector *Heartbeat* considera basicamente a verificação do incremento dos contadores de mensagens recebidas. Essa comparação é realizada através do uso de um intervalo de amostragem, que basicamente funciona como um *timeout*, onde são comparados os valores dos contadores, em dois instantes de tempo diferentes: se não houver incremento do contador de um processo, este será considerado falho.

5.2.6 Implementação do detector *ad-hoc "no message"*

A implementação do detector *ad-hoc "no message"* consiste basicamente na inclusão de um temporizador junto ao algoritmo de consenso, entre o envio da mensagem *estimate* e a recepção da mensagem *propose*. Se *propose* não chegar antes de ser atingido o *timeout*, o coordenador é considerado suspeito. Por outro lado, se a mensagem chegar enquanto o temporizador está em espera, este é anulado, e o processamento é concluído imediatamente, evitando assim uma espera desnecessária.

5.2.7 Implementação do detector *ad-hoc "heart-beat"*

O detector *ad-hoc "heart-beat"* foi implementado utilizando como base o detector *ad-hoc "no message"*. por outro lado, enquanto este detector não gerava mensagens de detecção, o *ad-hoc "heart-beat"* necessita uma mensagem específica para indicar que o coordenador está ativo. Na implementação realizada, a escolha de projeto foi de inserir esta mensagem de detecção no conjunto de mensagens controladas pelo objeto `ConsensusFactory`, de forma que a sua recepção se dá através dos mesmos mecanismos utilizados para as mensagens específicas do consenso. Isso aumenta a transparência da detecção (só é preciso conhecer a identidade do módulo de consenso), sendo condizente com a filosofia desta proposta. No entanto, como as mensagens de detecção utilizam os mesmos canais de processamento das mensagens do consenso, pode ocorrer a perda de desempenho do detector, pois gera uma disputa de recursos com o módulo de consenso.

Quando recebe mensagens *estimate*, o coordenador designa uma *thread* (objeto `SendThread`) para periodicamente enviar uma mensagem do tipo *"I am alive"*. O objeto `ConsensusFactory`, ao receber essas mensagens, redireciona-as para o método `heartbeat()` do objeto `Consensus`, que suspende o processo de suspeita até o próximo *timeout*. O coordenador cessa o envio destas mensagens apenas quando envia seu *propose*. Os campos da mensagem *heartbeat* são descritos na figura 5.13.

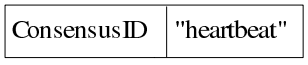

FIGURA 5.13 - Mensagem heartbeat do *ad-hoc "heart-beat"*

5.3 Aplicação de Testes

A aplicação de testes para o algoritmo de consenso foi construída com o objetivo de realizar diversas iterações do consenso e para isso, procurou deter-se apenas neste problema, uma vez que quaisquer funcionalidades adicionais poderiam prejudicar a avaliação das métricas deste trabalho. Por isso, foram utilizadas listas fixas para o controle dos membros do consenso, uma vez que não é função da aplicação de testes a modificação dinâmica destas propriedades.

Como objeto de acordo, foi utilizada um conjunto de frases, simulando participantes de um leilão. Assim, o primeiro processo envia como *estimate* a mensagem "Dou-lhe uma!", o segundo processo envia "Dou-lhe duas!", e assim sucessivamente. Após o consenso, cada processo exibe a sua sugestão e a decisão do acordo, para que possa ser feita a validação da saída dos processos.

Na figura 5.14, encontra-se o código-fonte básico da aplicação, sendo que apenas mudanças menores são realizadas para adaptar o uso de cada detector (especialmente no caso dos *ad-hoc*, onde não é necessária a inicialização do consenso).

```
public class leilao extends Thread {
  public FailureDetector FD = null;
  public ConsensusFactory CF = null;

  public leilao(String args[])  {
  // reads command-line parameters
  }

  public void initFD()  {
  // create a new FD object, and set the initial parameters for the Failure Detector
  }

  public void run()  {
    initFD();
    CF = new ConsensusFactory (FD,4023);
    Member votantes[] = new Member[5];
    for (int i=0; i< machineList.size(); ++i)
   {
      votantes[i]=new Member((machineId)machineList.elementAt(i));
    }
    Vector vet = new Vector();
    Vector vet2 = new Vector();
    Consensus leilao = CF.newConsensus();
    Vector vet3 = new Vector();
    vet3.addElement(new String ("dou-lhe tres!"));
    for (int i=0; i<200; ++i) {
      vet2 = null;
      vet2 = leilao.StartAgreement(votantes,3,vet3);
      System.out.println("[" + (String)(vet3.elementAt(0)) + "," + (String)(vet2.elementAt(0)) + "]");
   }
    leilao.finalize();
    System.exit(0);
  }
}
```

FIGURA 5.14 - Código-fonte da aplicação de testes

5.4 Alterações para a Experimentação

Enquanto os algoritmos implementados dos detectores e do consenso são funcionalmente completos, há a necessidade de adaptá-los para as experimentações realizadas, bem como para a tomada das medidas. As modificações mais intrusivas

foram realizadas durante a adaptação do algoritmo de consenso para as situações de testes *best case* e *worst case*, enquanto a tomada das medidas exigiu modificações bem menores e com menor impacto sobre o algoritmo original. A situação *normal case* não precisou de nenhuma alteração, pois é um exemplo de funcionamento habitual nos sistemas distribuídos.

5.4.1 Adaptação para a situação *best case*

Para simular a situação *best case* nas diversas iterações do algoritmo de testes, optou-se por fazer o consenso ignorar a saída dos detectores de defeitos. A outra opção possível, paralisar os detectores, não é desejável pois subtrai exatamente a carga de processamento que os detectores impõem ao algoritmo de consenso, quando em operação.

- **Detectores modulares**: a adaptação foi realizada no método `notifySuspicion()`, e consistiu basicamente em ignorar possíveis suspeitas que os detectores notificassem.
- **Detector *ad-hoc "no message"***: a adaptação neste detector foi sobre o temporizador que controla o *timeout* do coordenador. Se o coordenador não responder dentro do período previsto, não há notificação do defeito, apenas o temporizador é reiniciado até que o coordenador responda.
- **Detector *ad-hoc "heart-beat"***: a adaptação ocorreu sobre a saída do método `heartbeat()`, de forma que a ausência de mensagens *heartbeat* dentro do intervalo de espera não dispare a suspeita do coordenador.

5.4.2 Adaptação para a situação *worst case*

Para simular a situação *worst case* nas diversas iterações do algoritmo de testes, a primeira providência foi fazer com que o coordenador da primeira rodada parasse de responder. Como induzir o processo do coordenador a um colapso em uma aplicação com diversas iterações seria uma tarefa muito cara e demorada, optou-se por apenas simular esse colapso. Isto foi feito através da paralisação das atividades do coordenador e de seu detector, de forma que os demais detectores não pudessem mais obter indicações de seu funcionamento. Como o consenso neste caso deve ser resolvido na segunda rodada, a identificação do momento certo para falhar pode ser feita através de uma simples relação par/ímpar, obtida através da expressão *round mod 2*, que retorna o resto da divisão inteira do identificador da rodada por 2. Assim, quando o coordenador de uma rodada identifica que deve ser desabilitado, este paralisa seu detector e não envia mensagens para os demais processos (esse controle é feito dentro do método `GetEstimates()`). Quando na próxima rodada este processo receber a difusão confiável que contém a decisão do consenso, ativa novamente o seu detector e participa dos próximos consensos.

- **Detectores modulares**: além do método `GetEstimates()`, a adaptação foi realizada no método `notifySuspicion()`. Nele é feita a restrição sobre as suspeitas incorretas, impedindo que o coordenador da segunda rodada também seja suspeito. Se fosse permitido que ocorressem suspeitas em todas

as rodadas, haveria a possibilidade de prolongar o consenso por várias rodadas, impedindo a análise dos resultados coletados.

- **Detector *ad-hoc "no message"***: aqui o controle sobre as detecções nas rodadas pares foi feito sobre o temporizador que controla a resposta do coordenador.
- **Detector *ad-hoc "heart-beat"***: a adaptação ocorreu sobre a saída do método `heartbeat()`, impedindo que seja levantada uma suspeita sobre os coordenadores das rodadas pares.

5.4.3 Obtenção das medidas

A aquisição das métricas utilizadas neste trabalho para a comparação dos detectores foi feita tanto no nível do algoritmo quanto do sistema operacional.

A métrica *terminação do consenso* foi obtida a partir do intervalo de tempo que uma invocação do consenso levava para retornar o resultado à aplicação. Para isso, foram inseridos dois procedimentos, um no início da operação de consenso e outro junto ao fim da operação, sendo que cada um registra em uma variável o instante de sua ativação com uma precisão de milissegundos, obtido através do método `System.currentTimeMillis()` da máquina virtual Java. Quando o consenso era finalizado, o algoritmo registrava a diferença entre essas variáveis em um arquivo, através das classes de *log* do pacote *BW*[9], para posterior análise.

Já as métricas *tempo de CPU* e *quantidade de memória* foram obtidas através de uma ferramenta externa, que analisa os dados do sistema operacional. A ferramenta *top* emite como saída uma listagem periodicamente atualizada, que contém diversos parâmetros, entre eles o valor acumulado do tempo de CPU utilizado pela aplicação e a quantidade de memória ocupada naquele instante. Essa ferramenta é inicializada em cada máquina junto com a aplicação de testes, e tem sua saída filtrada e armazenada em um arquivo através de um *script* de *shell*.

A obtenção dos valores utilizados na comparação entre os detectores foi feita através da média dos valores registrados nos cinco processos: o tempo de terminação do consenso e o tempo de CPU são divididos pelo número de iterações do algoritmo, enquanto a quantidade de memória utilizada é obtida através da média dos valores registrados durante a execução da aplicação de testes.

5.5 Considerações Finais

Neste capítulo foram tratados os aspectos da implementação dos algoritmos, indo desde o protocolo de mensagens do consenso ao modelo de *threads* utilizado pelos detectores. Além disso, foi apresentada a aplicação de testes utilizada para a captura dos dados, bem como as técnicas utilizadas para simular as situações de testes consideradas.

No próximo capítulo será feita a análise dos resultados obtidos com cada modelo de detector de defeitos utilizado, para que em um segundo momento possam ser comparados. Ao fim do próximo capítulo será feita a confrontação entre as observações realizadas por Sergent *et al.* e as obtidas neste trabalho, sendo analisadas as diferenças e semelhanças destas observações.

---

9 desenvolvidas por Manuele Kirsch Pinheiro em sua dissertação de mestrado [PIN 01]

6 AVALIAÇÃO DOS RESULTADOS

No trabalho de Sergent *et al.* [SER 99] a análise dos resultados foi voltada à definição dos pontos ótimos de desempenho dos algoritmos, tendo assim uma profunda relação com os valores absolutos de tempo necessários para a terminação do consenso. Diversos fatores, no entanto, impedem que os resultados desta experiência possam ser livremente associados aos daquele trabalho. As opções de implementação e coleta dos resultados, como por exemplo o uso da linguagem Java e as situações de testes diferenciadas, tornam praticamente impossível comparar os trabalhos baseando-se apenas nos números obtidos. É claro, as diferenças significativas encontradas entre os dois trabalhos podem levar à análise da influência do ambiente sobre a experiência (Sergent *et al.* não consideraram o sistema operacional nas simulações), mas sempre é possível atribuir essa diferença às ferramentas e técnicas que foram utilizadas na implementação e nos testes.

Dessa forma, a melhor maneira de fazer uma análise dos resultados é através da avaliação do comportamento do consenso nos ambientes de teste quando se variam os parâmetros de entrada, uma vez que cada parâmetro utilizado induz diferentes cargas de processamento e diferentes exigências de eficiência para os detectores.

Neste capítulo serão analisados os resultados obtidos com cada detector, e em seguida estes serão comparados entre si e em relação ao trabalho de Sergent *et al*.

6.1 Análise dos Detectores

Nesta seção serão avaliados individualmente os resultados do consenso utilizando cada modelo de detector de defeitos, possibilitando a identificação de seu comportamento básico, de seus pontos de melhor desempenho e da carga que o detector representa junto ao sistema.

Os detectores serão avaliados segundo seu desempenho nas três situações de testes consideradas neste trabalho - melhor caso, pior caso e uso normal. Também será feito um acompanhamento dos indicadores auxiliares (tempo de CPU e quantidade de memória), a fim de validar ou complementar as observações.

6.1.1 Detector *Push*

O detector *Push* tem por característica o envio periódico de mensagens conhecidas como *heartbeat* ou "*I am alive!*". Tendo por característica o envio de mensagens para todos os processos, este modelo retira um pouco da transparência de operação, pois o processo que envia as mensagens deve obrigatoriamente ter algum conhecimento da identidade dos receptores. Além disso, como é o processo receptor que determina o *timeout* que será aplicado sobre as mensagens recebidas, há a necessidade de configurar bem tanto o emissor quanto o receptor, para não distorcer demais a relação entre o intervalo de envio e o *timeout*. Esta necessidade obriga também a operação ininterrupta dos detectores.

Entre as vantagens do detector *Push* está o tamanho reduzido das mensagens, que precisam apenas carregar o identificador do emissor (e dependendo da implementação, nem isto é necessário, pois estas informações podem ser obtidas junto ao protocolo de transporte). Além do custo menor de envio, devido ao tamanho das mensagens, há também um custo menor de processamento, uma vez que o envio e a detecção são atividades independentes. Isso aumenta a agilidade da detecção, pois a recepção das informações depende exclusivamente do tempo mínimo que a rede leva para entregar as mensagens.

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

As experiências conduzidas neste trabalho utilizaram dois valores para essa relação, $\Delta i = 98\% \ \Delta to$ e $\Delta i = 75\% \ \Delta to$. Isso significa que para um *timeout* fixo, foram testados intervalos de envio equivalentes a 98% e a 75% deste *timeout*. Como o intervalo de envio varia entre esses dois casos, também o desempenho é afetado, uma vez que uma taxa de envio mais alta tende a aumentar o tráfego na rede e a carga de processamento no sistema.

A figura 6.1 mostra o tempo de terminação do consenso no melhor caso (*best case*) quando se varia o *timeout* com relações de 98% e de 75%.

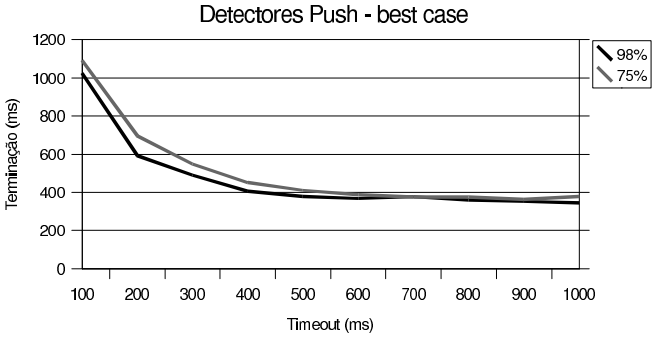

FIGURA 6.1 - Detectores *Push*: comparação *best case*

Na situação de testes *best case* é possível visualizar o impacto que o detector de defeitos tem sobre o desempenho do consenso. Isso é possível porque nesta situação de testes o detector não faz nenhuma suspeita, então o desempenho do consenso é influenciado apenas pela sobrecarga da troca de mensagens. Conforme observa-se na figura 6.1, não há quase diferença no impacto das diferentes relações de envio, mas quando o *timeout* for menor do que 300 ms o tempo de terminação do consenso aumenta, devido à grande quantidade de mensagens que precisam ser processadas e que estão trafegando na rede.

Na figura 6.2 são mostradas as curvas obtidas para essas mesmas relações de envio quando se considera o pior caso (*worst case*), ou seja, quando o coordenador da

primeira rodada falha, e este evento deve ser detectado para que o consenso seja resolvido na próxima rodada. Enquanto na situação *best case* era avaliada a sobrecarga de processamento que o detector causava, nesta situação é possível identificar a latência da detecção, pois quanto mais rápida for identificada a falha do coordenador (que tem uma relação direta com o *timeout*), mais cedo será possível passar para a próxima rodada e terminar o consenso em menos tempo.

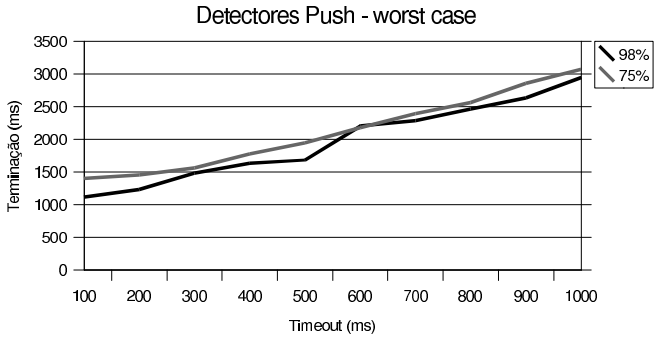

FIGURA 6.2 - Detectores *Push*: comparação *worst case*

Como previsto, quanto menor for o *timeout* (e mais rápido o envio das mensagens) melhor será o desempenho do consenso, pois o detector conseguirá detectar a falha do coordenador mais rapidamente. Aqui também se repete a pequena diferença entre as duas curvas, sendo que o detector com a relação de 75% apresenta índices levemente superiores devido sobretudo ao acréscimo de carga no sistema que o envio mais intenso de mensagens ocasiona.

A figura 6.3 apresenta o comportamento do detector em uma situação normal, onde apesar de não ocorrerem falhas, os detectores podem cometer enganos, e induzirem a novas rodadas do consenso.

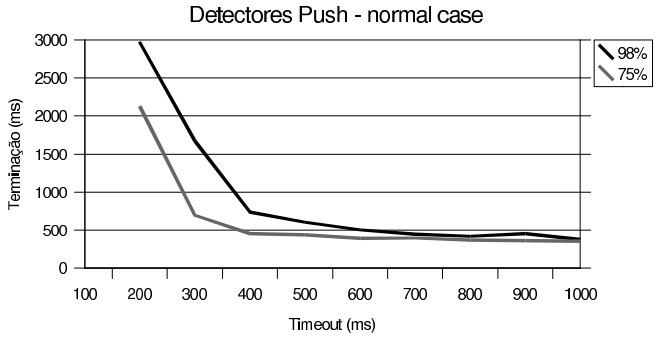

FIGURA 6.3 - Detectores *Push*: comparação *normal case*

A primeira observação interessante é a de que o desempenho da relação 98% é pior do que a relação 75%. De fato, quanto mais próximo é o *timeout* do tempo de envio, aumenta a chance de que as mensagens cheguem após o limite de tempo. Assim, os detectores fazem mais suspeitas incorretas, e o consenso tende a se estender por mais rodadas.

Não obstante, ambas curvas sofrem com o problema da taxa de envio muito elevada. Quando muitas mensagens são enviadas em pouco tempo, aumenta a sobrecarga do sistema e os detectores não conseguem processar todas com eficiência.

De fato, a importância desta sobrecarga do sistema é melhor observada na figura 6.4, onde são confrontadas as três situações de testes (*best case* – ***bc***, *worst case* - ***wc*** e *normal case* - ***nc***) para cada relação de envio/*timeout*.

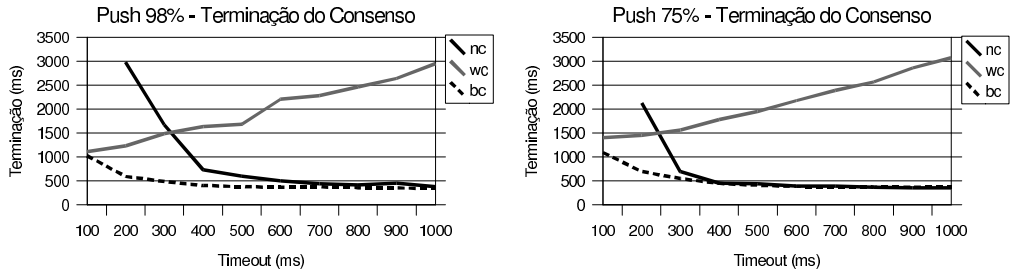

FIGURA 6.4 - Comparação das situações para os detectores *Push*

Enquanto a comparação entre as situações *best case* e *worst case* demonstra que os melhores parâmetros para conseguir uma latência menor na detecção são opostos aos parâmetros que visam reduzir a sobrecarga gerada pelos detectores, deve-se prestar especial atenção sobre o comportamento da situação *normal case*. Pode-se observar que para valores maiores de *timeout*, o desempenho médio da situação normal é quase justaposto à curva da situação *best case*, o que indica que na maioria das operações realizadas foi possível finalizar o consenso em apenas uma rodada. Entretanto, quando se diminui o *timeout*, a curva *normal case* aproxima-se dos valores do *worst case*, chegando até a ultrapassá-los. Quando a curva *normal case* ultrapassa a curva do *worst case* (em torno dos 300 ms para a relação 98% e dos 250 ms para a relação 75%) as operações de consenso levam mais de duas rodadas, em média (é possível afirmar isto porque o *worst case* sempre leva duas rodadas, um da suspeita de falha e outro para terminar).

A análise das métricas auxiliares também ajuda na verificação destas propriedades. A primeira destas métricas, o tempo de CPU, é muito importante porque indica o nível real de atividade dos algoritmos. Um processo que está bloqueado tem um tempo de finalização alto, mas seu consumo de CPU é bem menor. Isto ocorre porque o sistema operacional, ao dar o controle para o processo, verifica que este ainda não conseguiu sair da situação de bloqueio, então retira o controle deste processo e retorna em outro momento. Esta métrica também é eficaz para demonstrar o impacto global do detector, uma vez que são computados todos os esforços computacionais, como o tratamento de mensagens, o processamento das suspeitas, etc.

A figura 6.5 mostra o tempo de CPU utilizado por cada configuração do detector *Push*, quando expostas às três situações de teste.

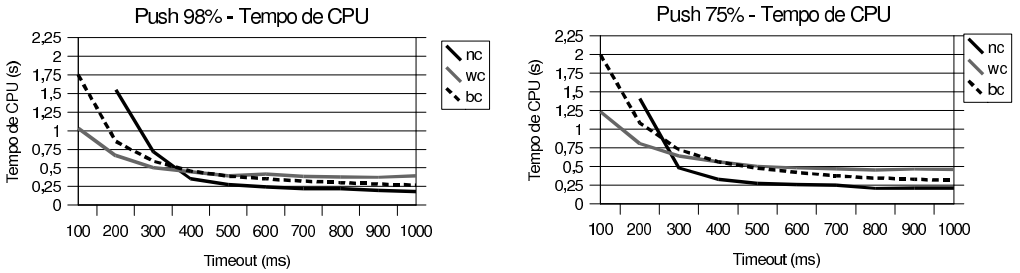

FIGURA 6.5 - Comparação do tempo de CPU dos detectores *Push*

Em ambas figuras, a situação *best case* segue a mesma tendência já observada para o tempo de terminação do consenso. Isso se deve ao grau de atividade elevado (afinal, se não há nenhuma suspeita, todo processamento ocorre no menor tempo possível, sem interrupções). Já a situação *worst case* apresenta uma curva diferente da observada anteriormente. A explicação para esse comportamento é de que, na situação *worst case,* os algoritmos são controlados pelo máximo *timeout* possível, pois só conseguem finalizar o consenso quando o *timeout* é ultrapassado, e a suspeita levantada. No gráfico do tempo de CPU o tempo transcorrido até atingir o *timeout* não é computado, pois o detector não está fazendo nenhum processamento. Isso também determina as diferenças entre as curvas *worst case* e *best case*, pois no *worst case* o consenso fica bloqueado boa parte do tempo, enquanto na situação *best case* ele está operando juntamente com o detector, o que aumenta a sobrecarga.

O comportamento da curva *normal case* segue o da curva do tempo de terminação do consenso, ou seja, para valores de *timeout* mais elevados o custo é baixo, uma vez que quase todas as operações conseguem finalizar em uma única rodada. Entretanto, abaixo de um certo valor de *timeout* (350 ms para a relação 98% e 300 ms para a relação 75%) o custo das múltiplas rodadas aumenta muito o tempo de CPU utilizado pelos algoritmos.

A última métrica a ser considerada é a da quantidade de memória ocupada pelo processos. A figura 6.6 mostra a quantidade média de memória utilizada pelos algoritmos (em % da memória RAM disponível em cada máquina). Para ambas relações de envio/*timeout* o comportamento das situações *best case* e *worst case* foi semelhante: como estas situações enfocam propriedades características (sobrecarga e latência da detecção), pode-se dizer que a pequena e constante diferença entre elas deve-se ao processamento extra que o *worst case* precisa fazer para tratar a detecção. O comportamento das curvas expressa o acúmulo de mensagens nos *buffers* e a necessidade de tratar mais mensagens devido à maior taxa de envio.

O comportamento da situação *normal case* não segue essa tendência, ao menos em boa parte da faixa de testes. Provavelmente, o custo das suspeitas incorretas e do tratamento destas, mesmo quando são raras, obriga os algoritmos a manterem uma ocupação mais constante da memória. Apenas quando a sobrecarga causada pela maior taxa de mensagens torna-se muito elevada, aumentando as suspeitas incorretas, é que o detector e o consenso consomem mais memória para realizar suas tarefas.

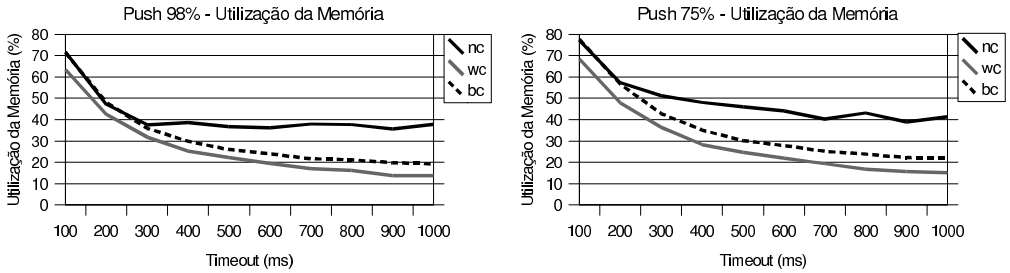

FIGURA 6.6 - Detectores Push: utilização da memória

6.1.2 Detector *Pull*

O detector *Pull* caracteriza-se por trabalhar através de requisições, ou seja, o processo de suspeita inicia quando o detector envia uma mensagem questionando um determinado processo. Se a resposta deste processo não chegar dentro do intervalo de tempo esperado, é iniciada a suspeita. Dessa forma, o tempo mínimo que o detector deve esperar compreende o tempo que a mensagem levará para ir, ser processada e voltar (esse tempo é comumente chamado de *roundtrip*). Se a aplicação exige uma detecção agressiva (*timeout* reduzido), o *roundtrip* representa uma limitação real para a aplicação.

Entre as vantagens do detector *Pull* está a transparência de operação, ou seja, um processo monitorado não precisa saber a identidade dos detectores que o monitoram, pois isso pode ser obtido através da mensagem de interrogação. Além disso, como cada detector é responsável pelo controle do *timeout*, seus parâmetros de *timeout* poderiam ser diferenciados, de acordo com as características de cada máquina. Uma outra possibilidade dos detectores *Pull* é que estes podem estabelecer intervalos entre os envios muito maiores do que o *timeout*, ou seja, os detectores podem fazer suas perguntas apenas quando precisam atualizar suas detecções.

Enquanto o detector *Push* necessita um intervalo de envio menor do que o *timeout*, o detector *Pull* não tem essa restrição. Na prática, podem ser usados tanto intervalos de envio menores quanto maiores. O uso de grandes intervalos de envio, ou seja, detecções bem esparsas, leva à diminuição da sobrecarga do sistema, mas faz com aumente muito o tempo de atualização de um detector. Já o uso de requisições demasiadamente próximas aumenta a sobrecarga do sistema e da rede.

Neste trabalho foram considerados intervalos de envio com valores próximos ao do *timeout*, para avaliar o impacto dos detectores em situações de sobrecarga. Essa escolha é justificada pelo fato de que relações muito grandes poderiam distorcer o comportamento tradicional dos detectores (que tenta equilibrar uma menor latência de detecção com uma boa precisão). Assim foram consideradas relações proporcionais entre os *timeouts* e os intervalos de envio, reduzindo o conjunto de testes (na seção 4.3.3 foram discutidos os motivos dessa escolha). As experiências conduzidas neste trabalho utilizaram dois valores para essa relação, $\Delta i = 150\% \Delta to$ e $\Delta i = 100\% \Delta to$. Com isso, é possível verificar como o comportamento do detector sob uso intenso e o desempenho são influenciados pela escolha do intervalo de envio.

A figura 6.7 mostra o comportamento do consenso na situação *best case*, onde pode ser avaliada a sobrecarga que os detectores impõem ao sistema, e por conseguinte, ao consenso.

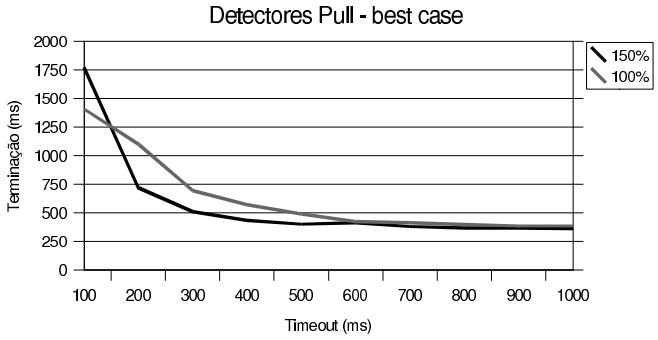

FIGURA 6.7 - Detectores *Pull*: terminação *best case*

Inicialmente, pode-se observar que nas situações onde a taxa de envio é mais alta, quanto maior for o intervalo entre os envios, melhor será o desempenho do consenso: isso pode ter conseqüência direta do menor número de mensagens a processar. Entretanto, o comportamento com *timeouts* abaixo de 300 ms mostrou uma modificação nesta tendência. Nestas situações, pesa especialmente a própria limitação imposta pelo *roundtrip* das mensagens: quando o *timeout* é muito pequeno, aumenta a probabilidade de que as respostas não cheguem a tempo. Como até mesmo a situação *best case*, que ignora as suspeitas do detector, está sujeita à carga de processamento dos detectores, quando ocorrem falsas suspeitas o detector ativa seus procedimentos de notificação. No caso do detector com a relação 100%, a probabilidade de manter mensagens nos *buffers* de recepção é maior do que a relação 150% proporciona. Por isso, o detector com a relação 150% apresenta um aumento mais significativo quando exposto a *timeouts* agressivos.

A figura 6.8 mostra o desempenho do detector na situação *worst case*. Nela pode-se observar que apesar de algumas variações (provavelmente, induzidas por variações no ambiente de testes), o comportamento do detector com ambas relações foi muito similar, indicando que a diferença entre as relações 100% e 150% não chega a influenciar muito no tempo de reação do detector, quando ocorre a falha do coordenador.

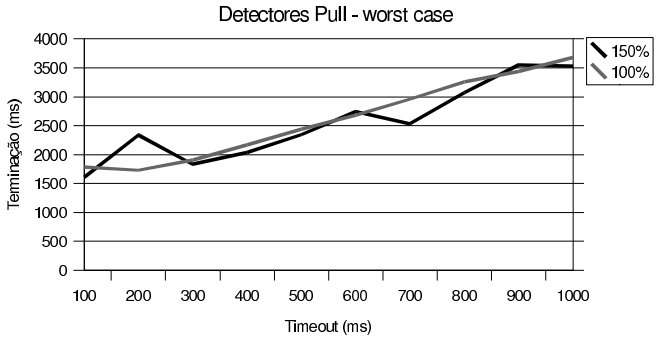

FIGURA 6.8 - Detector *Pull*: terminação *worst case*

Na figura 6.9, que representa o desempenho do detector na situação *normal case*, é possível observar claramente a limitação imposta pelo *roundtrip* das mensagens. Como o *roundtrip* médio é de aproximadamente 133 ms, quando utiliza-se *timeouts*

inferiores a esse valor as mensagens não conseguem chegar a tempo, então o consenso se estende por muitas rodadas antes de terminar.

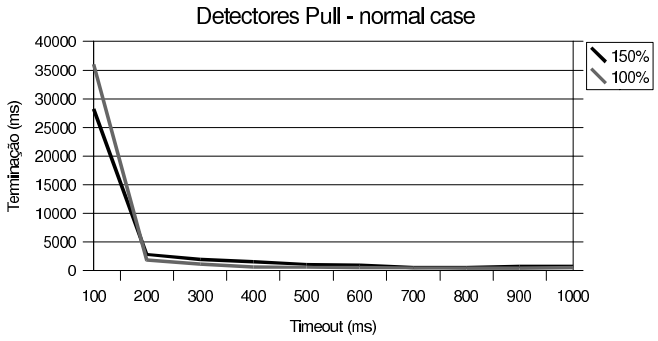

FIGURA 6.9 - Detectores *Pull*: terminação *normal case*

A figura 6.10 compara o desempenho dos detectores nas diversas situações de teste. Pode-se observar inicialmente que o comportamento do detector com a relação 150% tende a ser menos eficiente no *normal case*, apesar da menor sobrecarga que este deveria ter. De fato, essa diferença de aproximadamente um segundo é muito representativa, uma vez que as situações hipotéticas indicavam um comportamento semelhante. Como a situação *normal case* permite falsas suspeitas, essa diferença pode ser causada principalmente pela "demora" do detector a 150% em refazer sua visão, o que leva o consenso a se estender por mais rodadas. Outra possibilidade é a de que, ao se utilizar um *timeout* pequeno, é mais vantajoso ter mais mensagens sendo recebidas do que um intervalo grande entre as detecções: quanto mais mensagens um processo envia, maior é a chance de que nunca faltem mensagens para serem processadas pelo detector.

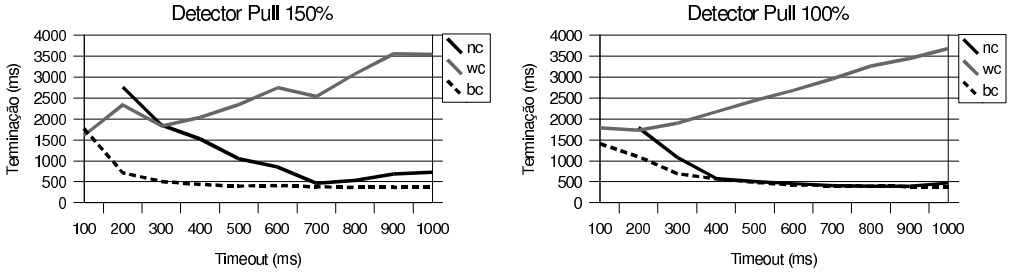

FIGURA 6.10 - Comparação das situações para os detectores *Pull* testados

A figura 6.11 mostra o tempo de CPU utilizado pelos detectores. Ela demonstra que a relação 150% tende a gerar uma menor sobrecarga sobre o sistema na situação *best case* e *worst case*. Por outro lado, seu comportamento na situação *normal case* abandona o comportamento do *best case* mais cedo do que a relação 100%, consumindo mais tempo de CPU, possivelmente devido a um maior número de suspeitas incorretas.

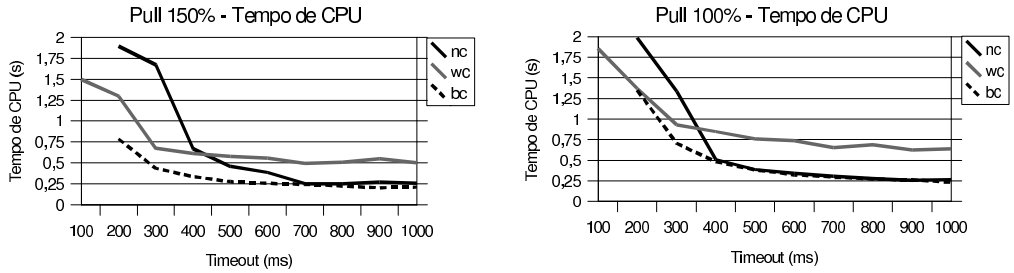

FIGURA 6.11 – Tempo de CPU

Quando utilizado um *timeout* inferior ao *roundtrip* médio das mensagens, o consenso não consegue ser finalizado na situação *normal case*. Até mesmo na situação *best case,* essa limitação teve impacto, pois apesar de serem ignoradas as suspeitas, todo o mecanismo de controle de *timeout* estava ativo, e o caráter agressivo do *timeout* e do intervalo de envio tornaram extremamente elevada a sobrecarga e o consumo de CPU neste caso. Apenas a situação *worst case* foi beneficiada pela impossibilidade da recepção das mensagens dentro do *timeout*, pois com isso o processo de suspeita do coordenador foi disparado mais cedo, mesmo que não tivesse transcorrido tempo suficiente para uma suspeita.

Na figura 6.12 é exibido o consumo de memória dos detectores. De forma geral, o detector com a relação 150% tem um consumo de memória menor, devido à redução do número de mensagens enviadas. Observa-se também que, ao contrário do detector *Push*, o consumo de memória da situação *best case* é a maior. A situação *worst case* é beneficiada pelo atraso natural das mensagens, que permite um processamento sem acúmulos de mensagens nos *buffers*. A situação *normal case* também é influenciada por esse fator, sendo que seu consumo de memória é maior quando ocorrem menos suspeitas incorretas: isso permite aos algoritmos de detecção e do consenso agirem com maior fluidez, sem situações de bloqueio e espera.

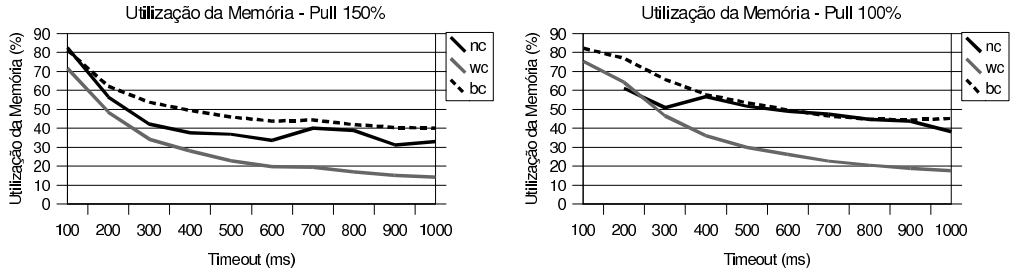

FIGURA 6.12 - Detectores *Pull*: utilização da memória

6.1.3 Detector *Adaptive Push*

Os detectores adaptativos utilizam uma técnica para compensar o fator de incerteza que existe na determinação do melhor conjunto de parâmetros para os detectores. Embora os resultados possam apresentar um comportamento semelhante em uma operação normal dos detectores, nem sempre os mesmos parâmetros podem ser repetidos nos experimentos, quando os detectores trabalham com diferentes aplicações e ambientes. É justamente em cima da diversidade de situações que os detectores

adaptativos são mais úteis. Ao invés de determinar parâmetros rígidos de operação, estes detectores permitem a manutenção dinâmica do *timeout* aplicado sobre as mensagens, podendo corrigir eventuais distorções originadas na transmissão destas. Essa adaptação dinâmica busca sempre o aumento da precisão dos detectores, uma vez que detecções incorretas têm um impacto muito negativo sobre o desempenho do consenso.

Como a definição destes parâmetros exige uma adaptação gradual, a fim de aumentar a precisão do detector sem no entanto onerar exageradamente a latência da detecção, os detectores adaptativos necessitam operar ininterruptamente. Isso significa que, talvez mais do que os outros modelos de detectores, os detectores adaptativos não podem parar e recomeçar a detecção freqüentemente, uma vez que demoram um certo tempo para encontrar a relação adequada de parâmetros. De fato, este funcionamento é o oposto do que ocorre com os detectores *ad-hoc*, que são invocados apenas quando o consenso necessita.

A avaliação dos detectores adaptativos considerou um valor de *timeout* inicial igual a 100 ms, sendo que cada incremento é de 50 ms. Variando-se o intervalo de envio, o detector deve primeiramente adaptar-se à situação: isto pode aumentar um pouco a média do tempo de finalização do consenso, mas permite que o detector tenha toda flexibilidade para se adaptar às variações que a transmissão das mensagens pode conduzir.

Na figura 6.13 pode-se observar o comportamento do detector *Adaptive Push* nas três situações de testes. Os resultados mostram que para a maior parte do experimento o detector conseguiu manter o desempenho do consenso estável, ou seja, o detector conseguiu adaptar-se a toda faixa do intervalo de envio, garantindo ao consenso um serviço eficiente. Apenas nas situações de maior sobrecarga esse desempenho diminuiu, provavelmente mais por causa desta sobrecarga do que pela incapacidade de adaptação do detector.

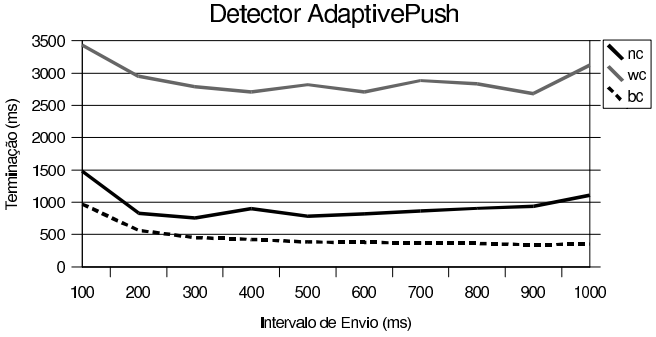

FIGURA 6.13 - Comparação das situações para um *Adaptive Push*

A maior diferença do comportamento deste detector com relação aos detectores comuns analisados anteriormente está na curva que representa a situação *normal case*. Ao invés de acompanhar a curva do *best case* pela maior parte do experimento, como faziam os outros detectores, a situação *normal case* coloca-se um pouco mais acima. Como um detector adaptativo comete algumas suspeitas incorretas até alcançar os parâmetros adequados, na sua curva de desempenho é representada a média dos tempos de toda a detecção, ou seja, são também incluídas as suspeitas errôneas.

A contrapartida desta adaptação é que na situação *worst case,* o detector acaba incrementando o *timeout* a cada falha do coordenador (e como são realizadas diversas operações, o *timeout* atinge valores bem altos). Assim, o detector acaba perdendo agilidade, retardando a finalização dos consensos nesta situação de falhas.

Já na figura 6.14 é possível identificar a curva de consumo de CPU das situações de teste. Pode-se observar que a diferença entre as situações *normal case* e *best case* é bem menos destacada do que no gráfico do tempo de terminação do consenso. Isso se explica principalmente pelo fato de que as detecções incorretas produzidas pelo detector, e que afetam diretamente o consenso, não são contabilizadas por essa métrica, uma vez que os processos ficam bloqueados enquanto não inicia a próxima rodada.

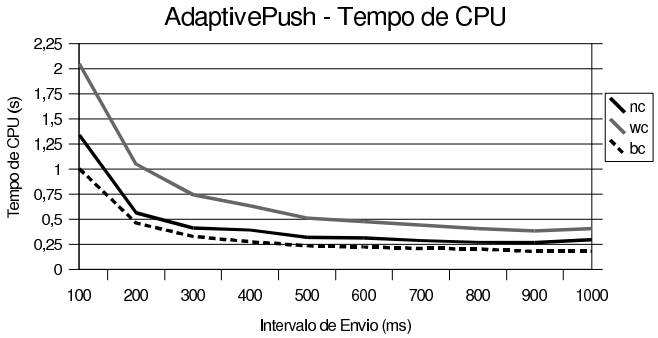

FIGURA 6.14 - Detector *Adaptive Push*: tempo de CPU

O consumo de memória deste detector pode ser visualizado na figura 6.15. Nela pode ser verificado que o consumo de memória também segue um comportamento bem definido, e apenas aumenta a sobrecarga quando o envio de mensagens é muito agressivo, devido ao consumo de recursos do sistema.

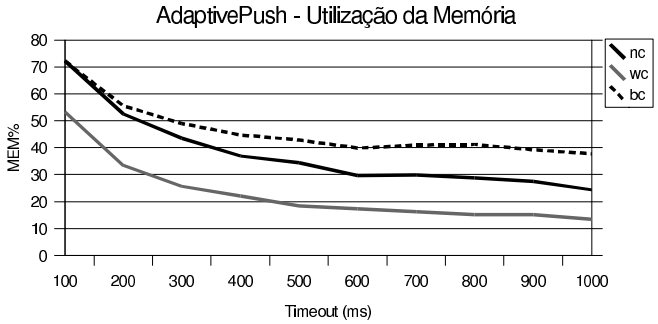

FIGURA 6.15 - Detector *Adaptive Push*: utilização da memória

6.1.4 Detector *Adaptive Pull*

Assim como no caso dos detectores *Push* e *Pull*, o detector *Adaptive Pull* difere do *Adaptive Push* principalmente pelo uso de dois passos de comunicação, com as vantagens e problemas inerentes desse modelo de comunicação. O *roundtrip* das mensagens limita a agilidade da detecção, mas o controle do *timeout* no próprio emissor

das requisições permite uma adaptação melhor ao comportamento da rede, compensando a limitação do *roundtrip*.

A figura 6.16 mostra o desempenho do consenso utilizando este detector nas três situações de testes consideradas neste trabalho. Enquanto o comportamento das curvas *best case* e *worst case* pode ser considerado estável (exceto quando a sobrecarga é muito grande), a curva *normal case* apresenta grandes variações.

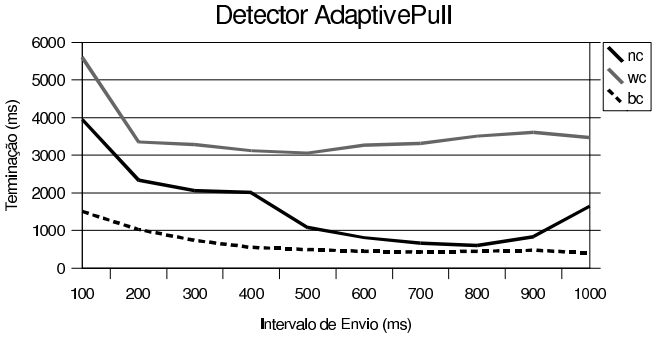

FIGURA 6.16 - Comparação das situações para *Adaptive Pull*

No entanto, quando são analisados trechos isolados desta curva, pode-se verificar que existem intervalos de relativa estabilidade. Primeiramente, se for considerado o trecho entre 500 e 900 ms de intervalo de envio, a curva apresenta-se estável e com um desempenho muito próximo ao da curva *best case*. Já quando o intervalo de envio é menor do que 500 ms, o desempenho começa a cair, motivado pelas falsas suspeitas e pela sobrecarga de processamento. Ainda assim, o detector *Adaptive Pull* procura manter o controle do consenso pois, em média, o tempo gasto com as rodadas canceladss devido às detecções incorretas na situação *normal case* é menor do que o custo da situação *worst case*.

Este aumento da sobrecarga, quando o intervalo de envio é inferior a 500 ms, pode ser validado através da figura 6.17, que indica o tempo de CPU utilizado pela aplicação de testes. Nesta figura nota-se claramente que a quantidade de processamento aumenta muito quando o detector trabalha com intervalos de envio menores do que 500 ms.

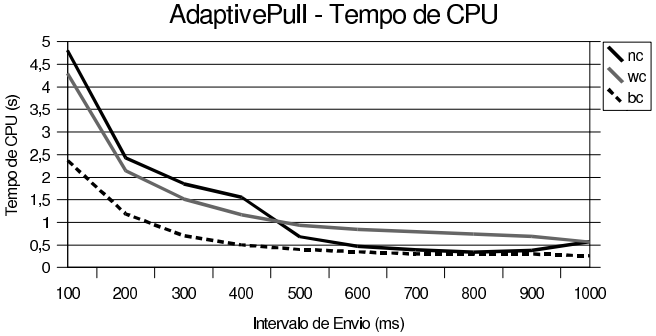

FIGURA 6.17 - Detector *Adaptive Pull*: tempo de CPU

O impacto do intervalo de envio também pode ser verificado na figura 6.18, que mostra a quantidade de memória utilizada pela aplicação. Nela pode-se identificar

especialmente o impacto sobre comportamento da situação *normal case:* a exemplo da figura 6.16, a curva da situação *normal case* distancia-se da curva *best case* quando o intervalo de envio utilizado é menor do que 500 ms.

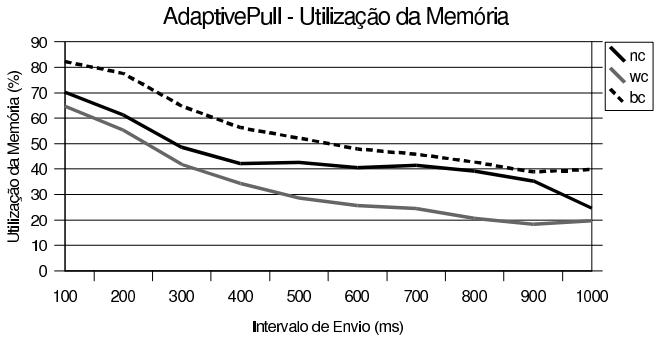

FIGURA 6.18 - Detector *Adaptive Pull*: utilização da memória

6.1.5 Detector *Heartbeat*

O detector *Heartbeat* é um caso à parte, no que se trata dos detectores testados. Seu mecanismo tem como objetivo principal explorar diversas técnicas que podem ser usadas para solucionar alguns problemas dos detectores tradicionais, e por isso, suas prioridades diferem dos demais detectores.

Basicamente, o detector *Heartbeat* utiliza três técnicas para a construção de seu mecanismo de detecção. A primeira delas envolve a detecção propriamente dita. A segunda técnica envolve a difusão das mensagens. A última técnica, que é usada para complementar a segunda, trata sobre a aquisição de conhecimento sobre os demais processos.

O mecanismo de suspeitas do *Heartbeat* é uma solução bem acertada, ao menos do ponto de vista da implementação e dos testes. De fato, ao invés de controlar se o recebimento das mensagens ocorre dentro de um *timeout* pré-estabelecido, o *Heartbeat* controla o número de mensagens recebidas em um determinado período. Isso é muito interessante, pois é uma maneira simples de abstrair alguns problemas comuns aos detectores tradicionais, como as velocidades de processamento e de transmissão.

O mecanismo de difusão também tem seus méritos, uma vez que sua intenção é a de diminuir o número de mensagens que trafegam na rede. Através deste mecanismo, um processo envia mensagens apenas para um pequeno grupo de outros processos (que podem ser fixos - vizinhos - ou escolhidos randômicamente). Embora esse modelo transforme a difusão das mensagens em um modelo probabilístico, a tendência é de que, após um determinado número de transmissões, todos elementos da rede tenham conhecimento das mensagens enviadas. No entanto, quando se trata da detecção de defeitos, esse modelo deve ser visto com ressalvas, uma vez que o número de passos de comunicação pode determinar um grande atraso, tanto no que se refere à disseminação das mensagens de "*I am alive*" quanto na detecção da falha de um processo.

O problema com o detector *Heartbeat* está localizado principalmente na maneira como as informações são obtidas. Uma vez que não há mais contato direto entre todos os processos, não é possível utilizar o mesmo mecanismo dos detectores tradicionais ("se recebi uma mensagem de p_i, então p_i está funcionando"). Cada processo passa a ter

a responsabilidade de avisar aos demais o recebimento de novas mensagens. para isso, a melhor maneira é utilizar as próprias mensagens de detecção trocadas entre os vizinhos. O detector *Heartbeat* utiliza este princípio, mas ao invés de enviar contadores, cada processo anexa à mensagem sua identificação, e a retransmite para seus vizinhos. Após algumas retransmissões cada mensagem carrega consigo a identificação do caminho que percorreu, e os detectores podem atualizar seus contadores com base nestes dados.

O problema é que cada mensagem é automaticamente retransmitida para os vizinhos por onde ainda não passou, de forma que uma única mensagem rapidamente será multiplicada em várias, de acordo com o número de vizinhos e de processos por onde deve passar. Como não há um controle de seqüência das mensagens, enquanto houver alguma mensagem que não circulou por toda a rede haverá o incremento dos contadores. Esse "tempo de estabilização" tem um grande impacto sobre a detecção dos defeitos, uma vez que após a falha de um processo é necessário esperar que todas as mensagens que ele enviou cessem de circular na rede. Além disso, como o conteúdo da mensagem cresce de acordo com o número de processos na rede, a análise de todos os campos das mensagens pode aumentar consideravelmente a sobrecarga do sistema.

Assim, a escolha do número de vizinhos é crucial uma vez que, ao reduzir o número de mensagens enviadas por vez, aumenta-se o número de passos de comunicação para que a mensagem esteja difundida entre todos os processos. A experiência deste trabalho considerou a opção que gera menos mensagens por vez, ou seja, considerou que um processo se comunica com apenas dois vizinhos (a possibilidade de utilizar um único vizinho foi descartada, pois seria criado um ponto único de falha que impossibilitaria a análise da situação *worst case*).

A figura 6.19 mostra claramente o impacto negativo do "tempo de estabilização" do detector *Heartbeat*. O tempo demasiadamente grande que o consenso leva para finalizar na situação *worst case* deve-se a este atraso em detectar a falha do coordenador. Além disso, por usar um modelo não determinístico de comunicação, há uma grande variação dos resultados.

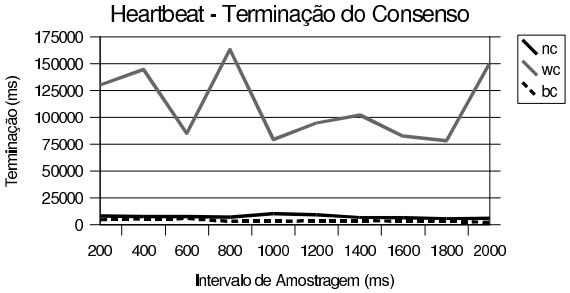

FIGURA 6.19 - Comparação das situações para o *Heartbeat*

Como a visualização das demais situações é comprometida pelos valores muito altos da situação *worst case*, a figura 6.20 exibe um detalhe onde aparecem apenas as duas outras situações. Nesta figura pode-se verificar que, de modo geral, o impacto da sobrecarga de processamento das mensagens apresenta um pequeno aumento quando utiliza-se uma amostragem mais agressiva, mas o maior responsável pela variação dos resultados continua sendo o modelo de comunicação empregado, que é não-determinístico.

Além disso, o tempo médio de finalização da situação *normal case* é consideravelmente maior do que a da *best case*, indicando que, apesar do problema que o "tempo de estabilização" representa para a detecção de defeitos, ocorrem suspeitas incorretas em quantidade considerável. Entretanto, seria necessário averiguar se essas suspeitas incorretas ocorrem durante toda a operação ou apenas enquanto as mensagens não conseguem alcançar todos os processos na rede, para ter certeza sobre as reais causas desta sobrecarga.

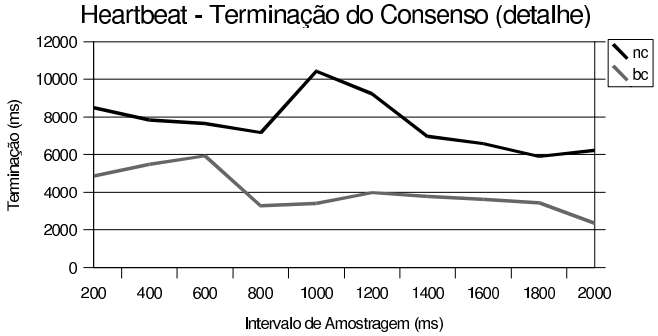

FIGURA 6.20 - Comparação das situações para o *Heartbeat* (detalhe)

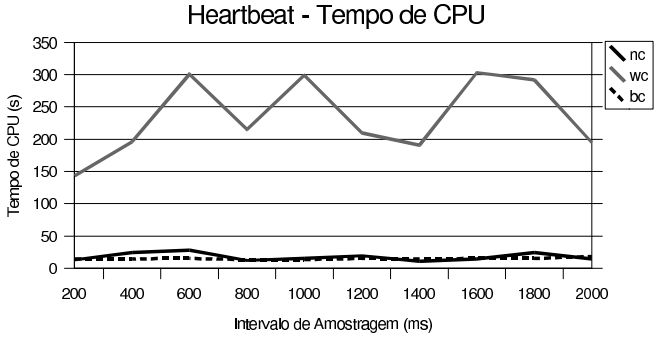

FIGURA 6.21 - Detector *Heartbeat*: tempo de CPU

Um reflexo dos problemas encontrados na terminação do consenso pode ser visto na avaliação do tempo de CPU utilizado pela aplicação. Na figura 6.21 pode-se observar que, apesar da demora em detectar o defeito do coordenador na situação *worst case*, a aplicação não ficou suspensa, como no caso dos outros detectores. De fato, a figura mostra que houve um intenso processamento, provavelmente originário na análise e troca de mensagens do detector. Como o algoritmo de consenso utilizado é semelhante para todas as experiências, esta também não parece ser a causa principal do problema. Já o tempo dos demais casos, melhor visualizado na figura 6.22, foi quase similar, apesar do indeterminismo que os testes na situação *normal case* apresentaram.

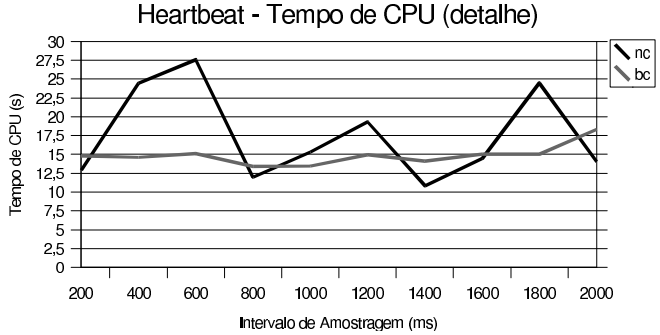

FIGURA 6.22 - Detector *Heartbeat*: tempo de CPU (detalhe)

Apenas na quantidade de memória que os processos ocuparam, não houve uma variação tão grande entre as situações de testes (figura 6.23): todas elas apresentaram índices muito elevados de consumo de memória, independente do intervalo de amostragem considerado. Isso é uma das principais demonstrações da sobrecarga de processamento que o modelo induz. Não obstante, algumas otimizações na implementação ainda podem reduzir o impacto deste detector.

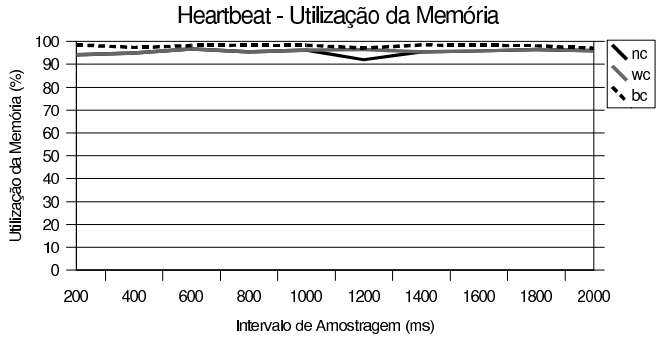

FIGURA 6.23 - Detector *Heartbeat*: utilização da memória

6.1.6 Detector *ad-hoc "no message"*

Os detectores *ad-hoc* são mecanismos de detecção altamente especializados, ou seja, são construídos para operar inseridos dentro de um algoritmo ou aplicação específicos. Comumente são sistemas otimizados para as tarefas e interações que realizam, de forma que fogem do conceito inicial dos detectores de defeitos, ou seja, a modularidade e reusabilidade (ou ao menos, o acesso compartilhado por várias aplicações).

O modelo de detector *ad-hoc "no message"* utilizado neste trabalho pode ser considerado o mais otimizado possível para o algoritmo de consenso de Chandra e Toueg. De fato, este detector tem em comum com os demais modelos apenas o mecanismo de suspeita; todas as demais características como, por exemplo, as mensagens de detecção, foram suprimidas em favor das estruturas e ações já existentes no algoritmo de consenso.

As desvantagens deste modelo concentram-se principalmente na ausência de contato entre os processos enquanto a operação não é finalizada. Com isso, se o coordenador demorar para enviar sua proposta, ele poderá ser considerado suspeito pois não respeitou o *timeout* máximo. Essa limitação é contornada no detector *ad-hoc "heart-beat"*, que será avaliado a seguir.

Além da restrição imposta pela existência de mensagens de detecção, existe a limitação imposta pelo meio de comunicação, uma vez que o *timeout* é contabilizado entre o envio do *estimate* e o recebimento do *propose*. Com isso, um algoritmo em operação normal deve respeitar a limitação do *roundtrip* da rede, caso contrário suas suspeitas incorretas serão extremamente freqüentes.

Por não fazer uso de mensagens próprias de detecção, nem de um elaborado controle de *timeout*, o impacto deste tipo de detector junto ao algoritmo de consenso é mínimo, e se reflete no comportamento das situações *normal case* e *best case*. Na situação *worst case*, a variação do desempenho refere-se sobretudo ao *timeout* utilizado, embora o maior custo das operações ainda esteja relacionado ao próprio tratamento das suspeitas e à condução de uma nova rodada. A figura 6.24 demonstra os tempos de finalização do consenso obtidos para este detector, nas três situações.

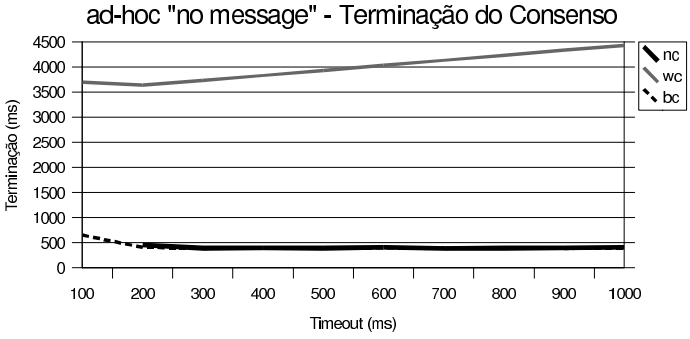

FIGURA 6.24 - Detector *ad-hoc "no message"*: terminação do consenso

Pode-se observar que o desempenho da situação *normal case* foi semelhante ao da situação *best case* em quase toda a extensão dos testes. Apenas quando o *timeout* utilizado foi menor do que o *roundtrip* médio da rede é que as detecções incorretas impediram a realização do consenso (e por isso, não há um valor de desempenho do consenso para o *timeout* de 100 ms). Isso leva à conclusão de que, nos testes realizados, o fator responsável pela diferença entre essas situações junto aos outros detectores foi mesmo a geração de detecções incorretas causada pela sobrecarga das mensagens de detecção. De fato, como este detector quase não influi no processamento, pode-se inferir a sobrecarga do algoritmo de consenso a partir das medições com este detector. Na figura 6.25, é mostrado o tempo de CPU consumido pelo detector *ad-hoc "no message"*: nas situações *best case* e *normal case,* a utilização do processador é menor do que o consumo da situação *worst case*. Este comportamento provavelmente é causado pela menor quantidade de detecções incorretas (ao menos, em comparação com os demais detectores). Como não se torna necessário disparar o processo de suspeita e não há nenhum dispositivo de detecção ativo, não há influência sobre o desempenho do consenso. Já no caso da situação *worst case*, os valores apresentados obrigatoriamente incluem o processamento necessário para tratar as suspeitas, o que aumenta um pouco a utilização da CPU.

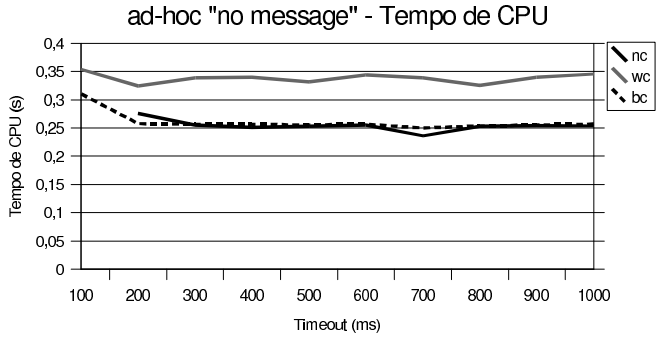

FIGURA 6.25 - Detector *ad-hoc "no message"*: tempo de CPU

Ainda assim, o baixo número de detecções incorretas resulta em um algoritmo que flui mais rapidamente, de forma que a aplicação de teste apresenta um consumo elevado de memória, quando não está na situação *worst case* (figura 6.26).

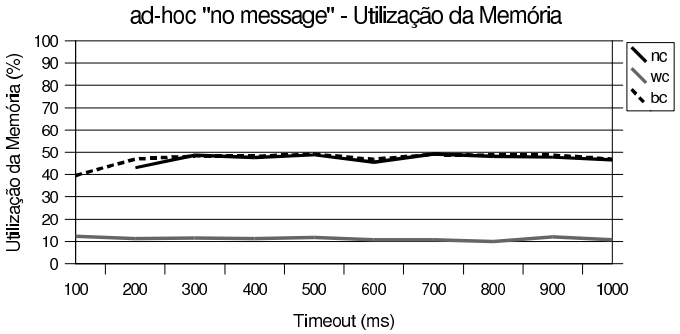

FIGURA 6.26 - Detector *ad-hoc "no message"*: utilização da memória

6.1.7 Detector *ad-hoc "heart-beat"*

O detector *ad-hoc "no message"* tem um problema potencial, que é a possibilidade de fazer uma detecção incorreta quando o coordenador demora a responder. Para tentar solucionar essa restrição, o detector *ad-hoc "heart-beat"* acrescenta ao mecanismo básico daquele detector uma rotina onde o coordenador fica enviando mensagens de "*I am alive*" enquanto não consegue obter o quórum necessário para divulgar sua proposta. Como o envio das mensagens de deteção fica restrito à execução do trecho do consenso compreendido entre as mensagens de *estimate* e de *propose*, sua influência no desempenho global do sistema deve ser menor, pois os detectores convencionais trocam essas mensagens durante todo o tempo.

Se o impacto esperado para este detector deveria ser menor do que o de outros detectores, os testes demonstraram que isto não ocorre. Segundo os dados apresentados na figura 6.27, as situações *best case* e *worst case* obtiveram bons índices, com tempos de finalização coerentes com as expectativas. No entanto, o comportamento da situação *normal case* demonstrou que a detecção no *ad-hoc "heart-beat"* pode ser influenciada por diversos fatores.

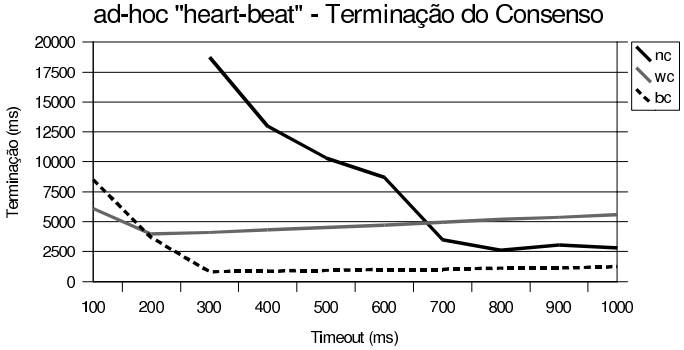

FIGURA 6.27 - Detector *ad-hoc "heart beat"*: terminação do consenso

Mesmo com os valores de *timeout* elevados, a curva da situação *normal case* indica a ocorrência de muitas detecções incorretas, diminuindo o desempenho do consenso. Em uma análise inicial, pode-se considerar três fatores como responsáveis por esse desempenho baixo: o *roundtrip* da primeira mensagem de detecção, a sobrecarga causada pela ativação das *threads* do detector (o detector é ativado e desativado a cada rodada) ou à própria construção do detector.

O tempo que a primeira mensagem de detecção leva para chegar aos processos é um fator muito importante para a operação do *ad-hoc "heart-beat"*, pois não se refere apenas ao tempo de transmissão das mensagens como ocorre com as demais. No caso da primeira mensagem, deve ser contado o tempo de envio da mensagem *estimate*, o processamento de seu conteúdo pelo coordenador e o tempo de transmissão destas mensagens. Pelo que indica a figura 6.27, a situação *normal case* requer que o *timeout* seja no mínimo 700 ms para a aplicação de testes, caso contrário a chance de suspeitas incorretas aumenta.

Já a ativação do detector a cada novo consenso (mais especificamente, do emissor das mensagens do coordenador) deve causar uma sobrecarga adicional à aplicação, que influenciaria a capacidade do detector agir dentro dos limites de tempo. A figura 6.28, entretanto, não mostra nenhuma sobrecarga excessiva de processamento para as situações *best case* e *worst case*, que também estão sujeitas a esse mecanismo de detecção. Na realidade, a influência deste fator deve ocorrer principalmente sobre o atraso das mensagens, uma vez que o *roundtrip* não contabiliza o tempo de inicialização das *threads*.

Além disso, o detector *ad-hoc "heart-beat"* foi construído utilizando o mecanismo e os serviços já existentes no algoritmo de consenso. Essa decisão de projeto procurou refletir a filosofia de integração que comanda os detectores especializados, mas por outro lado, pode ter contribuído para a redução da eficiência dos sistemas. Como as mensagens do detector devem utilizar os mesmos canais das mensagens do algoritmo de consenso, ou seja, são recebidas e processadas pelo objeto `ConsensusFactory` antes de serem entregues ao algoritmo de consenso, estas devem sofrer atrasos devido a esse processo.

A incidência deste atraso pode ser comprovada através da própria escolha da relação entre o *timeout* e o intervalo de envio. Conforme exposto na seção 4.3.3, testes preliminares indicaram que a relação $\Delta i = 98\% \Delta to$, não poderia ser utilizada, pois o

excesso de suspeitas incorretas impedia a terminação do consenso na situação *normal case*. Até mesmo com a relação $\Delta_i = 75\% \Delta_{to}$ não foi possível obter os dados para *timeouts* menores do que 300 ms, pois o consenso não conseguia terminar nenhuma operação. Como os detectores *Push* operam bem sob essas mesmas condições, o mau desempenho do *ad-hoc "heart-beat"* deve-se aos atrasos na entrega das mensagens. Os gráficos demonstram que mesmo utilizando uma relação menos agressiva ainda há a ocorrência de suspeitas incorretas, se o *timeout* for pequeno.

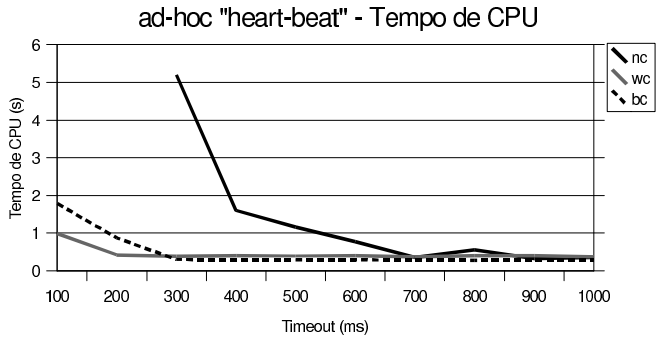

FIGURA 6.28 - Detector *ad-hoc "heart-beat"*: tempo de CPU

Também pode ser observado, na figura 6.29, o impacto da operação deste detector sobre o consumo de memória. Em um primeiro instante, quando se reduz o *timeout*, pode-se observar que a quantidade de memória utilizada pela situação *normal case* decai, aproximando-se da curva da situação *worst case*. Esse comportamento é justificado pelo aumento do número de detecções incorretas, que obrigam os processos a sincronizarem-se em uma nova rodada. Entretanto, quando o *timeout* é menor do que 700 ms, a utilização de memória volta a crescer. Esse comportamento leva a crer que, para esses valores, a sobrecarga induzida pelo aumento da taxa de mensagens e pelo processamento destas faz com que a sobrecarga dos detectores compense os resultados da sincronização das rodadas.

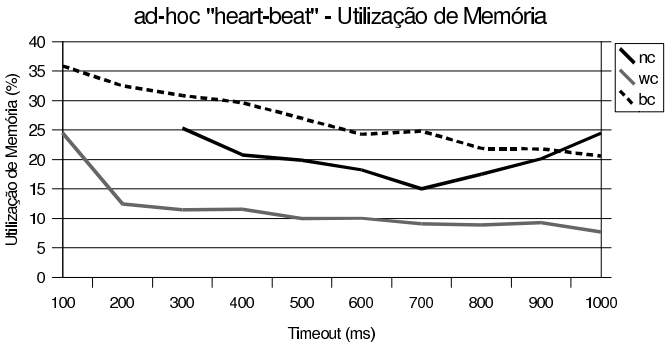

FIGURA 6.29 - Detector *ad-hoc "heart-beat"*: utilização da memória

6.2 Comparação entre os Detectores

As diferenças estruturais fazem com que cada detector tenha um comportamento próprio e reações que se adaptam a utilizações distintas. Entretanto, todos os detectores analisados neste trabalho utilizam *timeouts* em algum nível de operação, facilitando a sua comparação. Apenas os detectores adaptativos não se adaptam a uma comparação direta dos resultados, uma vez que sua adequação dinâmica do *timeout* impede a parametrização dos dados.

Neste capítulo, já foi analisado o comportamento individual de cada modelo de detector de defeitos, indicando a aplicabilidade deles sob condições variadas. A seguir serão comparados os seus comportamentos, de forma a identificar vantagens e desvantagens em relação aos demais detectores.

6.2.1 Detectores adaptativos

Devido ao uso do intervalo de envio (Δ_i) como seu parâmetro variável ao invés do *timeout* (Δ_{to}), os detectores adaptativos formam um grupo à parte nesta análise. Se por um lado isso impede a comparação direta entre todos os detectores, permite que sejam analisados mais cuidadosamente estes dois exemplos cuja característica marcante é a adaptação ao ambiente de operação.

- Terminação do consenso

Analisando a figura 6.30, observa-se que o detector *AdaptivePush* apresenta um desempenho constante na maior parte da experiência, sendo que, apenas quando o sistema começa a ficar sobrecarregado pela quantidade de mensagens, este detector perde um pouco de eficiência.

Já no caso do detector *Adaptive Pull*, o impacto da sobrecarga do sistema pode ser sentido mais cedo, fato que é justificado pelo processamento mais complexo que este detector executa. Apesar de todas as restrições abordadas, ousando-se uma comparação grosseira com os detectores equivalentes *Push* e *Pull*, o impacto dos detectores adaptativos apresenta sempre um pequeno acréscimo, que se deve sobretudo à maior complexidade que o mecanismo de adaptação dinâmica representa para o detector.

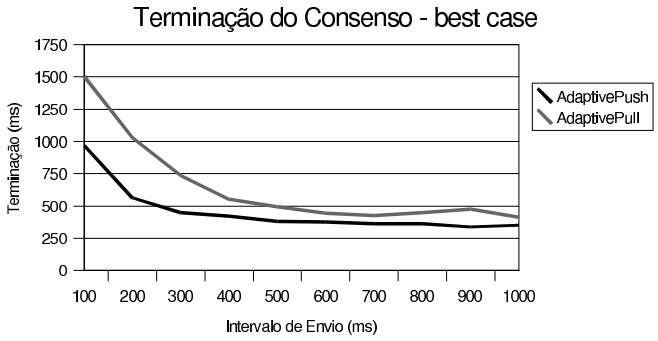

FIGURA 6.30 - Comparação dos detectores adaptativos no *best case*

Na situação *worst case*, apresentada na Fig. 6.31, ambos detectores conseguiram manter constante o desempenho do consenso e, de fato, a maior queda de desempenho registrada pelo detector *Adaptive Pull* ocorreu apenas quando este operava com intervalos de envio muito reduzidos.

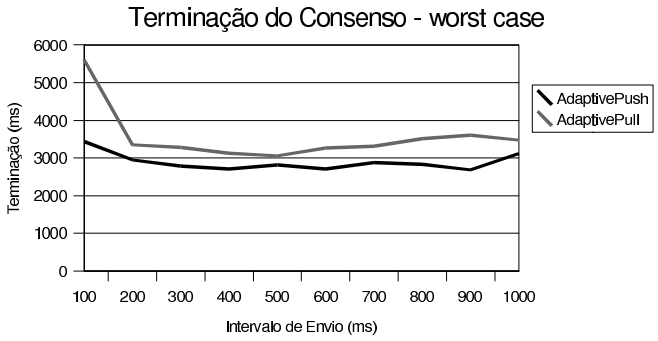

FIGURA 6.31 - Comparação dos detectores adaptativos no *worst case*

A análise da situação *normal case* (Fig. 6.32), ao contrário das demais situações, indica que o detector *Adaptive Pull* não consegue compensar todas as detecções incorretas, resultando em um comportamento instável. O detector *Adaptive Push*, que também deve ter sido influenciado pelas detecções incorretas, conseguiu compensá-las, e somente quando a sobrecarga de processamento do sistema tornou-se muito alta foram apresentados sinais desta influência.

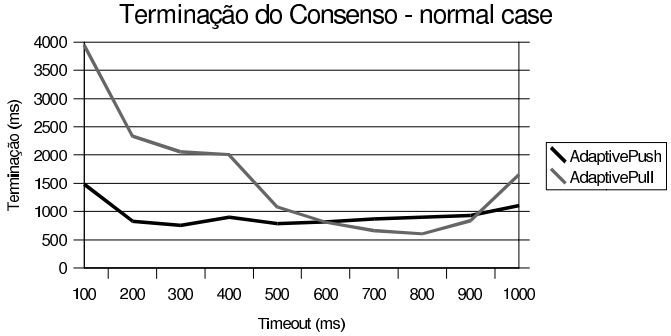

FIGURA 6.32 - Comparação dos detectores adaptativos no *normal case*

- Tempo de CPU

A variação apresentada pelo detector *Adaptive Pull* na situação *normal case* também pode ser observada através do tempo de CPU utilizado em cada operação de consenso, conforme a Fig. 6.33.

A partir destes gráficos, pode-se inferir que para o detector *AdaptivePull*, em intervalos de envio maiores que 500 ms, os valores de tempo de CPU são muito próximos aos apresentados pela situação *best case*. No entanto, para intervalos menores, os valores se aproximam dos apresentados pela situação *worst case*, denotando a influência das detecções incorretas.

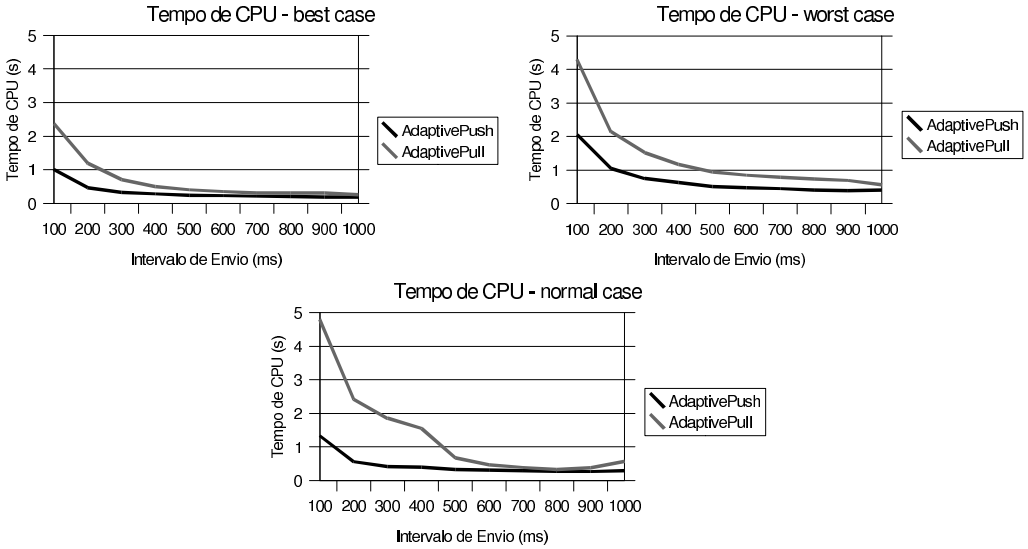

FIGURA 6.33 - Tempo de utilização da CPU: detectores adaptativos

Não obstante estas variações do detector *AdaptivePull*, pode-se afirmar que ambos detectores adaptativos conseguem controlar a geração de suspeitas incorretas, pois ao contrário dos demais detectores, as curvas não indicam ocorrência de múltiplas rodadas quando os detectores estão sobrecarregados.

- Utilização da memória

O consumo de memória dos detectores adaptativos, apresentado na figura 6.34, não traz mais indícios do que os já apresentados pelas demais métricas, e apenas demonstra que o detector *AdaptivePull* tende a consumir mais memória do que o detector *AdaptivePull*. Além disso, ambos parecem estar sujeitos aos mesmos fatores que influenciam essa utilização da memória (suas curvas variam com tendências semelhantes).

Conforme essas observações, pode-se afirmar que, entre os detectores adaptativos, o *AdaptivePush* apresenta o menor impacto sobre as operações de consenso, e ainda é capaz de manter uma operação estável sob várias situações e exigências. O detector *AdaptivePull*, devido ao seu maior tempo de terminação, representa uma solução menos apropriada quando há a necessidade de atualizações rápidas da visão. Entretanto, como é característico dos detectores *Pull*, se a aplicação permite verificações mais espaçadas, este detector pode continuar oferecendo um *timeout* baixo, independente do intervalo de envio.

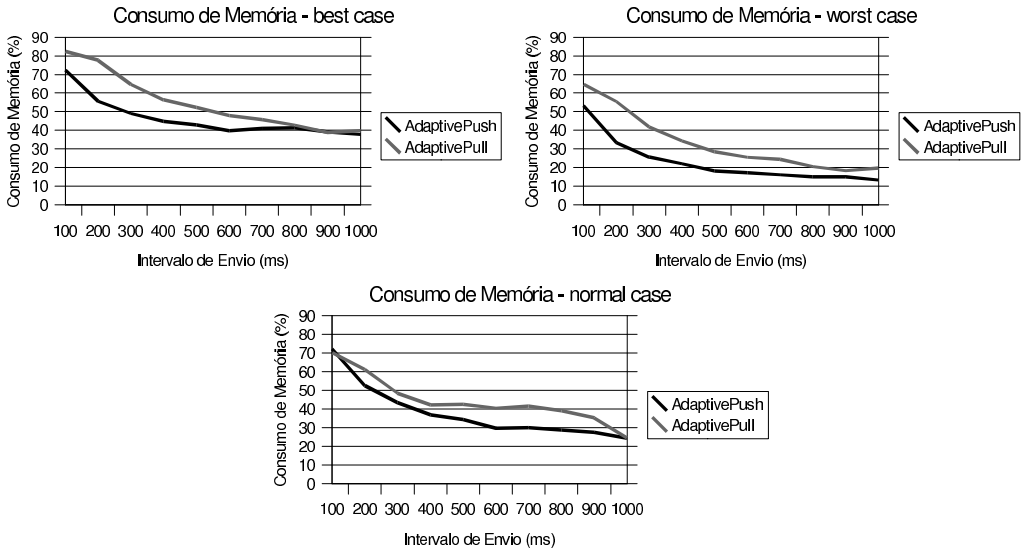

FIGURA 6.34 - Consumo de Memória: detectores adaptativos

6.2.2 Detectores com *timeout* fixo

Por envolver detectores com características e desempenhos muito variados, a comparação dos detectores que utilizam como parâmetro fixo o *timeout* teve que restringir a escala da visualização, e por isso não foram incluídos os resultados excessivamente elevados apresentados pelo detector *Heartbeat*. Esta escolha também contribui para que este detector não seja mal avaliado, uma vez que na análise individual foram identificados diversos pontos de otimização que podem melhorar seus resultados finais.

- Terminação do consenso

Na comparação dos resultados dos detectores para a situação *best case*, mostrada na figura 6.35, pode-se observar que, exceto o detector *ad-hoc "heartbeat"*, todos os demais seguem a mesma tendência. Enquanto o desempenho daquele detector pode ser justificado por sua construção, como explicado na seção 5.2.7, os demais detectores mostram variações de comportamento apenas quando sujeitos a situações de maior sobrecarga (menor *timeout*). Como os detectores "modulares" mantêm uma relação de proporcionalidade entre o *timeout* e intervalo de envio, um *timeout* menor induz a uma taxa maior de mensagens a serem processadas, em um período de tempo. A importância desta relação pode ser observada pelas próprias curvas de desempenho. Na maioria dos casos, os detectores com as relações mais agressivas (do ponto de vista da quantidade de mensagens) apresentam tempos de finalização ligeiramente superiores ao dos seus equivalentes com uma relação mais tolerante. Além disso, o detector *ad-hoc "no message"*, por não enviar mensagens de detecção, apresenta o melhor desempenho.

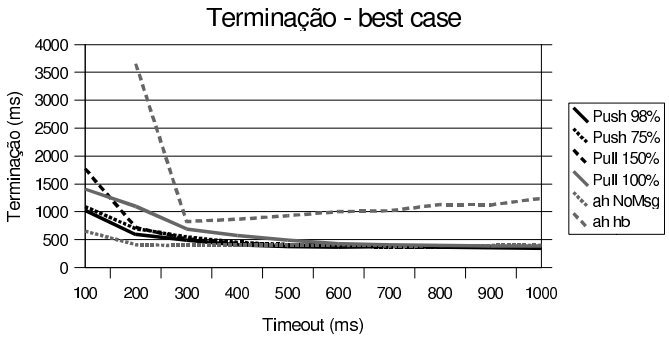

FIGURA 6.35 - Terminação do consenso: comparação *best case*

Já na figura 6.36, que mostra o tempo de terminação do consenso sob a situação *worst case*, observa-se que os detectores *ad-hoc* levaram mais tempo para detectar a falha do coordenador e iniciar a próxima rodada. Enquanto no caso do detector *ad-hoc "heart-beat"* essa diferença possa ser creditada aos atrasos embutidos no mecanismo de comunicação utilizado, não deveria haver comportamento semelhante para o detector *ad-hoc "no message"*. A explicação mais plausível é que a presença de suspeitas errôneas (que podem ser verificadas nas avaliações individuais de cada detector) colabora para suspeitar mais rapidamente de algum processo. O detector *ad-hoc "no message"*, que conta apenas com o *timeout* sobre as mensagens do consenso, não é capaz de suspeitar da falha em algum momento prévio, como fazem os demais detectores.

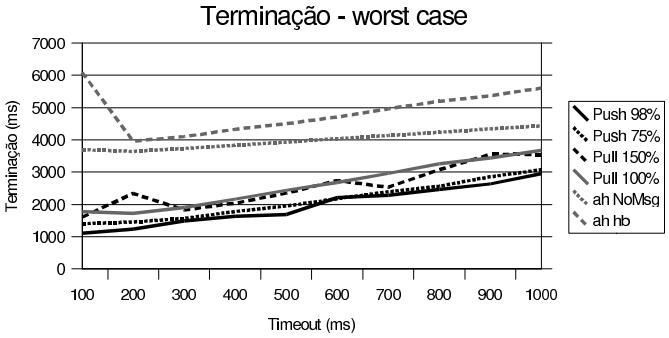

FIGURA 6.36 - Terminação do consenso: comparação *worst case*

Na figura 6.37 podem ser comparados os tempos de terminação do consenso na situação de testes *normal case*. Observa-se que a maioria dos detectores testados conserva um bom desempenho nesta situação, sendo as detecções incorretas as principais responsáveis pela diminuição da eficiência o consenso. A maior diferença destes dados com relação aos das demais situações de teste refere-se ao desempenho dos detectores com uma relação de envio/*timeout* mais agressiva, ou seja, com uma taxa maior de envio por unidade de tempo. Ao contrário das demais situações, onde a sobrecarga do sistema (situação *best case*) e a latência da detecção (situação *worst case*) eram as únicas influências no desempenho dos detectores, na situação *normal case* é importante observar a importância da relação de envio/*timeout* para evitar as detecções incorretas. Conforme exposto na seção 4.3.3, a escolha desta relação deve considerar os

possíveis atrasos de entrega das mensagens. Além disso, conforme o que já havia sido frisado na seção 3.7, a ocupação dos *buffers* de recepção tem uma grande importância para evitar suspeitas incorretas. Isto é comprovado pelos resultados da figura 6.37, onde pode-se verificar que os detectores *Push* 75% e *Pull* 100%, apesar de induzirem uma maior sobrecarga, geram menos detecções errôneas.

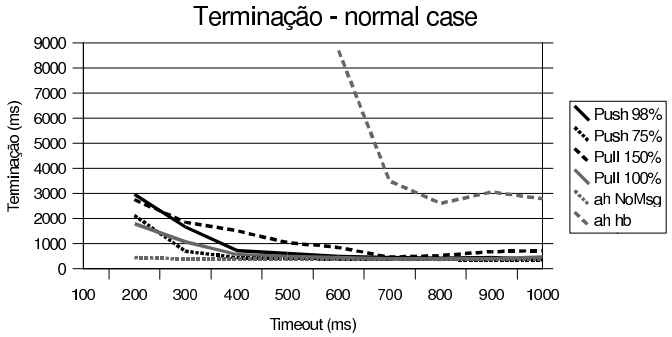

FIGURA 6.37 - Terminação do consenso: comparação *normal case*

- Tempo de CPU

O tempo de CPU utilizado pela aplicação permite identificar as causas que levam ao aumento do tempo de finalização do consenso, pois pode-se determinar os casos onde um processo é mais lento devido à sobrecarga de operações ou devido ao bloqueio do processamento (quando o algoritmo fica esperando alguma mensagem).

Conforme a Fig. 6.38, a maior utilização da CPU na situação *best case* é realizada pelos detectores *Push* e *Pull*, sendo que as relações de envio/*timeout* mais agressivas (75% para o detector *Push* e 100% para o detector *Pull*) apresentam um pequeno acréscimo nesta sobrecarga, em relação às outras relações testadas.

Assim, na situação *best case* os detectores *ad-hoc* apresentam os melhores resultados, confirmando a expectativa decorrente de seu projeto que é voltado para a redução do custo de processamento.

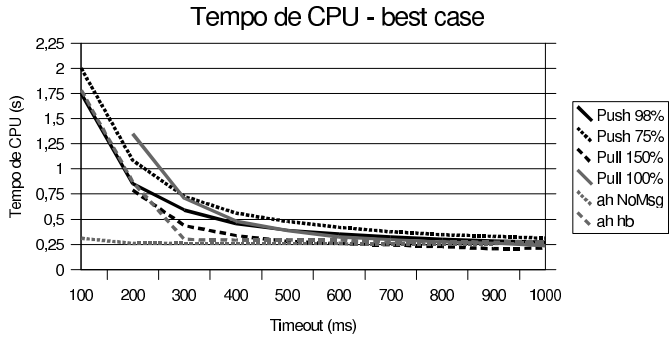

FIGURA 6.38 - Tempo de CPU: comparação *best case*

Na comparação para a situação *worst case*, apresentada na figura 6.39, além dos detectores *ad-hoc* manterem-se entre os mais econômicos, é possível observar também a influência dos modelos de comunicação sobre a detecção. Enquanto os detectores *Push* necessitam basicamente processar as mensagens recebidas e atualizar suas listas de suspeitos, os detectores *Pull* devem passar pelas duas etapas de comunicação antes de obter uma suspeita, de forma que a necessidade de processamento é maior nestes detectores.

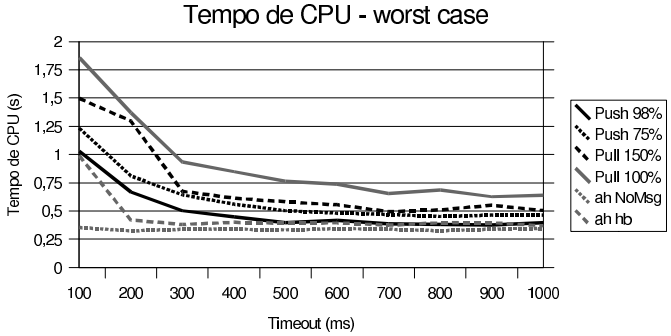

FIGURA 6.39 - Tempo de CPU: comparação *worst case*

Já na figura 6.40 são mostrados os tempos de CPU utilizados na situação *normal case*. Este gráfico confirma as observações sobre a terminação do consenso (figura 6.37), e inclui ainda as observações já realizadas sobre o tempo de CPU dos detectores na situação *best case* (figura 6.38). A partir deste conjunto de observações é possível verificar como a relação de envio/*timeout* é um fator determinante do desempenho dos detectores, uma vez que a quantidade de suspeitas incorretas depende muito da presença de mensagens nos *buffers* de entrada (a exemplo do que já foi apresentado na seção 3.7).

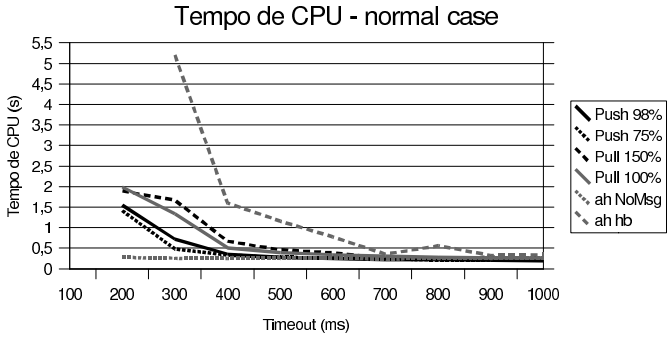

FIGURA 6.40 - Tempo de CPU: comparação *normal case*

- Utilização da memória

A primeira observação sobre os detectores, quando avaliados sob a métrica "utilização da memória", é de que os detectores *ad-hoc*, ao contrário dos demais, têm uma taxa fixa de utilização (visualizada nas figuras 6.41, 6.42 e 6.43). Entretanto, o

detector *ad-hoc "no message"* utiliza mais memória do que o detector *ad-hoc "heart-beat"*, fato que não era esperado.

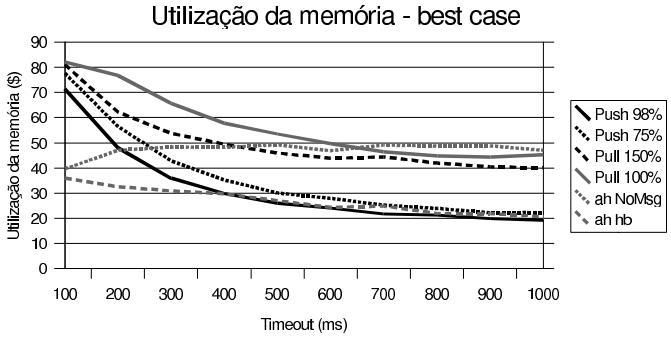

FIGURA 6.41 - Utilização da memória: comparação *best case*

Como o detector *ad-hoc "no message"* não utiliza mensagens de detecção, não é possível atribuir como justificativa o processamento destas. Além disso, a estrutura dos dois detectores *ad-hoc* não difere tanto, a ponto de justificar essa diferença. De fato, a única explicação encontrada para essa diferençaé a de que o detector *ad-hoc "no message"*, por não gerar tantas suspeitas incorretas quanto o outro detector *ad-hoc*, aumente o nível de atividade do consenso, aumentando também o consumo de recursos do sistema.

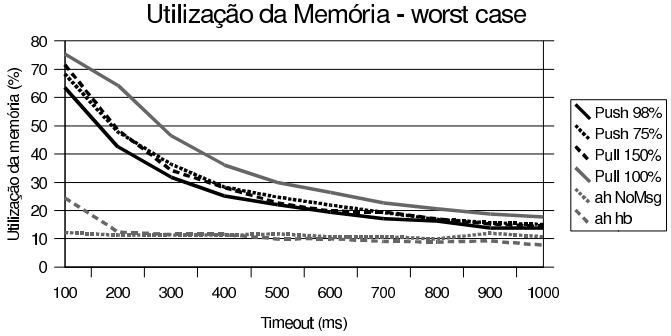

FIGURA 6.42 - Utilização da memória: comparação *worst case*

Outra observação importante para a escolha do modelo de detector mais apropriado a cada situação é a de que os detectores do modelo *Push* têm um menor consumo de memória do que os detectores *Pull*. Novamente, essa é uma característica que deve ser levada em conta na hora de implementar um sistema.

Os detectores *Pull*, entretanto, têm como vantagem a possibilidade de desassociar o intervalo de envio e o *timeout*, que embora não tenha sido melhor explorada nestes testes, pode ser de grande valia para a redução do impacto dos detectores de defeitos no custo dos sistemas.

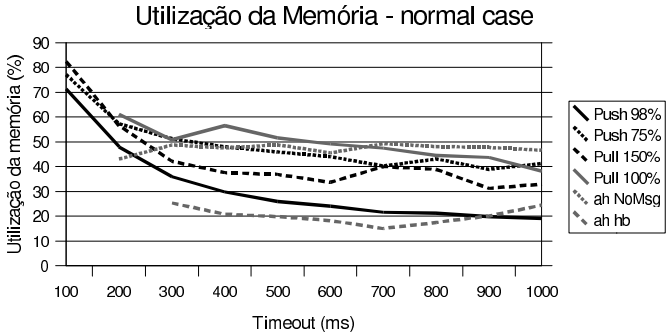

FIGURA 6.43 - Utilização da memória: comparação *normal case*

6.3 Considerações Finais

Neste capítulo, foram apresentadas a compilação dos resultados obtidos nos experimentos e algumas observações feitas sobre esses dados. Enquanto as análises aqui contidas descrevem o comportamento dos detectores e a sua relação com os demais, não foi possível fazer uma comparação direta com os resultados obtidos por Sergent *et al.* [SER 99], como havia sido proposto, uma vez que havia grandes diferenças de parametrização naquele trabalho. A ausência desta comparação, no entanto, não chegou a prejudicar a avaliação dos detectores, pois enquanto a análise individual de cada detector identificou suas vantagens e desvantagens, foi possível compreender as limitações dos detectores especializados, assim como verificar que os detectores modulares podem atingir um bom desempenho, se seus parâmetros forem bem escolhidos. Quanto aos detectores que não obtiveram bons resultados, não se deve desistir da sua utilização, e sim explorar ainda mais as causas dos seus problemas, pois a solução destes com certeza irá beneficiar todos os detectores de defeitos.

7 Conclusões

As operações de consenso constituem-se em um dos elementos básicos para a construção dos sistemas distribuídos, pois estão implícitas em todas as operações de acordo entre processos. No casos dos sistemas distribuídos assíncronos, o auxílio dos detectores de defeitos é essencial para que o consenso possa sobrepujar as limitações deste ambiente.

Enquanto os algoritmos de consenso são amplamente explorados e avaliados quanto ao custo de processamento e de comunicação, os diversos modelos de detectores de defeitos não contam com uma análise mais profunda, obtida através da comparação direta dos algoritmos.

Este trabalho procurou justamente preencher essa lacuna. Foram escolhidos alguns algoritmos que representam os modelos mais conhecidos de detectores de defeitos. Sobre esses detectores, foram feitas avaliações teóricas e práticas. Na avaliação teórica, foram consideradas as características dos algoritmos que têm influência sobre o desempenho dos detectores. Neste trabalho, foram avaliados o número de mensagens geradas pelos detectores, assim como o grau de latência, que representa o número de passos de comunicação necessários para que os algoritmos realizem suas tarefas.

Já na avaliação prática foram considerados o tempo de finalização do consenso, o tempo de CPU utilizado pelos processos e a quantidade de memória requerida. Essas métricas permitiram a avaliação dos detectores sob três situações distintas: a primeira situação avaliou a sobrecarga induzida pelos detectores, a segunda situação avaliou a latência da detecção, enquanto a terceira situação permitiu identificar a ocorrência das suspeitas incorretas, e suas conseqüências.

Para esta avaliação prática, foram implementados na linguagem Java os algoritmos dos detectores e do consenso, e os testes foram realizados sobre um conjunto de cinco processos, dispostos em máquinas distintas.

Entre os fatores que influenciaram negativamente o desempenho dos detectores e do consenso pode-se citar, em ordem de importância, as detecções incorretas, os atrasos na transmissão das mensagens, as limitações físicas do meio da comunicação e a sobrecarga induzida pelo processamento das mensagens.

Como as detecções incorretas normalmente são resultantes do atraso da entrega das mensagens, pode-se afirmar que os principais fatores que influenciam a detecção de defeitos estão diretamente relacionados com a troca de mensagens entre os detectores. De fato, a relação entre as suspeitas incorretas e os atrasos na transmissão tem um papel muito importante na definição dos parâmetros de operação dos detectores. Como a maioria dos modelos testados faz o controle de *timeout* diretamente sobre as mensagens, torna-se necessário escolher uma relação entre *timeout* e intervalo de envio de mensagens que possibilite compensar os possíveis problemas de atraso das mensagens.

Os detectores mais afetados pelas limitações físicas do meio de transmissão são aqueles que necessitam dois ou mais passos de comunicação. Isso ocorre no momento da definição de seus *timeouts*, pois estes detectores necessitam considerar o tempo de ida e volta das mensagens, de tal forma que os detectores ficam impedidos de realizar uma detecção mais agressiva, sob pena de gerar muitas detecções incorretas.

Já os detectores que utilizam apenas um passo de comunicação não são tão sujeitos a essas limitações físicas, mas devem tomar um cuidado especial com a sobrecarga produzida pelo excesso de mensagens enviadas.

A análise dos diferentes modelos de detectores de defeitos implementados para este trabalho possibilitou identificar algumas características marcantes de cada modelo, especialmente as relacionadas com os mecanismos de suspeita empregados.

Em primeiro lugar, os detectores adaptativos demonstraram ser bastante confiáveis pois, embora não tenham obtido os resultados mais eficientes, apresentaram um comportamento extremamente estável, independentemente da situação ou a carga de processamento imposta. Além disso, demonstraram uma ótima capacidade para minimizar a ocorrência de suspeitas incorretas, sem no entanto afetar a latência da detecção.

Já os detectores *Push* também destacaram-se pelo bom desempenho, que apenas foi influenciado pela sobrecarga de mensagens e pelos atrasos na entrega destas quando o *timeout* escolhido era muito curto.

Os detectores *Pull* apresentaram resultados próximos aos dos detectores *Push*, sendo que apenas a sobrecarga gerada pelo maior número de passos de comunicação impediu um melhor rendimento. De fato, essa influência pôde ser constatada através da utilização de memória, onde os detectores *Pull* apresentaram índices superiores aos dos detectores *Push*. O maior destaque deste modelo de detector, entretanto, refere-se à ocorrência de suspeitas errôneas. Ao contrário do que se esperava, este detector demonstrou que um maior intervalo entre as mensagens de deteção não compensa a diminuição da sobrecarga do meio de comunicação, pois os atrasos de entrega ainda são mais influentes na detecção.

O detector *Heartbeat*, escolhido para representar os detectores com características "*gossip*" de disseminação, apresentou os piores resultados entre todos os detectores analisados. No entanto, este detector não deve ser desconsiderado para algum trabalho futuro, pois os resultados demonstrara que o principal fator que reduziu o desempenho foi o mecanismo de disseminação de informações empregado na implementação. Isso significa que embora o modelo *gossip* reduza o número de mensagens enviadas em cada ciclo de detecção, o modelo de disseminação de informações foi extremamente mal dimensionado, afetando seriamente o desempenho do detector. De fato, se fosse excluído esse fator, a principal limitação deste detector seria na latência da detecção, pois o modelo *gossip* induz a um maior grau de latência.

Enquanto os detectores apresentados acima compartilham características modulares, ou seja, os detectores encapsulam os mecanismos de comunicação e de detecção a tal ponto que o consenso pode tratá-los apenas como módulos de detecção, existem ainda os detectores especializados, também chamados *ad-hoc*.

Geralmente considera-se que os detectores especializados estão inseridos no algoritmo de consenso, de tal forma que podem desempenhar suas funções de maneira mais eficiente e otimizada, compensando a perda de modularidade do detector e de portabilidade do consenso. De fato, essa vantagem pode ser reafirmada para o detector *ad-hoc "no message"* que, por não enviar nenhuma mensagem de detecção, não está sujeito à sobrecarga do processamento destas, nem aos atrasos de entrega. A ausência de mensagens próprias apenas tem influência sobre a latência da detecção, fato que pode ser negligenciado, considerando-se que este detector gera poucas suspeitas incorretas, as maiores responsáveis pela diminuição do desempenho do consenso.

Já no caso do detector *ad-hoc "heart-beat"* aquela afirmação não demonstrou sua veracidade. Este detector, apesar de estar inserido no algoritmo de consenso, teve seu desempenho reduzido por causa do excesso de detecções incorretas. Embora a causa imediata destas falsas suspeitas indicasse os atrasos na entrega das mensagens, foi verificado que estes atrasos eram causados pela disputa de recursos com o algoritmo de consenso. Assim, esse detector não foi capaz de otimizar o desempenho das operações de consenso, mesmo contando a seu favor o menor número de mensagens enviadas.

O conjunto destas análises permitiu reavaliar as observações feitas por Sergent *et al.* [SER 99]. Enquanto aquele trabalho indicava que as implementações especializadas eram as únicas capazes de atingir os melhores resultados, este trabalho demonstrou que ambas abordagens, detectores especializados e detectores modulares, podem atingir desempenhos muito bons. De fato, pode-se ainda considerar que o comportamento dos detectores modulares foi mais estável e independente da implementação. Isso se deve sobretudo à constante manutenção das listas de suspeitos, que permite a utilização de informações obtidas anteriormente para auxiliar a detecção no momento presente. Essas características reforçam a proposta de Felber *et al.* [FEL 99], onde os detectores de defeitos devem ser considerados "cidadãos de primeira classe", ou seja, serviços indispensáveis aos sistemas distribuídos e que estão sempre prontos a oferecer informações sobre as detecções para todas as aplicações do sistema. Relembrando o trabalho de Sergent *et al.*, onde foi afirmado que "detecção de defeitos e eficiência não são objetivos ortogonais", pode-se afirmar, a partir dos dados obtidos, que "a modularidade dos detectores e a eficiência também não são objetivos ortogonais".

Enquanto este trabalho responde algumas questões sobre os detectores de defeitos, ainda existem muitas tarefas a cumprir. Com base nas observações realizadas, os pesquisadores podem elaborar modelos de detectores de defeitos mais eficientes e otimizados, que agraciem, ao mesmo tempo, a redução da latência da detecção e a redução das suspeitas incorretas. Além disso, a partir de um modelo comportamental do sistema onde forem empregados os detectores, poderia ser feita uma análise mais completa, quantificando estatisticamente as relações *best/normal/worst case* de acordo com a sua ocorrência real. Isso permitiria uma comparação "direta" entre os detectores para um determinado cenário.

Se o objetivo das próximas pesquisas for de explorar valores absoluto de desempenho dos detectores, ou seus compromissos com metas de qualidade de serviço (QoS), pode ser interessante reconduzir os testes para outros ambientes, usando diferentes protocolos de comunicação ou linguagens de programação.

De aplicação imediata estão os sub-produtos deste trabalho, ou seja, as implementações dos detectores e do consenso, que podem ser usados para compor protocolos e ferramentas para uso em sistemas distribuídos, e em especial, ferramentas de comunicação de grupo.

Bibliografia

[AGU 96] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **Randomization and Failure Detection: A Hybrid Approach to Solve Consensus**. Ithaca:Cornell University, Jun. 1996. Disponível em: < http://cs-tr. cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl. cornell/TR96-1592 >. Acesso em: 11 jan. 2000.

[AGU 97a] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. **On the Weakest Failure Detector for Quiescent Reliable Communication**. Ithaca:Cornell University, Jul. 1997. Disponível em: < http://cs-tr. cs.cornell .edu:80/Dienst/UI/1.0/Display/ncstrl.cornell /TR97-1640 >. Acesso em: 11 jan. 2000.

[AGU 97b] AGUILERA, Marcos Kawazoe; CHEN, Wei; TOUEG, Sam. Heartbeat: a timeout-free failure detector for quiescent reliable communication. In: INTERNATIONAL WORKSHOP ON DISTRIBUTED ALGORITHMS, 11., Sep. 1997. **Proceedings...** Disponível em: < http://cs-tr. cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell /TR97-1631 >. Acesso em: 11 jan. 2000.

[BRA 00] BRAWERMAN, Alessandro; DUARTE Jr., Elias Procópio. A Synchronous Testing Strategy for Hierarchical Adaptive Distributed System-Level Diagnosis. In: IEEE LATIN AMERICAN TEST WORKSHOP, 1., Mar. 2000, Rio de Janeiro. **Proceedings...** Rio de Janeiro:IEEE, Mar. 2000. p. 154-161.

[CHA 96a] CHANDRA, Tushar Deepak; TOUEG, Sam. Unreliable Failure Detectors for Reliable Distributed Systems. **Journal of the ACM**, New York, v. 43, n. 2, p. 225-267, Mar. 1996. Disponível em: < http://cs-tr. cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/TR95-1535 >. Acesso em: 12 jan. 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, p. 685-722, Jul. 1996. Disponível em: < http:// cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/Display/ncstrl.cornell/ TR94-1426 >. Acesso em: 12 jan. 2000.

[CHE 00] CHEN, Wei; TOUEG, Sam; AGUILERA, Marcos Kawazoe. On the Quality of Service of Failure Detectors. In: INTERNATIONAL CONFERENCE ON DEPENDABLE SYSTEMS AND NETWORKS, Jun. 2000, New York. **Proceedings...** New York:IEEE, Jun. 2000.

[EST 00a] ESTEFANEL, Luiz Angelo Barchet. **Detectores de Defeitos Não Confiáveis**. UFRGS:Porto Alegre, Jan. 2000. 105p. Trabalho Individual Disponível em: < http://www.inf.ufrgs.br/~angelo/ publications/TIDetectores.zip >. Acesso em: 20 nov. 2000.

[EST 00b] ESTEFANEL, Luiz Angelo Barchet; JANSCH-PÔRTO, Ingrid. Comunicação Não Confiável em Detectores de Defeitos com Falhas por *Crash*. In: WORKSHOP DE TESTES E TOLERÂNCIA A

FALHAS, 2., Jul. 2000, Curitiba. **Anais...** Curitiba:UFPR, Jul. 2000. 123p, p. 64-69. Disponível em: < http://www.inf.ufrgs.br/~angelo/publications/Comunicacao.pdf >. Acesso em: 20 nov. 2000.

[EST 00c] ESTEFANEL, Luiz Angelo Barchet; JANSCH-PÔRTO, Ingrid. Avaliação Prática de um Detector de Defeitos: teoria *versus* implementação. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 2., Jul. 2000, Curitiba. **Anais...** Curitiba:UFPR, Jul. 2000. 123p, p. 70-75. Disponível em: < http://www.inf.ufrgs.br/~angelo/publications/Observacoes.pdf >. Acesso em: 20 nov. 2000.

[FEL 98] FELBER, Pascal. **The CORBA Object Group Service**. Lausanne:EPFL, 1998. Tese de Doutorado. Disponível em: < http://lsewww.epfl.ch/OGS/thesis/ >. Acesso em: 12 jan. 2000.

[FEL 99] FELBER, Pascal; DÉFAGO, Xavier; GUERRAOUI, Rachid; OSER, Philipp. Failure Detectors as First Class Objects. In: IEEE INTERNATIONAL SYMPOSIUM ON DISTRIBUTED OBJECTS AND APPLICATIONS, Edinburgh,Scotland, Sep. 1999, p. 132-141. **Proceedings...** Edinburgh:IEEE, 1999. Disponível em: < http://dlib.computer.org/conferen/doa/0182/pdf/01820132.pdf >. Acesso em: 20 nov. 2000.

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

[GAR 98] GARBINATO, Benoît. **Protocol Objects and Patterns for Structuring Reliable Distributed Systems**. Lausanne:EPFL, 1998. Tese de Doutorado. Disponível em: < http://lsewww.epfl.ch/garbinato/PhD/ >. Acesso em: 12 jan. 2000.

[GRE 00] GREVE, Fabiola; HURFIN, Michel; MACÊDO, Raimundo; RAYNAL, Michel. **Consensus Based on Strong Failure Detectors: Time and Message-Efficient Protocols**. Rennes:INRIA, Jan. 2000. 12p. Disponível em: < ftp://ftp.inria.fr >. Acesso em: 21 nov. 2000.

[GUE 97] GUERRAOUI, Rachid; SCHIPER, André. Consensus: The Big Misunderstanding. In: IEEE INTERNATIONAL WORKSHOP ON FUTURE TRENDS IN DISTRIBUTED COMPUTING SYSTEMS, Oct. 1997. **Proceedings...** Disponível em: < http://lsewww.epfl.ch/~rachid/papers/ftdcs2-97.ps >. Acesso em: 12 jan. 2000.

[GUO 98] GUO, Katherine Hua. Failure Detection. In: GUO, Katherine Hua. **Scalable Message Stability Detection Protocols**, Ithaca:Cornell University, May 1998. Tese de Doutorado. Disponível em: < http://cs-tr.cs.cornell.edu:80/Dienst/UI/ 1.0/Display/ncstrl.cornell/TR98-1684 >. Acesso em: 12 jan. 2000.

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

[HUR 00] HURFIN, Michel; MACÊDO, Raimundo; MOSTEFAOUI, Achour; RAYNAL, Michel. **A Sliding Round Window $\langle\rangle\mathcal{S}$-based Consensus Protocol**. Rennes: IRISA, Feb. 2000.

[JAI 91] JAIN, Raj. **The art of computer systems performance analysis:** techniques for experimental design, measurement, simulation, and modeling. New York:John Wiley, 1991. 685p.

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

[LAM 78] LAMPORT, Leslie. Time, clocks and the ordering of events in a distributed system. **Communications of the ACM**, v. 21, n. 7, p. 558-565, 1978.

[LUN 00] LUNG, Lao Cheuk; FRAGA, Joni da Silva. Detecção de Falha para Redes de Larga Escala no Fault-Tolerant CORBA. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 2., Jul. 2000, Curitiba. **Anais...** Curitiba:UFPR, Jul. 2000. 123p, p. 10-15.

[MAC 00] MACÊDO, Raimundo. Failure Detection in Asynchronous Systems. In: WORKSHOP DE TESTES E TOLERÂNCIA A FALHAS, 2., Jul. 2000, Curitiba. **Anais...** Curitiba:UFPR, Jul. 2000. 123p, p. 76-81.

[PIN 01] PINHEIRO, Manuele Kirsch. **Mecanismo de Suporte à Percepção em Ambientes Cooperativos**. Porto Alegre, RS: PPGC/UFRGS, 2001. Dissertação de mestrado.

[SAB 95] SABEL, Laura S.; MARZULLO, Keith. **Election vs. Consensus in Asynchronous Systems**, Ithaca:Cornell University, Feb. 1995. Disponível em: < http://cs-tr.cs.cornell.edu:80/Dienst/UI/1.0/ Display/ncstrl.cornell/TR95-1488 >. Acesso em: 11 jan. 2000.

[SCH 97] SCHIPER, André. Early Consensus in an asynchronous system with a weak failure detector. **Distributed Computing**, Springer-Verlag, v. 10, n. 3, p. 149-157, Apr. 1997.

[SER 99] SERGENT, Nicole; DÉFAGO, Xavier; SCHIPER, André. **Failure Detectors: implementation issues and impact on consensus performance**, Lausanne:EPFL, 1999. Disponível em: < http:// lsewww.epfl.ch/Research/Reports >. Acesso em: 15 mar. 2000.

[TUR 94] TUREK, John; SHASHA, Dennis. The Many Faces of Consensus in Distributed Systems. In: **Readings in Distributed Computing Systems**. New York:IEEE, 1994. p. 83-99

[VOG 96] VOGELS, Werner. World Wide Failures, In: ACM SIGOPS EUROPEAN WORKSHOP, 7., Sep. 1996. Connemara, Ireland. **Proceedings...** Connemara:ACM Press, Sep. 1996. Disponível em: < http:// mosquitonet.Stanford.EDU/sigops96/papers/vogels.ps >. Acesso em: 12 jan. 2000.