DisCusS: desenvolvendo um Serviço de Consenso genérico, simples e modular

Lásaro Jonas Camargos

Dissertação de Mestrado

Instituto de Computação
Universidade Estadual de Campinas

DisCusS: desenvolvendo um Serviço de Consenso genérico, simples e modular

Lásaro Jonas Camargos[1]

29 de Agosto de 2003

Banca Examinadora:

- Prof. Dr. Edmundo Roberto Mauro Madeira
  Instituto de Computação – UNICAMP (Orientador)
- Profa. Dra. Ingrid Jansch-Pôrto
  Instituto de Informática – UFRGS
- Prof. Dr. Luiz Eduardo Buzato
  Instituto de Computação – UNICAMP
- Prof. Dr. Ricardo Anido
  Instituto de Computação – UNICAMP (Suplente)

[1]**Trabalho com suporte parcial do CNPq, processo 133415/2001-5**

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Camargos, Lásaro Jonas

C14d DisCusS: desenvolvendo um Serviço de Consenso genérico, simples e modular / -- Campinas, [S.P. :s.n.], 2003.

Orientador : Edmundo Roberto Mauro Madeira

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Processamento distribuído. 2. Tolerância a falhas (Computação). 3. Software de sistemas. I. Madeira, Edmundo Roberto Mauro. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

DisCusS: desenvolvendo um Serviço de Consenso genérico, simples e modular

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Lásaro Jonas Camargos e aprovada pela Banca Examinadora.

Campinas, 1 de outubro de 2003.

Edmundo R. M. Madeira

Prof. Dr. Edmundo Roberto Mauro Madeira
Instituto de Computação – UNICAMP
(Orientador)

Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

TERMO DE APROVAÇÃO

Tese defendida e aprovada em 29 de agosto de 2003, pela Banca examinadora composta pelos Professores Doutores:

Profa. Dra. Ingrid Eleonora Schreiber Jansch Porto
UFRGS

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

Prof. Dr. Edmundo Roberto Mauro Madeira
IC - UNICAMP

Epígrafe

"I think that it's extraordinarily important that we in computer science keep fun in computing. When it started out, it was an awful lot of fun. Of course, the paying customers got shafted every now and then, and after a while we began to take their complaints seriously. We began to feel as if we really were responsible for the successful, error-free perfect use of these machines. I don't think we are. I think we're responsible for stretching them, setting them off in new directions, and keeping fun in the house. I hope the field of computer science never loses its sense of fun. Above all, I hope we don't become missionaries. Don't feel as if you're Bible salesmen. The world has too many of those already. What you know about computing other people will learn. Don't feel as if the key to successful computing is only in your hands. What's in your hands, I think and hope, is intelligence: the ability to see the machine as more than when you were first led up to it, that you can make it more."

Alan J. Perlis.

Resumo

Esta dissertação trata do processo de engenharia de um serviço de detecção de falhas compatível com FT-CORBA, a especificação para tolerância a falhas em CORBA, e de um serviço de consenso distribuído. Os serviços são independentes e fornecem diferentes propriedades para a aplicação cliente, dependendo dos módulos, com implementações de algoritmos diferentes, selecionados para uma *instanciação* destes serviços.

A arquitetura dos serviços é tal que a aplicação cliente não toma conhecimento dos algoritmos de detecção e consenso sendo executados, acessando-os por uma interface genérica.

Com o intuito de facilitar a escolha dos módulos dos serviços, apresentamos um pequeno estudo comparativo da influência de detectores de falhas adaptativos, aqueles que se adaptam para prover melhor qualidade de serviço na detecção, e não adaptativos sobre o desempenho dos algoritmos de consenso distribuído.

Abstract

This thesis is on the process of development of a *distributed consensus service* and its *fault detection service*, being the last one, compliant with FT-CORBA, the Fault Tolerant CORBA specification. These services are independent and, depending on the selected modules, offer different properties to client applications.

The presented architecture is defined in such a way that client applications do not know which algorithms are running, both for detection and for consensus, just accessing them through a generic interface.

To make easier the task of selecting modules in the services, we also present a comparative study of the influence of adaptive and non-adaptive failure detectors over the execution of consensus algorithms.

Agradecimentos

Gostaria de agradecer a todas as pessoas com quem convivi durante este período, que me ajudaram a aguentar a saudade de casa, ou que foram o motivo dela, do outro lado do rio Grande.

Obrigado a minha namorada Ewellyne, por me entender e apoiar, e por ter ficado sempre do meu lado, mesmo estando tão longe. Obrigado a D. Neuza e ao Sr. Eurípedes, pelo apoio que me deram, no início do começo.

Obrigado aos meus colegas de república Eduardo, Flávio, Flávio, Fernando, Marcelo, Marcelo, Luciana, Celso, Chenca, Bazinho, Quintão e Anibal, pela convivência quase sempre pacífica e por terem me feito rir muito.

Obrigado aos colegas do IC: os paraenses Fábio, Fernando e Schubert, pelas discussões filosóficas, pelos almoços de domingo e pelos bombons de cupuaçú; aos baianos Guido e Gustavo (sempre do contra); aos amigos Fernando e Rodrigo, pelas discussões científicas e pelos mangás; à Juliana, Evandro, Silvana, Bartho e alguns tantos outros colegas do IC, por serem colegas no IC.

Ainda no IC, agradeço aos professores e, em especial, ao Prof. Edmundo, pela paciência e orientação que modelou e culminou neste trabalho. Obrigado também aos membros da banca, cujas relevantes sugestões melhoraram este trabalho.

Agradeço ao CNPq pelo apoio financeiro, indispensável.

Finalmente, agradeço a minha família. Valeu por sempre acreditar em mim, Tião (meu irmão). E, como o mais importante agradecimento, agradeço a minha maior e melhor professora, Dona Margarida, minha mãe. Agradeço pelas lições de vida, pelo exemplo de corretude e perseverança, por compreender minha ausência e por todo o amor dedicado.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

"Comece do começo, passe pelo meio, vá até o final, e então pare."
Alice no País das Maravilhas - Lewis Carrol.

Sistemas computacionais são hoje usados nos mais diversos ramos de nossa sociedade. Há computadores executando operações "simples" como compra de passagens, controle de acesso a instalações, ou operações mais complexas como em sistemas de suporte a vida, mercado financeiro e controle de aviônicos, tornando-nos cada vez mais dependentes. Com o aumento de nossa dependência nestes sistemas computacionais, cresce também nossa vulnerabilidade às suas falhas. Tentamos então minimizar a possibilidade de ocorrência de falhas e, quando não houver mais meios para evitá-las, tratá-las de forma a manter a corretude e disponibilidade dos sistemas sendo usados. Esta é a propriedade que denominamos *Tolerância a Falhas* [Gär01].

A redundância pousa como a chave para a tolerância a falhas. Uma vez que a tarefa de impedir a ocorrência de falhas é nada trivial, replicamos os componentes sujeitos a falhas no sistema e os coordenamos para que mantenham consistência em suas ações. Assim, considerando baixa a probabilidade de falhas simultâneas em um grande número de réplicas, consegue-se mascarar a ocorrência de algumas destas falhas. Na coordenação destas réplicas surge outro problema: como garantir que uma mesma seqüência correta de operações seja executada em cada réplica, mantendo sua consistência? Este problema é redutível ao problema de *Consenso* [CT96].

O consenso pode ser visto como o *máximo divisor comum*, *i.e.*, a maior subunidade, de vários problemas de concordância (*agreement*) [MR99], como os problemas de *terminação atômica* (*weak atomic commitment*) [Gue95, Had86] e difusão totalmente ordenada (*total order broadcast*) [CT96]. Tais problemas surgem muito freqüentemente na implementação de sistemas distribuídos tolerantes a falhas. Percebe-se, portanto, que a existência de uma infra-estrutura para a resolução de problemas de consenso fazer-se-ia de grande utilidade no desenvolvimento de sistemas distribuídos tolerantes a falhas.

Fischer, Lynch e Patterson [FLP85] provaram a impossibilidade da resolução de consenso distribuído em sistemas assíncronos nos quais os processos estejam sujeitos a falhas. A impossibilidade é devida à dificuldade em se determinar quando um processo está falho ou simplesmente muito lento. Isso ocorre pela ausência de limites de tempo na execução de tarefas de processamento e transmissão de mensagens no sistema. Entretanto, Chandra e Toueg [CT96] mostraram que o consenso pode ser obtido em sistemas assíncronos adicionados de *detectores de falhas não confiáveis* (*unreliable failure detectors*), um *oráculo* distribuído que tenta adivinhar o estado funcional dos processos. Chandra e Toueg [CT96] apresentam também uma classificação dos detectores de falhas, e Chandra, Hadzilacos e Toueg [CHT96] mostram qual o detector mais simples que pode ser usado na resolução do consenso.

O trabalho de Guerraoui e Schiper [GS01] apresenta uma arquitetura para um serviço genérico de consenso distribuído que pode fazer uso de qualquer detector de falhas disponível no sistema. Tal trabalho, porém, apenas apresenta a arquitetura em um alto grau de abstração, não se preocupando com os algoritmos utilizados ou com uma implementação real do mesmo. A generalidade proposta por Guerraoui e Schiper [GS01] tem grande influência sobre o este trabalho.

O trabalho de Felber [Fel98] apresenta um serviço para replicação ativa de objetos, OGS (*Object Group Service*). O OGS faz uso de serviços de consenso e detecção de falhas para controlar os grupos de objetos. Ambos os serviços foram implementados em CORBA [OMG01a], aumentando a aplicabilidade destes serviços. CORBA, *Common Object Request Broker Architecture*, é uma arquitetura para acesso a objetos distribuídos com transparência de localização e linguagem de implementação. CORBA possui a especificação FT-CORBA [OMG01b], para o desenvolvimento de aplicações CORBA tolerantes a falhas. Contudo, OGS não é compatível com FT-CORBA, uma vez que é anterior a esta especificação.

Este trabalho apresenta serviços de consenso e de detecção de falhas [CM03] modelados com vistas a alcançar generalidade, simplicidade e modularidade. Tais serviços definem interfaces simples, mas com grande poder de abrangência sobre os algoritmos de consenso e detecção de falhas encontrados na literatura. Tais serviços podem ser usados para a resolução de diversos problemas de concordância, desde que haja *tradutores* destes problemas para o consenso. Aplicações *cliente* do serviço de consenso propõem ao *pool* de *objetos de consenso* suas estimativas de qual deve ser o valor acordado, e transmitem a este pool a responsabilidade pelo consenso. Os objetos de consenso, por sua vez, são completamente independentes de seus clientes, podendo ser *posicionados* estrategicamente de forma a aumentar a disponibilidade do serviço (por exemplo, em uma rede com alta disponibilidade). Atenção especial foi dada a detectores de falhas adaptativos no desenvolvimento do serviço de detecção de falhas. Em relação a FT-CORBA, este

trabalho apresenta algumas possibilidades para sua implementação utilizando um serviço de consenso.

A arquitetura proposta aqui é bastante modular e flexível, permitindo que módulos sejam escolhidos e encaixados de acordo com requisitos de qualidade de serviço e ambiente no qual o sistema irá operar. Surge desta flexibilidade o problema da seleção dos algoritmos mais adequados a cada ambiente. Neste trabalho, damos um primeiro passo na direção da avaliação de módulos, comparando a influência dos detectores de falhas em um serviço de consenso em termos de desempenho, considerando como métricas a freqüência máxima de execução (*throughtput* ou vazão) e o tempo mínimo para a obtenção de uma resposta do protocolo de consenso (latência).

Esta dissertação abrange várias áreas e, sendo assim, é preciso chamar à atenção o fato de que trabalhos relacionados são apresentados em vários pontos, de acordo com o tema sendo abordado. O Capítulo 2 apresenta conceitos básicos e trabalhos relacionados. O Capítulo 3 propõe um *framework* para construção de detectores de falhas adaptativos. O Capítulo 4 traz uma introdução sobre CORBA e FT-CORBA, enquanto o Capítulo 5 apresenta os serviços de detecção de falhas e consenso distribuído propostos, que são o tópico principal desta dissertação. O Capítulo 6 mostra um estudo sobre o impacto dos mecanismos de detecção de falhas na execução de algoritmos de consenso, e o Capítulo 7 conclui este trabalho, apresentando também pontos para possíveis trabalhos futuros.

Capítulo 2

Conceitos Básicos e Trabalhos Relacionados

"A constante de um homem é a variável de outro."
Alan Perlis.

Este capítulo apresenta uma série de conceitos e formalismos que serviram de base para o trabalho desenvolvido. Também apresenta um apanhado geral dos algoritmos de detecção de falhas e consenso distribuído encontrados na literatura.

2.1 Algoritmos Distribuídos

Segundo Gärtner [Gär99], sistemas distribuídos são um conjunto finito de processos que se comunicam por troca de mensagens através de um subsistema de comunicação. Cada um destes processos roda um algoritmo local, e a união destes algoritmos locais compõem um algoritmo distribuído. Aplicações que implementam algoritmos distribuídos, sistemas distribuídos, abstraem de seus usuários tal distribuição, sendo este usuário possivelmente outra aplicação

A aplicação dos sistemas distribuídos abrange um grande espectro de áreas, incluindo telecomunicações, computação científica e controle de processos em tempo real. Normalmente, requer-se que sistemas distribuídos trabalhem corretamente mesmo em sistemas heterogêneos e sujeitos a falhas em processos.

Dependabilidade e Tolerância a Falhas em Sistemas Distribuídos

A *dependabilidade* (*dependability*) de um sistema é a quantificação da confiança que pode ser (justificadamente) depositada no mesmo [Gär99, VR01], do quanto se pode depender

deste sistema. A quantificação estatística indicando a probabilidade do sistema estar funcional em um determinado instante é denominada *disponibilidade*.

Alguns autores chamam de falha, a imperfeição que leva um sistema a uma condição de *erro* e a uma manifestação deste erro na forma de um *defeito*. Hoje em dia, sistemas de software e hardware são construídos com subsistemas, peças menores, fazendo com que um defeito, em um nível inferior, apresente-se como uma falha ao nível superior. Sendo assim, consideramos uma *falha* como um desvio de execução de um sistema, em relação ao comportamento para o qual foi especificado. Um meio comum para aumentar a dependabilidade de um sistema é inserir-lhe capacidades de *tolerância a falhas*, a capacidade de continuar operante e correto mesmo na presença de falhas. Nenhum sistema pode ser totalmente tolerante a toda e qualquer falha (ou conjunto delas). A redundância é o mecanismo pelo qual se atinge a tolerância e o grau de tolerância é função desta redundância. Um componente redundante é desnecessário na execução normal do sistema, e sua atividade só é importante no caso de ocorrência de falha em outro componente.

Diz-se que um sistema mantém-se vivo se mantém suas propriedades *liveness*, se continua a prover sua funcionalidade, na ocorrência de falhas, isto é, se continua a avançar em sua computação. Se nenhum erro é introduzido pela falha, isto é, se o sistema continua a prover sua funcionalidade da forma como especificado, então diz-se que o sistema mantém suas propriedades *safety*, que mantém-se seguro [Gär99]. Gärtner [Gär99] formaliza quatro tipos de tolerância a falhas (Veja a Tabela 2.1). O tipo mais básico é a ausência de tolerância, combinação da ausência de segurança e de *liveness*. Se apenas a segurança é garantida, o sistema é *seguro contra falhas* (*fail safe*). Se apenas *liveness* mantém-se, o sistema é dito *não-mascarador*. Em contra partida, um sistema *mascarador* mantém-se vivo e seguro na presença de falhas.

	Vivo	Não Vivo
Seguro	Mascarador	Seguro
Não Seguro	Não Mascarador	Ausente

Tabela 2.1: Tipos de Tolerância a falhas.

2.2 Modelos de Sistema

O primeiro passo para a especificação de um sistema distribuído tolerante a falhas é a definição do modelo sobre o qual o sistema será construído. As duas questões mais importantes nesta definição são as características de sincronismo e o modelo de falhas

que deverá ser suportado. Estamos considerando neste trabalho, o modelo de troca de mensagens entre processos, sem nenhuma forma de memória compartilhada.

2.2.1 Sincronismo

A característica de sincronismo nos processadores e canais de comunicação vai do extremo do totalmente síncrono ao assíncrono. No primeiro, cada passo no processador ou transmissão de uma mensagem é executado em um tempo pré-determinado. No segundo, limites de tempo não são conhecidos para nenhuma destas operações e nada se pode afirmar quanto às velocidades relativas dos processadores. Pontos intermediários neste espectro compõem a classe dos sistemas *parcialmente* síncronos, nos quais relógios (parcialmente) sincronizados ou com um limite de discrepância conhecido podem existir, ou ainda, limites aproximados para troca de mensagens ou execução de instruções pelo processo podem existir [Lyn96].

O ambiente assíncrono é mais simples e geral que os demais, o que quer dizer que soluções encontradas neste ambiente também funcionarão em ambientes mais restritos. Apesar da dificuldade imposta pelo assincronismo, este ambiente é plausível, e pode ser exemplificado pela Internet.

2.2.2 Modelos de Falhas

Não há sistema que seja tolerante a qualquer tipo de falhas, pois há limitações quanto à redundância que pode ser aplicada em sua construção e execução. Logo, o desenvolvimento de um sistema tolerante a falhas deve ser pautado no tipo de falha que se deseja tolerar (leia-se, severidade). Componentes do sistema podem simplesmente parar de funcionar totalmente (*crash-stop*) ou parar de forma que dificulte a detecção da parada (*fail-stop*), parar intermitentemente (*omissão*), ou não trabalhar segundo o protocolo que deveria (*Bizantinas* [LSP95]), podendo tomar qualquer ação, inclusive malévolas.

2.3 Consenso

Os problemas de concordância (acordo, *agreement*) estão entre os mais interessantes na computação tolerante a falhas. Eles aparecem quando diversos processos devem tomar decisões consistentes. Tome por exemplo, a determinação de qual a próxima instrução a ser executada em uma máquina de estados replicada [Lam96], ou qual ação tomar na finalização de uma transação, *commit* ou *abort*. Grande parte dos problemas de concordância podem ser reduzidos ao problema do *consenso*.

O problema do consenso é definido da seguinte maneira: cada processo $p_i \in P$ inicialmente *propõe* um valor v_i, de um conjunto de possíveis valores, e os processos em P têm que se *decidir* por um valor v de forma que as seguintes propriedades sejam satisfeitas [Sch97]:

Terminação Todo processo correto chega a uma decisão em algum instante futuro

Validade Se um processo decide-se por v, então v é igual a algum v_i.

Concordância Todos os processos corretos decidem-se por um mesmo v.

Observe que a propriedade de *concordância* permite que processos incorretos decidam diferentemente dos corretos. Quando se considera ambientes com apenas falhas benignas (*fail-stop*), esta propriedade é suficientemente segura para garantir consistência. Entretanto, em sistemas onde outras falhas são permitidas, o seguinte reforço pode ser dado à concordância:

Concordância Uniforme Todos os processos que chegam a uma decisão, decidem-se por um mesmo valor v.

A versão de consenso definida com esta propriedade é conhecida como *Consenso Uniforme*. Embora possa parecer um problema mais difícil que o consenso não uniforme, Guerraoui [Gue95] demonstrou a equivalência destes dois problemas quando se faz uso de detectores de falhas não-confiáveis. Esta equivalência não ocorre quando se considera falhas bizantinas, no qual um processo pode agir como se tivesse falhado por parada, e ainda assim decidir por qualquer valor.

Em sistemas isentos de falhas, resolver o consenso é uma tarefa trivial. Considere por exemplo o protocolo simples em que todo processo envia sua estimativa inicial para todos os outros. A decisão é dada pela aplicação de uma função f no conjunto de estimativas recebidas. Este protocolo funciona tanto em ambientes síncronos como assíncronos, pois é garantida a chegada das mensagens enviadas.

Considere a introdução de falhas de parada nos processos executando o algoritmo anterior. Os processos têm que decidir quando parar de esperar por uma mensagem de um processo falho. No ambiente síncrono, cada processo sabe em que passo do protocolo todos os outros processos estão (ou deveriam estar), e consegue perceber a ocorrência de falhas. Entretanto, em ambiente assíncrono não há solução determinística para o consenso se os processos estão sujeitos a falhas, conforme demonstrado por Fischer, Lynch e Patterson [FLP85].

Como resultado desta impossibilidade, passou-se a tentar identificar os ambientes mais simples tais que permitam a resolução do consenso, isto é, aqueles que fossem o mais gerais possível.

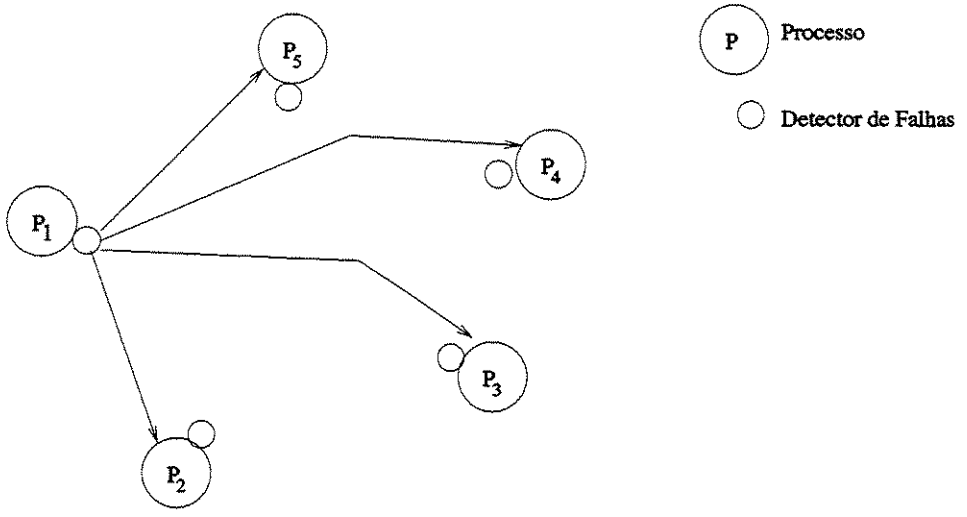


Figura 2.1: Detectores de Falhas

2.3.1 Sincronismo Parcial

Na busca por níveis intermediários de sincronismo, Dwork *et al* [DLS88] e Dolev *et al* [DDS87] definiram uma série de trinta e duas classes de sistemas parcialmente síncronos. Estas classes são encontradas pela combinação de cinco parâmetros, que podem assumir condições favoráveis ou não ($2^5 = 32$), sendo estes parâmetros os seguintes (a condição favorável é sempre a primeira): processadores síncronos ou assincronos; comunicação síncrona ou assincrona; ordenação FIFO ou não; comunicação por broadcast ou ponto-a-ponto; e, envio/recebimento de mensagens atômico ou não.

Em [DDS87], quatro destas classes são identificadas como modelos mínimos nos quais o consenso pode ser resolvido. *Mínimos* neste caso recebe o sentido de que a mudança de algum parâmetro de condição favorável para desfavorável em qualquer destas classes inviabiliza o consenso.

2.3.2 Detectores de Falhas

Chandra e Toueg [CT96] introduziram o conceito de *Detectores de Falhas Não-Confiáveis*. Um detector de falhas pode ser visto como *oráculo* distribuído, com módulos acoplados aos processos (ou objetos) do sistema, e que trabalha determinando o estado funcional dos outros processos (ver Figura 2.1).

Os detectores de falhas não podem ser implementados em sistemas totalmente assíncronos. Em ambientes parcialmente síncronos, nos quais os processos dispõem de *timers* precisos, um detector pode contar a passagem do tempo nos intervalos de comunicação com outros processos e, considerando um limite de tempo para estes intervalos, tentar

determinar se tais processos encontram-se falhos ou não [CHT96]. Esta determinação é por certo imprecisa, e os detectores podem voltar atrás em suas suspeitas tão logo percebam um erro. Entretanto, a despeito desta incerteza, a informação provida por estes detectores já é suficiente para que se alcance o consenso, salvo a restrição de que a maioria dos processos não sofra falhas [CT96, LSP95], e que os mesmos provejam certas propriedades mínimas, descritas adiante. Os detectores não confiáveis de certa forma *encapsulam* o sincronismo necessário à resolução do consenso.

Chandra e Toueg [CT96] classificaram os detectores de falhas segundo suas características de completude (*completeness*) e acurácia (*accuracy*), ou seja, a capacidade de suspeitar de um processo falho e a capacidade de não suspeitar de um processo correto, respectivamente. Alguns níveis destas propriedades são descritos a seguir:

Completude Forte A partir de algum instante futuro, *todo* processo falho é suspeito permanentemente por *todos* os processos corretos.

Completude Fraca A partir de algum instante futuro, *todo* processo falho é suspeito permanentemente por *algum* processo correto.

Precisão Forte *Todos* os processos são suspeitos somente após terem falhado.

Precisão Fraca *Algum* processo correto nunca é suspeito de ter falhado.

Precisão *Eventual* Forte A partir de algum instante futuro, *todos* os processos são suspeitos somente após falharem.

Precisão *Eventual* Fraca A partir de algum instante futuro, *algum* processo ativo nunca é suspeito antes de ter falhado.

Na prática, estas propriedades precisam ser válidas apenas por um período de tempo suficientemente longo para a resolução de uma instância de consenso. A Tabela 2.2 mostra as classes de detectores de falhas obtidas pelas combinações de acurácia e completude descritas.

Chandra e Toueg [CT96] demonstraram que completude *fraca* e *forte* são equivalentes. Isso ocorre devido a facilidade de propagação de uma suspeita para todos os outros detectores, fazendo com que todos suspeitem de um processo falho. Demonstraram também que classes com precisão *fraca* são "mais fracas" que classes com precisão *forte*, implicando que todo problema resolvível com detectores com precisão fraca pode ser resolvido com detectores com precisão forte (*eventual* ou não), embora o inverso não seja válido. A Figura 2.2 denota estas relações.

Em [CHT96] Chandra *et al* demonstram que $\Diamond W$ é o detector mais fraco com o qual se pode resolver consenso. Este detector de falhas é implementável em sistemas nos quais

Completude	Acurácia			
	Forte	Fraca	*Eventual* Forte	*Eventual* Fraca
Forte	Perfeito P	Forte S	Eventualmente Perfeito $\Diamond P$	*Eventual* Forte $\Diamond S$
Fraca	 Q	Fraco W	 $\Diamond Q$	*Eventual* Fraco $\Diamond W$

Tabela 2.2: Classes de detectores de falhas.

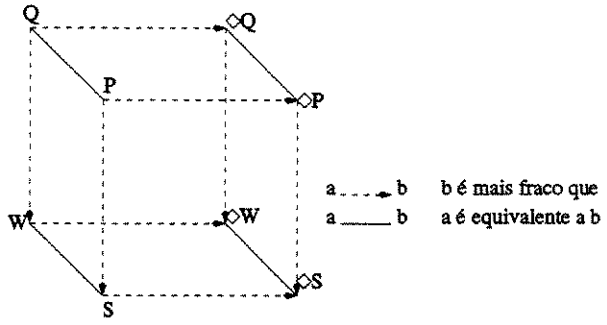


Figura 2.2: Relação entre classes de detectores de falhas.

há um limite superior de tempo para a transmissão de mensagens, mesmo que este limite seja desconhecido. Vários protocolos de consenso utilizam o detector equivalente, $\Diamond S$, aproveitando-se da completude forte.

2.3.3 Aleatorização

Em [AT96], Aguilera *et al.* propõem um protocolo de consenso que funciona sobre detectores de falhas não confiáveis durante rodadas boas (i. e., quando o coordenador não falha), e sobre oráculos aleatórios, uma função associada a cada processo e que gera um número aleatório toda vez que é invocada, durante rodadas com muitos erros. Desta forma, o protocolo sempre avança, mesmo que haja muitos erros dos detectores de falhas. Uma abordagem parecida é dada em [MRR01]. Este protocolo agrega oráculos aleatórios e oráculos de líderes, detectores que apontam um processo correto ao invés de apresentarem uma lista de suspeitos a cada vez que são invocados, e em situações com poucas falhas (menos de um terço dos objetos falham), pode ser terminado em apenas um passo de comunicação.

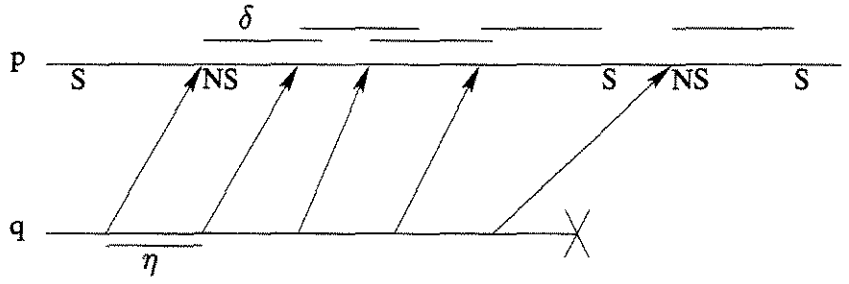


Figura 2.3: Detectores do tipo *heartbeat*

2.4 Trabalhos Relacionados: Algoritmos

Esta seção apresenta um apanhado geral dos algoritmos de detecção de falhas não confiáveis e de consenso baseados nestes detectores disponíveis na literatura. O conhecimento das características destes algoritmos facilita a escolha do mais adequado para um dado ambiente.

2.4.1 Detectores

Os esquemas de detecção de falhas, como será mostrado a seguir, evoluíram de configurações básicas a esquemas com garantias de qualidade de serviço, procurando minimizar e adaptarem-se às variações de carga que ocorrem na prática em sistemas distribuídos.

Detectores do Tipo *heartbeat*

Os detectores do tipo *heartbeat*, também conhecidos como *push*, executam um protocolo extremamente simples. Considere o caso trivial, no qual há dois módulos (processo, objetos) p e q no sistema, e p monitora q. O módulo q envia mensagens periodicamente, indicando a p que está vivo (*heartbeats*). Esta periodicidade é dada pelo parâmetro *entre_envios*, ou η. Alternativamente, as mensagens podem ser solicitadas por p, instanciando assim, um detector do tipo *pull*. p possui um temporizador que é (re)iniciado quando da chegada de uma mensagem. Se este temporizador atinge um valor acima de um *limite* (*timeout*) δ pré-estabelecido, p passa a considerar q como "suspeito de falha". Na Figura 2.3 vemos um exemplo de rodada deste algoritmo. Suspeitas indevidas podem ocorrer infinitamente, mas uma falha é sempre percebida em algum instante (pois as mensagens de monitoração param de ser enviadas após a falha).

Este detector, embora possua completude forte, não possui acurácia fraca ou forte,

eventual ou não. Somente com δ assumindo um valor alto, pode-se garantir que o detector não cometerá erros em períodos longos, o que também demanda conhecimento sobre a distribuição de atrasos das mensagens na rede. Em contrapartida, *timeouts* altos aumentam o tempo de detecção de falhas.

Chandra e Toueg

O detector de falhas proposto por Chandra e Toueg em [CHT96] estende os detectores do tipo *heartbeat*, alcançando acurácia *eventual* forte em ambientes nos quais há um limite de tempo de comunicação (mesmo que desconhecido). Ao perceber um erro, estes detectores incrementam o valor de δ. Assim, em algum momento futuro, o detector irá parar de cometer erros nas suspeitas, pois δ tornar-se-á maior que o limite superior do tempo de propagação da mensagem.

O grande problema destes detectores é que, na prática, tornam-se lentos na detecção de falhas após períodos de sobrecarga do sistema. Embora seja necessário um aumento da tolerância nos tempos de resposta durante a ocorrência de sobrecarga, este aumento não pode perpetuar-se por períodos de carga normal, principalmente por que estes são mais freqüentes e duradouros [AT96]. Deste problema, surge a idéia de detectores de falhas adaptativos [CTA02, BMP02, SM01a, SM01b, NJP02b], *i. e.*, que se adaptam às diferentes situações de carga do sistema.

Chen

Chen, Aguilera e Toueg [CTA02] apresentam um novo algoritmo de detecção que, como os anteriormente apresentados, envia mensagens de q para p com periodicidade η. Cada mensagem m_i é enviada por q no instante σ_i. Sendo $\tau_i = \sigma_i + \delta$, p suspeita de q no instante $t \in [\tau_i, \tau_{i+1})$, se e somente se m_i não foi recebida. Desta forma, mensagens antigas não enganam o detector, possivelmente diminuindo seu tempo de detecção. Na Figura 2.4 vemos como este algoritmo pode cometer erros (a mensagem enviada em σ_4 é recebida após τ_4) e como falhas são percebidas (nenhuma mensagem é enviada após σ_5).

A maior contribuição deste trabalho foi o estudo e definição de algumas métricas de qualidade de serviço (QoS) na detecção de falhas, e o estudo da qualidade de serviço de seu novo detector, resultando em um algoritmo estimador que toma como entrada alguns parâmetros da rede (probabilidade de perda de mensagens e a distribuição dos atrasos das mensagens) e requisitos de QoS (T_D – tempo de detecção -, T_{MR} – tempo de recorrência de erros – e T_M – duração do erro), e que gera valores de η e δ que satisfazem estes requisitos de QoS.

Quando as propriedades da rede não são conhecidas, uma avaliação do histórico de transmissão de mensagens na mesma pode ser usada para sua caracterização. Além disso,

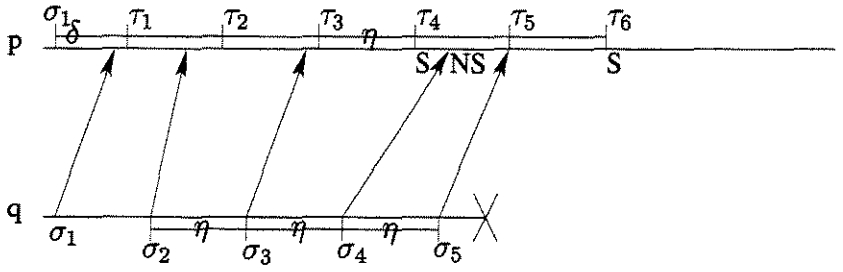


Figura 2.4: Detector proposto por Chen.

execuções subseqüentes do estimador, considerando variações da carga de serviço na rede, conferem adaptabilidade a este detector.

Bertier

Bertier *et al* [BMP02] propõem uma alternativa híbrida ao algoritmo de Chen [CTA02]. Embora use o estimador apresentado por Chen [CTA02] para calcular o provável instante de chegada de mensagens, usa o algoritmo de Jacobsen [PA00] para adicionar uma margem de segurança a este valor e calcular o δ. Este algoritmo responde mais rapidamente às variações de carga da rede, mas é um pouco mais lento que [CTA02] para baixar o δ após sobrecargas [BMP02].

ADAPTATION e DPCP

Os algoritmos propostos por Sotoma e Madeira [SM01a, SM01b] tentam, além de adaptarem-se às variações de carga na rede, minimizar a contribuição dos próprios detectores na degradação da mesma. Baseado no histórico de chegada das mensagens *heartbeat*, o algoritmo ajusta tanto os limites de espera para as mensagens seguintes, δ, quanto o intervalo *entre_envios*, η, no objeto monitorado.

ADAPTATION [SM01a] estima δ em função dos atrasos das três últimas mensagens recebidas, mais uma constante, que ajuda o algoritmo a lidar com pequenas variações na rede. Variações bruscas implicam em um evento *timeout* que, por sua vez, leva ao uso de um multiplicador no cálculo final de δ, ajustando o algoritmo a tais variações. DPCP [SM01b] usa apenas a última mensagem recebida no cálculo do novo valor de δ, ajustando-se mais rapidamente às variações. No trabalho [SM01b] há ainda a proposta de um valor mínimo que pode ser assumido por δ, o *minimum time unit* (MTU). A determinação do valor de MTU é dependente do compromisso de velocidade de detecção e degradação da rede.

Outros métodos para ajuste dos valores de *timeout* também são encontrados na literatura. Nunes e Jansch-Pôrto [NJP02a, NJP02b], por exemplo, constroem séries temporais a partir do histórico de *heartbeats* solicitados (método PULL), e utilizam estas séries para tentar predizer futuros atrasos na transmissão destas mensagens e, com isso, ajustar os parâmetros de detecção.

Detectores Especializados

Observando que os detectores de falhas também contribuem para a degradação dos recursos no ambiente onde são operacionais, Sergent *et al* [SDS99] propõem detectores que limitam o envio de mensagens a um determinado período de tempo, ou mesmo eliminam tais mensagens. Estes detectores são ditos *especializados* por trabalharem intimamente com algoritmos que os usam. O primeiro algoritmo proposto envia mensagens *heartbeat* apenas durante períodos críticos do algoritmo cliente, isto é, apenas nos intervalos em que falhas devam ser monitoradas para evitar bloqueio. Considere um sistema cliente/servidor: o cliente não precisa monitorar o servidor durante todo o tempo, podendo fazê-lo, apenas durante o intervalo entre o envio de uma requisição e sua resposta. O segundo algoritmo proposto por Sergent elimina inclusive estas mensagens, usando as mensagens do protocolo superior para monitorar falhas.

Na análise comparativa apresentada em [EJP01], estes detectores não se mostraram melhores que a abordagem convencional usando *heartbeats*, no que tange ao tempo de terminação do algoritmo que os utiliza [CT96]. Tal desempenho pode ser explicado pela lentidão na detecção, que só ocorre nos períodos críticos [EJP01]. Esta lentidão pode ser justificada pelo número reduzido de mensagens, cabendo ao desenvolvedor pesar prós e contras.

*H*eartbeat

O detector de falhas proposto por Aguilera *et al* [ACT97], embora denominado *H*eartbeat, segue uma abordagem diferente dos já apresentados. Enquanto os outros algoritmos preocupam-se com o atraso na entrega de mensagens *heartbeat*, o algoritmo *H*eartbeat preocupa-se com a quantidade de mensagens entregues. Uma mensagem *heartbeat* h, enviada por um processo p_i, passa por outros processos p_j, forçando-os a incrementar seu contador referente a p_i ($Cont_{p_j}[i] + +$). Quando uma aplicação requisita informações de seu detector, recebe o vetor $Cont$ e determina, baseado na variação dos contadores em relação ao último vetor recebido, quem deve considerar falho ou não.

Levando a adaptatividade em consideração, é proposto aqui a seguinte forma para análise da informação disponibilizada pelo detector:

- Sejam

- $Cont_1$ e $Cont_2$ os dois últimos vetores recebidos do detector Heartbeat;
- $Dif = Cont_1 - Cont_2$;
- $\max(X)$ e $\min(X)$ o maior e o menor valor no vetor X, respectivamente; e,
- c uma constante inteira positiva, que indica a fração do máximo incremento que deve ter ocorrido em todos os processos corretos. O processo de determinação desta constante é empírico e depende do ambiente, como a determinação do timeout inicial para algoritmos *heartbeat*.

- $\forall i$, se $Dif[i] < \max(Dif)/c$, então i é suspeito de falha.

Desta forma, se houver um aumento na carga de trabalho da rede, todos os contadores demorarão mais para serem incrementados, fazendo com que $\max(Dif)/c$ seja menor, aumentando a tolerância quanto ao menor incremento exigido. No caso de suspeitas incorretas, o valor de c pode ser incrementado, minimizando a possibilidade de erros. Assim, obtemos um detector que se ajusta para minimizar suspeitas incorretas e que se torna mais brando em períodos de sobrecarga do sistema.

2.4.2 Consenso

Esta seção apresenta os principais algoritmos de consenso baseados em detectores de falhas encontrados na literatura. A figura do coordenador, processo que centraliza a comunicação, está presente em todos os algoritmos. Todos eles seguem o paradigma do *rodízio de coordenadores*, isto é, alternam o papel de coordenador entre os diversos processos. O algoritmo de Brasileiro [BGMR00, BGMR01], também apresentado, é exceção tanto para o uso de detectores de falhas, quanto para o paradigma de *rodízio de coordenadores*. Sua presença aqui se justifica pela importância de algumas propriedades de que faz uso.

Chandra e Toueg

O algoritmo de Chandra e Toueg [CT96] é executado em rodadas assíncronas, sendo cada uma dividida em quatro fases. A comunicação é totalmente centralizada em um coordenador, isto é, toda comunicação ou *parte de*, ou *é direcionada ao* coordenador. O coordenador c_p é pré-determinado, por exemplo, pela função $c_p = r_p \ mod \ n$, sendo r_p a rodada atual e n o número de processos que participam do consenso. O algoritmo considera a existência de um detector de falhas da classe $\Diamond S$.

Fase 1 : Todos os processos enviam sua estimativa corrente e o número da última rodada na qual a estimativa foi atualizada (*timestamp*), para o coordenador.

Fase 2 : O coordenador recolhe uma maioria ($\lceil \frac{n+1}{2} \rceil$) de estimativas e propõe um dos valores com o maior *timestamp*.

Fase 3 : Todos os processos esperam a proposta do coordenador, enviando uma mensagem de reconhecimento quando a recebem (Ack) e assumindo a proposta do coordenador como sua estimativa corrente. Se o processo, antes de receber a mensagem, suspeita que o coordenador falhou, então envia uma mensagem de $NAck$ para o coordenador. Ao final desta fase todos os processos, com exceção do coordenador, passam para uma nova rodada.

Fase 4 : O coordenador espera por $\lceil \frac{n+1}{2} \rceil$ mensagens. Se nenhuma mensagem recebida for do tipo $NAck$, o coordenador envia sua estimativa como decisão para todos os processos via *difusão confiável*. Caso contrário, procede para uma nova rodada. Qualquer processo que receba a decisão do coordenador decide-se automaticamente pelo mesmo valor.

Uma instância de execução do algoritmo, exemplificando seu padrão de mensagens é exibida na Figura 2.5. Mensagens da primeira e segunda rodadas trafegam ao mesmo tempo nos canais de comunicação. Na parte esquerda da figura, o algoritmo transcorre sem falhas. Na parte direita, o processo P_1, o coordenador inicial em ambas as execuções, falha.

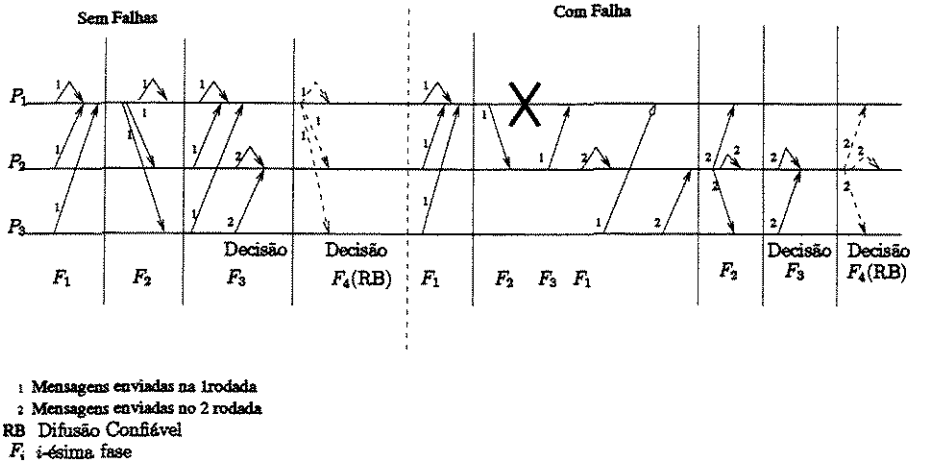


Figura 2.5: Algoritmo de Consenso proposto por Chandra e Toueg [CT96]

Uma possível e simples otimização para este algoritmo é a eliminação da primeira fase da primeira rodada, fazendo com que o coordenador tente impor sua estimativa aos outros participantes, diminuindo a latência do algoritmo [Sch97].

Uma propriedade interessante deste algoritmo é o travamento de um valor tão logo uma maioria o tenha aceitado, isto é, tão logo $\lceil \frac{n+1}{2} \rceil$ processos tenham aceitado um valor

como suas estimativas, este torna-se o único valor possível de ser decidido. Isso ocorre pois o coordenador, ao esperar uma maioria de estimativas e usar as que foram resultado de atualizações mais recentes, necessariamente escolhe este valor como sua proposição.

O algoritmo de Chandra e Toueg é facilmente ludibriado por suspeitas incorretas, que levam à emissão de *NAck*'s. Se um número suficiente de *Ack*'s foi gerado, mas um único *NAck* é recebido antes que todos os *Ack*'s tenham chegado ao coordenador, então a rodada é abandonada. Para minimizar este problema, pode-se adicionar uma lista de rodadas pendentes ao algoritmo e deixar um fio de execução (*thread*) independente tratar da chegada de *Ack*'s atrasados, permitindo a recuperação de rodadas erroneamente invalidadas e também que o coordenador avance mais rapidamente para uma nova rodada, acelerando a execução do protocolo como um todo. Esta alteração só é possível devido à propriedade do travamento da decisão, citada anteriormente, e o tamanho da lista de rodadas pendentes deve respeitar o compromisso "recursos (memória e processamento) X tempo de execução do algoritmo", sendo o *tamanho zero* equivalente ao protocolo original. Esta é a primeira vez que esta alteração é apresentada.

Schiper

O algoritmo proposto por Schiper [Sch97] tem como principal característica a descentralização do padrão de mensagens. O algoritmo pode ser descrito pelas seguintes fases:

Fase 1 : i)O coordenador envia sua estimativa para todos os processos. Cada processo que recebe a estimativa do coordenador reenvia-a a todos os outros processos. Se algum processo recebe a estimativa do coordenador por uma maioria de processos, decide-se por esta estimativa e a propaga. ii)Se algum processo passar a suspeitar do coordenador, então propaga esta suspeita para todos os demais. Qualquer processo que tenha recebido informes de suspeita de uma maioria de processos, passa para a Fase 2 e envia sua estimativa atual a todos os outros.

Fase 2 : Processos na Fase 2 coletam uma maioria de estimativas (para garantir que uma maioria de processos foi para esta fase) e aceitam estimativas que tenham vindo do coordenador. O próximo passo é prosseguir para a primeira fase da próxima rodada.

A Figura 2.6 mostra o padrão de mensagens do algoritmo na presença e na ausência de falhas. O caso com falhas pode gerar diversos padrões de mensagem pois soma-se ao assincronismo da rede, os possíveis instantes em que a falha ocorre: antes do coordenador enviar a primeira estimativa; entre o envio para diferentes receptores; e, após o envio. Na figura estão representados os casos em que a falha ocorre antes e após o envio da estimativa (no centro e à direita da figura).

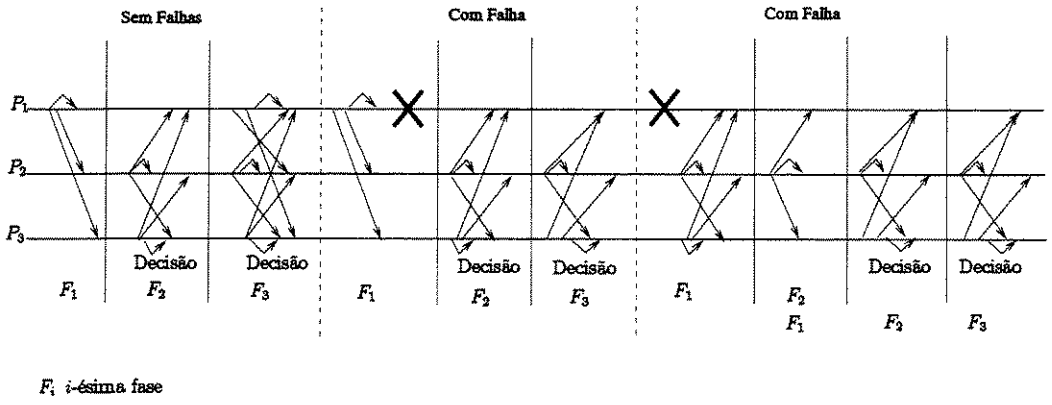


Figura 2.6: Algoritmo de Consenso proposto por Schiper [Sch97]

Como pode ser observado, este algoritmo possui menos passos de comunicação e é muito eficiente em sistemas nos quais haja a funcionalidade de *broadcast.*

Guerraoui

O trabalho de Guerraoui, Oliveira e Schiper [GOS98] apresenta variantes dos algoritmos de Chandra e Toueg [CT96] e de Schiper [Sch97], projetados para lidar com falhas por omissão de canais e de processos. Além disso, considera limitações nos buffers dos canais de comunicação, apresentando resultados pragmáticos e visivelmente aplicáveis em ambientes reais.

A confiabilidade em canais de comunicação é normalmente conseguida pelo reenvio de mensagens, mas o número de mensagens que permanece no *buffer* para ser reenviadas é limitado. Uma vez que o *buffer* esteja cheio, para aceitar novas mensagens, o canal deve descartar uma antiga. Cabe portanto ao algoritmo que usa o canal decidir quando uma mensagem já enviada, e possivelmente não entregue, deve ceder lugar a uma nova [GOS98]. Um canal de comunicação com *buffer* de tamanho k é chamado *k-stubborn*, e canais de comunicação confiáveis podem ser chamados ∞-*stubborn*.

Consenso baseado em Quorum

Mostefaoui e Raynal [MR99] apresentaram um algoritmo de consenso capaz de trabalhar com detectores tanto da classe $\Diamond S$, quanto da S.

O algoritmo calcula dinamicamente um conjunto Q_i de processos que precisam ter sua estimativa considerada para a obtenção do consenso, um quorum mínimo de opiniões. Ao usar detectores da classe S, Q_i corresponde ao conjunto de processos não suspeitos de falha pelo detector do processo. Como ao menos um processo correto nunca será suspeito por um detector da classe S, haverá sempre uma intersecção entre os conjuntos Q_i e Q_j

para qualquer par de processos corretos (p_i, p_j), condição necessária ao algoritmo. Quando considerando detectores da classe $\Diamond S$, Q_i precisa conter uma maioria de processos para garantir uma intersecção não vazia. Usar detectores S implica em permitir $f \leq n-1$, enquanto que $f < \frac{n+1}{2}$ é necessário quando do uso de detectores $\Diamond S$.

Este algoritmo pode ser considerado uma variante do algoritmo proposto por Schiper [Sch97], para suportar o uso de detectores S.

Paxos

Paxos [Lam98] é um algoritmo de consenso que tolera falhas na eleição de líderes (equivalentes aos coordenadores de outros algoritmos) necessária à chegada ao consenso. Neste algoritmo, um processo p que se julgue líder pergunta a uma maioria de processos P se estes estão dispostos a aceitar sua estimativa v_p. Caso suas respostas sejam afirmativas, o processo formaliza sua proposta, substituindo, caso necessário, v_p por v_i, onde v_i é a última proposta que os processos em P afirmaram estar dispostos a aceitar.

Os processos em P respondem afirmativamente a propostas que contenham *timestamps* maiores que a última proposta v_x que declarou estar disposto a aceitar e, junto a resposta, enviam também v_x. Ao receber uma formalização de uma proposta, estes processos em P a aceitam, caso não tenham se declarado propensos a aceitar propostas mais recentes que a proposta sendo formalizada, e repassam este aceite para todos os outros processos executando o algoritmo (uma versão otimizada deste algoritmo é também apresentada em [Lam98]). Processos decidem-se por um valor ao receber a confirmação de aceite de uma maioria de processos (P).

O processo de eleição de líderes pode ser executado por um detector de falhas denominado Ω que, ao invés de apontar uma lista de processos suspeitos de falhas, aponta um processo que não é suspeito, sendo facilmente criado a partir de um detector de falhas não confiável.

Brasileiro

Este algoritmo representa uma possível otimização no uso de qualquer outro algoritmo de consenso. Basicamente, tenta verificar se uma maioria absoluta ($> 2/3$) propôs o mesmo valor, decidindo por este valor ainda no primeiro passo. Quando esta propriedade não é verificada, recorre ao uso de outro algoritmo de consenso qualquer (Veja Figura 2.7). A grande limitação deste algoritmo é sua baixa resiliência (número de falhas suportado), $f < n/3$.

Observe que qualquer processo que decide no primeiro passo, decide-se pelo valor proposto pela maioria, e que se algum processo decidiu no primeiro passo, então todos os outros propõem este mesmo valor para o algoritmo de consenso usado no "nível" inferior.

```
function Consensus (v_i)
Task T_1 :
broadcast PROPOSED(v_i)
wait until ((n − f) PROPOSED messages have been received)
if these messages carry the same estimate value v then
  broadcast Decision(v)
  return(v)
else
  if (n − 2f) PROPOSED messages carry the same estimate value v then
    v_i ← v
  end if
  return(Underlying_Consensus(v))
end if
Task T_2 :
upon reception of Decision(v)
broadcast Decision(v)
return(v)
```

Figura 2.7: Algoritmo de Consenso proposto por Brasileiro [BGMR00, BGMR01]

Dada a propriedade de validade do consenso, este valor é decidido por todos os outros processos.

Assumindo que alguns valores tem mais *força* que outros (aborto é mais forte que *commit*), ou que alguns processos tem maior peso que outros, é possível baixar a restrição de $f < n/3$ para $f < n/2$ [BGMR00, BGMR01].

Outros Trabalhos

Nas seções anteriores apresentamos uma série de algoritmos de consenso baseados em detectores de falhas não-confiáveis. Esta lista, embora longe de ser exaustiva, referencia alguns dos trabalhos mais representativos da área. Estes trabalhos foram os primeiros, mas outros podem ser citados:

- Aguilera *et. al* [ACT98] aborda o consenso considerando o modelo de falhas *crash/-recovery*.

- uma série de trabalhos segue a linha do trabalho de Mostefaoui e Raynal [MR99], permitindo a configuração do algoritmo para trabalhar em diferentes ambientes. Hurfin *et. al.* [HMR00] apresenta um protocolo de consenso que pode trabalhar tanto com detectores $\Diamond S$ quanto com S, e utilizar tanto um padrão de mensagens centralizado quanto um distribuído. Mostefaoui, Rajsbaum e Raynal [MRR01]

apresentam um protocolo que pode trabalhar com combinações de detectores não confiáveis, aleatórios e de líderes, e com diferentes graus de resiliência. O trabalho de Aguilera e Toueg [AT96] combina detectores de falhas não confiáveis e oráculos aleatórios.

- Hurfin e Raynal [HR97] apresentam um algoritmo descentralizado semelhante ao *Early Consensus* de Schiper [Sch97], tendo como diferenças o fato de confiar mais nos detectores de falhas que usa, segundo Guerraoui [RMA+00], e de tentar remover o passo de propagação da opinião do coordenador quando avança para uma nova rodada.

- Rodrigues [Rod03], Pereira *et. al.* [PRO03] e Vicente e Rodrigues [VR02] apresentam trabalhos que consideram a semântica da aplicação e abordagem otimista para entrega confiável e ordenada de mensagens e, embora não tratem diretamente de consenso, abrem possibilidades para o uso daquelas abordagens também neste problema.

Capítulo 3

Framework para Detecção Adaptativa de Falhas

No estudo dos detectores de falha apresentados no Capítulo 2, percebe-se alguns comportamentos básicos que os compõem. A partir desta percepção, foi construído um *framework* simples para a composição de detectores de falhas com comportamento adaptativo. Este *framework* é descrito a seguir.

Em primeiro lugar, foram identificados os seguintes módulos em um sistema de detecção de falhas:

Monitor - Componente que observa outros objetos tentando identificar suas falhas.

Monitorado - Componente sujeito a falhas, as quais devem ser reportadas ao sistema

Máquina de Eventos - Componente que gera eventos indicando recepção de mensagens ou atraso na mesma. O Monitor é o consumidor destes eventos.

3.1 *Framework*

O objeto monitorado é provavelmente o mais simples e que requer menos adaptações para que um detector ou outro seja instanciado. Em momentos pré-estabelecidos, envia mensagens para o objeto monitor. Para tanto, basta conhecer o intervalo que deve haver entre o envio de mensagens (entre_envios) e um referencial para o instante zero (zero).

```
class Monitorado {                                                    1
  private:                                                            2
  bool monitorar = false;                                             3
                                                                      4
  void iniciar(time_t entre_envios, time_t zero) {                    5
    monitorar = true;                                                 6
    ulongint id = 1;                                                  7
```

```
    while(monitorar) {                                          8
      wait(zero + (id * entre_envios));                         9
      heartbeat(id);                                            10
      id++;                                                     11
    }                                                           12
  };                                                            13
  ...                                                           14
};                                                              15
```

As mensagens enviadas pelo objeto monitorado podem gerar dois tipos de eventos: VIVO, indicando a chegada de uma mensagem dentro do intervalo de tempo em que é esperada, e ATRASADO, indicando que o limite de tempo de espera de uma mensagem foi ultrapassado. Estes eventos podem ser definidos assim:

```
#define VIVO 1;                                                 1
#define ATRASADO 2;                                             2
                                                                3
#typedef struct {                                               4
  ulongint ID;                                                  5
  time_t instante;                                              6
  enum tipo {VIVO, ATRASADO};                                   7
                                                                8
  union {                                                       9
    time_t limite;                                              10
  } extra;                                                      11
} evento_t;                                                     12
```

Estes eventos são gerados por um fio de execução especializado do lado do objeto monitor, que conhece os limites de tempo declarados para a recepção de eventos VIVO e o tipo de eventos em que o objeto está interessado em dado momento.

```
void espere_evento(evento_t *evento, int mascara, Monitor *mtr, Monitorado *mtd);   1
```

Internamente a este procedimento, mensagens antigas são descartadas, não gerando eventos VIVO injustificadamente. Este procedimento não é adotado por todos os detectores disponíveis na literatura, não permitindo sua instanciação através deste *framework*. Mas sem dúvida, a incorporação desta característica leva a uma maior precisão na detecção de falhas, levando portanto, à variações mais eficientes destes detectores.

O monitor possui um série de variáveis que configuram seu funcionamento, e que são acessíveis ao código do método `espere_evento`.

```
class Monitor {                                                 1
  private:                                                      2
  //Quem monitorar?                                           3
  Monitorado *mtd;                                              4
                                                                5
  //Qual o contador da última mensagem recebida?              6
  ulongint ult_recebido;                                        7
                                                                8
  //mtd é suspeito de falha?                                  9
  bool suspeito = false;                                        10
                                                                11
  //Rodar laço de monitoração?                                12
  bool monitorar = false;                                       13
```

```
14
15  //Para quando é a próxima mensagem?
16  time_t limite = 0;
17  //Tempo extra para  a chegada da mensagem.
18  time_t delta;
19  //Fator de incremento de delta.
20  time_t delta_inc;
21
22  //Último atraso de comunicação calculado.
23  time_t atraso;
24
25  //Período entre envios de mensagens do monitorado (Eta).
26  time_t entre_envios;
27  //Instante inicial no processo de monitoração.
28  time_t zero;
29
30  //Tipos de eventos a serem esperados.
31  bitmasc_t masc;
32
33  //Referência para o último evento.
34  evento_t *evento;
35
36  //Função de ajuste dos parâmetros do algoritmo.
37  Adaptador adaptador;
38
39  //Histórico de chegada de mensagens.
40  ArrayCircular *historico;
```

Estes parâmetros serão necessários ou constantes para algumas instâncias de detectores. O procedimento de monitoração é descrito a seguir.

```
1  void Monitore(&Monitorado mtd,
2                time_t zr,
3                time_t dlt,
4                time_t dlt_inc,
5                time_t e_envios,
6               &Adaptador adp,
7                int tam_hist)
8  {
9    ID ultimo_recebido = 0;
10   suspeito = true;
11   monitorar = true;
12
13   monitorado = mtd;
14   zero = zr;
15   delta = dlt;
16   delta_inc = dlt_inc;
17   entre_envios = e_envios;
18   adaptador = adp;
19   historico = new ArrayCircular(tam_hist);
20
21   Monitorado.iniciar(entre_envios, zero);
22
23   masc = VIVO;
24
25   while (monitorar) {
26     espere_evento(&evento, masc, this, &monitorado);
27
```

```
      if(evento->tipo == ATRASADO) {                                    28
        //Se nao era suspeito, passa a ser.                             29
        if(!suspeito) {                                                 30
          suspeito = true;                                              31
          masc = VIVO;                                                  32
        }                                                               33
      } else if(evento->tipo == VIVO) {                                 34
        historico.add(evento);                                          35
        if(adaptador != null)                                           36
          adaptador.estimar(this);                                      37
                                                                        38
        masc = VIVO|ATRASADO;                                           39
                                                                        40
        //Se era suspeito antes.                                        41
        if(suspeito) {                                                  42
          delta += delta_inc;                                           43
          suspeito = false;                                             44
        }                                                               45
        limite += delta;                                                46
      }                                                                 47
    }                                                                   48
  }                                                                     49
  ...                                                                   50
};                                                                      51
```

A cada evento VIVO o algoritmo tem a chance de ajustar seus parâmetros através do Adaptador. É neste trecho que os algoritmos se diferenciam, acessando e, possivelmente, modificando o estado interno do Monitor.

No código seguinte vê-se um exemplo de algoritmo de adaptação, que usa o último evento VIVO recebido para calcular o tempo de chegada do próximo evento.

```
class basico: Adaptador {                                               1
  static void estimar(&Monitor mtr) {                                   2
    atraso = mtr->evento->instante - (zero + mtr->evento->id * mtr->entre_envios);   3
    mtr.limite = zero + (mtr->evento->id + 1) * mtr->entre_envios + atraso;          4
  };                                                                    5
};                                                                      6
```

Este estimador corresponde ao estimador de [CTA02], conforme apresentado em [BMP02], considerando histórico igual a um.

Esta função poderia fazer uso da referência que o monitor tem do objeto monitorado, e ajustar também seus parâmetros. No entanto, é necessário ter-se em mente que o tempo de propagação de mensagens pode ser diferente em cada um dos sentidos e que estes ajustes só surtirão efeito após sua propagação para o objeto monitorado.

3.2 Utilização

O *framework* apresentado pode ser utilizado para a instanciação de detectores de falhas adaptativos ou não. Sua instanciação é efetuada pela invocação do método `Monitore`, definido previamente. Este método define variáveis que vão ser utilizadas para ajustes do

detector, como o fator de incremento do *timeout*, delta, e um fator de incremento deste delta. Todavia, o parâmetro mais importante é a função de adaptação do algoritmo que pode, inclusive, ser nulo.

A definição deste método de forma tão genérica permite que uma série de algoritmos sejam implementados com uma mesma interface, facilitando o desenvolvimento de código que utilize estes detectores, e possibilitando a troca dos mesmos sem que haja necessidade de recodificação. A idéia presente aqui é que detectores de falhas são peças menores, que devem ser utilizadas para a composição de serviços mais complexos, como um serviço de detecção de falhas.

Capítulo 4

CORBA e FT-CORBA

Common Object Request Broker Architecture (CORBA) é uma infra-estrutura para desenvolvimento de sistemas distribuídos que provê transparência de localização e de linguagem na invocação de métodos de objetos distribuídos. Objetos CORBA são entidades identificáveis que provém um ou mais serviços que podem ser requisitados por um cliente. Estes objetos ficam acoplados a um barramento virtual distribuído provido por um ORB, *Object Request Broker*. Com o objetivo de mascarar também a ocorrência de falhas através do ORB, a especificação CORBA foi estendida em sua versão 3.0 pela especificação FT-CORBA, que trata da replicação de objetos e sua manipulação para tentar tornar as falhas transparentes ao cliente.

Este capítulo apresenta inicialmente a arquitetura CORBA e, posteriormente, a de FT-CORBA.

4.1 Arquitetura

As invocações de métodos de objetos CORBA trafegam pelo ORB, propiciando à aplicação cliente, semântica parecida àquela do acesso a objetos locais. Mais ainda, o barramento mascara a linguagem de implementação dos objetos. Desta forma o modelo cliente/servidor é abstraido, simplificando o desenvolvimento de aplicações distribuídas.

Além de tratar de mecanismos de invocação remota, CORBA define também uma série de peças para construção de aplicações distribuídas como serviços de objetos, facilidades comuns e interfaces de domínios.

Serviços de objetos são peças básicas como serviço de nomes, eventos, *trading* e concorrência.

Facilidades comuns são peças mais complexas, mas também com finalidades gerais como internacionalização e suporte a agentes móveis.

Interfaces de domínio estão em um nível de abstração mais alto e têm seu uso restrito a certos domínios de problema como transporte, finanças e telecomunicações.

Apesar da abrangência do modelo, ele não fornecia inicialmente mecanismos para tolerância a falhas. Depois de alguns trabalhos tentarem suprir esta deficiência [Fel98, NGS00], CORBA acabou por ser estendida com a especificação de FT-CORBA (*Fault Tolerant CORBA*)[OMG01a, OMG01b], abordada na seção seguinte.

Como alternativas de abordagens para a adição de funcionalidades ao CORBA, temos:

Abordagem de Integração - a adição direta da funcionalidade ao cerne do ORB;

Abordagem de Serviço - a adição de objetos de serviço com a funcionalidade;

Abordagem de Interceptação - a interceptação de invocações à funcionalidade e *tradução* da mesma para os mecanismos já presentes no ORB; e,

combinações destas abordagens.

Embora possa ter problemas de desempenho, acreditamos que a abordagem de serviços é a melhor para a implementação de FT-CORBA e de outras funcionalidades em geral. Esta abordagem permite o desenvolvimento de soluções mais portáveis, podendo inclusive ser usada em ORB's antigos, minimizando o custo da solução, embora isso não seja possível para o caso específico de FT-CORBA.

A especificação CORBA define alguns princípios para o projeto de objetos de serviços e suas interfaces [OMG97].

Modelo CORBA - Serviços são compostos por um ou mais objetos CORBA, e cada objeto possui uma única interface. Os serviços devem ser construídos segundo alguns preceitos do modelo CORBA como: separação entre interfaces e implementação; a definição do tipo do objeto feita pela interface que implementa; uso de herança múltipla de interfaces; clientes são dependentes de interfaces e não de implementações; e, uso de sub-tipos como mecanismo de extensão, evolução e especialização.

Serviços básicos e flexíveis - Serviços devem solucionar bem *um* problema, da forma mais simples possível. Funcionalidades complexas devem ser providas pela combinação de serviços.

Serviços Genéricos - Não deve haver dependência entre o tipo de dado que é manipulado e o serviço.

Implementações locais e remotas - Serviços devem ser estruturados como objetos CORBA, que podem ser acessados local ou remotamente.

Qualidade de serviço é uma característica da implementação - A interface é independente de requisitos de qualidade de serviço, *i.e.*, a interface não varia para implementações que provêem diferentes qualidades de serviço.

Objetos compõem um serviço - O serviço deve ser construído de forma a prover diferentes interfaces para os diferentes tipos de clientes.

Interfaces para *callback* - Quando o serviço exigir que o cliente execute alguma operação, deve disponibilizar uma interface que defina esta operação, a ser implementada pelo cliente.

Ausência de Identificadores Globais - Os serviços não devem se basear na existência de identificadores globais para elementos. Quando existirem, identificadores devem ser restritos a um contexto.

Busca é ortogonal ao uso de um serviço .

4.2 Tolerância a Falhas

Tolerância a falhas em FT-CORBA é alcançada pelo uso de redundância de software e detecção e recuperação de falhas, isto é, pela replicação de objetos e por mecanismos que permitam detectar falhas nestes objetos e substituí-los, em caso de falhas, por objetos corretos. Os principais conceitos necessários ao entendimento de tais mecanismos são apresentados a seguir.

Replicação e Grupos de Objetos - Os objetos CORBA são replicados e combinados em grupos de objetos (objeto replicado), identificados por um *Interoperable Object Group Reference*, IOGR. Este IOGR é composto por informações que identificam e possibilitam o acesso aos membros do grupo. A abstração de grupos permite que a replicação e a ocorrência de falhas em objetos servidores sejam transparentes à aplicação cliente.

Domínios de Tolerância a Falhas - Vários objetos replicados podem ser colocados em um mesmo domínio e controlados por um mesmo gerente de replicação, conseguindo assim, maior escalabilidade e facilidade de controle.

Propriedades de Tolerância a Falhas - Cada objeto replicado é associado a um conjunto de propriedades de tolerância a falhas (p. ex. número mínimo de réplicas). Estas propriedades ditam o gerenciamento dos objetos replicados.

Consistência Forte de Réplicas - Objetos replicados devem manter seus estados de forma consistente, ainda que sejam feitas requisições ou que falhas ocorram. Isto quer dizer que quando um objeto falha e outro assume (replicação passiva), este deve ter exatamente o mesmo estado que tinha aquele antes da falha. E que se vários objetos estão atendendo requisições em paralelo (replicação ativa), então devem chegar a um mesmo estado ao final do atendimento de cada requisição.

A atuação dos componentes de FT-CORBA pode ser dividida em gerenciamento de replicação, detecção de falhas e recuperação de falhas, discutidos a seguir.

Gerenciamento de Replicação - Consiste do gerenciamento dos grupos de objetos e da criação e remoção de objetos destes grupos baseado nas propriedades de tolerância a falhas. Um objeto gerente de replicação, **Replication Manager**, é responsável por estas atividades em cada domínio de tolerância a falhas.

Detecção de Falhas - Objetos **FaultDetector** monitoram os objetos replicados, e iniciam a cadeia de criação, propagação e análise de relatórios de falhas.

Recuperação de Falhas - Para a recuperação de falhas e manutenção da consistência, FT-CORBA define mecanismos de captura de estado de uma réplica, e formas de transferir este estado para outras réplicas.

A Figura 4.1 apresenta a arquitetura de FT-CORBA. O **Replication Manager** gerencia os grupos de objetos controlando a criação e destruição das réplicas ($\mathbf{S}_i$) através das fábricas (**F**). Quando necessário, o controle destes grupos pode ser assumido pela aplicação, controlando o ciclo de vida dos objetos através da interface `GenericFactory`, disponibilizada também pelo **Replication Manager**.

Detectores de falhas (**FD**) são dispostos em dois níveis: locais, que monitoram diretamente os objetos do sistema; e globais, que monitoram os próprios detectores locais, para impedir que suas falhas mascarem falhas em outros processos. Estes detectores informam falhas ao **Fault Notifier**, que as processa e repassa ao **Replication Manager** ou a qualquer outra aplicação que tenha se registrado para tal, junto ao **Fault Notifier**. Nada impede que outros níveis sejam adicionados ao esquema de detecção de falhas.

Interfaces especiais devem ser implementadas pelas réplicas, permitindo que seu estado interno seja capturado e re-inserido pelos mecanismos de de *Logging* e *Recovery*.

Algumas limitações de FT-CORBA são apontadas pela própria especificação, e se devem à superficialidade intencional com que alguns tópicos são tratados. Esta abordagem permite o desenvolvimento de extensões e soluções proprietárias para problemas específicos, mas limita a compatibilidade com sistemas legados e com ORBs de outros fabricantes. Como exemplo destas limitações pode-se citar a exigência do uso de um mesmo

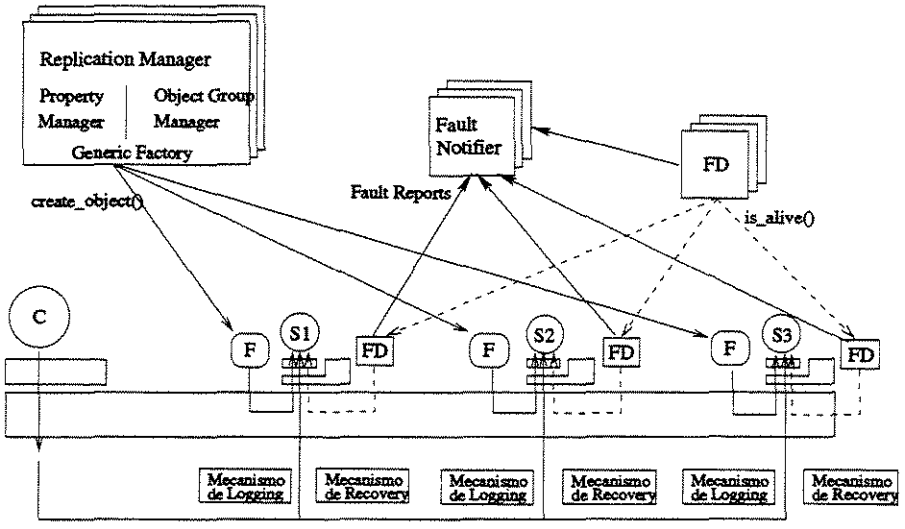


Figura 4.1: Arquitetura FT-CORBA.

ORB para todos os objetos replicados de um mesmo domínio de tolerância a falhas. Esta exigência vai claramente contra a independência proposta por CORBA.

Outras limitações são a necessidade de comportamento determinístico por parte das aplicações, pois de outra forma não seria possível manter a consistência das réplicas, e a não abordagem de falhas de particionamento de rede e falhas correlacionadas, devido a complexidade do tratamento destes problemas. Ainda, a especificação não define claramente como efetuar o controle dos grupos de objetos que compõem objetos replicados.

Operações como adição e remoção de réplicas, de um grupo, comportamento determinístico e consistência são terreno fértil para utilização de consenso, e é justamente esta abordagem que exploramos neste trabalho. A Seção 5.2.6 aborda, de forma mais detalhada, como um serviço de consenso distribuído (em especial o apresentado aqui) pode ser utilizado na implementação de FT-CORBA.

Implementações de FT-CORBA não são ainda abundantes, apesar desta especificação ser de 2001. Diana [SNT02] apresenta uma implementação de FT-CORBA e testes para determinação da sobrecarga introduzida pela tolerância a falhas na execução normal do sistema. Diana [SRNTG03] estende seu trabalho prévio [SNT02], introduzindo um algoritmo de consenso (Paxos [Lam98]) para obter acordo sobre a resposta a uma requisição do cliente, acordo sobre o estado interno dos processos e não determinismo nos servidores.

GroupPac [LFSP00, LCLPS01, UFS03] é uma infra-estrutura para construção de aplicações tolerantes a falhas em CORBA. Além de aderir à especificação Fault Tolerant CORBA, GroupPac propõe extensões e adaptações necessárias ao desenvolvimento de sistemas de larga escala tolerantes a falhas.

IRL [BCP02], da Universidade de Roma, é uma implementação de FT-CORBA totalmente portável e que permite a integração de objetos utilizando diferentes ORBs em um mesmo objeto replicado. DOORS [NGS00], por sua vez, é uma infra-estrutura desenvolvida em paralelo a especificação FT-CORBA e que foi adaptada para se tornar aderente a esta. Esta tarefa foi simplificada pelo fato de DOORS ter contribuído em muito naquela especificação.

Capítulo 5

Serviços de Detecção de Falhas e Consenso Distribuído

"Computadores não resolvem problemas, executam soluções."
Laurent Gasser

Este capítulo apresenta a arquitetura dos serviços de consenso distribuído e detecção de falhas propostos no contexto desta dissertação. O projeto desta arquitetura e sua pormenorização na forma de um *framework*, também descrito neste capítulo, foram pautados nos seguintes requisitos:

Objetos Selecionáveis - Os serviços devem ser compostos de peças menores, e diferentes implementações de uma mesma peça devem ser possíveis. O usuário deve poder escolher dentre estas diferentes implementações a que melhor atenda aos seus requisitos de qualidade de serviço e desempenho.

Serviços Independentes - Seguindo o mesmo princípio de *Objetos Selecionáveis*, os serviços devem ser construídos de forma a permitir seu uso independentemente. Como o serviço de consenso está *sobre* o serviço de detecção, diferentes implementações deste devem poder ser utilizadas por aquele.

Extensibilidade - Possibilidade de adição de novos módulos, com novas implementações para uma interface. Isto aumenta a usabilidade do sistema, permitindo sua adaptação a ambientes para os quais não havia sido inicialmente pensado, e a otimização de módulos já existentes.

Suporte a Detectores de Falhas Adaptativos - Requisitos de qualidade de serviço em ambientes reais, isto é, sujeitos a variações de carga, implicam em adaptação do sistema em tempo de execução. Detectores adaptativos têm comportamento e

interface diferentes das abordagens convencionais, mas devem ser contemplados no desenvolvimento de sistemas reais.

Compatibilidade com FT-CORBA - Dada a grande difusão de CORBA no desenvolvimento de sistemas distribuídos, a compatibilidade com esta especificação aumenta consideravelmente a usabilidade deste modelo.

Quanto ao modelo computacional, apesar de termos apresentado seus diferentes tipos (Seção 2.2), não definiremos um em específico a ser seguido. Tal modelo deverá ser definido pelo usuário de nossa arquitetura, resultando em restrições quanto aos módulos do sistema, com implementações de diferentes algoritmos, a serem selecionados.

5.1 Arquitetura dos Serviços

A baixa troca de informação e a independência lógica entre os algoritmos de consenso e detecção de falhas permitem que os serviços sejam implementados independentemente, sem prejuízos ao desempenho. Esta estrutura possibilita ainda o uso dos serviços em ambientes especializados e sua distribuição entre várias estações. Exemplos de utilização desta funcionalidade incluem o uso de hardware detector de falhas especializado e a disposição dos serviços em estações com diferentes propriedades, como por exemplo a presença de canais confiáveis (úteis ao serviço de consenso mas não ao serviço de detecção de falhas [EJP01]) e canais síncronos (que possibilitam a construção de detectores de falhas com precisão mais forte).

A Figura 5.1 apresenta a arquitetura dos serviços propostos para consenso distribuído e detecção de falhas, sendo que alguns objetos e relacionamentos duplicados foram propositalmente omitidos para simplificar a visualização. Nesta figura podemos observar os três componentes principais da arquitetura: **Cliente**; DisCusS, serviço de consenso; e FuSe, o serviço de detecção de falhas. **Cliente**, é a aplicação que faz uso do serviço de consenso para resolver algum problema de concordância. O serviço de consenso, ao receber uma invocação do **Cliente**, executa um algoritmo de consenso e devolve o resultado para o **Cliente**, e para tanto, utiliza-se dos detectores de falhas disponibilizados pelo serviço de detecção de falhas, responsável por monitorar todos os objetos importantes do sistema e por notificar suas falhas.

As seções seguintes pormenorizam cada um destes serviços e especificam a interação do **Cliente** com os mesmos.

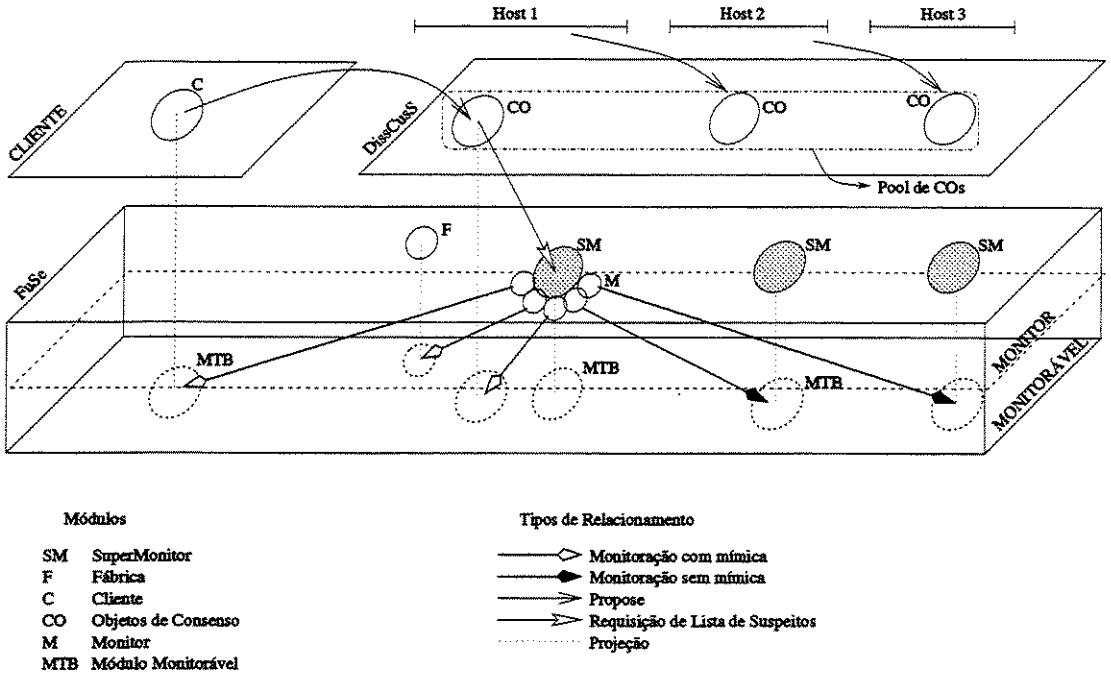


Figura 5.1: Arquitetura.

5.1.1 Serviço de Detecção de Falhas - FuSe

O serviço de detecção de falhas, FuSe, fica no nível mais baixo da arquitetura, servindo de anteparo às *projeções monitoráveis* (MTB) dos objetos dos níveis superiores. FuSe atua sobre estas projeções, não se importando com o tipo do objeto sendo monitorado. Os monitores (M), objetos que implementam os detectores de falhas, são quem efetivamente monitoram objetos, sempre com granularidade um para um. Os monitores são criados por fábricas (F) que disponibilizam diferentes tipos de monitores. Conjuntos de monitores (possivelmente heterogêneos) são gerenciados por super-monitores (SM), responsáveis por disponibilizar listas de objetos suspeitos de falhas. Quando necessário, o super-monitor pode imitar o comportamento de falha de objetos que esteja monitorando, generalizando falhas de um objeto para um grupo, e possibilitando a utilização de diversas granularidades de monitoração.

Os objetos que compõem o próprio FuSe também são monitorados e, quando necessário, replicados para eliminar a presença de pontos únicos de falha. O super-monitor, por exemplo, pode ser monitorado para impedir que uma falha sua leve à inutilização de todos os objetos que monitora. Fábricas são sempre monitoradas por monitores criados por outra fábrica, impedindo que suas falhas passem desapercebidas.

Os super-monitores reportam uma lista de objetos suspeitos de estarem falhos aos seus clientes, aplicações do nível superior ou outros super-monitores. Esta reportação pode ser

efetuada de forma solicitada (pull) ou quando o super-monitor julgar necessário (push). O tipo de monitor usado, por sua vez, é totalmente transparente ao **Cliente**.

5.1.2 Serviço de Consenso Distribuído - DisCusS

O serviço de consenso distribuído, DisCusS, permite que qualquer problema tradutível ao consenso possa ser resolvido.

Cada cliente (C) é associado inicialmente a um objeto de consenso (CO). Os clientes são os responsáveis por instanciar uma resolução do consenso, tendo duas alternativas para fazê-lo:

Instanciação Explícita - todos os clientes propõem suas estimativas para o valor de consenso a seus objetos de consenso, ou

Instanciação Implícita - um cliente indica ao *pool* de objetos de consenso (CO) que precisa resolver uma instância de consenso, e os objetos do *pool* se comunicam com seus clientes, obtendo suas estimativas.

Objetos de consenso podem, em algumas circunstâncias, usar valores padrões para suas estimativas, não necessitando entrar em contado com seus respectivos clientes.

Cada cliente necessita conhecer apenas seu objeto de consenso, enquanto os objetos de consenso precisam, além de conhecer seus clientes, conhecer uns aos outros. O próprio serviço pode ser usado para a remoção e adição de membros neste grupo, desde que sejam tomadas precauções para que estas operações não levem à parada do sistema (por exemplo, levando a um número muito pequeno de objetos, ou a diferentes ordens entre processamento de mensagens e operações de exclusão/inclusão de membros no grupo). Com o conhecimento mútuo entre os participantes, qualquer protocolo de comunicação pode ser estabelecido, permitindo a implementação de qualquer algoritmo de consenso.

Isolar os clientes dos objetos que realmente resolvem o consenso possibilita que o protocolo execute, mesmo quando vários destes clientes falharem, situação bastante plausível quando se considera o uso do serviço por clientes espalhados na Internet, por exemplo. Ainda, este isolamento permite o uso de desconexões planejadas dos clientes.

Cada instância de consenso possui um identificador associado. É responsabilidade dos clientes gerar identificadores distintos para instâncias diferentes de consenso, e apenas um identificador para uma mesma instância. Este requisito só é problemático quando uma mesma aplicação trabalha com mais de um problema de acordo ao mesmo tempo, podendo levar a situações em que uma estimativa possa ser aceita como decisão, e não ser aplicável a todos os clientes. Para contornar este problema, diferentes associações com o *pool* de objetos de consenso podem ser estabelecidas.

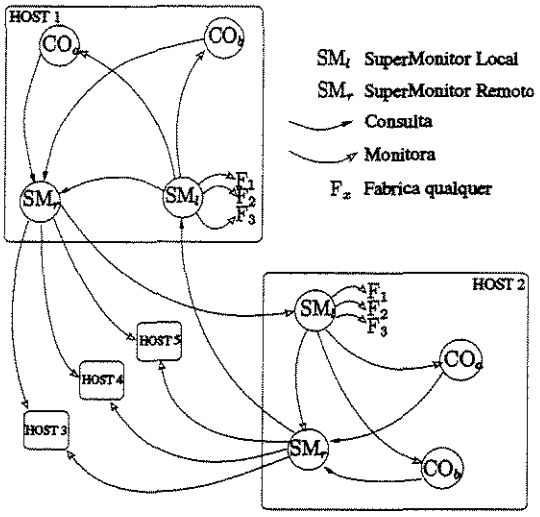


Figura 5.2: Exemplo de utilização de DisCusS + FuSe.

Os objetos de consenso são clientes do nível de detecção de falhas, fazendo consultas aos super-monitores. Um mesmo super-monitor pode ser compartilhado por mais de um objeto de consenso, e este último pode ser cliente de mais de um super-monitor. Considere a situação apresentada na Figura 5.2.

Neste exemplo, cada *host* usa dois super-monitores, responsáveis por monitorar objetos locais e objetos remotos. Estes super-monitores são compartilhados pelos objetos de consenso, que sabem como discernir entre os objetos que lhe são importantes e os que não são. A vantagem deste compartilhamento é a economia de recursos da rede, já que mensagens de um *host* para o outro são enviadas em conjunto. Neste caso, pode não convir ao super-monitor fazer monitoramento com mímica. Além disso, esta estrutura só faz sentido em condições especiais, como por exemplo, se o super-monitor responsável pelos objetos remotos estiver em um hardware dedicado.

5.2 *Framework*

O mapeamento da arquitetura proposta em um modelo orientado a objetos é relativamente direto. Esta seção, apresenta um *framework* resultante desta modelagem, segundo a notação UML [B.99], sendo que tal modelagem visa atender da melhor forma possível aos requisitos da arquitetura citados anteriormente.

5.2.1 Visão Geral

Algumas das características da arquitetura correspondem a soluções documentadas como *padrões de projeto* [GHJV94], que são descrições formais de soluções simples e reconhecidamente eficientes em projeto de software orientado a objetos. Por exemplo, graças ao *Strategy Pattern*, detecção de falhas e consenso são ocultados por uma interface genérica. Do lado monitorado, *Strategy* permite que um mesmo objeto responda a diferentes detectores, com requisitos de qualidade de serviço diferentes. Mais ainda, são facilmente cambiáveis, provendo meios para rápida reconfiguração, uma vez que todos são acessados por uma interface comum. O uso de fábricas abstratas [GHJV94] e de gerenciadores de detectores de falhas [SB96] permite tratar criação, manuseio e destruição de objetos em alto nível, tornando possível agrupar diferentes tipos de detectores (PULL e PUSH, adaptativos e não adaptativos), com interfaces comuns, trabalhando em paralelo e cooperativamente.

O diagrama de classes da Figura 5.3 está organizado em três partes: detecção de falhas (F), consenso distribuído (D) e controle (C). Cada uma destas partes é abordada nas próximas seções. Note que neste diagrama, classes com sufixo _i são implementações das interfaces de mesmo nome sem o sufixo. Estas seções apresentam também partes das definições em IDL (*Interface Definition Language*) das interfaces presentes no *framework*. As definições completas se encontram no Anexo B.

5.2.2 FuSe

Os super-monitores apresentados na arquitetura são mapeados para objetos **SuperMonitor_i**, que implementam as interfaces `SuperMonitor`, que lhes confere a capacidade de controle de um conjunto de monitores, e `Notifier`, que lhe dá a capacidade de enviar relatórios de falhas a clientes previamente registrados via esta interface.

O controle do conjunto de monitores, isto é, adição, remoção, ativação e inativação, em conjunto ou individual, são apresentados no código a seguir. Uma associação monitor/-monitorável é criada para cada objeto a ser monitorado e recebe um identificador único no contexto do super-monitor que a controla. A interface também especifica o protocolo para consulta sobre falhas de um objeto específico ou de todos os objetos monitorados.

```
interface SuperMonitor {                                                    1
  void start_monitoring_all();                                              2
                                                                            3
  void stop_monitoring_all();                                               4
                                                                            5
  void start_monitoring(in FUSEObjectUID uid);                              6
                                                                            7
  void stop_monitoring(in FUSEObjectUID uid);                               8
                                                                            9
  FUSEObjectUID add_monitoring_assoc(in GeneralMonitorStrategy monitor,     10
```

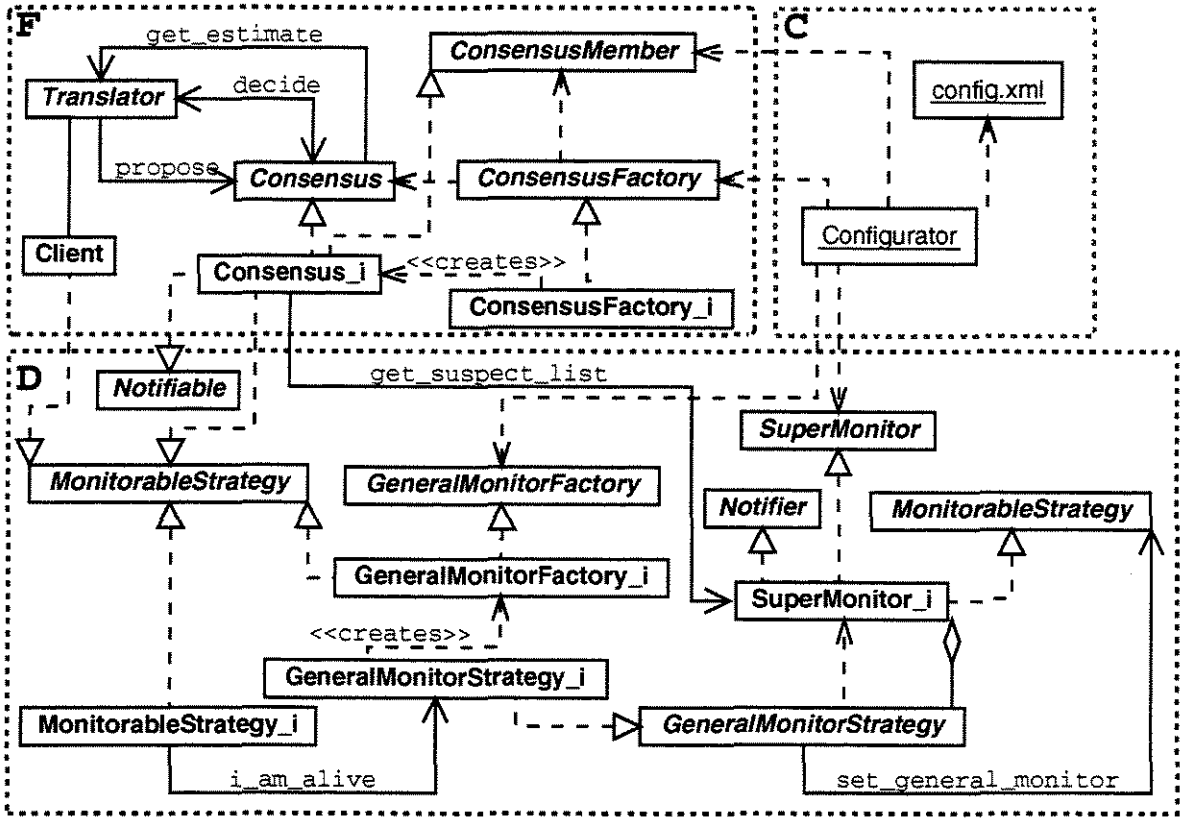


Figura 5.3: *Framework.*

```
                                in MonitorableStrategy monitorable);
  Boolean remove_monitoring_assoc(in FUSEObjectUID uid);

  Boolean is_it_suspect (in FUSEObjectUID uid);

  SuspectList get_suspect_list();
};
```

A interface `Notifier` apenas introduz a funcionalidade de registro como interessado em receber notificações de falhas, como se vê na IDL seguinte. Somente objetos notificáveis, isto é, que implementam a interface `Notifiable` podem se registrar.

```
interface Notifier {
  void  add_notifiable(in Notifiable ntb);
};
```

Como estão também sujeitos a falhas e também precisam ser monitorados, objetos **SuperMonitor_i** também implementam a interface `MonitorableStrategy`, que equivale à projeção monitorável descrita na arquitetura. Esta interface, como visto a seguir, disponibiliza métodos para configuração dos parâmetros de monitoração e para adição e remoção de monitores para o qual mensagens de *heartbeat* devem ser enviadas. Tais configurações

podem ser feitas dinamicamente em tempo de execução, característica importante para detectores adaptativos que manipulam o lado monitorado como ADAPTATION [SM01a] e DPCP [SM01b]. A interface apresenta também o método `is_alive`, que pode ser usado para monitoração PULL do objeto que a implementar.

```
interface MonitorableStrategy {
  boolean is_alive();

  oneway void set_monitoring_interval(in MonitorUID uid,
                                      in TimeBase::TimeT mi);

  TimeBase::TimeT get_monitoring_interval(in MonitorUID uid);

  oneway void run(in MonitorUID uid);

  oneway void stop(in MonitorUID uid);

  MonitorUID add_general_monitor(in GeneralMonitorStrategy gen_mon_stg);

  oneway void remove_general_monitor(in MonitorUID uid);
};
```

Monitores e fábricas de monitores são respectivamente mapeados para implementações das interfaces `GeneralMonitorStrategy` e `GeneralMonitorFactory`. Ambos também implementam a interface `MonitorableStrategy`. A interface `GeneralMonitorStrategy` disponibiliza métodos para ativação e desativação do monitor, bem como para recepção de mensagens de *heartbeat* do objeto monitorado. A interface `GeneralMonitorFactory`, por outro lado, apenas estabelece o protocolo genérico para a criação de monitores.

```
interface GeneralMonitorStrategy {
  void start();

  void stop();

  void set_super_monitor(in SuperMonitor sm);

  void set_monitorable(in MonitorableStrategy ms,
                       in FUSEObjectUID mtb_uid);

  oneway void i_am_alive(Long message_id);
  ...
};

interface GeneralMonitorFactory{
  GeneralMonitorStrategy create_general_monitor();
};
```

As interfaces `Notifier` e `Notifiable` permitem que objetos registrem-se para receber relatórios de falhas, podendo processá-los e reenviá-los quando necessário, montando esquemas hierárquicos de comunicação de falhas. Como os super-monitores implementam `Notifier`, seus clientes tem a possibilidade de receber relatórios de falhas tanto por informativos assíncronos, quanto perguntando ao super-monitor quando julgarem necessário.

A interface `Notifiable` consiste de apenas um método para recepção dos informativos.

```
interface Notifiable {                                          1
  void notify(in SuspectList suspect_list);                     2
};                                                              3
```

5.2.3 DisCusS

O lado cliente da arquitetura é mapeado em um objeto **Client**, associado a um objeto que implementa a interface `Translator`. Esta interface registra métodos que permitem que o objeto de consenso requeira a estimativa do **Client** e também que lhe comunique a decisão obtida pelo protocolo de consenso.

```
interface Translator {                                          1
  Value get_estimate(inout ConsensusID cid);                    2
                                                                3
  void decide (in ConsensusID cid, in Value decision);          4
};                                                              5
```

Objetos de consenso são mapeados em objetos do tipo **Consensus_i**, que implementam as interfaces `Consensus`, `ConsensusMember` e `Notifiable`. `Consensus` define o protocolo para instanciações implícitas (`launch`) e explícitas (`propose`) do algoritmo de consenso, e um método para que o cliente obtenha o resultado do algoritmo (`decide`).

```
interface Consensus {                                           1
  void launch (in Translator translator);                       2
                                                                3
  void propose (in Value estimate, in ConsensusID cid);         4
                                                                5
  Value decide (in ConsensusID cid);                            6
};                                                              7
```

Observe que um método para comunicação da decisão está presente em cada uma das interfaces anteriores, permitindo que a proposição para o algoritmo e a comunicação do resultado sejam totalmente assíncronos.

`ConsensusMember` define métodos para inclusão e remoção de outros objetos de consenso e de super-monitores no conjunto de objetos acessados por um `ConsensusMember`. Desta forma, o objeto toma conhecimento dos outros em seu mesmo *pool*, e de super-monitores que deve consultar para receber relatórios de suspeitas de falhas.

```
interface ConsensusMember {                                     1
  FUSEObjectUID add_super_monitor (in SuperMonitor sm);         2
                                                                3
  Boolean  remove_super_monitor (in FUSEObjectUID smid);        4
                                                                5
  void add_consensus_member(in ConsensusMember member,          6
                            in ConsensusMemberUID cmuid);       7
                                                                8
  Boolean remove_consensus_member(in ConsensusMemberUID cmuid); 9
};                                                              10
```

A complexidade do problema de concordância é ocultada pela implementação de `Translator`. É esta implementação a responsável por converter o problema em instâncias do problema de consenso. Dados dependentes do problema e um identificador para estes dados devem ser colocados em uma variável do tipo `Value`, que será usada como proposição para o valor de consenso. Este identificador pode ser um número *hash* do dado (o que permite a comparação de igualdade entre os valores), um identificador do processo cliente, ou ainda qualquer outra informação que possa ser utilizada pelo algoritmo de consenso. Desta forma, podem ser feitas tanto implementações genéricas de algoritmos de consenso, quanto implementações específicas para um tipo de dados dependente do problema. A estrutura `Value` é definida conforme a IDL seguinte.

```
struct Value {                                    1
  Long   ID;                                      2
  Any    data;                                    3
};                                                4
```

Para a comunicação entre objetos de consenso, protocolo algum foi definido. Desta forma, não se criam restrições à implementação dos algoritmos, devendo estes próprios proverem seus meios de comunicação.

5.2.4 Configurador

Os relacionamentos entre objetos são estabelecidos e mantidos por uma entidade externa, um *configurador*. No diagrama apresentado na Figura 5.3, o configurador é personificado pelo objeto **Configurator**. Cabe a este objeto estabelecer os relacionamentos iniciais e agir, se for o caso, na ocorrência de falhas. O diagrama de classes da Figura 5.3 foi organizado para evidenciar a dependência entre DisCusS, FuSe e o configurador. Entretanto, a existência desta entidade configuradora exige que o sistema passe por um período inicial sem ocorrência de falhas, possibilitando que todos os relacionamentos (como por exemplo: A *monitora* B; A, B, C e D *pertencem ao pool*; A *usa detectores do tipo* X, *criados por FacX, para monitorar* B) entre objetos sejam estabelecidos. Todas estas informações são passadas ao configurador no início da execução do sistema. Um arquivo de configuração exemplo é apresentado no Anexo A. Todavia, na implementação protótipo, estas configurações foram passadas como parâmetros de linha de comando.

5.2.5 Diagramas de Seqüência

Uma vez apresentados os objetos do sistema e suas inter-dependências, faz-se necessária a apresentação da interação entre estes objetos. Os diagramas a seguir apresentam a interação entre os objetos no nível de detecção de falhas (Figura 5.4) e de consenso (Figura 5.5).

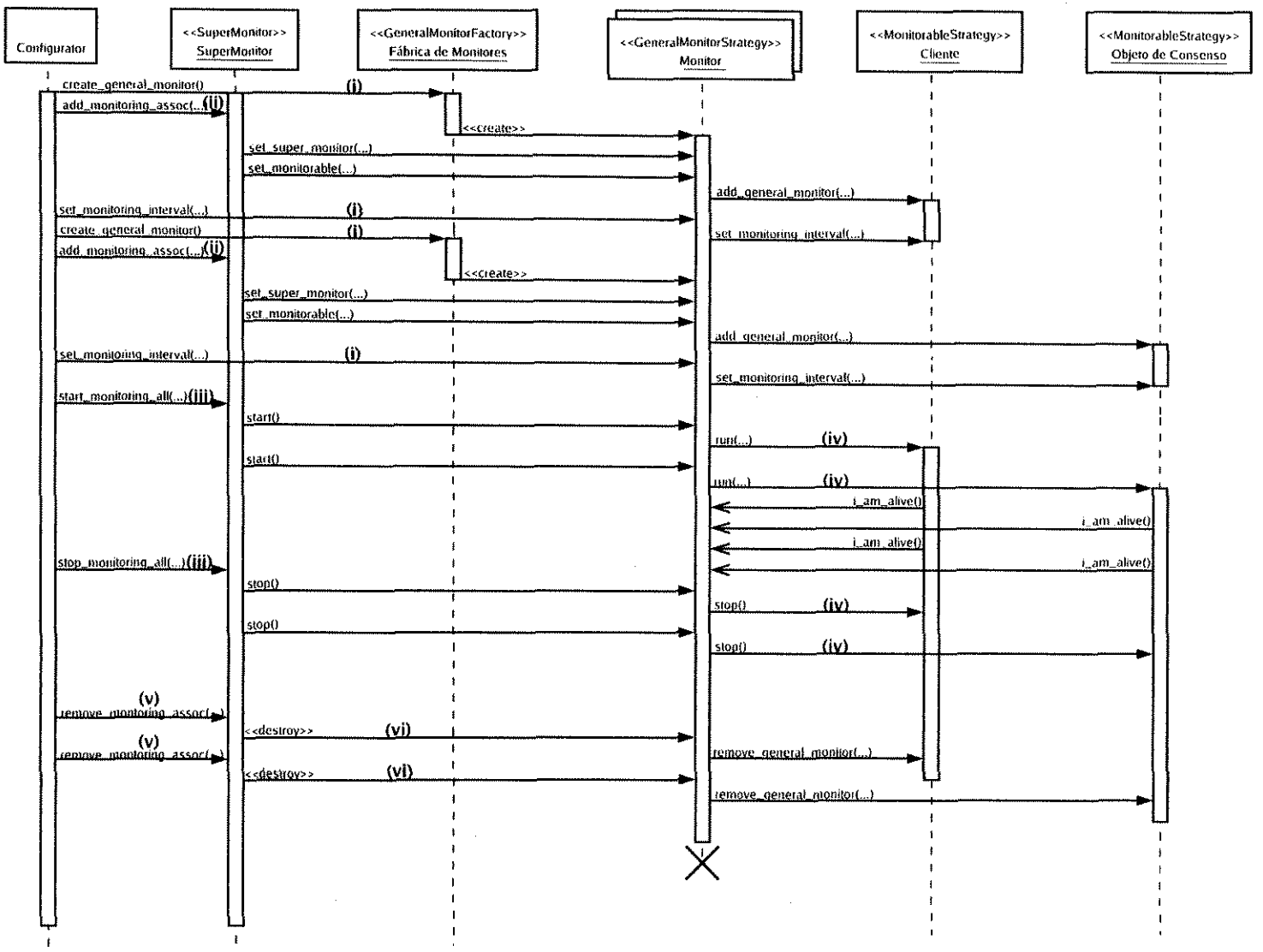


Figura 5.4: Interação entre objetos de FuSe.

Na Figura 5.4, vê-se o configurador inicializando os demais objetos (i) e indicando ao super-monitor quais os objetos devem ser monitorados e por quais monitores (ii). É também o configurador que, neste exemplo, inicializa e indica o início da execução dos monitores (iii) que, por sua vez, disparam a parte monitorável do cliente e dos outros objetos de consenso (iv).

Quando uma associação monitor/monitorável é eliminada (v), os monitores são automaticamente destruídos (vi).

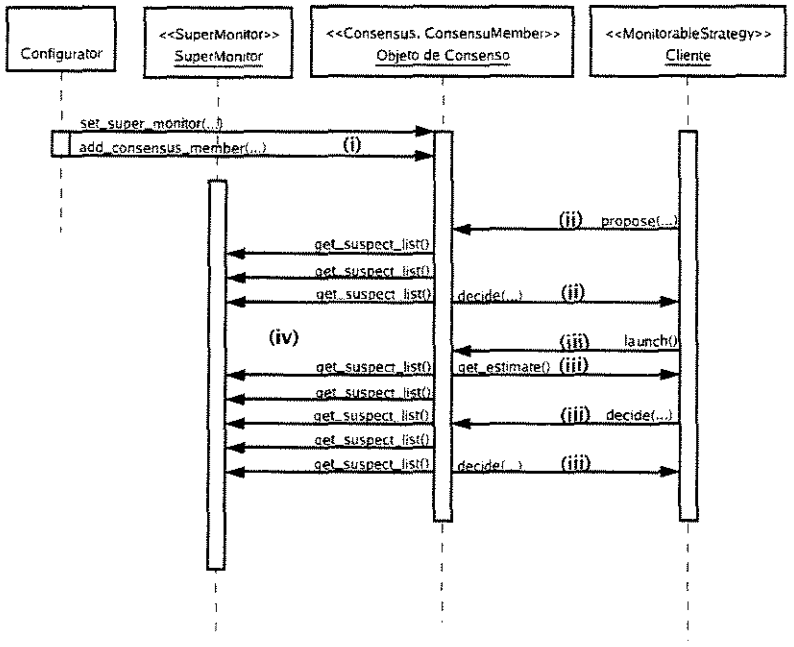


Figura 5.5: Interação entre objetos de DisCusS.

Na Figura 5.5, vê-se as mensagens trocadas pelo nível de consenso. Observe que algumas mensagens duplicadas foram eliminadas para simplificar a visualização (mensagens para adicionar outros membros do consenso (i)). Neste diagrama, podemos observar duas instâncias de consenso, sendo a primeira explícita (ii), e a segunda implícita (iii). Durante uma instância de consenso, o objeto de consenso consulta constantemente seu super-monitor (iv).

5.2.6 Implementando FT-CORBA

Nesta seção é mostrado como um serviço de consenso distribuído pode ser usado na implementação da especificação FT-CORBA.

Como citado na Seção 4.2, FT-CORBA permite a organização dos detectores de falhas de forma hierárquica. Este modelo é totalmente compatível com a arquitetura pro-

posta aqui. Processos podem ser monitorados por detectores distintos, tornando fácil a replicação de detectores globais. O **FaultNotifier** pode ser implementado como uma especialização de um detector de falhas ou por um super-monitor, uma vez que, pela implementação das interfaces `Notifier` e `Notifiable`, estão aptos a receber e enviar notificações de falha. Como foi idealizado, FuSe permite diversos esquemas de granularidade de monitoração, como i) um para um, ii) um para muitos, iii) muitos para um, iv) muitos para muitos ou misturas destes, conforme apresenta a Figura 5.6 (algumas setas para objetos replicados foram removidas para simplificar a visualização). A notificação de falhas pode ser configurada nos mesmos esquemas.

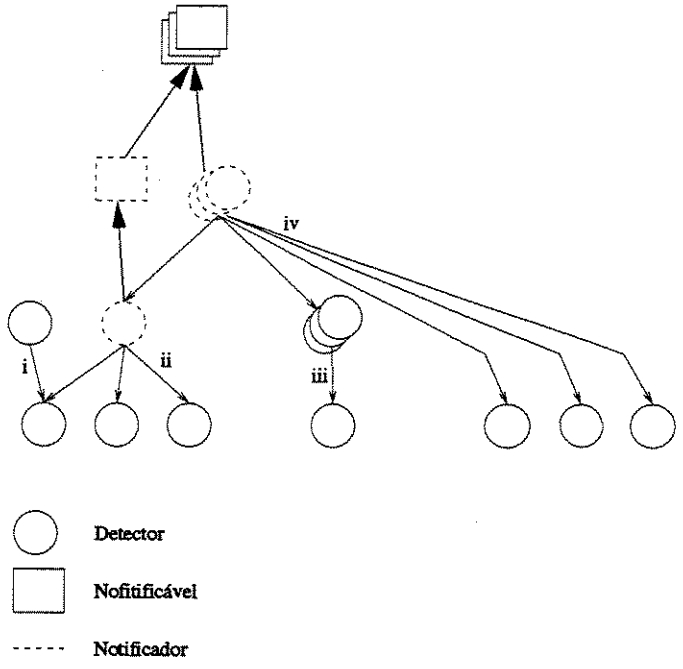


Figura 5.6: Exemplos de esquemas de monitoração e notificação.

FT-CORBA é baseado no modelo PULL de monitoração, no qual o detector de falhas periodicamente pergunta ao objeto monitorado se ainda está "vivo". A própria especificação prevê uma extensão para a utilização do modelo PUSH, mas ainda assim, nada diz com respeito a qualidade de serviço e a detecção adaptativa. O trabalho [SM01a] propõe variantes adaptativas para ambos os modelos. As definições para detectores de falhas feitas aqui, levaram em conta os modelos PULL e PUSH e suas variantes adaptativas, resultando na definição da interface `MonitorableStrategy`. A interface `PullMonitorable` definida em FT-CORBA pode ser vista como uma especialização desta interface.

Para que se eliminem pontos simples de falha (SPOF), FT-CORBA dita que **Fault Notifier** e **Replication Manager** devem ser replicados, ainda que devendo agir logicamente como apenas um objeto. Algoritmos de consenso podem ser úteis para manter

a consistência entre estas réplicas. O consenso pode ser usado também na resolução do problema de pertinência de grupo (*Group Membership*), executada pelo **Replication Manager**.

A manutenção da consistência entre *logs* na implementação dos mecanismos de recuperação é outro ponto de potencial uso do consenso, bem como para promover a entrega atômica de mensagens entre réplicas, garantindo a consistência forte entre as mesmas, para coordenar ações após a detecção de uma falha, e para implementar (ou mesmo substituir) o estilo de replicação `ACTIVE WITH VOTING` [OMG01b], apresentado na especificação.

5.3 Aspectos de Implementação

As implementações dos protótipos de FuSe e DisCusS usaram o ORB TAO [Sch02, SNG+03] (The ACE ORB), compatível com a especificação CORBA 2.6. TAO é um ORB construído sobre o *framework* ACE (Adaptive Communication Environment), focado no desenvolvimento de aplicações distribuídas com alto desempenho e em tempo real, que disponibiliza implementações de uma série de padrões de projeto na área de telecomunicações e redes, e que se casa perfeitamente com o foco deste trabalho: desenvolvimento de aplicações distribuídas com código reutilizável e alto desempenho. O uso de ACE simplificou a resolução de alguns problemas de implementação como gerenciamento de fios de execução, objetos ativos [LS95] e filas de eventos. A linguagem de desenvolvimento usada foi a C++, por razões de desempenho e disponibilidade de TAO.

O serviço de nomes do TAO, compatível com CosNaming [OMG01a], foi usado na localização de objetos, mesmo não sendo tolerante a falhas. O serviço de nomes é utilizado principalmente no início da operação do sistema. Como este período é estável, isto é, sem falhas (veja 5.2.4), não há a possibilidade de que este serviço seja causa de não funcionamento do sistema. Contudo, quando não se estende o período de estabilidade ao funcionamento do serviço de nomes, ou se considera a inclusão de novos objetos em tempo de execução mais adiantado, é necessário o uso de alguma implementação tolerante a falhas do serviço de nomes, como por exemplo [LFF+99].

O **Configurator**, responsável por estabelecer os relacionamentos entre objetos, informa aos objetos de consenso quais os outros objetos do mesmo *pool*. Estas informações são obtidas de um arquivo de configuração, como o exemplo apresentado no Anexo A.

Cada objeto de consenso conhece os identificadores de todos os objetos de consenso de seu *pool*. Estes identificadores são únicos e gerados previamente a execução do sistema, variando de 0 a $n-1$, sendo n o número de objetos de consenso no sistema. A atribuição destes identificadores aos objetos é feita pelo configurador.

Para permitir a execução concorrente de várias instâncias de consenso, foi implementado um mecanismo de multiplexação e enfileiramento das mensagens baseado nos

identificadores de instância de consenso que estas mensagens carregam, definido como `typedef Long ConsensusID`. Todas os métodos da interface `Consensus` utilizam este identificador para evitar ambiguidade (as definições destes métodos podem ser encontradas no Anexo B, na Seção "Definições Relacionadas ao Objeto de Consenso"). Vários fios de execução são compartilhados pelas instâncias de consenso, que desenfileram suas mensagens e processam-nas, descartam-nas caso não sejam mais úteis, ou reenfileram-nas caso possam ser úteis no futuro. A quantidade de fios de execução é ditada por políticas do próprio ORB [Sch02, SNG+03],

Para a implementação do algoritmo de consenso uma interface foi criada, estendendo as interfaces a serem implementadas por objetos de consenso (`ConsensusMember`, `Consensus` e `MonitorableStrategy`) e adicionando as funcionalidades específicas do algoritmo.

Para possibilitar a análise comparativa da influência da carga de trabalho infligida pelos detectores de falhas no desempenho dos algoritmos de consenso, foi escolhido o algoritmo *early consensus* [Sch97] para ser implementado no DisCusS, já que este algoritmo faz uso de *broadcast* na troca de mensagens e, sendo assim, é mais sensível à variação de carga da rede. Este algoritmo possui dois fios de execução que trabalham i) esperando a chegada de uma mensagem de decisão e ii) tentando chegar ao consenso por si próprio. Mais especificamente, o segundo fio trabalha processando eventos (suspeita de falhas e entrega de mensagens), postergando aqueles que não devem ser processados ainda e desconsiderando eventos sem relevância (velhos). Neste protótipo, o primeiro fio de execução, que espera a chegada de decisões, foi implementado como um evento a mais a ser tratado no segundo fio, minimizando a sobrecarga causada pela concorrência. O algoritmo usa os identificador único dos objetos de consenso como identificador do dado para a tomada de decisão, sendo que o identificador original enviado pelo cliente é sobreposto pelo algoritmo.

Para a troca de mensagens entre objetos no nível de detecção de falhas, mensagens *one-way* foram utilizadas, conforme observável nas IDL's, com qualidade de serviço `SYNC_NONE`, o que quer dizer que a invocação não sofre nenhum tipo de sincronização, não garantindo nem mesmo o envio da mensagem pelo ORB. Para a implementação do algoritmo de Schiper, foram considerados canais de comunicação entre os objetos de consenso como sendo confiáveis.

5.4 Trabalhos Relacionados: Arquitetura

Algumas arquiteturas também fazem uso do consenso como forma de resolver outros problemas em sistemas distribuídos. Nesta seção, algumas delas serão apresentadas.

5.4.1 *Generic Consensus Service*

Em [GS01], Guerraoui e Schiper apresentam uma arquitetura para um serviço genérico de consenso. Basicamente, tal trabalho divide os papéis presentes em um problema de consenso em *iniciadores*, *clientes* e *servidores*. Iniciadores são processos que avisam aos clientes que uma nova rodada de consenso está se iniciando, e que estes devem enviar suas proposições para os servidores. Cada servidor é envolto em um *filtro* que transforma o dado em uma entrada utilizável pelo serviço. O inconveniente desta abordagem é que o serviço de consenso deve estar ciente do tipo de dado que os clientes mandam para que possa tranformá-lo em algo utilizável. Em contrapartida, a abordagem utilizada neste trabalho força o cliente a entregar dados compatíveis com o serviço, isolando a parte variável do sistema no lado cliente. Além disso, há neste trabalho também o objetivo de uma especificação mais detalhada do serviço, apresentando uma interface de programação, mas sem perder o foco na generalidade do serviço.

5.4.2 *Object Group Service*

O *Object Group Sevice*[Fel98] é um serviço para replicação ativa de objetos que faz uso tanto de um serviço de detecção de falhas quanto de um serviço de consenso. Entretanto, o serviço de detecção não é compatível com FT-CORBA, uma vez que seu desenvolvimento se deu anteriormente a esta especificação. Enquanto o serviço de consenso do OGS (OConsS) foi planejado para dar suporte ao serviço de replicação, o trabalho apresentado aqui foi focado em prover um serviço com partes selecionáveis para ambientes específicos, e com uma interface genérica o suficiente para ser utilizado na resolução de diversos problemas e em diversos ambientes. OConsS decide sobre dados específicos da aplicação, o que quer dizer que para utilizá-lo isoladamente ao OGS, tem-se que especializar seu protocolo para trabalhar com o novo tipo de dado. DisCusS, por outro lado, decide sobre uma representação do dado da aplicação (um hash, por exemplo), que vai junto com o próprio dado (Ver a definição de `Value` no Anexo B). Obviamente especializações que manipulem dados específicos da aplicação também são implementáveis.

5.4.3 Bast

Desenvolvido pela *École Polytechnique Fédérale de Lausanne*, Bast[Gar98] é um arcabouço implementado em Smalltalk e em Java que fornece abstrações para tolerância a falhas em sistemas distribuídos. Bast usa um serviço de consenso para prover camadas de ordenação total de mensagens, *view synchrony* e terminação atômica. Estes serviços são apresentados encapsulados em *protocol objects*, classes com implementações de protocolos distribuídos e em conjunto com *protocol patterns*, esquemas para organização dos protocolos.

5.4.4 Thunderbolt

Thunderbolt [PAP99] é um middleware que provê um conjunto de serviços como *commit* atômico, *view synchrony* e difusão atômica, todos implementados sobre um serviço de consenso. Sua arquitetura é baseada em um barramento virtual semelhante ao barramento disponibilizado por um ORB. A principal restrição ao uso deste sistema é sua abordagem monolítica e, aparentemente, restrita a uma implementação, em detrimento de um modelo conceitual.

5.4.5 *Generic Agreement Framework*

O *General Agreement Framework* [HRTM99, HMRT99, GHRT00] usa uma abordagem diferente das anteriores para a resolução de problemas de concordância. Ao invés de traduzir os problemas para instâncias de consenso, ele permite a configuração de uma série de *parâmetros de versatilidade* para que o protocolo resolva diretamente o problema proposto. No fundo, estas configurações criam apenas variantes do algoritmo de Chandra&Toueg [CT96] e, embora esta abordagem seja extremamente útil para a criação de protocolos *ad hoc*, ela não permite a incorporação de algumas otimizações como a proposta por Schiper [Sch97], que baixa a dependência dos participantes do consenso em um coordenador, e por Brasileiro *et al.* [BGMR00, BGMR01], que permite que a decisão seja alcançada em apenas um passo de comunicação quando certas condições se apresentam.

5.4.6 Phoenix

Comunicação de grupo é uma abordagem muito utilizada na construção de sistemas que usam replicação para proverem tolerância a falhas. Sistemas de comunicação de grupo normalmente implementam esquemas de difusão atômica para elementos de um grupo, normalmente dinâmico, controlado por algum algoritmo de *group membership*.

O sistema de comunicação de grupo Phoenix [Mal96] usa um serviço de consenso como base para uma pilha de protocolos com *group membership* e difusão atômica. O algoritmo de Chandra e Toueg [CT96] é usado na implementação do serviço de consenso de Phoenix, sendo fechado para uso pelos níveis superiores.

Em tal arquitetura, o uso do DisCusS como alternativa para o serviço de consenso de Phoenix adicionaria flexibilidade na escolha de protocolos alternativos para ambientes variados, sem afetar o uso pelos níveis superiores. Esta abordagem provavelmente lhe conferiria maior usabilidade com melhor desempenho, uma vez que algoritmos mais propícios poderiam ser usados.

Capítulo 6

Impacto dos Mecanismos de Detecção de Falhas na Execução do Consenso Distribuído

"Counting in octal is just like counting in decimal, if you don't use your thumbs."
Tom Lehrer

A detecção de falhas é feita pela observância da chegada de mensagens do objeto monitorado ao detector. A periodicidade destas mensagens varia de acordo com o algoritmo de detecção, mas também com a qualidade de serviço que se espera do detector. Quanto mais freqüente o envio de mensagens, mais rápido o detector conseguirá tomar conhecimento de falhas. Contudo, esta velocidade tem um preço: a detecção de falhas consome uma fatia proporcional a esta velocidade em largura de banda da rede, bem como de recursos computacionais nas estações. Sendo assim, é importante tentar identificar o quanto outros serviços, em especial o de consenso, sofrem degradação em função da carga criada pelos detectores, mesmo sabendo que um detector pode ser compartilhado por diversas aplicações, o que justifica também o estudo de seu desempenho independentemente de clientes.

Poucas avaliações desta influência, dos detectores sobre outros servços, são encontradas na literatura. Sergent *et. al.* [SDS99] relatam simulações e Esteffenel [EJP01], medições em um sistema real. Nas seções seguintes são reportadas medições independentes feitas sobre o protótipo da implementação da arquitetura proposta, em uma rede local, e simulações sobre uma rede geograficamente distribuída. Até onde sabemos, nenhum outro trabalho comparou o impacto de detectores adaptativos e não-adaptativos no consenso.

6.1 Testes

Serviços concorrentes como detecção de falhas e consenso distribuído concorrem não somente por largura de banda nos canais de comunicação, como também por recursos de processamento e memória. Para medir o impacto dos mecanismos de detecção de falhas sobre o desempenho do serviço de consenso, foram utilizadas métricas que levem em conta a contenda por recursos no sistema apresentadas por Úrban [UDS00]: latência e vazão(*throughput*). É importante notar que o trabalho [UDS00] apresenta uma visão analítica destas métricas, usando-as para comparar diversos algoritmos de *broadcast*, enquanto aqui, usamo-las para comparar diferentes execuções do mesmo algoritmo de consenso, variando-se apenas os detectores de falhas que o suportam.

Como se pode esperar, quanto maior a carga de trabalho, menor a vazão do serviço. Em períodos de grande uso da rede, mensagens dos detectores de falhas contribuem para um aumento deste uso. Além disso, o atraso na transmissão destas mensagens também aumenta. Logo, é necessário que se aumente o intervalo entre envios de mensagens durante alta carga, e que se tolere maiores atrasos nestas mensagens. Os detectores de Sotoma e Madeira [SM01a, SM01b] trabalham desta forma. Estes detectores tentam, basicamente, estimar um novo intervalo entre-envios e o valor de *timeout*, baseado nos tempos de transmissão das últimas mensagens, aumentando estes intervalos se as mensagens estão atrasadas ou diminuindo-os caso as mensagens cheguem muito rapidamente, provendo uma melhor velocidade na detecção de falhas. Nestes testes, tentou-se verificar se esta abordagem é realmente boa, comparando-a com um detector de falhas clássico (*heartbeat-like*) e com um detector nulo, que não emite mensagens. De fato, o detector *heartbeat* é apenas uma versão simplificada (sem adaptação) do primeiro, e o segundo, apenas uma execução do primeiro em que a detecção não é iniciada.

No serviço de consenso, o algoritmo de Schiper [Sch97] foi usado pois, como já dito, devido ao seu padrão de mensagens distribuído, se torna mais sensível ao tráfego da rede, permitindo uma melhor observação da influência da contenda no sistema causada pelos detectores de falhas.

Todos os testes foram executados em estações Intel Celeron 1200Mhz, rodando Linux 2.4.18, sobre uma rede 10BaseT em períodos de baixa utilização (*round-trip* máximo de 0,2 ms).

6.1.1 Testes sobre o Consenso

Os testes sobre o consenso foram realizados em dois cenários: I – *sem falhas* e *sem suspeitas erradas*, pois todas as suspeitas foram desconsideradas; e, II – *com falhas do coordenador* e *sem suspeitas erradas*. Em II, todas as falhas ocorreram no coordenador antes do início do algoritmo de consenso, logo, quando o algoritmo se iniciava, todos os detectores já

haviam percebido sua falha. Este modelo foi seguido para garantir que todas as rodadas sendo comparadas tinham exatamente o mesmo padrão de falhas.

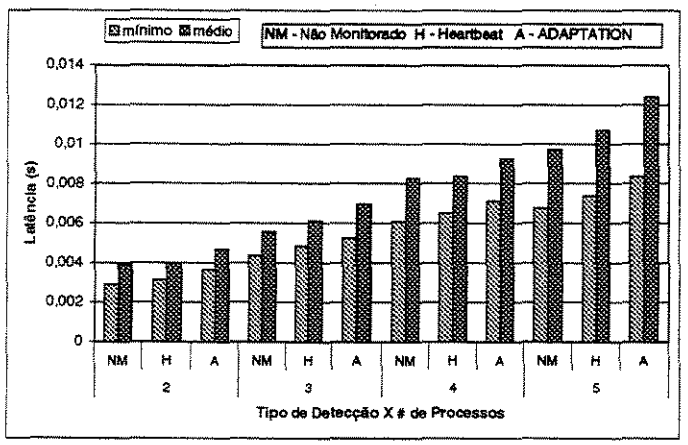
Os resultados apresentados correspondem à média do tempo de execução de pelo menos 1000 execuções seqüenciais de operações de consenso. Como em algumas aplicações, como servidores replicados, é a primeira resposta para uma requisição que é importante, foram mostrados também o tempo médio dos tempos mínimos (tempo para chegada da primeira resposta) para execução do consenso.

A execução dos testes para a tomada de tempos exigia que a seqüencia de instâncias de consenso começasse a ser resolvida ao mesmo tempo. Como os relógios das estações não estavam perfeitamente sincronizados, optou-se por sincronizar as execuções por meio de uma mensagem a todos os envolvidos. Ao invés de usar uma aplicação externa que enviasse esta mensagem a todos os processos, optou-se por rodar n (o número de membros do consenso) instâncias de consenso, antes das instâncias que sofreram tomadas de tempo. Neste esquema, cada um dos n processos é responsável por coordenar uma instância, e como não há falhas ou suspeitas erradas, cada instância só termina quando o coordenador envia a decisão. Isto garante que após a execução da n-ésima instância, todos os processos estão prontos para executar a seqüencia seguinte.

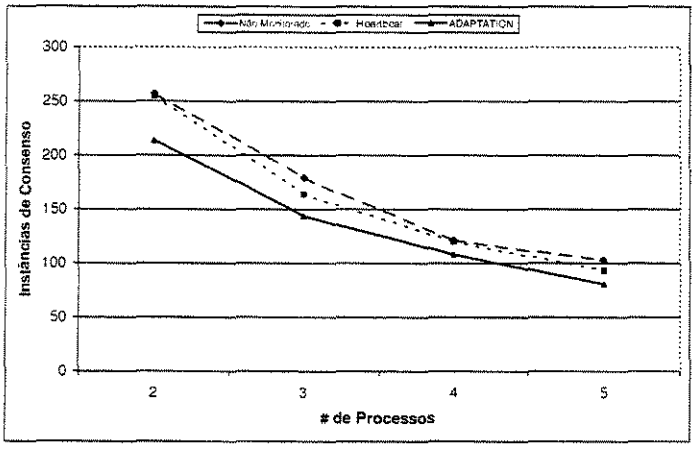
Cenário I. O gráfico na Figura 6.1.a mostra a latência da execução do consenso no cenário I, com 5 estações envolvidas. Os testes envolveram detectores *heartbeat-like* (H), ADAPTATION (A) e um detector oco, sem monitoração real (NM). O η (intervalo entre envios) do detector *heartbeat* foi ajustado para 20 ms e seu timeout para 40 ms. Para o detector ADAPTATION, introduzimos um limite inferior para seu η, ajustado também para 20 ms. Este limite inferior foi proposto em [SM01b] como MTU (*minimum time unit*). Observe que os valores de η são duas ordens de grandeza maiores que o valor de *round trip*, e foram escolhidos para não inundar completamente a rede com mensagens dos detectores, assim como o valor de timeout de *heartbeat*, 200 vezes maior que o rtt. É importante que se lembre que todas as suspeitas inválidas são descartadas neste teste, implicando que variações nos tempos de execução resultam tão somente da contenda por recursos.

Nota-se facilmente que o mecanismo de detecção tem influência sobre a latência do consenso, mesmo em um ambiente sem tráfego de fundo (outras mensagens no sistema de comunicação). Há uma relação constante NM<H<A que pode ser explicada pelo número de mensagens e pelos recursos que os detectores usam. O algoritmo ADAPTATION usa 33% mais mensagens que o *heartbeat*, já que estima seu novo η a cada 3 mensagens recebidas, o que também demanda mais processamento.

Outro fato já esperado é que quanto maior o número de processos envolvidos, maior a latência, já que o número de mensagens necessárias para a monitoração e para o algoritmo de consenso cresce exponencialmente em função de n. A Figura 6.1.b apresenta a vazão



(a)



(b)

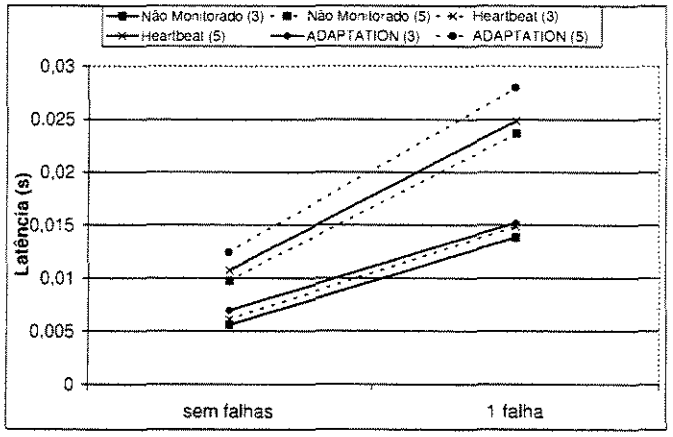
Figura 6.1: Testes sem falhas e sem suspeitas erradas.

do consenso (instâncias por segundo) no cenário I.

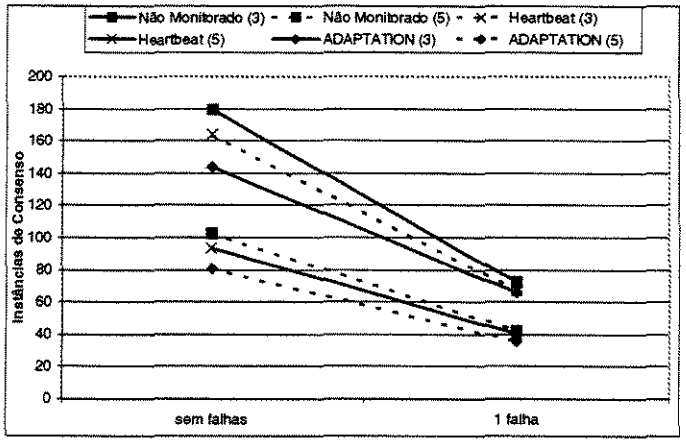
Cenário II. Falhas forçam os algoritmos de consenso a uma nova rodada, com um novo coordenador. Suspeitas de falhas incorretas podem ter o mesmo custo para o algoritmo que falhas reais. O algoritmo de Schiper é especialmente robusto quanto a falhas incorretas, pois uma maioria de processos tem que se decidir por passar para uma nova rodada antes que qualquer processo o faça. Esta robustez tem o custo de no mínimo $\lceil \frac{n+1}{2} \rceil$ mensagens em *broadcast*. O custo desta transição de rodadas é evidenciado pelos testes no cenário II, apresentados na Figura 6.2.

Os gráficos na Figura 6.2 mostram um grande decréscimo no desempenho do sistema quando se introduzem falhas, para três e cinco objetos de consenso, chegando a uma redução de 50% na vazão do algoritmo.

O custo proveniente de falhas não pode ser descartado. O custo de suspeitas incorretas,



(a)



(b)

Figura 6.2: Testes com falhas do coordenador e sem suspeitas incorretas.

entretanto, traz custos *desnecessários* à computação, inviabilizando rodadas ou gerando mensagens desnecessárias. Logo, minimizar a ocorrência destas suspeitas incorretas é algo de extremo interesse quando se faz uso de detectores de falhas não confiáveis.

6.1.2 Testes Sobre a Detecção de Falhas em Redes Locais

Uma vez que foi constatado que a ocorrência de suspeitas erradas tem um alto custo em termos de vazão do serviço de consenso, foram executados testes para medir a freqüência de ocorrência de suspeitas indevidas em cada um dos detectores.

Para a execução de ADAPTATION, usou-se 20 ms como limite inferior para η e, após algumas execuções, constatou-se que o valor médio desta variável era de 20,44 ms devido aos ajustes que ocorrem durante sua execução. Para manter justa a comparação,

atribuímos 23 ms como η para o algoritmo *heartbeat*, dando-lhe ligeira vantagem sobre o outro. Isto permite que os dois algoritmos reportem falhas dentro de praticamente o mesmo intervalo, permitindo usar o número de suspeitas incorretas como parâmetro de comparação.

A Figura 6.3 apresenta o resultado da monitoração de falhas por oito minutos em um período de utilização normal das estações envolvidas (as estações são usadas pelos alunos de pós-graduação do IC). Os testes com os dois detectores foram realizados em paralelo, monitorando uma mesma estação.

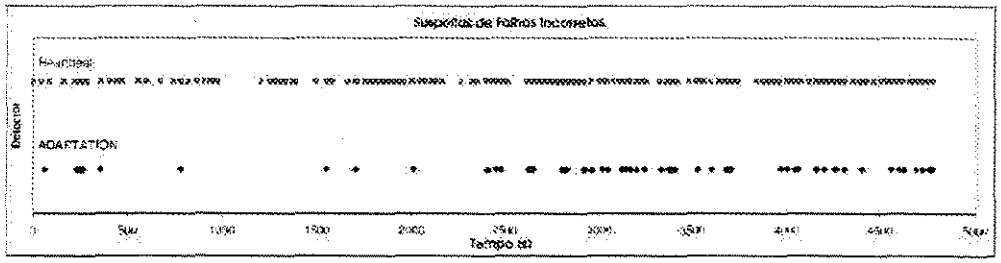


Figura 6.3: Ocorrência de suspeitas erradas.

A linha representando a ocorrência de suspeitas de falhas no detector *heartbeat* é claramente mais densa do que a linha do detector ADAPTATION. Em números, o detector *heartbeat* cometeu 262 suspeitas incorretas, contra 64 do ADAPTATION. Um detector com um alto número de suspeitas erradas irá, provavelmente, forçar o algoritmo de consenso a descartar mais rodadas. A conclusão direta é que em tais ambientes, o detector ADAPTATION conduzirá a um melhor desempenho do algoritmo de consenso. O uso de um timeout maior para o detector *heartbeat* melhoraria seu desempenho, mas isto ocorreria ao preço de um aumento no tempo de detecção de falhas.

O gráfico na Figura 6.4 mostra a latência de execução do consenso para 5 processos em um ambiente sem falhas, com variação de carga no sistema e sujeito à ocorrência de suspeitas de falhas incorretas. A carga foi introduzida pela instanciação de processos computacionalmente intensivos, seguindo a distribuição de Poisson. A utilização da UCP foi mantida na média de 90% e 100% para carga média e alta, respectivamente.

Avaliação dos Resultados

É perceptível pela Figura 6.4 que o desempenho de ADAPTATION varia em relação ao de *heartbeat*, quando variada a carga de trabalho. Com baixa carga, *heartbeat* apresenta melhor desempenho, mas a diferença diminui até se inverter levemente quando a carga sobe.

Quando em carga total, ADAPTATION apresenta um desempenho levemente melhor que *heartbeat*. O pequeno tamanho desta diferença pode ser justificado pelo fato do

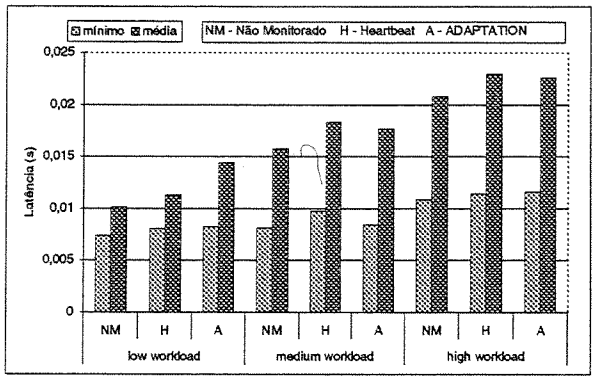


Figura 6.4: Latência na presença de carga de trabalho variável.

algoritmo de consenso de Schiper possuir a característica de não ser muito sensível a suspeitas incorretas por parte do detector de falhas. Suspeita-se que para um algoritmo como o de Chandra e Toueg, ou em redes com maior variação do atraso de comunicação, esta diferença seria mais significativa.

6.1.3 Testes sobre Detecção de Falhas em WANs

Para tentar confirmar as suspeitas apresentadas na última seção, esta seção apresenta simulações de uma série de algoritmos de detecção de falhas em uma rede geograficamente distribuída (WAN).

Para o desenvolvimento destes testes, um simulador foi desenvolvido na linguagem Python. Este simulador foi construído nos moldes do framework para detecção de falhas apresentado no Capítulo 3, e considera quatro parâmetros de entrada para cada algoritmo: entre_envios - intervalo entre o envio de mensagens *heartbeat*; timeout - tempo que o monitor espera antes de suspeitar do objeto monitorado; Δ - tempo adicionado ao timeout para encontrar o tempo que o monitor realmente espera por uma mensagem; e Δ^* - valor do qual Δ é incrementado a cada suspeita inválida detectada. O termo TIMEOUT é usado para a soma de timeout e Δ, enquanto Σ é usado para representar a soma das durações dos períodos em que o detector suspeitou indevidamente do objeto monitorado, durante um dado período de monitoração.

Os tempos de propagação das mensagens de *heartbeat* são lidos de um arquivo de *logs* de atrasos de comunicação, e os mesmos atrasos são utilizados para diferentes algoritmos, propiciando um bom parâmetro de comparação. Os atrasos podem ser também gerados aleatoriamente, seguindo alguma distribuição estatística.

Nas simulações apresentadas aqui, foi utilizado um *log* com o *round trip time* entre a Universidade Federal do Rio Grande do Sul e a Universidade Federal do Pará (Ponto de Presença da RNP no Pará), coletados por Nunes e Jansch-Pôrto, conforme apresen-

tados em [NJP02b]. Embora estes tempos equivalham ao tempo de ida e volta de uma mensagem entre as duas universidades, foram utilizados como o tempo de propagação em apenas um sentido nos testes apresentados aqui. Isto é totalmente plausível, se for considerado na simulação, um canal de comunicação igual a soma dos dois sentidos do canal de comunicação entre as universidades.

A Figura 6.5 apresenta os atrasos de comunicação (rtt) experimentados no dia 27 de Junho de 2001, tendo sido enviada 1 mensagem a cada segundo. A média para estes atrasos foi de 404 ms, com desvio padrão de 491 ms. A figura apresenta também esta média, e a soma da média e desvio padrão (895 ms).

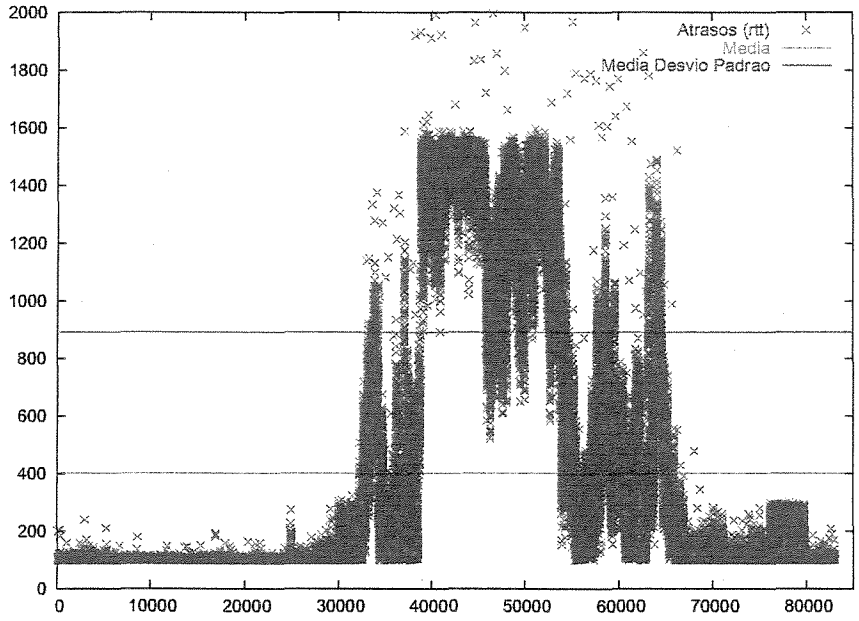


Figura 6.5: Atrasos entre UFRGS e POP/PA (21 de Junho de 2001).

Os gráficos a seguir apresentam o comportamento de vários detectores de falhas perante os atrasos apresentados na Figura 6.5, isto é, apresentam a variação de seus valores de TIMEOUT, apresentados nos gráficos com o nome dos detectores, e os períodos em que o detector de falhas suspeitou indevidamente do objeto monitorado.

A Figura 6.6 apresenta um detector *heartbeat* clássico, isto é, que mantém constante o seu valor de TIMEOUT ($\Delta = 0$ e $\Delta^\star = 0$). Como seria óbvio esperar, sempre que a carga de trabalho impõe atrasos maiores que seu valor de timeout, suspeitas incorretas são geradas. O valor de timeout do deste detector foi ajustado para 895 ms, a soma da média dos atrasos mais seu desvio padrão. Para este detector, $\Sigma_{hearbeat} = 15388,058$ segundos.

O gráfico seguinte (Figura 6.7) corresponde ao detector de falhas apresentado por

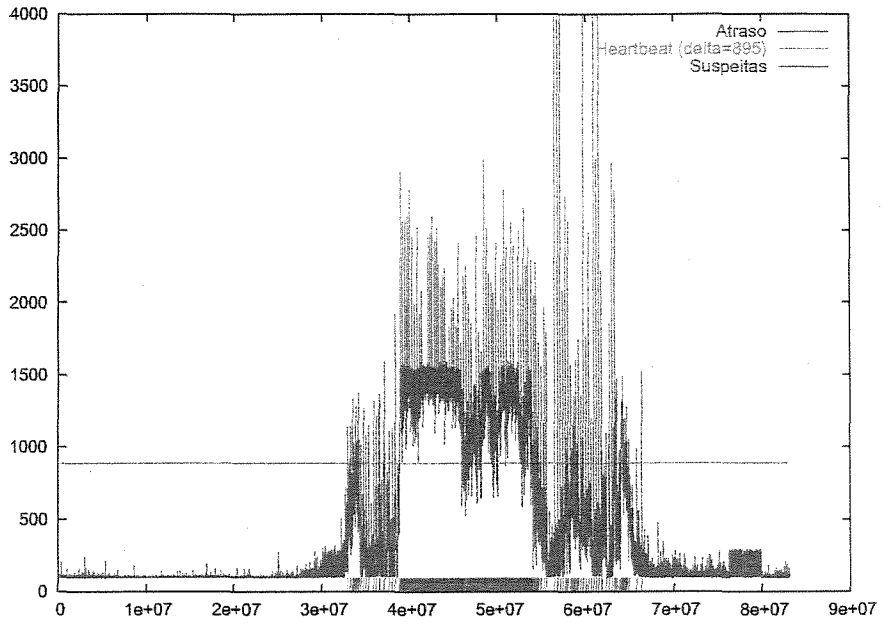


Figura 6.6: Detector *heartbeat* (timeout = 895 ms).

Chandra e Toueg [CHT96], que incrementa seu valor de TIMEOUT a cada confirmação de suspeita incorreta (no experimento, $\Delta^* = 10$), obtendo $\Sigma_{C\&T} = 203,807$.

Os gráficos seguintes apresentam detectores do tipo ADAPTATION. Nestes testes, houve adaptação apenas no timeout, sem alteração do intervalo entre-envios, uma vez que o *log* utilizado considera intervalo de 1 segundo entre mensagens durante toda sua execução.

O primeiro algoritmo, Figura 6.8, não obteve um resultado muito bom, uma vez que sempre tenta baixar seu valor de timeout, mesmo que este esteja próximo do limite inferior dos tempos de atraso. Este detector manteve uma suspeita indevida por $\Sigma = 20551$ segundos, o que equivale a quase um quarto dos 86400 segundos de um dia.

Os gráficos na Figuras 6.9, 6.10 6.11, equivalem a uma variação do algoritmo ADAPTATION, no qual um Δ igual a 100 ms, 200 ms e 300 ms foi usado, respectivamente.

Nestas variações, o período de suspeita indevida caiu para $\Sigma_{\Delta=100} = 2571,08$ segundos, $\Sigma_{\Delta=200} = 186,345$ segundos e $\Sigma_{\Delta=300} = 135,980$ segundos, respectivamente.

Uma variação do algoritmo baixou estes tempos para $\Sigma_{\Delta=100} = 386,713$ segundos, $\Sigma_{\Delta=200} = 120,523$ segundos e $\Sigma_{\Delta=100} = 107,631$ segundos. Esta variação, denominada ADAPTATION+, consistiu em dar mais ênfase ao último atraso experimentado que aos anteriores (o último atraso tem o dobro do peso dos anteriores). Seus comportamentos são apresentados nos gráficos seguintes (Figuras 6.12, 6.13 e 6.14)

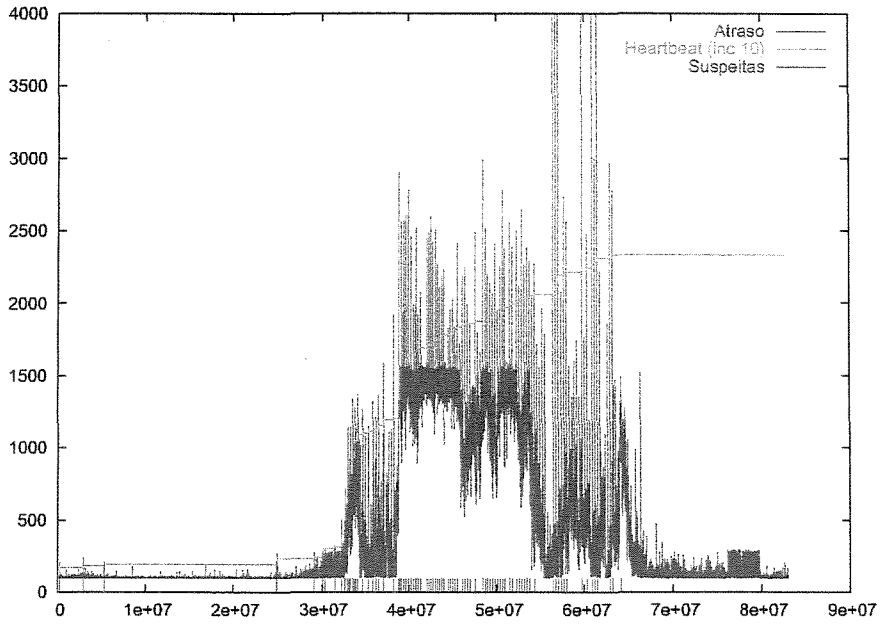


Figura 6.7: Detector *heartbeat* ($\Delta^\star = 10$ ms).

A Tabela 6.1.3 seguinte aponta os parâmetros usados nos algoritmos e os resultados dos testes.

Avaliação dos Resultados

Comparando-se os detectores *heartbeat* e ADAPTATION, percebe-se que o último provê tempos de detecção muito menores que o primeiro, ao custo de um aumento de Σ de um quarto, o que reflete diretamente na precisão do detector. Ao se considerar um Δ significativo em ADAPTATION, sua precisão é aumentada drasticamente. Com um $\Delta = 100$ ms, $\Sigma_{ADAPTATION}$ cai de 20.551 segundos para 2.571 segundos. Aumentando este tempo extra Δ para 300 segundos, $\Sigma_{ADAPTATION}$ cai para 135 segundos. Estes 300 ms somados ao valor de timeout calculado por ADAPTATION são ainda muito inferiores ao timeout do algoritmo *heartbeat*, 895 ms.

$\Sigma_{ADAPTATION} = 135$ segundos é menor também que a mesma medida no algoritmo de Chandra e Toueg que, embora se inicie com um tempo de detecção pequeno (pois seu valor de TIMEOUT é pequeno), aumenta-o indefinidamente ($\Delta^\star = 10$), tornando-se um detector muito lento para ser utilizado na prática. Claro que um reajuste de Δ é possível, mas esta seria também uma variação adaptativa do algoritmo.

Por fim, uma variação de ADAPTATION, denominada ADAPTATION+ que dá mais ênfase ao últimos atrasos detectados foi introduzida, obtendo resultados ainda melhores

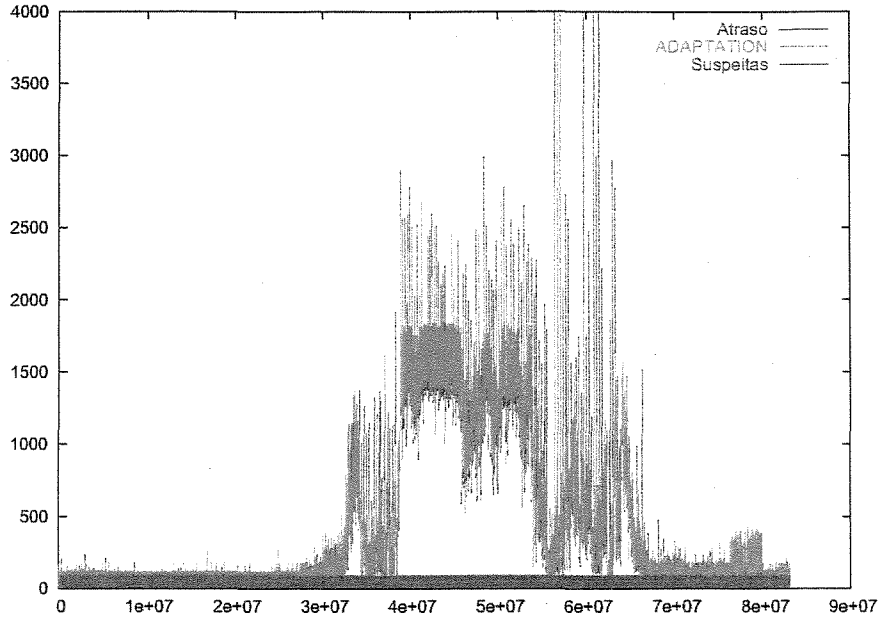


Figura 6.8: Detector *ADAPTATION*.

que aqueles obtidos por ADAPTATION, principalmente quando se compara os resultados dos algoritmos rodando com o tempo de timeout acrescido de $\Delta = 100$ ms, baixando o tempo Σ de 2571 segundos para 386 segundos.

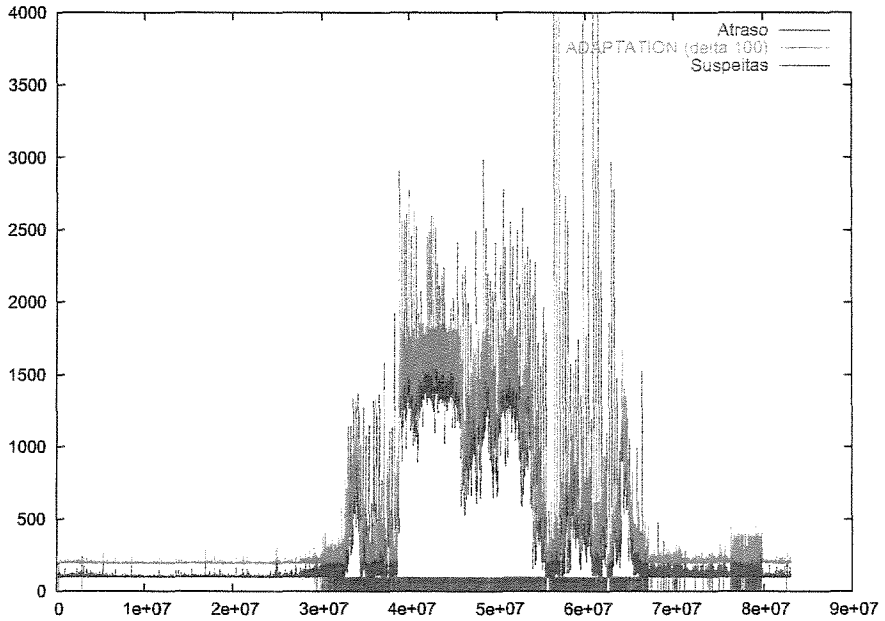


Figura 6.9: Detector *ADAPTATION* ($\Delta = 100$ ms).

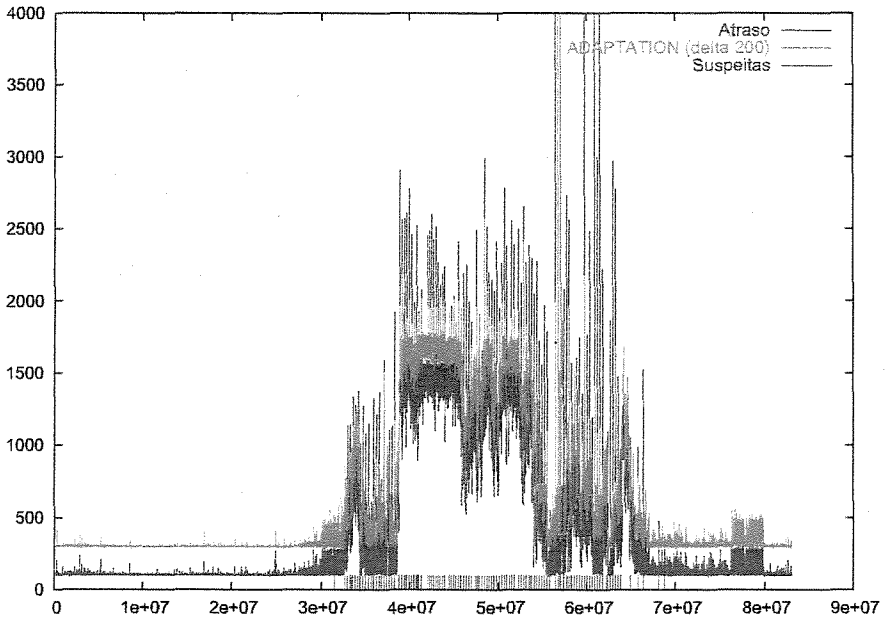


Figura 6.10: Detector *ADAPTATION* ($\Delta = 200$ ms).

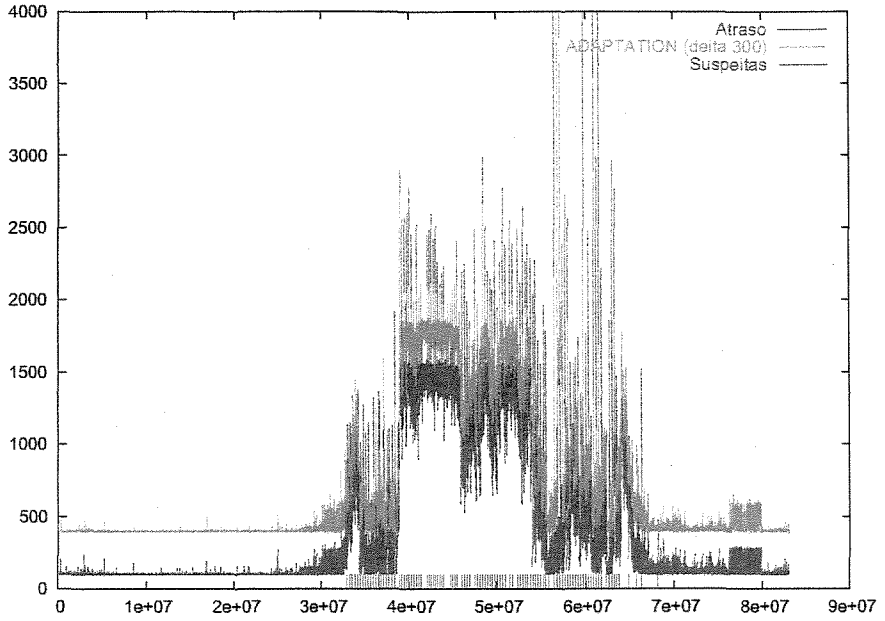


Figura 6.11: Detector *ADAPTATION* ($\Delta = 300$ ms).

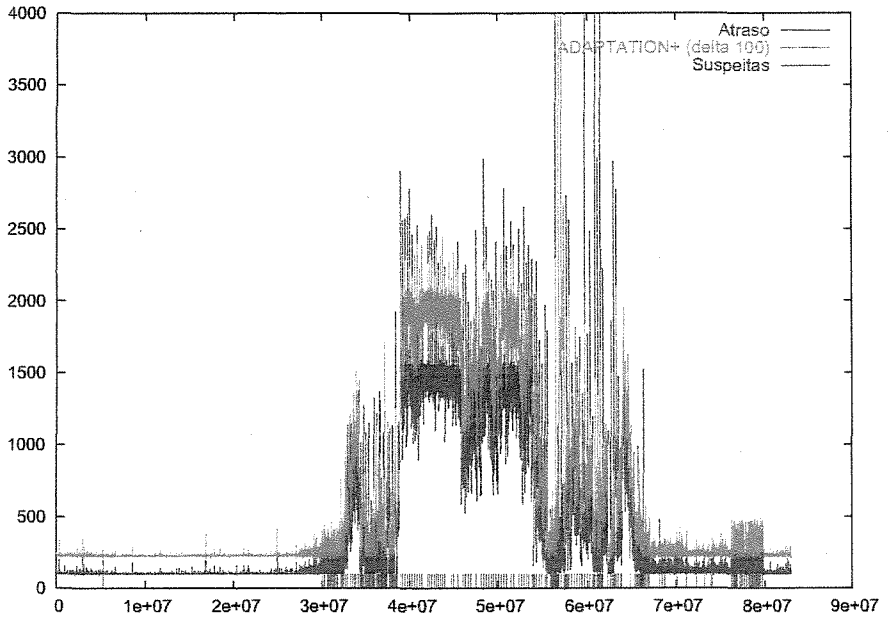


Figura 6.12: Detector *ADAPTATION+* ($\Delta = 100$ ms).

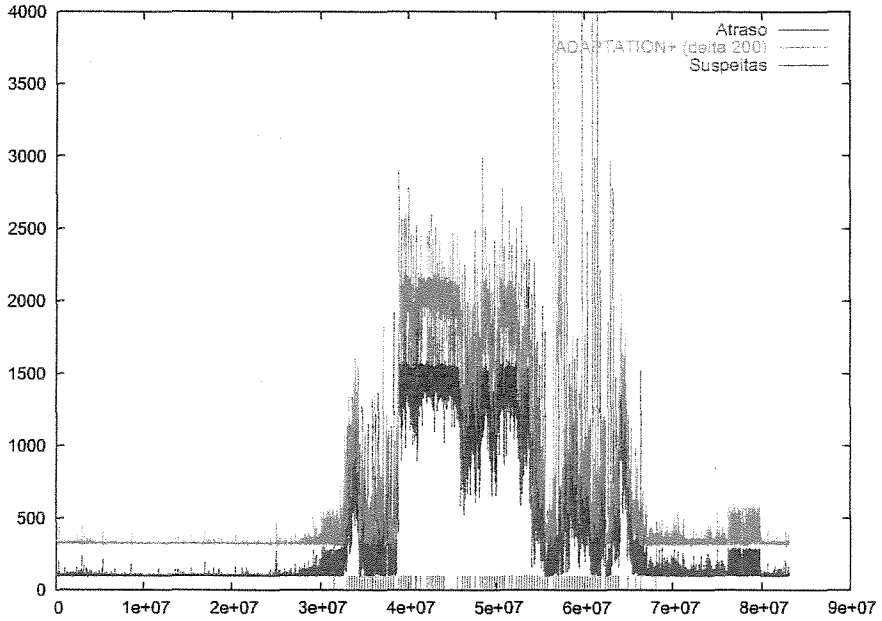


Figura 6.13: Detector *ADAPTATION+* ($\Delta = 200$ ms).

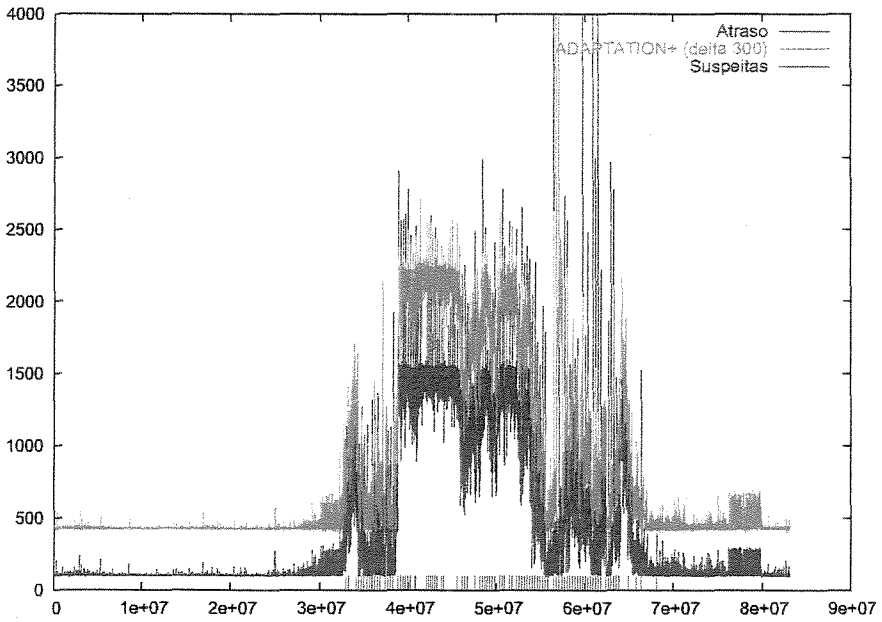


Figura 6.14: Detector *ADAPTATION+* ($\Delta = 300$ ms).

Algoritmo	timeout (ms)	Δ (ms)	$\Delta^{\star}$ (ms)	Σ (s)
Heartbeat	895	0	0	15388,058
	150	0	10	203,807
ADAPTATION	150	0	0	20551,0
	150	100	0	2571,08
	150	200	0	186,345
	150	300	0	135,980
ADAPTATION+	150	100	0	386,713
	150	200	0	120,523
	150	300	0	107,631

Tabela 6.1: Comparação dos algoritmos de detecção

Capítulo 7

Conclusão

Problemas de concordância estão entre os mais importantes enfrentados na construção de sistemas distribuídos. Difusão atômica, *commit* atômico e *group membership* são exemplos comumente encontrados na construção de sistemas distribuídos tolerantes a falhas. Tais problemas são caracterizados por um conjunto de processos tentando entrar em acordo sobre um valor (o conteúdo de uma variável, a próxima ação a ser executada, finalizar ou não uma transação). Em muitos sistemas, algoritmos específicos para estes problemas são desenvolvidos. Muitos destes são, entretanto, redutíveis a um problema mais simples, conhecido como *problema do consenso*, muito estudado, mas ainda de uso relativamente pequeno na prática.

Este trabalho apresentou um serviço de consenso genérico, denominado DisCusS (*Distributed Consensus Service*) e um serviço de detecção de falhas chamado FuSe (*Fault Detection Service*), do qual DisCusS é cliente. Por genérico, entenda-se que o serviço pode ser utilizado na resolução dos diversos problemas redutíveis ao consenso, como citado anteriormente. Esta generalidade é obtida pela utilização de um módulo tradutor, que converte um dado problema de concordância para uma instância de consenso. Este módulo, localizado no lado cliente do sistema, acessa um, de um *pool* de objetos de consenso, e delega a ele a tarefa de chegar a uma decisão. Como vantagens deste modelo pode-se citar o assincronismo entre cliente e objetos de consenso, que permite que clientes possam se desconectar durante a execução do serviço e voltar posteriormente para obter a decisão de uma dada instância de consenso. Além disso, este modelo permite que o serviço de consenso possa ser posicionado em uma rede com alta disponibilidade ou de características bem conhecidas, e ainda suportando a heterogeneidade ou propensão a falhas no ambiente dos clientes.

A arquitetura destes serviços, seus relacionamentos e principais módulos foram também apresentados e, baseado nesta arquitetura, foi ainda apresentado um *framework* definido segundo o paradigma de orientação a objetos. A definição deste *framework* foi

pautada em cinco requisitos: i)objetos selecionáveis; ii)independência entre serviços; iii)extensibilidade; iv)detecção adaptativa de falhas; e v)compatibilidade com FT-CORBA. Estes requisitos visam proporcionar maior configurabilidade e aplicabilidade ao modelo, permitindo a utilização de diferentes implementações e/ou de diferentes algoritmos simultaneamente, tanto para detecção de falhas quanto para o consenso, e a compatibilidade com a tecnologia CORBA, altamente difundida no desenvolvimento de sistemas distribuídos.

Foi apresentada também uma implementação protótipo do *framework* definido e, através de testes feitos sobre esta implementação, um estudo sobre a influência que os detectores de falhas têm sobre o desempenho do serviço de consenso. Deste estudo conclui-se que detectores de falhas adaptativos tem grande potencial de aplicabilidade, sendo de especial interesse em sistemas com carga de trabalho alta e sujeita a variações constantes.

Ainda, foi mostrado como a especificação FT-CORBA poderia ser implementada sobre DisCusS e FuSe, já que esta descreve um sistema de detecção de falhas e exige que alguns problemas de consistência e replicação de objetos sejam resolvidos, problemas estes mapeáveis para o consenso.

Contudo, o trabalho apresentado aqui possui limitações e, conseqüentemente, várias oportunidades para trabalhos futuros. A falta de uma interface para definição de requisitos de qualidade de serviço é uma destas limitações pois, embora as interfaces definidas para o serviço de detecção de falhas permitam o ajuste dinâmico dos parâmetros de detecção (entre-envios e *timeout*), elas não definem meios para ajuste de parâmetros como *tempo de detecção*, *taxa de recorrência de erros* e *duração de erros* [CTA02]. Também o estudo de qualidade de serviço na execução do consenso deve ser trabalhado, possibilitando por exemplo, a especificação de uma *taxa máxima de inviabilização de rodadas.*

Considerando aspectos mais práticos, a implementação de outros algoritmos, tanto de detecção de falhas quanto de consenso, deve ser o próximo passo. Mais implementações permitiriam um estudo mais aprofundado da influência dos detectores de falhas na execução de algoritmos de consenso com diferentes padrões de comunicação e, assim, a identificação de pares de algoritmos detecção/consenso que melhor se aplicam a ambientes específicos.

Também com o objetivo de melhor comparar os algoritmos, usar simuladores que permitam a produção e reprodução de padrões de falhas mais complexos, permitindo avaliar os tempos de resposta dos algoritmos para falhas que ocorram durante e em vários pontos de suas execuções, uma vez que os testes apresentados usaram padrões de falhas simples (falha do coordenador antes da invocação do algoritmo de consenso). A experimentação utilizando padrões de falhas complexos implicaria em, por exemplo, inserção de pontos de sincronização nos algoritmos, o que deterioraria seu desempenho e resultaria em testes não representativos de execuções normais, isto é, sem os pontos de sincronização.

Quanto à arquitetura proposta, é preciso determinar seu custo em relação a uma abordagem não tão genérica e modular. Uma vez que a modularização introduz indireções, é preciso verificar se a sobrecarga adicionada por estas indireções não inviabiliza a utilização deste modelo.

A efetivação da implementação de FT-CORBA utilizando os serviços apresentados é outro ponto interessante para trabalho futuro, assim como a aplicação dos serviços no desenvolvimento de uma aplicação real.

Referências Bibliográficas

[ACT97] Marcos K. Aguilera, Wei Chen, e Sam Toueg. Heartbeat: a timeout-free failure detector for quiescent reliable communication. Em *Proc. 11th International Workshop on Distributed Algorithms*, Setembro 1997.

[ACT98] Marcos K. Aguilera, Wei Chen, e Sam Toueg. Failure detection and consensus in the crash-recovery model. Em *Proc. of the 12th International Symposium on Distributed Computing*, Setembro 1998.

[AT96] Marcos K. Aguilera e Sam Toueg. Randomization and failure detection: A hybrid approach to solve consensus. Em Özalp Babaoglu e Keith Marzullo, editores, *Proceedings of the 10th International Workshop on Distributed Algorithms (WDAG96)*, volume 1151, páginas 29–39, Bolonha, Itália, 9–11 1996. Springer.

[B.99] Oestereich B. *Developing Software with UML.* Addison-Wesley, Harlow, England, 1999.

[BCP02] R. Baldoni, C.Marchetti, e S.Tucci Piergiovanni. Asynchronous active replication in three-tier distributed systems. Em *Prooceedings of the 2002 Pacific Rim International Symposium on Dependable Computing (PRDC'02)*, 2002.

[BGMR00] Francisco Brasileiro, Fabíola Greve, Achour Mostefaoui, e Michel Raynal. Consensus in one communication step is possible. Relatório Técnico 1321, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Abril 2000.

[BGMR01] F. Brasileiro, F. Greve, A. Mostefaoui, e M. Raynal. Consensus in one communication step. Em *6th Parallel Computing Technologies, 6th International Conference, PaCT 2001*, number 2127 in LNCS, páginas 42–50, Barcelona, Espanha, Setembro 2001. Springer Verlag.

[BMP02] Marin O. Bertier M. e Sens Pierre. Implementation and performance evaluation of an adaptable failure detector. Em *Proc. of the Int. Conference*

on Dependable Systems and Networks (DSN'02), páagina 354, Washington, D.C., USA, Junho 2002.

[CHT96] Tushar Deepak Chandra, Vassos Hadzilacos, e Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[CM03] L. J. Camargos e E. R. M. Madeira. Discuss and fuse: considering modularity, genericness and adaptation in the development of consensus and fault detection services. Em Rogerio de Lemos, Taisy Weber, e João Batista Camargo Jr., editores, *Proceedings of the Latin American Symposium on Dependable Computing (LADC 2003)*, LNCS, São Paulo, SP - Brasil, Outubro 2003. Springer-Verlag. Aceito para publicação.

[CT96] Tushar Deepak Chandra e Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM (JACM)*, 43(2):225–267, 1996.

[CTA02] Wei Chen, Sam Toueg, e Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51(5):561–580, 2002.

[DDS87] Danny Dolev, Cynthia Dwork, e Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM (JACM)*, 34(1):77–97, 1987.

[DLS88] Cynthia Dwork, Nancy Lynch, e Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM (JACM)*, 35(2):288–323, 1988.

[EJP01] L. A. B. Esteffenel e Ingrid Jansch-Pôrto. On the evaluation of failure detectors performance. Em *Proc. of IX Simpósio Brasileiro de Computação Tolerante a Falhas*, Florianópolis, Brasil, Março 2001.

[Fel98] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. Tese de Doutorado, École Polytechnique Fédérale de Lausanne, Suíça, 1998. Número 1867.

[FLP85] Michael J. Fischer, Nancy A. Lynch, e Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, 1985.

[Gar98] B. Garbinato. *Protocol Objects & Patterns for Structuring Reliable Distributed Systems*. Tese de Doutorado, École Polytechnique Fédérale de Lausanne, Suíça, 1998. Número 1801.

[GHJV94] Erich Gamma, Richard Helm, Ralph Johnson, e John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1994.

[GHRT00] Fabíola Greve, Michel Hurfin, Michel Raynal, e Frédéric Tronel. Primary component asynchrounous group membership as an instance of a generic ageement framework. Relatório Técnico 1292, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Janeiro 2000.

[GOS98] R. Guerraoui, R. Oliveira, e A. Schiper. Stubborn communication channels. Relatório Técnico 98/272, École Polytechnique Fédérale de Lausanne, Suíça, Março 1998.

[Gär99] Felix C. Gärtner. Fundamentals of fault-tolerant distributed computing in asynchronous environments. *ACM Computing Surveys (CSUR)*, 31(1):1–26, 1999.

[Gär01] Felix C. Gärtner. A gentle introduction do failure detectors and related subjects. Relatório Técnico, Darmstadt University of Technology, Abril 2001.

[GS01] Rachid Guerraoui e André Schiper. The generic consensus service. *IEEE Transactions on Software Engineering*, 27(1):29–41, Janeiro 2001.

[Gue95] Rachid Guerraoui. Revisiting the relationship between non-blocking atomic commitment and consensus. Em *Proceedings of the International Workshop on Distributed Algorithms*, LNCS. Springer-Verlag, 1995.

[Had86] Vassos Hadzilacos. On the relationship between the atommic commitment and consensus protocols. Em B. Simons e A. Spector, editores, *Proceedings of the Workshop on Fault-Tolerant Distributed Computing*, number 448 in LNCS, Pacific Grove, CA, Março 1986. Springer-Verlag 1990.

[HMR00] M. Hurfin, A. Mostefaoui, e M. Raynal. A comprehensive approach for failure detector-based consensus protocols. Relatório Técnico 1303, Institut de Recherche en Informatique et Systèmes Aléatoires, 2000.

[HMRT99] Michel Hurfin, Raimundo Macêdo, Michel Raynal, e Frédéric Tronel. A general framework to solve agreement problems. Relatório Técnico 1231, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Fevereiro 1999.

[HR97] Michel Hurfin e Michel Raynal. Fast asynchronous consensus based on a Weak failure detector. Relatório Técnico 1118, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Julho 1997.

[HRTM99] Michel Hurfin, Michel Raynal, Frédéric Tronel, e Raimundo Macêdo. A general framework to solve agreement problems. Em *Proc. of the 18th IEEE Symposium on Reliable Distributed Systems*, páginas 55–65, Lausanne, Suíça, Outubro 1999.

[Lam96] B. W. Lampson. How to build a highly available system using consensus. Em Babaoglu e Marzullo, editores, *10th International Workshop on Distributed Algorithms (WDAG 96)*, volume 1151, páginas 1–17. Springer-Verlag, Berlin Germany, 1996.

[Lam98] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems (TOCS)*, 16(2):133–169, 1998.

[LCLPS01] Joni Fraga Lau Cheuk Lung, Ricardo Padilha, e Luciana Souza. Adaptando as especificações ft-corba para redes de larga escala. Em *Anais do XIX Simpósio Brasileiro de Redes de Computadores - SBRC'2001 - SBC*, Belo Horizonte, MG, Brasil, Maio 2001.

[LFF+99] C. Lung, J. Fraga, Jean-Marie Farines, M. Ogg, e A. Ricciardi. Cosnamingft - a fault-tolerant corba naming service. Em *Proc. 18th IEEE International Symposium on Reliable Distributed Systems (SRDS'99)*, páginas 254–262. IEEE Computer Society, 1999.

[LFSP00] Lau Cheuk Lung, Joni Fraga, Luciana Souza, e Ricardo Padilha. Grouppac: Um framework para implementação de aplicações tolerantes a falhas. Em *Anais do XXVI Conferencia LatinoAmericana de Informatica - CLEI'2000*, Estado de México, México, Setembro 2000.

[LS95] R. Greg Lavender e Douglas C. Schmidt. Active object: an object behavioral pattern for concurrent programming. *Proc. Pattern Languages of Programs*, 1995.

[LSP95] Lamport, Shostak, e Pease. The byzantine generals problem. Em N. Suri, C. J. Walter, e M. M. Hugue, editores, *Advances in Ultra-Dependable Distributed Systems*. IEEE Computer Society Press, 1995.

[Lyn96] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann, Abril 1996. ISBN 1558603484.

[Mal96] C. P. Malloth. *Conception and Implementation of a Toolkit for Building Fault-Tolerant Distributed Applications in Large Scale Networks.* Tese de Doutorado, École Polytechnique Fédérale de Lausanne, Suíça, Setembro 1996.

[MR99] Achour Mostefaoui e Michel Raynal. Solving consensus using chandra-toueg´s unreliable failure detectors: A general quorum-based approach. Relatório Técnico 1254, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Julho 1999.

[MRR01] Achour Mostefaoui, Sergio Rajsbaum, e Michel Raynal. A versatile and modular consensus protocol. Relatório Técnico 1427, Institut de Recherche en Informatique et Systèmes Aléatoires, Université de Rennes, Dezembro 2001.

[NGS00] B. Natarajan, A. Gokhale, e D. C. Schmidt. DOORS: Towards high-performance fault tolerant CORBA. Em *Proceeding of the 2nd International Symposium on Distributed Objects and Applications (DOA´00)*, Setembro 2000.

[NJP02a] R. C. Nunes e I. Jansch-Pôrto. Modelling communication delays in distributed systems using time series. Em *21st Symposium on Reliable Distributed Systems (SRDS2002)*, Suita, Japão, Outubro 2002.

[NJP02b] R. C. Nunes e I. Jansch-Pôrto. Non-stationary communication delays in failure detectors. Em *3rd IEEE Latin-American Test Workshop (LATW2002)*, Montevideo, Uruguay, Fevereiro 2002.

[OMG97] CORBAservices: CORBA Object Services Specification. Relatório Técnico, Object Management Group, Julho 1997.

[OMG01a] The Common Object Request Broker: Architecture and Specification. Versão 2.6, Object Management Group, Dezembro 2001.

[OMG01b] Fault Tolerant CORBA. CORBA 2.6. Versão 2.6, Object Management Group, Dezembro 2001.

[PA00] V. Paxson e M. Allman. Request for comments: 2988. Relatório Técnico, Network Working Group, 2000.

[PAP99] H. Praveen, S. Arvindam, e S. Pokarna. Thunderbolt: A consensus-based infrastructure for loosely coupled cluster computing. Em *Proc. of the 6th International Conference on High Performance Computing (HiPC 99)*, 1999.

[PRO03] J. Pereira, L. Rodrigues, e R. Oliveira. Semantically reliable multicast: Definition, implementation and performance evaluation. *IEEE Transactions on Computers, Special Issue on Reliable Distributed Systems*, 52(2):150–165, Fevereiro 2003.

[RMA+00] Guerraoui R., Hurfin M., Mostefaoui A., Oliveira R., Raynal M., e Schiper A. Consensus in asynchronous distributed systems: A consise guided tour. Em *Distributed Systems*, number 1752 in LNCS, páginas 33–47. Springer-Verlag, 2000.

[Rod03] L. Rodrigues. The road to a more configurable and adaptive communication and coordination support. Em *Proceedings of the 9th Workshop on Future Trends of Distributed Computing Systems*, páginas 16–22, San Juan, Porto Rico, Maio 2003.

[SB96] P. Sommerlad e F. Buschmann. Manager design pattern. Em *3rd annual PLoP*, Allenton Park, Illinois, Setembro 1996.

[Sch97] Andre Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, 1997.

[Sch02] Douglas C. Schmidt. http://www.cs.wustl.edu/~schmidt/tao.html. Internet site, Março 2002.

[SDS99] Nicole Sergent, Xavier Défago, e André Schiper. Failure detectors: implementation issues and impact on consensus performance. Relatório Técnico, École Polytechnique Fédérale de Lausanne, Switzerland, 1999.

[SM01a] Irineu Sotoma e Edmundo R. M. Madeira. ADAPTATION - Algorithms to ADAPTive FAulT MonItOriNg and Their Implementation on CORBA. Em *Proceedings IEEE of the 3rd International Symposium on Distributed Objects and Applications (DOA '01)*, páginas 219–228, Roma, Itália, Setembro 2001.

[SM01b] Irineu Sotoma e Edmundo R. M. Madeira. DPCP(Discard Past Consider Present) - A Novel Approach to Adaptive Fault Detection in Distributed Systems. Em *Proceedings of the 8th IEEE Workshop on Future Trends of Distributed Computing Systems (FTDCS'2001)*, páginas 76–82, Bolonha, Itália, Novembro 2001.

[SNG+03] Douglas C. Schmidt, Balachandran Natarajan, Aniruddha Gokhale, Chris Gill, e Nanbor Wang. Tao: A pattern-oriented object request broker for distributed real-time and embedded systems. *IEEE Distributed Systems Online*, 2003. ISSN: 1541-4922.

[SNT02] Diana Szentiványi e Simin Nadjm-Tehrani. Building and evaluating a fault-tolerant corba infrastructure. Em *Proceedings of the Workshop on Dependable Middleware-Based Systems (WDMS'02)*, Washington, DC, Junho 2002.

[SRNTG03] Diana Szentiványi, Isabelle Ravot, Simin Nadjm-Tehrani, e Rachid Guerraoui. Dependable distributed middleware: Pay now or pay later! Em *Poster Session at Middleware 2003, in Middleware 2003 Companion*, páagina 331, Rio de Janeiro, Brasil, Junho 2003.

[UDS00] P. Urbán, X. Défago, e A. Schiper. Contention-aware metrics for distributed algorithms: Comparison of atomic broadcast algorithms. Em *Proc. of the 9th IEEE International Conference on Computer Communications and Networks (IC3N 2000)*, páginas 582–589, Outubro 2000.

[UFS03] LMCI UFSC. Página do projeto grouppac. Sítio Internet, Junho 2003. http://grouppac.sourceforge.net/.

[VR01] Paulo Veríssimo e Luís Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, Janeiro 2001. ISBN 0-7923-7266-2.

[VR02] P. Vicente e L. Rodrigues. An indulgent uniform total order algorithm with optimistic delivery. Em *Proceedings of the 21th IEEE Symposium on Reliable Distributed Systems (SRDS'02)*, páginas 92–101, Osaka, Japão, Outubro 2002.

Anexo A: Arquivo de Configuração.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE conf SYSTEM "file:/mnt/1/Proj/FUSE/willbe/nConf.dtd">
<conf>
    <!--Primeiramente, definimos algumas variáveis. -->
    <define>
        <var name="IP1" val="10.0.0.1"/>
        <var name="IP2" val="10.0.0.2"/>
        <var name="IP3" val="10.0.0.3"/>
        <!--Isso varia de máquina para máquina.-->
        <var name="LOC_IP" val="IP1"/>
        <var name="REM_IP1" val="IP2"/>
        <var name="REM_IP2" val="IP3"/>
        <!--Todos os executáveis são relativos a este endereço.-->
        <var name="BASE_DIR" val="/mnt/1/dist"/>
    </define>
    <!--O segundo passo é iniciar estes programas, atribuindo-lhes nomes (-n $<name>) -->
    <objects>
        <object>
            <name>PUSH_ADAPT_FAC_1</name>
            <binary>/paf.exe</binary>
        </object>
        <object>
            <name>PUSH_FAC_1</name>
            <binary>/pf.exe</binary>
        </object>
        <object>
            <name>PUSH_FAC_2</name>
            <binary>/pf.exe</binary>
        </object>
        <object>
            <name>CONSENSUS</name>
            <binary>/consensus.exe</binary>
        </object>
    </objects>
    <!--Com todos iniciados, configurar o FuSe. -->
    <fuse>
        <factories>
            <factory>
                <name>PUSH_ADAPT_FAC_1</name>
                <type>PUSH ADAPTATION</type>
            </factory>
            <factory>
                <name>PUSH_FAC_1</name>
                <type>PUSH</type>
            </factory>
```

```
    <factory>
        <name>PUSH_FAC_2</name>
        <type>PUSH</type>
    </factory>
</factories>
<super_monitor>
    <!--Consenso-->
    <name>SM_$LOC_IP</name>
    <monitor>
        <name>LOC_DEC_1</name>
        <type>PUSH</type>
        <!--Se a fábrica não é especificada, pode-se fazer b
         alanceamento de carga, elegendo uma pelo tipo.-->
        <factory>PUSH_FAC_1</factory>
        <monitored>PUSH_CONSENSUS</monitored>
        <!--Em ms.-->
        <interval>1000</interval>
        <notifiable>QUEM</notifiable>
    </monitor>
    <!--PUSH_FAC_2 -->
    <monitor>
        <name>LOC_DEC_2</name>
        <type>PUSH</type>
        <factory>PUSH_FAC_1</factory>
        <monitored>PUSH_FAC_2</monitored>
        <interval>1000</interval>
        <notifiable/>
    </monitor>
    <!--PUSH_FAC_1-->
    <monitor>
        <name>LOC_DEC_3</name>
        <type>PUSH</type>
        <factory>PUSH_FAC_2</factory>
        <monitored>PUSH_FAC_1</monitored>
        <interval>1000</interval>
        <notifiable/>
    </monitor>
    <!--PUSH_ADAPT_FAC_1-->
    <monitor>
        <name>LOC_DEC_4</name>
        <type>PUSH</type>
        <factory>PUSH_FAC_2</factory>
        <monitored>PUSH_ADAPT_FAC_1</monitored>
        <interval>1000</interval>
        <notifiable/>
    </monitor>
    <monitor>
        <name>REM_DEC_1</name>
        <type>PUSH ADAPTATION</type>
        <factory>PUSH_ADAPT_FAC_1</factory>
        <monitored>$REM_IP1</monitored>
        <interval>1000</interval>
        <notifiable/>
    </monitor>
    <monitor>
        <name>REM_DEC_2</name>
        <type>PUSH ADAPTATION</type>
        <factory>PUSH_ADAPT_FAC_1</factory>
```

```
                <monitored>$REM_IP2</monitored>
                <interval>1000</interval>
                <notifiable/>
            </monitor>
        </super_monitor>
    </fuse>
    <!--Próximo passo, configurar o programa de consenso -->
    <discuss>
        <!--Configura o objeto já iniciado... -->
        <name>CONSENSUS</name>
        <!--com o identificador 0... -->
        <id>0</id>
        <!--e o faz consultar o super-monitor seguinte.-->
        <super_monitor>SM_$LOC_IP</super_monitor>
    </discuss>
    <!--Em caso de falhas, o sistema deve iniciar novos objetos e
    substituir os falhos. -->
</conf>
```

Anexo B: Definições das IDL's.

Definições Básicas

As seguintes definições estendem FT-CORBA para a utilização de algoritmos adaptativos de detecção de falhas. Estas definições foram inicialmente apresentadas em [SM01a]

```
//Basic definitions from FT-CORBA.
typedef unsigned short FaultMonitoringStyleValue;
const FaultMonitoringStyleValue PULL = 0;
const FaultMonitoringStyleValue PUSH = 1;
const FaultMonitoringStyleValue NOT_MONITORED = 2;

//Extensions for ADAPTATION algorithms.
const FaultMonitoringStyleValue PULL_ADAPTATION = 3;
const FaultMonitoringStyleValue PUSH_ADAPTATION = 4;

struct FaultMonitoringIntervalAndTimeoutValue {
  TimeBase::TimeT     monitoring_interval;
  TimeBase::TimeT     timeout;
};

enum MonitorableStatus {SUSPECTED, ALIVE, NOT_MONITORED};
```

Definições Relacionadas ao Monitor

As interfaces seguintes definem o protocolo de utilização de FuSe.

```
//The FaUlt detection SErvice.
module FUSE{
  interface MonitorableStrategy;

  typedef Long MonitorUID;
  typedef Long FUSEObjectUID;
  typedef sequence<MonitorableStrategy> SuspectList;

  interface SuperMonitor {
    void
    start_monitoring_all ();

    void
    stop_monitoring_all ();

    void
    start_monitoring (in FUSEObjectUID uid);
```

```
      void
      stop_monitoring (in FUSEObjectUID uid);

      //To add/remove a Monitor <-> Monitorable association.
      FUSEObjectUID
      add_monitoring_assoc (in GeneralMonitorStrategy monitor,
                            in MonitorableStrategy monitorable);

      Boolean
      remove_monitoring_assoc (in FUSEObjectUID uid);

      //Invoked  by monitors.
      void change_suspect_list (in FUSEObjectUID uid,
                                in MonitorableStatus status);

      //To query about a specific Monitorable.
      Boolean
      is_it_suspect (in FUSEObjectUID uid);

      //A complete list of fault suspected processes.
      SuspectList get_suspect_list();
    };


    interface GeneralMonitorStrategy {
      //Called by SuperMonitor.
      //Used to manipulate the fault monitoring parameters.
      void set_monitoring_interval(in FaultMonitoringIntervalAndTimeoutValue mi);

      FaultMontoringIntervalAndTimeout get_monitoring_interval();

      void start();

      void stop();

      //Creates an association between this monitor
      //and its correspondent SuperMonitor.
      void set_super_monitor(in SuperMonitor sm);

      //Inform which monitorable this monitor must
      //monitor and what is its identifier on the
      //SuperMonitor.
      void set_monitorable(in MonitorableStrategy ms,
                           in FUSEObjectUID mtb_uid);

      //Called by Monitorable. The message_id can
      //be used to identify out dated messages.
      oneway void i_am_alive(Long message_id);
    };

    interface GeneralMonitorFactory{
      GeneralMonitorStrategy create_general_monitor();
    };
}
```

Definições Relacionadas ao Objeto Monitorado

```
module FUSE {
  interface MonitorableStrategy {
    boolean is_alive();

    oneway void
    set_monitoring_interval(in MonitorUID uid,
                            in TimeBase::TimeT mi);

    TimeBase::TimeT
    get_monitoring_interval(in MonitorUID uid);

    oneway void
    run (in MonitorUID uid);

    oneway void
    stop(in MonitorUID uid);

    MonitorUID
    add_general_monitor(in GeneralMonitorStrategy gen_mon_stg);

    oneway void
    remove_general_monitor(in MonitorUID uid);

  };

  interface Notifier {
    FUSEObjectUID
    add_notifiable(in Notifiable ntb);

    Boolean
    remove_notifiable(in FUSEObjectUID ntb_uid);

  };

  interface Notifiable {
    void
    notify(in SuspectList suspect_list);
  };
}
```

Definições Relacionadas ao Objeto de Consenso.

```
module DISCUSS {
  typedef Long ConsensusID;
  typedef Long ConsensusMemberUID;

  struct Value {
    Long   ID;
    Any    data;
  };

  interface Translator {
    //for implicit instantiation.
    Value get_estimate(inout ConsensusID cid);
```

```
    void decide (in ConsensusID cid, in Value decision);
  };

  interface Consensus {
    //For implicit invocation.
    void launch (in Translator translator);

    //For explicit invocation.
    void propose (in Value estimate, in ConsensusID cid);

    Value decide (in ConsensusID cid);
  };

  interface ConsensusMember {
    FUSEObjectUID add_super_monitor (in SuperMonitor sm);

    Boolean  remove_super_monitor (in FUSEObjectUID smid);

    void add_consensus_member(in ConsensusMember member,
                              in ConsensusMemberUID cmuid);

    Boolean remove_consensus_member(in ConsensusMemberUID cmuid);
  };
};
```