Leandro Pacheco de Sousa

Um Serviço Distribuído de Detecção de Falhas Baseado em Disseminação Epidêmica

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre. Programa de Pós-Graduação em Informática, Setor de Ciências Exatas, Universidade Federal do Paraná.

Orientador: Prof. Dr. Elias Procópio Duarte Jr.

Curitiba

2009

Sumário

Resumo

Problemas de acordo são blocos fundamentais na construção de sistemas distribuídos tolerantes a falhas. É fato que vários destes problemas, dentre os quais o do consenso, não podem ser resolvidos de maneira determinística em sistemas assíncronos na presença de falhas. Os detectores de falhas não-confiáveis foram propostos como uma maneira de se contornar esta impossibilidade. Estes detectores são como oráculos que, quando consultados, retornam informações sobre falhas de outros processos. Este trabalho apresenta um serviço de detecção de falhas baseado em disseminação epidêmica. Para utilizar o detector, um processo precisa implementar o serviço e participar de um grupo de detecção. A qualquer momento, um processo pode consultar seu detector e obter uma lista de processos considerados falhos ou corretos. O funcionamento do algoritmo pode ser alterado através de parâmetros do serviço. Este serviço foi implementado para a plataforma JXTA. Um simulador também foi implementado para permitir a avaliação do algoritmo para um número maior de processos. Resultados experimentais são apresentados para diferentes valores dos parâmetros do detector.

Abstract

Agreement problems are fundamental blocks for the design and implementation of fault-tolerant distributed systems. Some of these problems, such as consensus, cannot be deterministically solved in asynchronous systems subject to process crashes. To circumvent this impossibility, the concept of unreliable failure detectors was proposed. An unreliable failure detector is an oracle that can provide information about process crashes. This work presents a failure detection service based on a gossip strategy. To use the detector, a process must implement the service and join a detection group. At any time, a process can ask its detector and obtain a list of processes considered failed or correct. The detection algorithm can be configured by changing service parameters. The service was implemented in the JXTA platform. A simulator was also implemented so the detector could be evaluated for a larger number of processes. Experimental results are shown for a variety of detector parameter values.

Capítulo 1

Introdução

Redes de computadores têm se tornado não apenas extremamente populares mas também pervasivas, estando presentes nas mais diversas atividades de indivíduos e organizações. Estas redes se tornam cada vez maiores, mais complexas e mais poderosas. O maior exemplo disso é a explosão no crescimento da Internet na última década. A disponibilidade de recursos é cada vez maior, mas devido à complexidade e heterogeneidade dos sistemas conectados, a utilização destes recursos não é uma tarefa fácil. Além disso, o funcionamento correto dos sistemas distribuídos que executam nestas redes é crítico para seus usuários [41].

Grande parte dos problemas relacionados aos sistemas distribuídos requer algum tipo de coordenação entre os diversos componentes [28, 49], chamados aqui de *processos*. Estes processos se comunicam através de *canais de comunicação*. Tanto os processos quanto os canais de comunicação podem sofrer falhas. Quando um processo sofre uma falha, ele é dito *falho*, caso contrário, o processo está *correto*. Desse modo, um requisito fundamental para a coordenação entre os processos é que estes conheçam os estados uns dos outros, para que estes possam tomar as providências necessárias em caso de falha. Em alguns tipos de sistemas distribuídos, esta pode ser uma tarefa difícil ou mesmo impossível. Este é o caso dos *sistemas assíncronos*. Neste tipo de sistema, tanto os processos quanto os canais de comunicação podem se comportar de maneira arbitrariamente lenta, tornando um processo falho indistinguível

de um processo muito lento. Este problema é conhecido na literatura como a *impossibilidade FLP* [26]. Este resultado torna impossível a resolução de uma classe de problemas distribuídos de extrema importância, os chamados *problemas de acordo*. Dentre os problemas de acordo, o *consenso* é um dos mais fundamentais. De maneira simplificada, o consenso requer que os processos participantes cheguem a uma decisão em comum, dados os votos iniciais de cada processo.

Como uma maneira de contornar a *impossibilidade FLP* e tornar possível a resolução do consenso, Chandra et al. criaram os *detectores de falhas não-confiáveis* [13]. Cada processo tem acesso a seu próprio detector, que funciona como um tipo de oráculo que fornece informações sobre o estado dos demais processos. Estes detectores são ditos não-confiáveis pois podem cometer *enganos*. Um detector comete um engano ao informar que um processo correto está falho. Desse modo, os detectores de falhas precisam fornecer certas garantias, que são traduzidas em propriedades de *completude* e *exatidão*. As propriedades de completude são relacionadas à capacidade do detector de identificar falhas, isto é, o quão completa é a informação sobre os processos falhos. Por outro lado, as propriedades de exatidão informam sobre a capacidade do detector de não cometer enganos, ou seja, o quão precisa é a informação sobre as falhas. Um detector, para ser de alguma utilidade, deve fornecer alguma garantia para os dois tipos de propriedade. De acordo com as combinações destas propriedades, os detectores podem ser organizados em diversas classes.

Em [13, 11], Chandra et al. provam que, em um sistema distribuído assíncrono acrescido de detectores de falhas não-confiáveis, o problema do consenso pode ser resolvido mesmo na presença de falhas. Os autores ainda provam qual é a classe de detectores mais fraca que pode ser utilizada para possibilitar o consenso. A partir deste trabalho inicial, muitos outros surgiram estudando a relação entre determinados problemas de acordo e a classe de detectores mais fraca com a qual uma solução pode ser obtida. Dentre estes, podemos citar os casos da difusão atômica [13], da confirmação atômica não-bloqueante [30, 32] e a gestão de grupos [28, 29, 12]. Deste modo, os detectores de falhas são utilizados como uma abstração das

propriedades temporais de um sistema. Enquanto estas propriedades são específicas de cada implementação e sistema, os detectores de falhas são entidades abstratas que fornecem informações sobre os processos. Um problema de acordo pode ser resolvido independentemente das propriedades do sistema, caso este forneça o detector adequado. Assim, detectores de falhas são ferramentas de grande importância teórica no estudo e comparação de problemas de acordo.

É fato que, em sistemas completamente assíncronos, detectores de falhas que forneçam alguma garantia de completude e exatidão *não podem ser implementados*. Ainda assim, o estudo da implementação destes detectores é de grande importância. Algoritmos que fazem o uso de um detector são mais genéricos, pois não precisam se preocupar com as caracterísitas temporais do sistema. Outro fator importante é que, sabendo que tipo de detector pode ser implementado em um determinado modelo de sistema, pode-se saber quais tipos de problema podem ser resolvidos no mesmo. Estudos sobre a implementação de detectores em alguns modelos mais fortes de sistema podem ser encontrados na literatura. Podemos citar a implementação em sistemas de sincronia parcial [13, 40, 39], em sistemas com propriedades probabilísticas [14, 35] e em sistemas onde a troca de mensagens segue um determinado padrão [44].

Do ponto de vista prático, ou seja, da implementação em sistemas reais, os detectores de falhas também são uma abstração interessante. Estes sistemas são normalmente assíncronos e dificilmente podem ser feitas garantias sobre as propriedades temporais dos mesmos. Por outro lado, estes sistemas geralmente alternam entre períodos de instabilidade e estabilidade. Assim, um detector normalmente irá fornecer informações aproximadas sobre os processos, mas durante períodos estáveis suficientemente longos, os detectores podem fornecer informações corretas [47]. Além disso, detectores proporcionam modularidade na implementação de aplicações distribuídas. Estas se tornam menos complexas, pois não precisam se preocupar com certas características do sistema (que são abstraídas na entidade do detector). Outro

ponto importante é que várias aplicações podem se utilizar de um mesmo módulo de detecção, evitando a redundância nas implementações.

Um tipo de sistema distribuído que vem ganhando crescente importância é o *peer-to-peer* [46], ou simplesmente P2P. Estes são sistemas dinâmicos caracterizados pela auto-organização de seus componentes, chamados *peers*, com o objetivo do compartilhamento de seus recursos, sejam estes conteúdo, tempo de processamento ou qualquer outro tipo de serviço. A plataforma JXTA [3] é uma plataforma aberta que tem como objetivo facilitar o desenvolvimento de aplicações no modelo P2P. Ela é formada por um conjunto de protocolos que visam facilitar a localização de recursos e a comunicação entre os *peers*.

Este trabalho apresenta a especificação e implementação de um serviço de detecção de falhas baseado em disseminação epidêmica. O protocolo de detecção de falhas é probabilístico, se utilizando de uma estratégia de envio de *gossips* [34]. Para utilizar o detector, um processo precisa implementar o serviço e participar de um grupo de detecção. A qualquer momento, um processo pode consultar seu detector e obter uma lista de processos considerados falhos e corretos. O funcionamento do algoritmo pode ser alterado através de parâmetros do serviço. O serviço de detecção foi implementado para a plataforma JXTA. Um simulador também foi implementado, utilizando a biblioteca SMPL [42]. Resultados experimentais são apresentados tanto para a implementação JXTA quanto para a simulação. Estes experimentos visam avaliar a implementação do serviço e o funcionamento do algoritmo de detecção para diferentes configurações dos parâmetros do detector.

Os resultados da simulação indicam que o serviço proposto é escalável com o número de *peers*, tanto em relação à troca de mensagens quanto à qualidade da detecção de falhas. Estes resultados também mostram que a estratégia de disseminação epidêmica é robusta e apresenta vantagens significativas em grupos com grande número de *peers*. Porém, os experimentos realizados para a plataforma JXTA apontam para o uso elevado de processamento pela implementação do serviço de detecção, mesmo para um grupo pequeno de *peers*.

Este trabalho está organizado da seguinte maneira. No capítulo 2, é apresentada uma visão geral do conceito de sistemas distribuídos e de detectores de falhas não-confiáveis. São mostradas as relações entre detectores e alguns problemas de acordo e, finalmente, são apresentadas algumas implementações de detectores disponíveis na literatura. No capítulo 3, a plataforma JXTA é descrita juntamente com seus protocolos. No capítulo 4 é descrito o serviço de detecção JXTA-FD bem como sua implementação para a plataforma JXTA e o simulador. Também são apresentados resultados experimentais e de simulação. O capítulo 5 conclui este trabalho e apresenta algumas considerações finais.

Capítulo 2

Detectores de Falhas

Os detectores de falhas são uma abstração criada para contornar o resultado da impossibilidade FLP, que diz que o problema do consenso não pode ser resolvido em sistemas completamente assíncronos onde processos podem falhar. Este capítulo apresenta uma explicação sobre os modelos de sistema distribuído e a impossibilidade FLP, e fornece a especificação de detectores de falhas não-confiáveis, bem como sua classificação, relação com problemas de acordo e algumas implementações propostas na literatura.

2.1 Sistemas Distribuídos

Este trabalho considera um *sistema distribuído* como um conjunto de processos que se comunicam por meio de troca de mensagens [33]. Estas mensagens são transmitidas através de *canais de comunicação*.

Os sistemas distribuídos podem ser classificados de acordo com suas propriedades temporais. Estas propriedades normalmente se referem a duas características: latência entre o envio e a entrega de uma mensagem e a velocidade relativa de execução dos processos [25]. Um sistema é chamado *síncrono* quando existem limites superiores conhecidos para as duas características. Por outro lado, um sistema é dito *assíncrono* quando não assume hipótese

alguma sobre qualquer destas propriedades, tanto a latência de transmissão de mensagens quanto o tempo de processamento não são limitados.

Entre os dois extremos de sistemas síncronos e sistemas assíncronos, temos os sistemas *parcialmente síncronos*. Vários modelos podem ser utilizados para a especificação destes sistemas [23]. Em [25] dois modelos de sincronia parcial são definidos. No primeiro modelo, existem limites superiores para a latência na transmissão de mensagens e para o tempo de execução dos processos, mas estes limites não são conhecidos. No segundo modelo, os limites existem e são conhecidos, mas só são respeitados após um tempo desconhecido, chamado *Global Stabilization Time* (GST). Chandra et al. definem em [13] um modelo mais fraco que generaliza os dois modelos. Neste, os limites existem mas não são conhecidos, e só são respeitados após o GST.

A elaboração de algoritmos e protocolos para ambientes completamente assíncronos é de grande interesse da comunidade de sistemas distribuídos. Isto pois, dada a resolução de um problema para um sistema assíncrono, esta também pode ser aplicada a sistemas com propriedades temporais mais rígidas. Por outro lado, a impossibilidade de uma solução para um dado sistema também é aplicável a sistemas com propriedades temporais mais relaxadas.

Outra característica importante de um sistema distribuído é o seu *modelo de falhas*. Um modelo de falhas define quais os tipos de falha um processo pode sofrer. Na literatura, os principais tipos de falha reportados [28] são: falhas por parada (ou *crash*), por omissão, de desempenho e bizantinas. Uma *falha por parada* ocorre quando a execução de um processo é totalmente interrompida após um certo instante, e este entra em colapso. A *falha por omissão* ocorre quando um processo deixa de executar alguma ação prevista. Em uma *falha de desempenho*, um processo não executa alguma ação dentro do tempo previsto. Finalmente, o caso de *falha bizantina* [38], além de incluir os outros tipos de falha, prevê o caso em que um processo executa ações fora de sua especificação, ou seja, exibe comportamento arbitrário. Um processo que falha é dito *falho*. Caso contrário, dizemos que o processo é *correto* ou *sem-falha*. Neste trabalho, somente consideramos o caso de falhas por parada.

O comportamento dos canais de comunicação é outro ponto importante na definição de um modelo de sistema distribuído. Na literatura, o mais comum é assumir canais de comunicação assíncronos e confiáveis. Um canal é *confiável* quando não existe perda de mensagens. Uma mensagem enviada por um processo é, em algum momento, entregue ao destinatário. Se o canal é *assíncrono*, uma mensagem pode levar um tempo arbitrariamente longo para ser entregue ao seu destino. Outro canal usado com frequência na literatura é o *canal com perdas equitáveis (fair lossy)*. Neste tipo de canal, se um processo envia uma mensagem uma infinidade de vezes, esta é recebida uma infinidade de vezes em seu destino. Os casos de mensagens corrompidas, duplicadas ou criadas pelo canal de comunicação normalmente são desconsiderados, pois estes problemas podem ser facilmente contornados com o uso de mecanismos de verificação. Neste trabalho, consideramos que os canais de comunicação são assíncronos e confiáveis.

2.2 O Problema do Consenso

Os problemas de acordo são blocos fundamentais na implementação de sistemas distribuídos tolerantes a falhas [33]. De maneira geral, o objetivo destes problemas é fazer com que os processos cheguem a alguma decisão em comum sobre algum valor ou conjunto de valores. Dentre estes problemas, o consenso é um dos mais importantes. O *problema do consenso* [13] é composto de duas primitivas: *propose(v)* e *decide(v)*, sendo que *v* pertence a um dado conjunto de valores possíveis. Um processo *propõe* um valor *v* executando *propose(v)* e *decide* por um valor *v* quando executa *decide(v)*. Um protocolo de consenso possui as seguintes propriedades:

Terminação (Termination) - Todo processo em algum momento decide por um valor;

Integridade Uniforme (Uniform Integrity) - Todo processo decide no máximo uma vez;

Acordo (Agreement) - Dois processos corretos não decidem por valores diferentes;

Validade Uniforme (Uniform Validity) - Se um processo decide por *v*, então *v* foi proposto por algum processo.

Esta definição de consenso permite que processos falhos decidam por valores diferentes. Uma especificação mais forte do consenso, chamada *consenso uniforme*, exclui esta possibilidade [47]. Ela substitui a propriedade de acordo pela seguinte propriedade:

Acordo Uniforme (Uniform Agreement) - Dois processos *(corretos ou não)*, não decidem por valores diferentes.

Neste trabalho, o termo consenso é usado para se referir ao consenso uniforme.

Como veremos a seguir, o problema do consenso *não possui solução* em sistemas distribuídos assíncronos.

2.2.1 A Impossibilidade FLP

Um dos resultados teóricos mais importantes com relação ao problema do consenso foi apresentado por Fischer, Lynch e Paterson em [26]. Os autores provam que, dado um sistema completamente assíncrono, o consenso não pode ser resolvido dada a presença de uma única falha. Esta impossibilidade é conhecida na literatura como a *impossibilidade FLP*. Ela é consequência do fato de um processo falho ser indistinguível de um processo muito lento em um sistema assíncrono.

2.3 Os Detectores Distribuídos

A impossibilidade FLP pode ser contornada em sistemas assíncronos por meio de duas abordagens [41]. A primeira é o relaxamento das propriedades do consenso. Como exemplos, temos o problema do *k-acordo (k-agreement ou k-set agreement)* [18] e o problema do *acordo aproximado (approximate agreement)* [24]. A segunda abordagem utiliza a adição de garantias e propriedades ao sistema para fortalecer o modelo. Neste caso, podemos citar a resolução do

consenso em sistemas de sincronia parcial [23, 25], o uso de geradores de números aleatórios [9] e o uso de detectores de falhas não-confiáveis.

Os *detectores de falhas não-confiáveis* foram inicialmente propostos por Chandra e Toueg em [13] e [11] como uma maneira de se contornar a impossibilidade FLP. Um detector de falhas é como um oráculo que fornece informações sobre quais processos estão falhos. Os autores consideram detectores de falhas distribuídos e não-confiáveis. Cada processo tem acesso a um módulo de detecção de falhas. Cada módulo monitora um subconjunto dos processos do sistema e mantém uma lista de processos suspeitos de estarem falhos. Os detectores são *não-confiáveis* pois podem cometer enganos. Um *engano* ocorre quando um detector suspeita de um processo correto. Dizemos que um processo *confia* em outro processo se este não é suspeito por seu detector de falhas. Um detector pode repetidamente inserir e remover processos de sua lista de suspeitos.

2.3.1 As Propriedades dos Detectores de Falhas

Em [13], os detectores de falhas foram separados em classes de acordo com suas propriedades de *completude* (*completeness*) e *exatidão* (*accuracy*). De maneira simplificada, a completude garante que processos falhos sejam suspeitos e a exatidão restringe os enganos que um detector pode cometer. As propriedades de completude consideradas pelos autores são duas:

Completude Forte (Strong Completeness) - Todo processo que falha é, após um tempo, permanentemente suspeito por *todos* os processos corretos;

Completude Fraca (Weak Completeness) - Todo processo que falha é, após um tempo, permanentemente suspeito por *algum* processo correto.

As propriedades de exatidão consideradas são quatro:

Exatidão Forte (Strong Accuracy) - Nenhum processo é suspeito antes de falhar;

Exatidão Fraca (Weak Accuracy) - Algum processo correto nunca é suspeito;

Exatidão Forte Após Um Tempo (Eventual Strong Accuracy) A partir de um certo tempo, nenhum processo correto é suspeito por outro processo correto;

Exatidão Fraca Após Um Tempo (Eventual Weak Accuracy) - A partir de um certo tempo, algum processo correto nunca é suspeito por outro processo correto.

As propriedades de exatidão forte e exatidão fraca são chamadas *propriedades de exatidão perpétuas (perpetual accuracy properties)* [40].

2.3.2 A Classificação de Detectores de Falhas

De maneira isolada, é trivial garantir tanto a completude forte quanto a exatidão forte. Para a completude, basta que o detector suspeite permanentemente de todos os processos. Para a exatidão, basta que o detector nunca suspeite de qualquer processo. Um detector, para ser de alguma utilidade, precisa satisfazer garantias tanto de completude quanto de exatidão. Desse modo, oito classes de detectores surgem da combinação destas propriedades. As classes e suas notações são mostradas na Tabela 2.1.

Completude	Exatidão			
	Forte	Fraca	Forte Após um Tempo	Fraca Após um Tempo
Forte	Perfeito $\mathcal{P}$	Forte $\mathcal{S}$	Perfeito Após um Tempo $\Diamond\mathcal{P}$	Forte Após um Tempo $\Diamond\mathcal{S}$
Fraca	$\mathcal{Q}$	Fraco $\mathcal{W}$	$\Diamond\mathcal{Q}$	Fraco Após um Tempo $\Diamond\mathcal{W}$

Tabela 2.1: Classes de detectores de falhas definidas pelas propriedades de completude e exatidão.

Em [13], Chandra et al. definem a *redutibilidade* entre detectores de falhas. Dados dois detectores $\mathcal{D}$ e $\mathcal{D}'$, dizemos que $\mathcal{D}'$ é *redutível* a $\mathcal{D}$ se existe algum algoritmo que transforma $\mathcal{D}$ em $\mathcal{D}'$. Neste caso, qualquer problema que pode ser resolvido com o uso de $\mathcal{D}'$ também pode ser resolvido usando $\mathcal{D}$, dizemos que $\mathcal{D}'$ é *mais fraco* que $\mathcal{D}$ e usamos a notação $\mathcal{D} \geq \mathcal{D}'$. Se $\mathcal{D} \geq \mathcal{D}'$ e $\mathcal{D}' \geq \mathcal{D}$, dizemos que $\mathcal{D}$ e $\mathcal{D}'$ são *equivalentes* e utilizamos a notação $\mathcal{D} \cong \mathcal{D}'$.

A relação de redutibilidade também pode ser aplicada às classes de detectores. Dadas duas classes $\mathcal{C}$ e $\mathcal{C}'$, se para cada detector $\mathcal{D} \in \mathcal{C}$ existe um detector $\mathcal{D}' \in \mathcal{C}'$ tal que $\mathcal{D} \geq \mathcal{D}'$, dizemos que $\mathcal{C}'$ é *mais fraca* que $\mathcal{C}$ e utilizamos a notação $\mathcal{C} \geq \mathcal{C}'$. Se $\mathcal{C} \geq \mathcal{C}'$ e $\mathcal{C}' \geq \mathcal{C}$ dizemos que as duas classes são *equivalentes* e usamos a notação $\mathcal{C} \cong \mathcal{C}'$.

Ainda em [13], os autores propõem um algoritmo que transforma qualquer detector que satisfaça a completude fraca em um que satisfaz a completude forte. O algoritmo proposto preserva as propriedades de exatidão do detector original. Dado este algoritmo, juntamente com a transformação trivial da completude forte para fraca, eles chegam à conclusão de que as duas propriedades de completude são, na realidade, equivalentes. Com este resultado, podemos estabelecer as seguintes relações entre as classes de detectores: $\mathcal{P} \cong \mathcal{Q}$, $\mathcal{S} \cong \mathcal{W}$, $\diamondsuit\mathcal{P} \cong \diamondsuit\mathcal{Q}$ e $\diamondsuit\mathcal{S} \cong \diamondsuit\mathcal{W}$. Devido a estas relações de equivalência, a grande maioria dos trabalhos na literatura se limitam ao estudo das classes da primeira linha da Tabela 2.1.

2.4 A Relação Entre Problemas de Acordo e os Detectores de Falhas

Esta seção expõe alguns dos resultados propostos na literatura sobre a resolução de diferentes problemas de acordo com a utilização de detectores de falhas.

2.4.1 O Consenso

Chandra et al. em [11, 13] provam que o detector mais fraco capaz de resolver o problema do consenso, como definido na seção 2.2, é o detector $\diamondsuit\mathcal{W}$. Como os autores também provam a equivalência entre as classes de detector $\diamondsuit\mathcal{W}$ e $\diamondsuit\mathcal{S}$, muitos trabalhos consideram esta última como a mais fraca a resolver o consenso.

Outra classe de detectores frequentemente relacionada à resolução do consenso é a classe Ω, dita *ômega*. Esta classe foi inicialmente proposta por Chandra et al. em [11]. Um módulo

de detecção Ω executando em um processo p, quando consultado, retorna o identificador de um processo, chamado de $leader_p$. Estes detectores possuem a seguinte propriedade:

- Existem um processo correto l e um tempo t tais que, a partir de t, para todo processo vivo p, $leader_p = l$.

O detector de falhas Ω pode ser visto como um mecanismo de *eleição de líder*. Os detectores Ω de cada processo, após um tempo, passam a retornar o mesmo $leader_p$. Um algoritmo que faz uso do detector Ω para a resolução do consenso pode ser encontrado em [45].

Em [11, 15], é provada a equivalência entre as classes Ω e $\diamondsuit\mathcal{W}$. Isto é, as duas classes possuem o mesmo poder computacional. Qualquer protocolo de consenso baseado em detectores das classes Ω, $\diamondsuit\mathcal{W}$ e $\diamondsuit\mathcal{P}$ necessita de uma maioria de processos corretos.

2.4.2 A Difusão Confiável

A *difusão confiável (reliable broadcast)* [36] é um mecanismo de comunicação fundamental para a construção de sistemas distribuídos tolerantes a falhas. De maneira simplificada, ela garante que os processos corretos entregam o mesmo conjunto de mensagens. A difusão confiável é formalmente definida por meio de duas primitivas: *broadcast(m)* e *receive(m)*. Um processo *envia por difusão* uma mensagem *m* quando executa *broadcast(m)*. Um processo *entrega* uma mensagem *m* quando termina a execução de *receive(m)*. As propriedades da difusão confiável são:

Validade (Validity) - Se um processo correto envia por difusão uma mensagem *m*, então ele em algum momento entrega *m*;

Acordo (Agreement) - Se um processo correto entrega uma mensagem *m*, então todo processo correto entrega *m* em algum momento;

Integridade (Integrity) - Para cada mensagem *m*, todo processo correto entrega *m* no máximo uma vez, e somente se *m* foi enviada anteriormente por algum processo.

O problema da difusão confiável *pode ser resolvido em sistemas assíncronos* sem a adição de propriedades adicionais, satisfeita a condição de que a falha dos canais de comunicação não particionem a rede [36]. Ainda em [36], os autores propõem um algoritmo que implementa o mecanismo de difusão confiável.

2.4.3 A Difusão Confiável Uniforme

A *difusão confiável uniforme (uniform reliable broadcast)*, aqui chamada de URB, é uma primitiva de comunicação semelhante à difusão confiável, porém mais forte que ela. Ela requer que, se um processo qualquer entrega uma mensagem *m*, então todo processo correto deve entregar *m* [7]. Formalmente, ela é composta pelas primitivas *broadcast(m)* e *deliver(m)*. As propriedades da URB são:

Validade (Validity) - Se um processo envia por difusão uma mensagem *m*, então ele entrega *m* em algum momento;

Acordo Uniforme (Uniform Agreement) - Se algum processo entrega uma mensagem *m*, então todo processo correto entrega *m* em algum momento;

Integridade Uniforme (Uniform Integrity) - Para cada mensagem *m*, todo processo entrega *m* no máximo uma vez, e somente se *m* foi enviada anteriormente por algum processo.

A diferença entre a difusão confiável e a URB está no comportamento dos processos falhos. Na difusão confiável, processos falhos podem entregar mensagens de maneira arbitrária, sem que estas sejam entregues pelos processos corretos ou mesmo sem que tenham sido enviadas por algum processo. Este comportamento não é permitido na versão uniforme das propriedades de Acordo e Integridade.

Em [7], Aguilera et al. apresentam uma nova classe de detectores de falhas chamada θ. Os autores provam que os detectores pertencentes à classe θ são os detectores mais fracos que podem resolver o problema da URB. Um detector da classe θ possui as seguintes propriedades:

θ-*completude (*θ-*completeness)* - A partir de um certo tempo, processos corretos não confiam em nenhum processo que falha.

θ-*exatidão (*θ-*accuracy)* - Se existe algum processo correto, então, em todo tempo, todo processo confia em ao menos um processo.

A θ-exatidão permite que processos confiem em processos falhos. O conjunto de processos no qual um processo p confia pode mudar ao longo do tempo e não é necessariamente o mesmo que de um outro processo q.

Larrea et al. provam em [40] que, em um modelo de sincronia parcial como o de [13], uma maioria de processos corretos é condição suficiente e necessária para a implementação do θ

Um algoritmo que implementa a URB com o uso de um detector θ foi proposto em [7].

2.4.4 A Difusão Atômica

A *difusão atômica (atomic broadcast)* é um mecanismo de comunicação de grande importancia na construção de sistemas distribuídos tolerantes a falhas. Suas propriedades formais são todas as propriedades da difusão confiável mais a seguinte propriedade:

Ordem Total (Total Order) - Se dois processos corretos p e q entregam duas mensagens m e m', então p entrega m antes de m' se e somente se q entrega m antes de m'.

Em [13], Chandra et al. provam que o problema da difusão atômica é equivalente ao problema do consenso em sistemas assíncronos. A prova é obtida com a redução dos problemas um ao outro. Desse modo, assim como para o consenso, os detectores $\diamondsuit\mathcal{S}$ são os mais fracos para a resolução da difusão atômica.

2.4.5 A Difusão Atômica Seletiva

A *difusão atômica seletiva (atomic multicast)* é um mecanismo de comunicação semelhante à difusão atômica. A diferença está no fato de que em um mecanismo de difusão seletiva,

as mensagens são entregues somente a um grupo determinado de processos. Desse modo, podemos dizer que a difusão comum (broadcast) é um caso especial de difusão seletiva na qual o grupo de destinatários das mensagens é o conjunto de todos os processos [20]. Quando um processo envia uma mensagem por difusão atômica seletiva, pode-se dizer que ele envia a mensagem *por multicast*. O conjunto dos processos destinatários de uma mensagem m é chamado de $dest(m)$. O problema da difusão atômica seletiva possui quatro propriedades [20]:

Validade (Validity) - Se um processo correto envia uma mensagem m por multicast, então algum processo correto em $dest(m)$ entrega m em algum momento.

Acordo Uniforme (Uniform Agreement) - Se um processo entrega uma mensagem m, então todos processos corretos em $dest(m)$ entregam m em algum momento.

Integridade Uniforme (Uniform Integrity) - Para qualquer mensagem m, todo processo p entrega m no máximo uma vez, e somente se m foi previamente enviada por algum processo e p pertence a $dest(p)$.

Ordem Total Uniforme (Uniform Total Order) - Se dois processos p e q entregam duas mensagens m e m', então p entrega m antes de m' se e somente se q entrega m antes de m'.

As propriedades uniformes citadas anteriormente possuem variantes não-uniformes, ou seja, que não limitam o comportamento de processos falhos. Algumas classes de problema necessitam das propriedades uniformes, mas nos casos em que elas não são necessárias, pode-se utilizar as seguintes propriedades [20]:

Acordo (Agreement) - Se um processo *correto* entrega uma mensagem m, então todos processos corretos em $dest(m)$ entregam m em algum momento.

Integridade (Integrity) - Para qualquer mensagem m, todo processo *correto* p entrega m no máximo uma vez, e somente se m foi previamente enviada por algum processo e p pertence à $dest(p)$.

Ordem Total (Total Order) - Se dois processos *corretos* p e q entregam duas mensagens m e m', então p entrega m antes de m' se e somente se q entrega m antes de m'.

A implementação de protocolos de difusão atômica seletiva com detectores de falhas depende da especificação do mesmo. De acordo com as propriedades adotadas (algumas das quais não são mencionadas neste trabalho), o protocolo pode ser implementado com detectores não-confiáveis ($\Diamond\mathcal{S}$ e $\Diamond\mathcal{P}$). Um estudo extensivo das características e propriedades dos mecanismos de difusão seletiva e de algoritmos e protocolos para a implementação desta primitiva pode ser encontrado em [21].

2.4.6 A Confirmação Atômica Não-Bloqueante

O problema da *confirmação atômica não-bloqueante (non-blocking atomic commit)*, aqui chamado NBAC, surgiu no contexto da área de bancos de dados. Todos os processos responsáveis por uma dada transação deveriam chegar à mesma conclusão, validar ou abortar a mesma. Inicialmente, cada processo escolhe entre *sim* (validar) ou *não* (abortar), com base em seu conhecimento local, e emite seu voto. Depois, de acordo com o conjunto de votos emitidos, os processos devem chegar à decisão final de *abortar (abort)* ou *validar (commit)*. Formalmente, o NBAC possui as seguintes propriedades:

NBAC-Terminação (Termination) - Todo processo correto decide em algum momento.

NBAC-Validade (Validity) - O valor decidido é *commit* ou *abort*.

NBAC-Justificativa (Justification) - Se um processo decide por *commit*, todos processos votaram *sim*.

NBAC-Obrigação (Obligation) - Se todos processos votaram *sim* e não houve falhas, então o valor decidido é *commit*.

NBAC-Acordo (Agreement) - Dois processos não decidem por valores diferentes.

A solução clássica para o problema da confirmação atômica é o protocolo de *confirmação em duas fases (two-phase commit)* [27]. O problema desta solução é que, na presença de falhas, o protocolo é *bloqueante*. Um protocolo é *não-bloqueante* quando a decisão entre abortar ou validar é obtida mesmo perante a ocorrência de falhas. Para se resolver o NBAC

em um sistema assíncrono com falhas, é necessária a presença de detectores de falhas. Os protocolo de *confirmação em três fases (three-phase commit)* [48] são não bloqueantes, mas precisam de um detector da classe $\mathcal{P}$ [30]. Em [30], Guerraoui et al. provam que um detector da classe $?\mathcal{P}$ é necessário, porém não suficiente, para a resolução do NBAC. Somente são considerados detectores *atemporais (timeless)*, ou seja, que não fornecem informação sobre quando uma falha ocorre. Os detectores da classe $?\mathcal{P}$, chamados *anonimamente perfeitos (anonymously perfect)*, possuem as seguintes propriedades:

Completude Anônima (Anonymous Completeness) - Se uma falha ocorre, todo processo correto é, após um tempo, informado de que alguma falha ocorreu.

Exatidão Anônima (Anonymous Accuracy) - Uma falha só é detectada se algum processo falhou.

Em [30] e [32], é provado que a classe de detectores mais fraca suficiente para a resolução do NBAC é a $?\mathcal{P} + \diamondsuit\mathcal{S}$. Os detectores desta classe são os detectores que satisfazem todas as propriedades das classes $?\mathcal{P}$ e $\diamondsuit\mathcal{S}$. Os autores também mostram que esta classe é estritamente mais fraca que a classe $\mathcal{P}$. Uma solução utilizando a classe $?\mathcal{P} + \diamondsuit\mathcal{S}$ é descrita em [32]. Intuitivamente, o que ela faz é dividir o problema em duas partes. A primeira se utiliza do $?\mathcal{P}$ para descobrir se houve alguma falha durante a votação. A segunda parte se utiliza do consenso, e consequentemente do $\diamondsuit\mathcal{S}$, para chegar à decisão final.

2.4.7 A Gestão de Grupos

Em sistemas distribuídos, processos podem ser organizados em grupos. Normalmente um grupo é responsável por uma determinada computação ou pelo fornecimento de algum serviço ou recurso. Ao longo da execução do sistema, a composição de um grupo pode sofrer alterações. Processos podem entrar ou sair do grupo, e processos que falham devem ser removidos. A composição do grupo em um dado instante [28] é chamada de *visão*. Cada processo do grupo mantém uma visão local da composição do grupo. Um serviço de *gestão de compo-*

sição de grupos (group membership) tem o objetivo de manter visões *consistentes* e *justas* [28]. As visões são consistentes se são concordantes entre si. Uma visão é dita justa se reflete mudanças reais na composição do grupo.

Em um sistema em que a rede pode se particionar devido à falha de processos, os grupos de processos também podem sofrer particionamentos. Isto gera a necessidade de duas abordagens para a gestão da composição de grupos [28]: *gestão com componente primária (primary group membership)* e *gestão com componentes particionáveis (partitionable group membership)*.

A *gestão com componente primária* garante que somente uma visão existe em cada instante. Esta visão engloba a componente majoritária do sistema. A sequência de visões é *totalmente ordenada*. Este tipo de gestão é ideal para sistemas nos quais a ocorrência de falhas é baixa e as aplicações possuem requisitos fortes de consistência [28]. Em [12], Chandra et al. provam a impossibilidade da gestão com componente primária em sistemas assíncronos com falhas por parada. Esta impossibilidade surge da impossibilidade do consenso. Detectores de falhas podem ser utilizados para a resolução do problema. Como processos falhos precisam ser removidos, um detector completamente confiável, ou seja $\mathcal{P}$, seria necessário para a implementação da gestão de componente primária. Para contornar isto, a especificação do problema pode ser enfraquecida. Se a exclusão de processos suspeitos for permitida, um detector $\Diamond\mathcal{S}$ pode ser usado para a implementação da gestão com componente primária [29]. Especificações deste tipo de serviço podem ser encontradas em [12] e [29].

Em sistemas muito dinâmicos e nos quais falhas são frequentes, a estratégia de *gestão com componentes particionáveis* pode ser mais apropriada [28]. Este tipo de gestão permite a existência de visões distintas em um dado instante, ditas *visões concorrentes*. Visões concorrentes permitem a existência de múltiplos *subgrupos*, cada um com sua visão. Cada subgrupo deve continuar fornecendo o serviço ou recurso pelo qual o grupo é responsável. Não existe comunicação entre estes subgrupos. Em um determinado momento, estes subgrupos podem vir a se combinar novamente. Neste caso, o serviço de gestão deve fornecer um processo de

reconciliação das visões (group merging). Um serviço de gestão com componentes particionáveis escapa à impossibilidade FLP, pois não precisam manter as visões locais em acordo. Ainda sim, alguma garantia deve ser fornecida pelo serviço, de modo que este mantenha alguma coerência nos estados do sistema e seja de utilidade às aplicações [28].

2.4.8 A Comunicação Quiescente

A comunicação quiescente não é um problema de acordo propriamente dito, mas também pode ser implementada com o uso de detectores de falha não-confiáveis. Um algoritmo de comunicação é dito *quiescente (quiescent)* se este, após um tempo, pára de enviar mensagens. Em um sistema assíncrono com falhas por parada, um algoritmo de comunicação confiável e quiescente é impossível [5]. Aguilera et al. em [5] propõem o uso de detectores de falhas não-confiáveis para a resolução do problema da comunicação quiescente. As implementações de detectores de falhas em sistemas distribuídos não são quiescentes. Ainda sim, os autores sugerem a utilização destes para a comunicação quiescente. Isto pois um dado detector normalmente é utilizado por diversas aplicações, e precisa continuar executando e verificando o estado de outros processos enquanto o sistema estiver funcionando. Em [6], os autores mostram que a classe de detectores mais fraca para a implementação da comunicação quiescente é a classe $\diamondsuit\mathcal{P}$. Este resultado é obtido levando em consideração somente detectores que emitem uma lista de suspeitos.

Em [5], um outro tipo de detector é proposto pelos autores. Este detector é chamado *Heartbeat*, e sua característica mais importante é que ele pode ser implementado em sistemas assíncronos com enlaces do tipo *fair lossy*, sem o uso de *timeouts*. O detector Heartbeat fornece aos processos, para cada outro processo q sendo monitorado, um valor inteiro não negativo hb_q chamado valor de *heartbeat*. As propriedades formais do detector Heartbeat são:

HB-Completude (HB-Completeness) - Em todo processo correto, o valor de *heartbeat* hb_q é limitado se q está falho.

HB-Exatidão (HB-Accuracy) - Em todo processo correto, o valor de *heartbeat* hb_q para todo q não diminui. Em todo processo correto, o valor de *heartbeat* hb_q não é limitado se q é um processo correto.

De maneira simplificada, estas propriedades garantem que o valor de *heartbeat* hb_q em um dado processo correto aumenta indefinidamente enquanto o processo q não falha e pára de aumentar quando o processo q falha. Implementações quiescentes de comunicação confiável ponto-a-ponto e por difusão, se utilizando de um detector hearbeat, são propostas em [5].

2.4.9 Outros Problemas

Diversos trabalhos sobre a relação entre problemas e detectores de falha podem ser encontrados na literatura. Dentre estes, podemos citar [28, 47, 22, 31, 19, 16, 21].

2.5 A Implementação de Detectores de Falhas

Detectores de falhas não podem ser implementados em sistemas assíncronos, devido à impossibilidade FLP. Ainda sim, o estudo de detectores de falhas do ponto de vista prático é de extrema importância na implementação de soluções para os problemas de sistemas distribuídos. Chandra et al. mostram em [13] que detectores de falhas não-confiáveis podem ser implementados em sistemas de sincronia parcial. No modelo de sistema adotado pelos autores, existem limites superiores tanto para transmissão de mensagens quanto para o tempo de execução dos processos, mas estes são desconhecidos e só valem após um tempo de estabilização, chamado GST. Desse modo, uma implementação de um detector que faz o uso de *timeouts* pode, após um tempo, passar a detectar a ocorrência de falhas dos processos.

Em sistemas reais, o modelo de sincronia parcial teórico não é respeitado. Porém, estes sistemas normalmente alternam entre períodos de estabilidade e instabilidade. Durante um período de instabilidade, o sistema é completamente assíncrono. Já durante um período de estabilidade, pode-se dizer que o sistema respeita algumas propriedades temporais, assim de-

tectores de falhas são possíveis. Se este período for longo o suficiente, problemas como o do consenso podem ser resolvidos nestes sistemas [47].

A grande vantagem do uso de detectores de falhas em sistemas reais está na modularidade que eles proporcionam à implementação de algoritmos distribuídos. As características temporais do sistema são encapsuladas pela entidade do detector de falhas. Isto torna possível a elaboração de algoritmos que podem ser executados em qualquer tipo de sistema, desde que este forneça um detector de falhas adequado ao algoritmo em questão. Outra vantagem do uso dos detectores é que um mesmo detector pode ser utilizado simultaneamente por diversas aplicações, evitando a duplicação de mecanismos e, consequentemente, de mensagens.

Nesta seção, serão abordados alguns algoritmos e implementações de detectores de falhas não-confiáveis que foram propostos na literatura.

2.5.1 O Detector de Chandra e Toueg

Em [13], Chandra et. al. propõem um algoritmo para a implementação de um detector de falhas da classe $\diamondsuit\mathcal{P}$ em um sistema de sincronia parcial. O algoritmo pode ser visto na Figura 2.1.

O algoritmo executa em cada processo p. Para cada outro processo q sendo monitorado, um valor de *timeout*, $\Delta_p(q)$, é mantido. Cada processo p envia periodicamente mensagens de *heartbeat* para todos os outros processos. Basicamente, o que o algoritmo faz é verificar quando um processo q não responde a mais do que $\Delta_p(q)$ unidades de tempo. Quando isto ocorre, o valor de $\Delta_p(q)$ é incrementado e o processo q é marcado como suspeito. Se um processo q responde dentro do tempo esperado, este é removido da lista de suspeitos. Após o GST, os valores de *timeout* para cada processo irão aumentar até o ponto em que são maiores que o limite superior do tempo de transmissão dos *heartbeats*.

Esta implementação de detector não é viável em um sistema real, pois o valor de *timeout* somente aumenta ao longo do tempo, e de maneira ilimitada.

Every process p executes the following:

$output_p \leftarrow \emptyset$
for all $q \in \Pi$ **do**
  $\Delta_p(q) \leftarrow$ default time-out interval
end for

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

|| *Task 2:* **repeat periodically**
for all $q \in \Pi$ **do**
  if $q \notin output_p$ **then**
    p did not receive "q-is-alive" during the last $\Delta_p(q)$ ticks of p's clock
    $output_p \leftarrow output_p \cup \{q\}$
  end if
end for

|| *Task 3:* **when** receives "q-is-alive" for some q
if $q \in output_p$ **then**
  $output_p \leftarrow output_p - \{q\}$
  $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
end if

Figura 2.1: Algoritmo de Chandra e Toueg para a implementação de um detector $\Diamond\mathcal{P}$ em modelos de sincronia parcial.

2.5.2 Qualidade de Serviço e o Detector NFD-E

Para o uso prático de detectores de falhas, aplicações podem ter necessidades além das propriedades eventuais dos detectores, propostas em [13]. Aplicações podem ter restrições temporais, e um detector que possui um atraso muito grande na detecção de falhas pode não ser suficiente. Por este motivo, Chen et al. propõem em [14] métricas para a *qualidade de serviço (quality of service)*, ou simplesmente QoS, de detectores de falhas.

De maneira geral, as métricas de QoS para um detector de falhas buscam descrever a *velocidade (speed)* e a *exatidão (accuracy)* da detecção. Em outras palavras, as métricas definem quão rápido o detector detecta uma falha e quão bem este evita enganos.

No modelo proposto pelos autores, o detector de falhas de um processo p possui, para cada outro processo q, um valor de estado S ou T, sendo S para suspeito e T para correto.

Uma *transição* ocorre quando o detector altera o estado de um processo q. Quando o estado passa de T para S ocorre uma *S-transição*. Quando o estado passa de S para T ocorre uma *T-transição*.

Chen et al. propõem três métricas de QoS primárias e quatro métricas derivadas. A primeira métrica primária é relacionada à velocidade de detecção:

T_D *: Tempo de detecção (detection time)* - É o período de tempo entre o tempo que um processo q falha e outro processo p detecta esta falha permanentemente. Mais especificamente, é o tempo entre a falha de um processo q e a S-transição final do detector em p.

As outras métricas de QoS são relacionadas à exatidão do detector, e levam em consideração *execuções sem falha (failure-free runs)*, ou seja, execuções em que os processos não falham. As duas métricas primárias restantes são:

T_{MR} *: Tempo de recorrência de enganos (mistake recurrence time)* - É o período de tempo entre dois enganos consecutivos. Mais especificamente, é o tempo de uma S-transições até a próxima.

T_M *: Duração de engano (mistake duration)* - É o tempo que um detector leva para corrigir um engano. Isto é, o tempo entre uma S-transição e a próxima T-transição.

Com base nas três métricas primárias, mais quatro métricas derivadas são definidas:

λ_M *: Taxa média de enganos (average mistake rate)* - Mede com que frequência o detector comete enganos. Mais especificamente, quantas S-transições ocorrem por unidade de tempo.

P_A *: Probabilidade da exatidão de uma consulta (query accuracy probability)* - É a probabilidade da saída do detector estar correta em um tempo qualquer aleatório.

T_G *: Duração de um período bom (good period duration)* - Um *período bom* é um intervalo de tempo em que o detector não comete falhas. Mais precisamente, é o tempo entre uma T-transição e a próxima S-transição.

T_{FG} *: Duração restante de um período bom (forward good period duration)* - Dado um tempo qualquer em que um processo p é considerado correto no detector, é o tempo restante até o fim do período bom, ou seja, até a próxima S-transição de p.

Todas as métricas foram definidas com base em um modelo de sistema probabilístico, utilizando a teoria de processos estocásticos. Deste modo, cada uma das métricas é definida como uma variável aleatória. A definição formal do sistema e das métricas pode ser encontrada em [14].

O Detector NFD-E

Chen et al. propõem em [14] um novo detector de falhas, chamado NFD-E, que pode ser configurado de acordo com os parâmetros de QoS necessários para a aplicação em questão. Este detector visa o sistema probabilístico proposto pelos autores. É assumido que cada processo possui um relógio local, não sincronizado com os demais, mas que estes relógios executam na mesma velocidade, isto é, os relógios não apresentam *drift* em relação uns aos outros.

O algoritmo do NFD-E pode ser visto na Figura 2.2. Este algoritmo ilustra o caso onde um processo q monitora outro processo p. A saída do detector é representada pela variável *output*. Esta variável possui valor *S* se p for suspeito e *T* caso contrário. O NFD-E faz o uso de *heartbeats*, que são enviados por p em intervalos de η unidades de tempo. O funcionamento do algoritmo é baseado numa estimativa do tempo de chegada dos *heartbeats* por q. Este tempo equivale aos termos τ do algoritmo. De maneira simplificada, durante o intervalo de tempo $[\tau_i, \tau_{i+1})$, q confia em p se e somente se q recebeu de p um *heartbeat* m_x, tal que $x \geq i$.

O detector NFD-E faz o uso de um configurador para calcular seus parâmetros de acordo com a QoS desejada. Os parâmetros são os termos α e τ do algoritmo. Tanto este configurador quanto o próprio algoritmo fazem uso de um estimador do comportamento probabilístico dos *heartbeats*. A Figura 2.3 ilustra a relação entre estes componentes. Detalhes sobre o configurador e o estimador podem ser encontrados em [14].

Process p executes the following: {Using p's local clock}

for all $i \geq 1$, at time $i.\eta$ **do**
  send heartbeat m_i to q
end for

Process q executes the following: {Using q's local clock}

|| **initialization:**
$\tau_0 \leftarrow 0$
$l \leftarrow -1$ {l keeps the largest sequence number in all messages q received so far}

|| **upon** τ_{l+1} = the current time
$output \leftarrow S$ {suspect p since no message received is still fresh at this time}

|| **upon** receipt of message m_j at time t
if $j > l$ **then**
  $l \leftarrow j$
  $\tau_{l+1} \leftarrow EA_{l+1} + \alpha$ {set the next freshness point}
  if $t < \tau_{l+1}$ **then**
    $output \leftarrow T$ {trust p since m_j is still fresh at time t}
  end if
end if

Figura 2.2: Algoritmo do detector NFD-E.

2.5.3 Detector de Falhas Baseado em Disseminação Epidêmica

Em [50], van Renesse et al. propõem um algoritmo de detecção de falhas que se utiliza de uma estratégia de disseminação epidêmica (*gossip*). De acordo com esta estratégia, processos enviam mensagens para um grupo de outros processos, estes escolhidos de maneira aleatória. O algoritmo proposto possui, informalmente, as seguintes propriedades:

1. A probabilidade de um engano é independente do número de processos.
2. O algoritmo é resistente a perdas de mensagens e falhas de processos. Isto é, uma porcentagem pequena de falhas de processos e perdas ou atrasos de mensagens não provoca o aumento no numero de enganos.
3. Se o *drift* dos relógios for insignificante, o algoritmo detecta falhas com uma probabilidade conhecida de enganos.

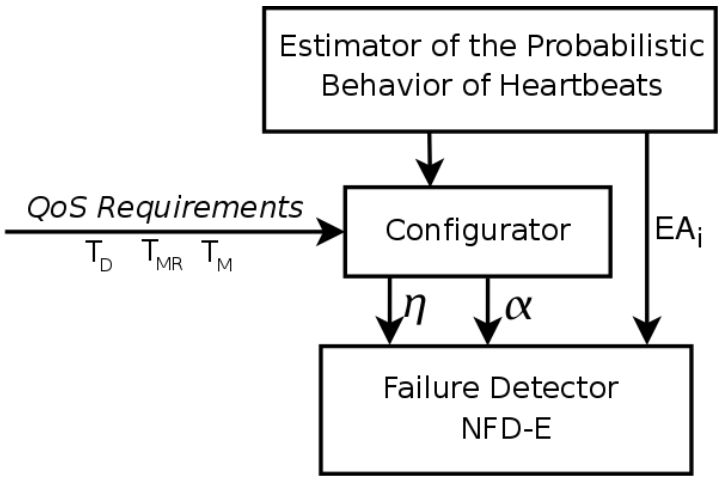

Figura 2.3: Relação entre os componentes do detector de falhas NFD-E.

4. O tempo de detecção aumenta em $O(n\log n)$ com o número de processos.
5. A banda utilizada pelo algoritmo aumenta no máximo linearmente com o número de processos.

O funcionamento do algoritmo é basicamente o seguinte:

- Cada processo mantém uma lista contendo, para cada processo que este conhece, o identificador do processo e um contador, chamado *heartbeat*. Este também mantém um *heartbeat* para si próprio.
- A cada T_{gossip} unidades de tempo, cada membro incrementa seu próprio *heartbeat*, escolhe aleatoriamente um processo de sua lista aleatoriamente e envia para ele uma mensagem de *gossip*, contendo sua lista de processos e *heartbeats*.
- Ao recebimento de uma mensagem de *gossip*, o processo une a lista recebida com sua própria lista, mantendo os *heartbeats* de maior valor.
- De tempos em tempos, cada processo envia por *broadcast* a sua lista. Este procedimento é feito para que o processo seja inicialmente reconhecido por seus vizinhos e para que o algoritmo possa se recuperar de possíveis particionamentos na rede.
- Cada membro também mantém, para cada processo conhecido, um timestamp com o último tempo em que o *heartbeat* para este processo aumentou. Se mais que T_{fail}

unidades de tempo se passaram desde o último aumento, este processo é considerado falho.

Este algoritmo não verifica o estado dos enlaces entre os processos. Se dois processos A e B não se comunicam diretamente, mas ambos podem se comunicar com um terceiro processo C, A e B não irão suspeitar um do outro como falhos. Supondo a existência de algum algoritmo de roteamento entre os processos, como ocorre na Internet, este tipo de situação se resolverá automaticamente.

Os autores sugerem ainda uma melhora para o algoritmo, para o caso do mesmo ser utilizado em um ambiente como o da Internet. Mais especificamente para o caso dos processos estarem espalhados em diferentes domínios e sub-redes. A otimização sugerida está em manter, juntamente com a lista original de processos, uma segunda lista, que identifica a sub-rede de cada processo. Esta lista é enviada juntamente com a primeira nas mensagens de *gossip*.

O funcionamento do algoritmo modificado dentro de cada sub-rede é o mesmo do original. A diferença está para mensagens enviadas entre as sub-redes. Para que o caminho entre duas sub-redes não seja sobrecarregado por mensagens de *gossip*, a probabilidade de um processo enviar uma mensagem para outra sub-rede é tal que, na média, somente uma mensagem de *gossip* é enviada para cada outra sub-rede a cada rodada. Em redes com mais níveis hierárquicos, esta técnica pode ser adaptada para a comunicação entre componentes de cada nível, contanto que os processos consigam identificar a localização dos outros processos dentro da hierarquia. Esta modificação possui a desvantagem de que o tempo de disseminação de informação entre componentes da rede aumenta.

Em [50] mais detalhes sobre o algoritmo e uma análise do mesmo podem ser encontrados.

2.5.4 O Detector do Protocolo SWIM

Das et al. propõem em [17] um protocolo de gestão da composição de grupos, chamado SWIM. De acordo com os autores, este é um protocolo de *consistência fraca*, isto é, a visão de cada processo não precisa ser consistente com a visão nos demais processos.

O protocolo SWIM é dividido em duas partes: um detector de falhas e um protocolo de disseminação da composição do grupo. Este detector de falhas foi proposto inicialmente em [35]. A especificação do algoritmo de detecção de falhas é mostrada na Figura 2.4.

Every process M_i executes the following:

$pr \leftarrow 0$ {period number}

|| *Task 1:* **Every T' time units**
$pr \leftarrow pr + 1$
Select random member Mj from view
Send a ping(M_i, M_j, pr) message to M_j
Wait for the worst-case message round-trip time for an ack(M_i, M_j, pr) message
if have not received an ack(M_i, M_j, pr) message yet **then**
  Select k members randomly from view
  Send each of them a ping-req(M_i, M_j, pr) message
  Wait for an ack(M_i, M_j, pr) message until the end of period pr
  if have not received an ack(M_i, M_j, pr) message yet **then**
    Declare M_j as failed
  end if
end if

|| *Task 2:* **On receipt of a ping-req(M_m, M_j, pr) ($M_j \neq M_i$)**
Send a ping(M_i, M_j, M_m, pr) message to M_j
On receipt of an ack(M_i, M_j, M_m, pr) message from M_j
  Send an ack(M_m, M_j, pr) message to M_m

|| *Task 3:* **On receipt of a ping(M_m, M_i, M_l, pr) message from a member** M_m
Reply with an ack(M_m, M_i, M_l, pr) message to M_m

|| *Task 4:* **On receipt of a ping(M_m, M_i, pr) message from a member** M_m
Reply with an ack(M_m, M_i, pr) message to M_m

Figura 2.4: Algoritmo do detector de falhas do protocolo SWIM.

No algoritmo do detector de falhas, os valores T' e k são parâmetros que devem ser determinados antes do início da execução. T' é a duração de um período do protocolo e k é o tamanho do subgrupo de detecção de falhas. Em [35], os autores mostram como estes valores podem ser definidos, de acordo com as necessidades da aplicação e o comportamento da rede.

Além do detector de falhas, o protocolo SWIM possui a componente de disseminação da composição do grupo. Esta componente faz o uso de uma técnica de *piggybacking*. O *piggybacking* consiste em enviar mensagens de um protocolo dentro de mensagens de um outro protocolo. No caso do protocolo SWIM, as mensagens da componente de disseminação são enviadas dentro das mensagens do detector de falhas (*ack*'s, *ping*'s e *ping-req*'s). Com isto, não são necessárias mensagens adicionais para a realização da disseminação. Desse modo, a informação é disseminada de uma maneira *epidêmica* (*infection-style*). Este estilo de disseminação é resistente a falhas de processos e perdas de mensagens. Em [17] os autores fazem uma análise mais detalhada das características deste método de disseminação.

O protocolo SWIM também faz algumas modificações ao comportamento do detector da Figura 2.4. Para diminuir o número de suspeitas falsas, causando a retirada de processos corretos de visões do grupo, um subprotocolo de suspeita é proposto. De maneira simplificada, o funcionamento deste subprotocolo é o seguinte:

- Considere um processo p qualquer executando o protocolo de detecção de falhas do SWIM. Quando p deixa de receber um *ack* de outro processo q, seja como resposta de um *ping* ou indiretamente através de um *ping-ack*, p não marca q como falho. p marca q como *suspeito* e dissemina esta informação através da componente de disseminação do SWIM. Qualquer processo que recebe essa informação também marca o processo como suspeito.
- Um processo suspeito continua fazendo parte da visão local, e é tratado de maneira semelhante a um processo correto. Processos suspeitos continuam sendo possíveis alvos para *ping*'s.

- Quando um processo p recebe um *ack*, direta ou indiretamente, de um outro processo q considerado suspeito, p marca q como correto e dissemina a informação de que q está *vivo*. Qualquer processo que recebe esta informação também marca q como correto.
- Processos marcados como suspeitos na visão local de um processo expiram após um determinado *timeout*. Se este tempo de *timeout* expira antes do recebimento de mensagens confirmando que o processo está vivo, este é considerado falho, sendo removido da visão. Como nos outros casos, a confirmação da falha do processo é disseminada para os outros processos.
- Como um processo pode alternar entre o estado de suspeito e correto inúmeras vezes ao longo da execução do algoritmo, é necessário um critério de ordenação nas mensagens do protocolo de suspeita. Este critério é um *número de encarnação (incarnation number)* global para cada processo. O número de incarnação de um processo p só pode ser incrementado pelo próprio processo p. Isto ocorre toda vez que p recebe uma mensagem informando que p é suspeito. Quando o número de incarnação é incrementado, p dissemina o novo valor para os outros processos. Este número é utilizado para marcar as mensagens do protocolo de suspeita. Mensagens com número de incarnação maior tem precedência.

A especificação completa de cada componente do SWIM pode ser encontrada em [17].

2.5.5 Uma Implementação Assíncrona de Detectores

A abordagem para a implementação de detectores proposta por Mostefaoui et al. em [44] é diferente das apresentadas anteriormente. Todas as implementações apresentadas se utilizam de *timeouts* para detectar falhas em processos. O uso dos *timeouts* pressupõe que o sistema, após um tempo, se comporte de maneira síncrona (sincronia parcial). Mostefaoui et al. propõem a

implementação de um detector onde as propriedades temporais do sistema não são levadas em consideração.

A implementação proposta pelos autores se utiliza de um mecanismo de *requisição/resposta (query/response)*. Todo processo p possui a capacidade de enviar por *broadcast* uma mensagem de QUERY() e esperar pelas mensagens correspondentes de RESPONSE() de $(n-f)$ processos, onde n é o número total de processos e f o número máximo de processos que podem falhar. Mensagens de RESPONSE() recebidas após as $(n-f)$ primeiras são descartadas. Um processo só pode emitir uma nova QUERY() quando recebeu as $(n-f)$ respostas da QUERY() anterior. Assume-se que a resposta do próprio processo sempre chega entre as $(n-f)$ primeiras.

Mostefaoui et al. expressam quatro propriedades que, se satisfeitas, possibilitam a implementação de detectores das classes $\mathcal{S}$, $\diamondsuit\mathcal{S}$, $\mathcal{P}$ e $\diamondsuit\mathcal{P}$ respectivamente. As propriedades são:

$\mathcal{PR}(\mathcal{S})$ - Existem um processo p_i e um conjunto Q com ao menos $(f+1)$ processos tais que, após o início da execução, cada processo $p_j \in Q$ recebe uma resposta de p_i para todas suas mensagens de QUERY(), até que p_j possivelmente falhe.

$\mathcal{PR}(\diamondsuit\mathcal{S})$ - Existem um processo p_i e um conjunto Q com ao menos $(f+1)$ processos tais que, após um tempo desconhecido t, cada processo $p_j \in Q$ recebe uma resposta de p_i para todas suas mensagens de QUERY(), até que p_j possivelmente falhe.

$\mathcal{PR}(\mathcal{P})$ - Para todo processo p_i, existe um conjunto Q com ao menos $(f+1)$ processos tal que, após o início da execução, cada processo $p_j \in Q$ recebe uma resposta de p_i para todas as suas mensagens de QUERY(), até que p_j possivelmente falhe.

$\mathcal{PR}(\diamondsuit\mathcal{P})$ - Para todo processo p_i, existe um conjunto Q com ao menos $(f+1)$ processos tais que, após um tempo desconhecido t, cada processo $p_j \in Q$ recebe uma resposta de p_i para todas suas mensagens de QUERY(), até que p_j possivelmente falhe.

O algoritmo proposto pelos autores em [44] que implementa o detector de falhas com base nas propriedades $\mathcal{PR}()$ é exposto na Figura 2.5.

Every process p_i executes the following:

$not\_rec\_from_i \leftarrow \emptyset$
$suspected_i \leftarrow \emptyset$

|| *Task 1:* **Repeat forever**
for all j **do**
  send QUERY() to p_j
end for
wait until (RESPONSE($not\_rec\_from_j$) received from $(n-f)$ distinct processes)
let I = the set of processes from which p_i received a RESPONSE message
let X = the set of the $not\_rec\_from_j$ sets received from $p_j \in I$
let Π = the set of all processes
$not\_rec\_from_i \leftarrow \Pi - I$
$suspected_i \leftarrow \bigcap_{ls \in X} ls$

|| *Task 2:* **On receipt of** QUERY() **from** p_j
send RESPONSE($not\_rec\_from_i$) to p_j

Figura 2.5: Algoritmo do detector de falhas baseado nas propriedades $\mathcal{PR}()$.

Capítulo 3

A Plataforma JXTA

Sistemas *peer-to-peer*, ou P2P, são sistemas distribuídos onde os nós participantes, chamados *peers*, são capazes de se auto-organizar com o objetivo de compartilhar seus recursos [8]. Sistemas P2P são dinâmicos, e devem ser tolerantes a falhas e a mudanças na composição do sistema. A plataforma JXTA, ou simplesmente JXTA, foi desenvolvida com o objetivo de facilitar a implementação de aplicações no modelo P2P. Ela é formada por um conjunto de protocolos que buscam facilitar a localização de recursos e a comunicação entre *peers*. As principais características do JXTA são [1]:

Interoperabilidade - Peers são capazes de se localizar e comunicar independentemente do endereçamento da rede e dos protocolos físicos utilizados.

Independência de Plataforma - O JXTA foi projetado para ser independente de linguagem e plataforma de implementação e de protocolos de transporte.

Ubiquidade - O JXTA tem o objetivo de poder ser utilizado em qualquer tipo de dispositivo digital indepentendemente de plataformas específicas.

O restante deste capítulo expõe alguns dos principais componentes do JXTA.

3.1 A Arquitetura do JXTA

Os *peers* em uma rede JXTA, com o uso dos protocolos da plataforma, se organizam em uma rede *overlay* virtual, permitindo que estes possam se comunicar diretamente, independentemente da estrutura física da rede [4]. O JXTA permite inclusive a comunicação entre *peers* que se encontram por trás de NATs e *firewalls*. A Figura 3.1 mostra a organização de uma rede JXTA exemplo.

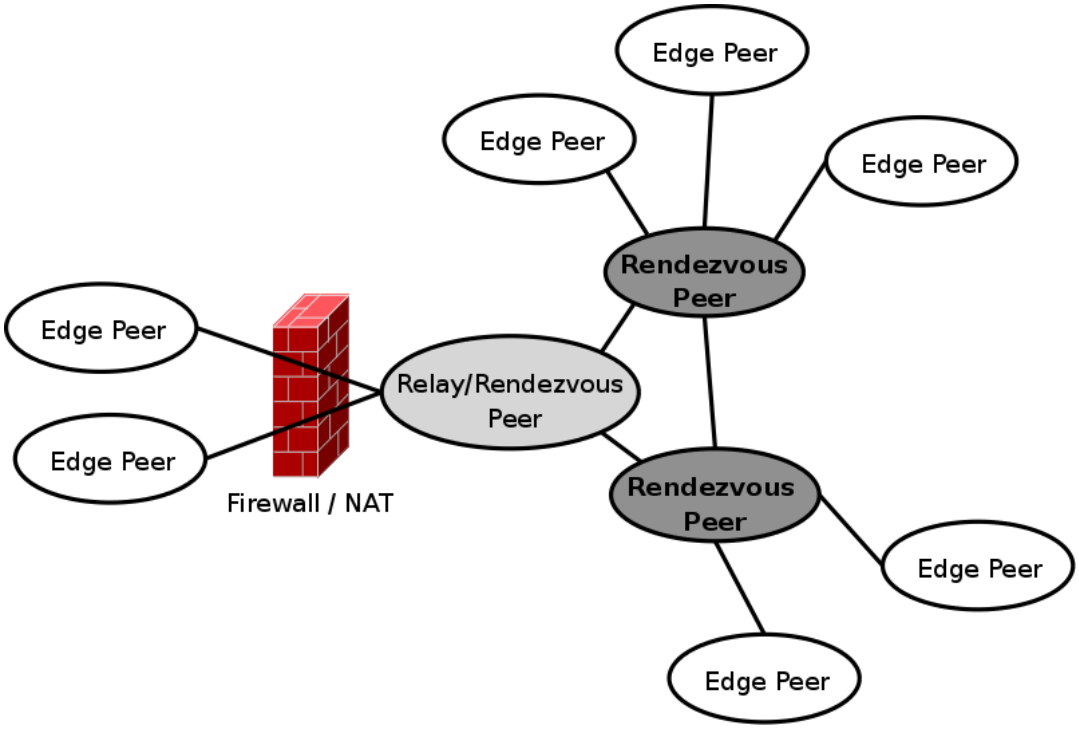

Figura 3.1: Exemplo de configuração de uma rede JXTA.

3.2 Peers, Peer Groups e Módulos

Cada participante de uma rede JXTA, que implementa um mínimo dos protocolos padrões definidos pela plataforma, é chamado de *peer*. *Peers* que possuam algum interesse ou objetivo em comum, seja o compartilhamento de recursos ou a implementação de algum tipo de aplicação ou serviço, podem se organizar em grupos, chamados *Peer Groups*. Um *Peer Group*

é definido por um conjunto de *módulos* (*modules*), os quais os *peers* participantes do grupo precisam implementar.

Um *módulo* é uma implementação de um comportamento qualquer dentro do JXTA. Os protocolos padrão do JXTA e quaisquer serviços disponibilizados por *peers* ou por *Peer Groups* são implementados como módulos. Três outros termos relacionados a um módulo são definidos pela plataforma JXTA: *classe de módulo* (*module class*), *especificação de módulo* (*module specification*) e *implementação de módulo* (*module implementation*). A explicação sobre cada um destes termos é feita na Seção 3.4, pois suas definições são ligadas ao conceito de *anúncios*.

3.3 Super Peers

Algumas funcionalidades da plataforma JXTA são dependentes de alguns *peers* especiais, ditos *super peers*. Um *super peer* é um *peer* que possui uma ou mais das seguintes atribuições [1]:

Relay - Armazena e transmite mensagens de *peers* que não possuem acesso direto a outras partes da rede devido a *firewalls* e *NATs* por exemplo.

Rendezvous - São os *peers* de infraestrutura mais utilizados. Auxiliam no armazenamento e na busca de anúncios. Também são os responsáveis pelo mecanismo de envio de mensagens por *broadcast* disponível no JXTA. Estes *peers* mantém um mapa local da topologia formada pelo conjunto de *peers rendezvous* que este conhecem. Esta informação é utilizada para a disseminação controlada de mensagens e para o mecanismo de busca de anúncios. Os *rendezvous* são os principais responsáveis pela organização da rede JXTA, funcionando como um ponto de encontro entre *peers* que não se encontram na mesma rede física. É comum *peers rendezvous* também atuarem como *relay*.

Proxy - São utilizados por *peers* que não possuem a capacidade de interagir diretamente com a plataforma JXTA, por não implementarem algum protocolo necessário. Este pode ser o caso de peers que não possuem muito poder de processamento ou energia, como por exemplo

sensores, onde a implementação de funcionalidades muito complexas não é viável. Estes *peers* são chamados *peers de borda mínimos* (*minimal egde peers*).

Os *peers* comuns são ditos *peers de borda* (*egde peers*). Em grande parte dos casos, os *peers* deste tipo são a maioria em uma rede JXTA.

3.4 IDs e Advertisements

No JXTA, todas as entidades definidas pela plataforma são representadas por *IDs*. IDs são por exemplo utilizados para identificar de maneira única *peers*, *peer groups*, *pipes* e *módulos*. Um ID JXTA é normalmente representado por uma URN [43] (*Uniform Resource Name*), uma forma de URI [10] (*Uniform Resource Identifier*) que tem o objetivo de ser um identificador de recursos persistente e independente de localização [4]. Estes IDs são manipulados como URIs em formato de *strings*. Três operações são suportadas: comparação, resolução e decomposição. A comparação é feita por igualdade de *strings*. A resolução é o ato de descobrir o recurso ao qual um ID se refere. Por fim, um ID pode ser decomposto e interpretado (por exemplo, descobrir o *Peer Group* ID através de um *Peer* ID).

Outra característica fundamental da plataforma JXTA é o uso de *anúncios* (*advertisements*). Um anúncio é um documento de meta-dados utilizado no JXTA para descrever um recurso [4]. Estes anúncios são representados por documentos XML (*Extensible Markup Language*) [2]. Desse modo, os anúncios herdam todas as vantagens e desvantagens de documentos XML. Serviços JXTA podem criar novos tipos de anúncio a partir de tipos existentes ou criando tipos novos. Os anúncios definidos pela plataforma JXTA são:

Peer Advertisement - Descreve as propriedades de um *peer* e os recursos e serviços que este disponibiliza ao seu grupo.

Peer Group Advertisement - Descreve as propriedades de um *Peer Group* e possíveis parâmetros para seus serviços.

Module Class Advertisement - Descreve uma classe de módulos. Mais especificamente, este anúncio descreve qual o comportamento e a interface de uma dada classe de módulos. As classes de módulo são normalmente utilizados para identificar os módulos dos quais um dado serviço ou aplicação JXTA dependem. Este anúncio tem o objetivo de ser utilizado por humanos para criar novos módulos com comportamento semelhante.

Module Specification Advertisement - Descreve uma especificação de módulo. Mais especificamente, descreve o protocolo que deve ser implementado por um dado módulo. Mais de uma especificação pode existir para uma dada classe de módulos, e todas devem ser compatíveis. Este anúncio tem como objetivo ser utilizado por humanos para a criação de novas implementações compatíveis com uma dada especificação.

Module Implementation Advertisement - Descreve uma implementação de módulo. Este anúncio se refere a uma implementação específica de uma dada especificação de módulo. Desse modo, mais de uma implementação pode existir para uma dada especificação. Este anúncio também deve fornecer uma maneira de se conseguir as informações necessárias à implementação. Como exemplo, uma implementação em Java poderia fornecer o endereço de download de um pacote *.jar* com a implementação do módulo. Outro exemplo seria colocar todo o código necessário dentro do próprio anúncio.

Para a descoberta de novos anúncios para algum tipo de recurso, os *peers* se utilizam do *serviço de discovery* (*discovery service*). Este serviço faz uso do protocolo PDP, que é descrito na Seção 3.5. Para publicar um anúncio, um *peer* tem a opção de publicar localmente, isto é, armazenar o anúncio e responder em caso de uma requisição de anúncio, ou publicar remotamente, na qual o *peer* envia o anúncio para um ou mais outros *peers* do *Peer Group*.

3.5 Os Protocolos JXTA

As funcionalidades providas pela plataforma JXTA são definidas por um conjunto de protocolos. Estes protocolos definem uma série de mensagens XML que são trocadas pelos *peers* para implementar as funcionalidades específicas. Os protocolos definidos pelo JXTA são [1, 4]:

Peer Resolver Protocol

O *Peer Resolver Protocol* ou PRP, fornece um mecanismo genérico de *requisição/resposta* (*query/response*) aos *peers* do JXTA. *Peers* podem definir e realizar qualquer tipo de requisição dentro de um *Peer Group* e, depois, obter as respectivas respostas. Estas requisições podem ser direcionadas a *peers* específicos ou a todos os *peers* do grupo. Uma requisição não garante uma resposta. Um *peer* pode receber zero, uma ou mais respostas para uma dada requisição feita.

Peer Discovery Protocol

O protocolo *Peer Discovery Protocol*, ou PDP, é utilizado pelos *peers* para encontrar anúncios que tenham sido publicados em outros *peers*. Estes anúncios podem representar *peers*, *peer groups*, *módulos*, *pipes* e qualquer outro tipo de recurso que seja descrito por um anúncio. Este protocolo faz uso do PRP para suas requisições, desse modo, uma requisição por anúncios não garante o recebimento de respostas.

Peer Information Protocol

O *Peer Information Protocol*, ou PIP, fornece uma maneira de se conseguir mais informações sobre um dado *peer* conhecido. A implementação deste protocolo é opcional. PIP faz uso do PRP para enviar suas requisições, logo zero ou mais respostas podem ser recebidas por uma requisição do PIP. A informação contida em uma mensagem de resposta contém informações

sobre o estado do *peer*, tais como tempo de funcionamento (*uptime*), número de mensagens enviadas e recebidas, tempo de recebimento da última mensagem, etc. As mensagens de requisição e resposta do PIP contém campos opcionais que as aplicações podem utilizar conforme necessário para expandir sua funcionalidade.

Rendezvous Protocol

O *Rendezvous Protocol*, ou RVP, é dividido em três partes [4]:

- O protocolo *PeerView* é utilizado pelos *peers rendezvous* para se organizar com os outros *rendezvous* conhecidos. Cada *peer rendezvous* mantém uma visão local que forma um mapa da topologia formada pelos *rendezvous* conhecidos. Estes *peers* periodicamente trocam mensagens de *PeerView* para manter as visões atualizadas. Este protocolo é opcional.
- O protocolo *Rendezvous Lease Protocol* é um protocolo que possibilita *peers* que não são *rendezvous* a se inscreverem para participar do mecanismo de propagação de mensagens. *Peers rendezvous* podem fornecem um *lease* para outros *peers*. Durante a duração do *lease*, estes podem enviar e receber mensagens pelo mecanismo de propagação. Este protocolo é opcional.
- O protocolo *Rendezvous Propagation Protocol* é obrigatório. Ele permite que os *peers* façam a propagação controlada de mensagens dentro de um *Peer Group*. Um *peer* pode determinar a origem e destino de uma mensagem, os *peers* pelos quais uma mensagem já passou e qual a distância restante que a mensagem pode percorrer antes de expirar.

Pipe Binding Protocol

O protocolo *Pipe Binding Protocol*, ou PBP, é utilizado pelos *peers* para estabelecer canais de comunicação virtuais, chamados *pipes*, entre *endpoints*. Um *endpoint* é uma abstração cons-

truída sobre as interfaces nativas fornecidas pelos *peers* [51]. Os *endpoints* são responsáveis por criar, enviar, receber e consumir mensagens dentro da plataforma JXTA. Um *pipe* é definido por um *anúncio de pipe* (*Pipe Advertisement*), e o protocolo PBP é responsável por *ligar* (de *bind*) as pontas de um *pipe* a seus respectivos *endpoints*.

Endpoint Routing Protocol

O protocolo *Endpoint Routing Protocol*, ou ERP, é utilizado pelos *peers* para encontrar o caminho até outro *peer* qualquer. Ele é utilizado para que *peers* que não possuem uma conexão direta possam se comunicar. A informação de roteamento inclui uma sequência de *peers* pelos quais uma mensagem deve passar para chegar até o destino.

3.6 A Implementação

Apesar de ser uma especificação aberta, a implementação do JXTA mais madura atualmente é a implementação na linguagem Java, mais especificamente na plataforma *Java Standard Edition*. Esta implementação também é conhecida como JXSE. O arcabouço provido pela implementação JXSE fornece a implementação de toda a funcionalidade descrita neste capítulo. Com o uso desta implementação, aplicações podem facilmente criar um *peer* JXTA e se comunicar com outros *peers*, que podem ser encontrados na rede local ou através de um *peer rendezvous*. O JXSE é a implementação JXTA utilizada neste trabalho.

Capítulo 4

O Serviço de Detecção de Falhas Baseado em Disseminação Epidêmica

Este capítulo descreve a implementação do serviço de detecção de falhas baseado em disseminação epidêmica proposto. O serviço foi implementado para a plataforma JXTA, sendo chamado JXTA-FD. Primeiramente é exposto o algoritmo de detecção utilizado. Em seguida, é detalhado o funcionamento do serviço JXTA-FD. Por fim, é descrita a implementação do serviço proposto para a plataforma JXTA, bem como resultados experimentais.

4.1 O Algoritmo de Detecção

O algoritmo de detecção implementado pelo serviço JXTA-FD é baseado no algoritmo proposto por van Renesse et al. [50], descrito na Seção 2.5.3. O monitoramento entre os processos do sistema utiliza uma estratégia de *heartbeats*, que são propagados através de disseminação epidêmica (*gossip*). O algoritmo completo do serviço pode ser visto na Figura 4.1.

O sistema considerado para a execução do algoritmo é representável por um grafo completo, isto é, cada processo pode se comunicar com qualquer outro processo. Esta consideração foi feita com base em funcionalidades providas pela plataforma JXTA. O sistema é assíncrono; mais especificamente, o sistema tem propriedades probabilísticas tanto para o atraso dos ca-

Every JXTA-FD instance executes the following:

|| *Initialization:*
$table \leftarrow$ **new** HBTable
$heartbeat \leftarrow 0$
$timeOfLastBcast \leftarrow 0$
start tasks *ReceiverTask*, *GossipTask*, *BroadcastTask* and *CleanupTask*

|| *ReceiverTask:* **whenever a gossip message** m **arrives**
for all <*ID*, *hbvalue*> $\in m$ **do**
  $table.update(ID, hbvalue)$
end for
if m is a broadcast **then**
  $timeOfLastBcast \leftarrow$ current time
end if

|| *GossipTask:* **repeat every** $GOSSIP\_INTERVAL$ **units of time**
if $table$ is not empty **then**
  $numberOfTargets \leftarrow min(FANOUT, table.size())$
  $targets \leftarrow$ choose $numberOfTargets$ random elements from $table.get\_ids()$
  for all $t \in targets$ **do**
    send gossip message to t
  end for
  $heartbeat \leftarrow heartbeat + 1$
end if

|| *BroadcastTask:* **repeat every** $BCAST\_TASK\_INTERVAL$ **units of time**
if $shouldBcast()$ **then**
  send gossip message by broadcast
  $timeOfLastBcast \leftarrow$ current time {not necessary if the process receives its own broadcasts}
end if

|| *CleanupTask:* **repeat every** $CLEANUP\_INTERVAL$ **units of time**
for all $id \in table.get\_ids()$ **do**
  $timeFromLastUpdate \leftarrow$ current time - $table.get\_tstamp(ID)$
  if $timeFromLastUpdate \geq REMOVE\_TIME$ **then**
    remove id from $table$
  end if
end for

Figura 4.1: Algoritmo de detecção de falhas do serviço JXTA-FD.

nais de comunicação quanto para a falha de processos. Falhas de processos são falhas por parada. Processos que falham podem retornar ao sistema com um novo identificador. Cada processo executa uma instância do algoritmo de detecção.

O funcionamento do algoritmo é divido em quatro rotinas que executam em paralelo: *ReceiverTask*, *GossipTask*, *BroadcastTask* e *CleanupTask*, além de um procedimento de inicialização. As seções seguintes detalham as estruturas de dados, rotinas do algoritmo e funcionamento do detector.

4.1.1 Estruturas de Dados

A principal estrutura de dados utilizada pelo algoritmo é chamada *HBTable*. Uma *HBTable* tem por finalidade armazenar valores de *heartbeat* de outros processos juntamente com o tempo da última atualização de cada um. Cada *heartbeat* é representado por um valor inteiro positivo. Uma *HBTable* é implementada como uma tabela *hash* que utiliza como chave um identificador de processo, aqui chamado de *ID*, e como valor uma tupla composta por dois valores inteiros, um representando o valor do *heartbeat* e o outro o *timestamp* da última atualização. A notação <*hbvalue*, *tstamp*> é utilizada para representar esta tupla.

Uma *HBTable* fornece cinco operações básicas: *update*(*ID*, *hbvalue*), *get_hbvalue*(*ID*), *get_tstamp*(*ID*), *size*() e *get_ids*(). A operação *update*(*ID*, *hbvalue*), quando invocada, primeiro verifica se o valor *hbvalue* passado é maior do que o armazenado para aquele *ID*. Se sim, o novo valor é armazenado e o *timestamp* correspondente é alterado para o tempo atual. Caso a *HBTable* não possua um valor para o *ID* passado, uma tupla com o *hbvalue* passado e o tempo atual é inserida na estrutura. As operações *get_hbvalue*(*ID*) e *get_tstamp*(*ID*) retornam os valores de *hbvalue* e *tstamp*, respectivamente, para um dado *ID*. Por fim, *size*() retorna o número de entradas na tabela e *get_ids*() retorna o conjunto dos *ID*s contidos na mesma.

Cada instância do algoritmo faz uso de uma *HBTable* e de dois inteiros, *localHB* representando o valor atual de *heartbeat* do processo local e *timeOfLastBcast* representando o

tempo no qual foi recebida a ultima mensagem de difusão. $timeOfLastBcast$ é utilizado na decisão de quando um processo deve fazer uma nova difusão ou não.

4.1.2 Inicialização

Ao início da execução do algoritmo de detecção, um procedimento de inicialização é executado. Este procedimento tem como objetivo inicializar as estruturas de dados necessárias e disparar as outras rotinas do algoritmo de detecção. Primeiramente, uma *HBTable* é criada, o valor de heartbeat local e o valor $timeOfLastBcast$ são inicializados para 0. Em seguida, as demais rotinas do algoritmo são iniciadas e passam a executar paralelamente.

4.1.3 ReceiverTask

A rotina *ReceiverTask* é executada toda vez que uma mensagem de *gossip* é recebida, inclusive no caso das mensagens ocasionais enviadas por difusão. Cada mensagem de *gossip* é composta de um conjunto de tuplas <*ID*, *hbvalue*>, sendo que cada uma representa o *heartbeat* de um dado processo. Quando uma mensagem de *gossip* chega, a operação *update*(*ID*, *hbvalue*) da *HBTable* é chamada para cada uma das tuplas, atualizando as informações de *heartbeat* contidas na tabela. Quando a mensagem recebida é de difusão, o processo também incrementa o valor $timeOfLastBcast$.

4.1.4 GossipTask

A rotina *GossipTask* é executada periodicamente, a cada $GOSSIP\_INTERVAL$ intervalos de tempo. Ela é responsável pelo envio das mensagens de *gossip* aos outros processos. A cada execução, a rotina primeiramente verifica se *HBTable* está vazia. Se sim, não há nada a ser feito, pois nenhum outro processo é conhecido. No caso de existir alguma entrada na tabela, $FANOUT$ processos são escolhidos aleatoriamente do conjunto de processos conhecidos (ou menos, no caso de o número de entradas na *HBTable* ser insuficiente). Uma mensagem de

gossip é enviada para cada um dos processos selecionados. A mensagem de *gossip* enviada é construída com as informações contidas na $HBTable$. Para cada entrada na tabela, uma tupla <*ID*, *hbvalue*> é adicionada à mensagem. Também é adicionada uma tupla com o *ID* e *heartbeat* do processo local. Após o envio da mensagem, o processo incrementa seu valor local de $heartbeat$.

4.1.5 BroadcastTask

Para que os processos possam se encontrar inicialmente, e para que a saída do detector se estabilize mais rapidamente em caso de um número alto de falhas simultâneas, a rotina *BroadcastTask* é executada periodicamente. A cada execução, existe a chance de uma mensagem de *gossip* do algoritmo ser enviada para todos os outros processos, através de um mecanismo de difusão. A probabilidade da difusão ser efetuada é calculada com base em parâmetros do algoritmo e no tempo de chegada da última mensagem de difusão recebida. Esta probabilidade visa evitar difusões simultâneas e muito frequentes.

Para a implementação do algoritmo, o JXTA-FD utiliza a mesma função de probabilidade proposta por van Renesse et. al. em [50], $p(t) = (t/BCAST\_MAX\_PERIOD)^{BCAST\_FACTOR}$, onde t é igual ao número de unidades de tempo decorridas da última difusão recebida e $BCAST\_MAX\_PERIOD$ e $BCAST\_FACTOR$ são parâmetros do algoritmo. A cada execução da rotina *BroadcastTask* uma mensagem de *gossip* é enviada por difusão com probabilidade $p(t)$. Desse modo, o tempo médio entre o envio de difusões depende diretamente da frequência de execução da rotina (controlada pelo parâmetro $BCAST\_TASK\_INTERVAL$), do número de processos no sistema e dos parâmetros do algoritmo. $BCAST\_MAX\_PERIOD$ é o intervalo máximo entre cada difusão, e quando t se aproxima deste valor, a probabilidade $p(t)$ tende a 1. $BCAST\_FACTOR$ deve ser um valor real positivo, e determina o quão antes de $BCAST\_MAX\_PERIOD$ as difusões tendem a ser enviadas. Quanto maior o valor de

BCAST_FACTOR, mais próxima do valor *BCAST_MAX_PERIOD* fica a duração do intervalo esperado entre as difusões.

Como exemplo, para os valores *BCAST_TASK_INTERVAL* de 1 unidade de tempo, *BCAST_MAX_PERIOD* de 20 unidades de tempo e um conjunto de 1000 processos, para se obter um intervalo de aproximadamente 10 unidades de tempo entre uma difusão e a próxima, *BCAST_FACTOR* deve ser aproximadamente 10.43 [50].

4.1.6 Saída do Detector

A qualquer momento durante a execução do algoritmo, um processo pode consultar seu detector local pelo conjunto de processos suspeitos ou corretos. Para determinar estes processos, a tabela *HBTable* é percorrida e, para cada entrada, o tempo decorrido desde a sua última atualização é calculado. Se este tempo for maior ou igual a *SUSPECT_TIME*, o processo é considerado suspeito. Caso contrário, ele é considerado correto.

4.1.7 CleanupTask

A rotina *CleanupTask* é responsável por remover entradas antigas da *HBTable* do processo local. A cada *CLEANUP_INTERVAL* unidades de tempo, a tabela é percorrida e entradas que não foram atualizadas a mais de *REMOVE_TIME* unidades de tempo são excluídas. Este mecanismo é importante pois processos considerados suspeitos também são utilizados pelo mecanismo de *gossip* como possíveis alvos, e uma tabela contendo um número muito alto de entradas inválidas (processos falhos) pode causar um impacto negativo na exatidão do detector. O valor *REMOVE_TIME* é um parâmetro de configuração do algoritmo.

4.2 O Serviço de Detecção JXTA-FD

O algoritmo de detecção descrito na Seção 4.1 foi implementado como um serviço para a plataforma JXTA, chamado JXTA-FD. O monitoramento entre *peers* é feito dentro do contexto de um *Peer Group*. Um *peer* que deseja monitorar e ser monitorado por outros *peers* deve inicialmente entrar no grupo desejado e então carregar e inicializar o serviço de detecção dentro do grupo.

Quando o serviço JXTA-FD é carregado e inicializado dentro de um grupo, o algoritmo de detecção é iniciado e todos os *peers* que estiverem também executando o serviço dentro do grupo passam a monitorar uns aos outros. Um único *peer* pode fazer parte de múltiplos grupos monitorados, mas uma instância do serviço deve ser executada para cada grupo e estas são independentes umas das outras.

A interface provida pelo serviço de detecção JXTA-FD é composta de dois métodos: *getSuspectPeers()* e *getCorrectPeers()*. Quando invocadas pelo *peer*, estas funções retornam, respectivamente, uma lista composta dos identificadores de *peers* considerados suspeitos dentro do grupo e uma lista com os identificadores de *peers* considerados corretos.

O JXTA-FD não fornece um mecanismo de saída explícita do grupo de monitoramento. Se um *peer* deseja interromper o detector, ou seja, deixar de monitorar e ser monitorado pelos outros *peers*, basta interromper o serviço JXTA-FD e deixar o *Peer Group*. Após um tempo, os detectores dos demais *peers* do grupo irão suspeitar deste *peer* e em seguida remover por completo as informações do mesmo.

O comportamento do serviço JXTA-FD pode ser controlado através de alguns parâmetros de configuração. Os parâmetros mais importantes são três: $GOSSIP\_INTERVAL$, $FANOUT$ e $SUSPECT\_TIME$. O primeiro controla o intervalo entre o envio de mensagens de *gossip* pela rotina *GossipTask*. O segundo controla quantas mensagens são enviadas em cada execução da mesma. Por último, $SUSPECT\_TIME$ determina quantas unidades de tempo sem atualização de *heartbeat* são necessárias para se considerar um *peer* como suspeito.

Alguns outros parâmetros são disponibilizados pelo serviço. *CLEANUP_INTERVAL* e *REMOVE_TIME* dizem respeito à rotina *CleanupTask*, qual o período de tempo entre cada execução e quantas unidades de tempo sem atualização são necessárias para remoção das informações de um *peer* do serviço local. *BCAST_TASK_INTERVAL*, *BCAST_MAX_PERIOD* e *BCAST_FACTOR* controlam a rotina *BroadcastTask*, basicamente definindo com que frequência os *peers* enviam mensagens por difusão. A Seção 4.1.5 descreve a função de cada um destes valores.

Os parâmetros do serviço para um dado grupo devem ser decididos a priori, antes da inicialização do mesmo. Uma maneira de se fazer isso utilizando o JXTA é colocando os valores na descrição do *Advertisement* do grupo. Durante a execução do algoritmo, os parâmetros da instância local do serviço podem ser alterados pelos *peers*, porém nenhum mecanismo de coordenação ou acordo destes valores é disponibilizado.

4.3 A Implementação

O serviço JXTA-FD foi implementado para a plataforma JXTA, versão 2.5, utilizando a linguagem Java. Este serviço foi implementado como um módulo (*Module*) JXTA, chamado JxtaFD. O monitoramento entre *peers* é feito dentro do contexto de um *Peer Group*, e um módulo do JXTA-FD deve ser carregado e inicializado para cada grupo monitorado. Somente *peers* que estejam executando o módulo participam do algoritmo.

A estrutura *HBTable* foi implementada com o uso de uma *ConcurrentHashMap* (estrutura provida pelo Java), que é uma tabela *hash* feita para acessos concorrentes. As chaves da tabela são *PeerIDs*. Cada *peer* é identificado de maneira única por um dado *PeerID*.

A comunicação entre os *peers* do grupo é feita com o uso de *Pipes*. Mensagens de *gossip* são enviadas através de *Pipes* do tipo *Unicast*. O *Pipe Unicast* para um dado *peer* é gerado através do *PeerID* deste *peer*. Deste modo, para que um *peer* possa enviar uma mensagem de *gossip* para outro, basta que este conheça o *PeerID* do destinatário. Mensagens de *broadcast*

são enviadas através de um *Pipe* do tipo *Propagate*. Este *Pipe* é único para cada *PeerGroup*, e é gerado com base no *PeerGroupID* deste grupo. O uso de *Pipes* JXTA possibilita a comunicação direta entre dois *peers* quaisquer, visto que mensagens enviadas através de *Pipes* são propagadas por *relays* e *rendezvous* quando necessário, ou seja, se caso os *peers* não consigam se comunicar diretamente.

As rotinas do algoritmo de detecção foram implementadas para execução em paralelo. A rotina *ReceiverTask* foi implementada como duas *Threads*, uma com um *loop* esperando por mensagens vindas do *Pipe Unicast* e outra mensagens vindas do *Pipe Propagate*. Cada *Thread* é responsável por fazer o devido processamento de mensagens de *gossip* ou *broadcast*. Para a implementação das rotinas *GossipTask*, *BroadcastTask* e *CleanupTask* foram utilizados *TimerTasks*. Um *TimerTask* funciona de maneira semelhante a uma *Thread*, mas o inicio de sua execução pode ser agendado com o uso de *Timers*. Toda vez que uma *TimerTask* inicia a execução, o início da próxima execução é agendado no *Timer*.

Para consultar o serviço de detecção, o *peer* pode fazer uso de dois métodos providos pelo módulo JxtaFD, *suspectPeers*() e *correctPeers*(). Estes métodos retornam, respectivamente, um *Set* dos *PeerIDs* de processos suspeitos pelo detector local e um *Set* de processos considerados corretos.

O comportamento do detector é controlado através de atributos do módulo JxtaFD, porém, estes valores só devem ser alterados antes do início da execução do detector.

4.4 Avaliação e Resultados Experimentais

Para a avaliação do serviço de detecção de falhas proposto, experimentos foram realizados para a implementação na plataforma JXTA e para um simulador, implementado com o uso da biblioteca de eventos discretos SMPL [42]. Os experimentos têm como objetivo avaliar a implementação JXTA e o impacto dos parâmetros do serviço no funcionamento do detector.

Nos experimentos realizados, foram avaliadas duas estratégias diferentes para a configuração do detector. Na primeira, a cada rodada do algoritmo é enviada apenas uma mensagem de *gossip*. Para aumentar a exatidão do detector, o intervalo entre o envio das mensagens de *gossip* (parâmetro $GOSSIP\_INTERVAL$) é reduzido. Na segunda estratégia, o intervalo entre o envio de mensagens de *gossip* é mantido fixo enquanto o *fanout* do algoritmo (parâmetro $FANOUT$), ou seja, o número de mensagens de *gossip* enviadas a cada rodada, é incrementado. As comparações são feitas de modo que a banda utilizada, isto é, a quantidade de tuplas do tipo <*ID*, *hbvalue*> enviada pelos peers em um dado intervalo de tempo, seja a mesma para os dois casos. Estas estratégias são representadas nos gráficos por *Gossip* e *Fanout*, respectivamente. Até o restante deste capítulo, a quantidade de tuplas do tipo <*ID*, *hbvalue*> enviadas pelos *peers* é chamada simplesmente de "banda utilizada".

Para a simulação do atraso e perda de mensagens, foi implementado como um parâmetro da execução dos experimentos um valor que controla a porcentagem de mensagens perdidas pelos *peers*. Cada mensagem recebida, tanto na simulação quanto na implementação JXTA, tem uma chance de ser descartada. Esse mecanismo foi adotado para simplificar a implementação e a avaliação dos resultados, visto que mensagens suficientemente atrasadas causam o mesmo impacto de mensagens perdidas no funcionamento do detector.

4.4.1 Resultados da Implementação JXTA

Os experimentos realizados na plataforma JXTA tiveram como principal objetivo a avaliação da implementação do serviço de detecção. Inicialmente, estava prevista a realização de experimentos com *peers* executando em múltiplos *hosts*, em localizações distintas. Dificuldades na implementação e no uso de *peers relay*, juntamente com a indisponibilidade dos servidores *relays* públicos, resultaram na execução dos testes em um ambiente mais simples. Os experimentos foram realizados para um grupo de *peers* executando em um único *host* e o mecanismo de descarte de mensagens é utilizado para representar o atraso e a perda de mensagens. Os

gráficos destes experimentos apresentam os resultados obtidos e um intervalo de confiança de 95%.

Uso de CPU e Memória

O objetivo destes experimentos é avaliar o uso de CPU e memória pelo serviço de detecção e plataforma JXTA. Os experimentos foram realizados em uma máquina Intel Core2 Quad Q9400, 2.66GHz, com 4GB de memória RAM. O experimento foi realizado com 10 *peers* executando o serviço de detecção por 15 minutos. A cada 1 segundo, são obtidos os dados relacionados ao uso de CPU e memória. Todos os *peers* executam o detector com os mesmos parâmetros. O tempo para suspeitar de um *peer*, $SUSPECT\_TIME$ é de 5 segundos. O tempo para remoção de um *peer* suspeito, $REMOVE\_TIME$, é de 20 segundos. Cada *peer* realiza 1 consulta por segundo ao seu detector. Os parâmetros $BCAST\_MAX\_PERIOD$ e $BCAST\_FACTOR$ são 20 e 4.764 respectivamente.

A Figura 4.2 mostra o impacto do dos parâmetros do detector no uso de CPU. O gráfico demonstra que o uso de CPU é diretamente proporcional ao uso de banda pelo serviço de detecção. Nos testes realizados, pouca diferença é observada entre as duas estratégias do detector. É possível que, dado um grupo maior de *peers*, a diferença entre as duas estratégias se torne mais expressiva. Do mesmo modo, pode-se observar na Figura 4.3 que o uso de memória também acompanha o uso de banda do detector.

Na Figura 4.4 é mostrado o uso de CPU ao longo de uma das execuções do experimento. Um pico bastante alto pode ser observado ao início da execução. Isto se deve principalmente ao processo de inicialização da plataforma JXTA. Após essa inicialização, o uso de CPU se mantém relativamente estável.

Estes resultados demonstram que a plataforma JXTA consome uma quantidade de recursos considerável. Vale ressaltar que a quantidade de *peers* utilizada é pequena e estes participam de apenas um grupo de detecção. Mesmo considerando que as porcentagens na Figura 4.2 representam o uso de apenas um dos núcleos do processador, o valor de 4% é bas-

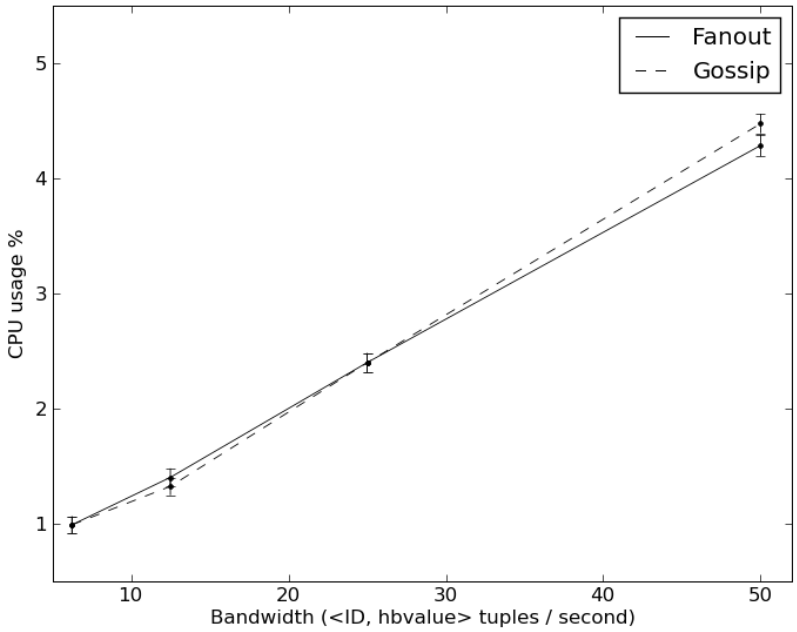

Figura 4.2: Impacto dos parâmetros do detector no uso de CPU.

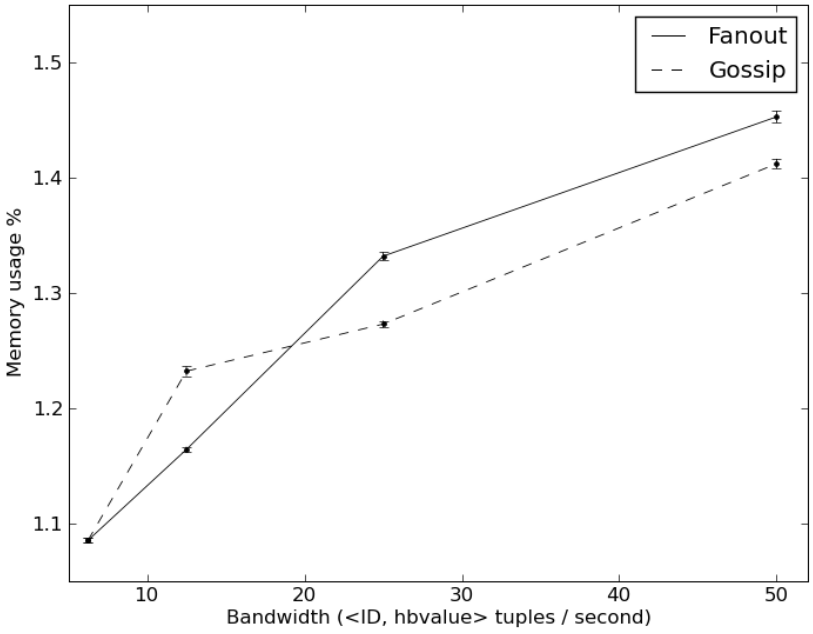

Figura 4.3: Impacto dos parâmetros do detector no uso de memória.

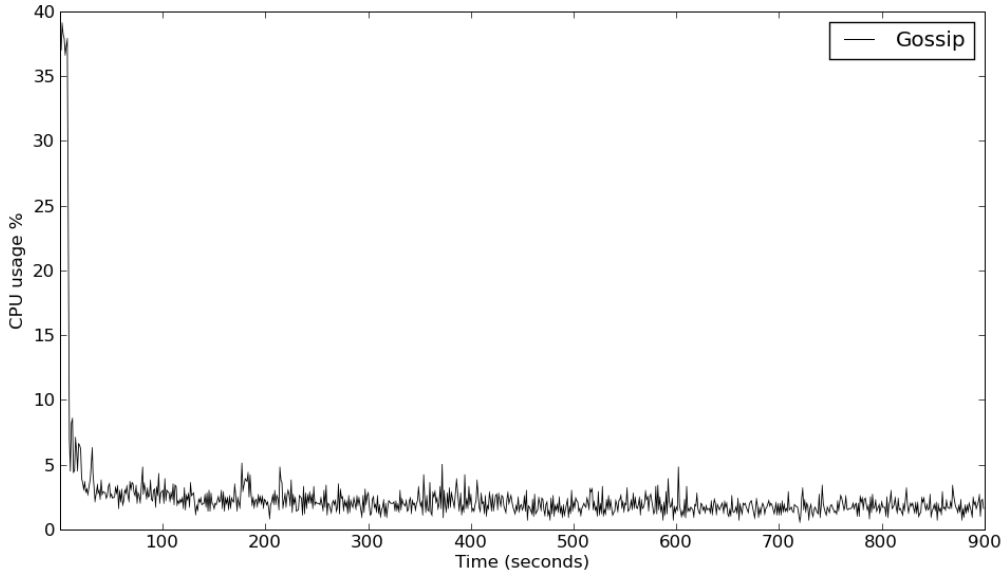

Figura 4.4: Uso de CPU ao longo da execução do experimento. É utilizada a estratégia *Gossip* e a banda é constante, com um *GOSSIP_INTERVAL* de 400 milissegundos.

tante alto para um serviço básico, que é normalmente utilizado para implementação de outros algoritmos.

Probabilidade de Enganos

Estes experimentos visam validar a implementação do serviço de detecção e verificar qual o impacto dos parâmetros do detector e do número de falhas no número de enganos do detector. Um engano ocorre quando um *peer* correto é suspeito pelo detector. Nestes experimentos, os *peers* nunca falham. Desse modo, qualquer suspeita do detector é um engano. O cenário dos experimentos é o mesmo dos experimentos de uso de CPU, os testes são executados para 10 *peers*, e os valores dos parâmetros fixos são os mesmos.

A Figura 4.5 mostra a probabilidade de uma consulta ao detector ser um engano, para uma quantidade de perda de mensagens igual a 30%. É possível observar a melhora na exatidão do detector com o aumento do *fanout* ou na frequência do envio de mensagens de *gossips*. A probabilidade de enganos para o valor de banda 12.5 é aproximadamente 0.02700 para a

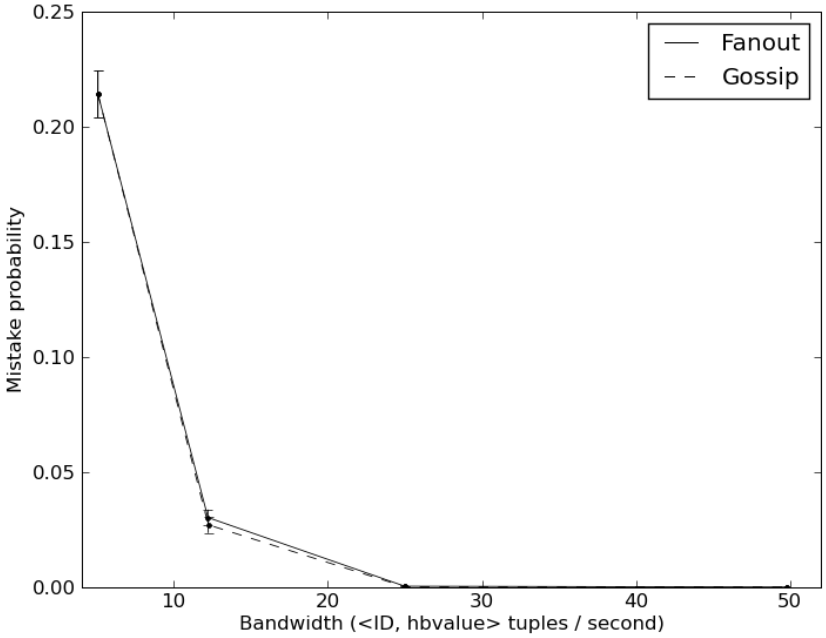

Figura 4.5: Variação no número de enganos de acordo com a banda utilizada. Experimento realizado com 30% das mensagens sendo descartadas.

estratégia *Gossip* e 0.03019 para a estratégia *Fanout*. Para o valor de banda 25, os valores são 0.00015 e 0.00035, respectivamente. Para o valor de banda 50, não foram detectados enganos. Pode-se observar que, para um grupo tão pequeno de *peers*, a diferença entre as duas estratégias não é muito expressiva. Apesar disso, a estratégia de *Gossip* obteve um resultado melhor, com menos de 50% na quantidade de enganos para o valor de banda 25.

Na Figura 4.6 é mostrado o impacto da probabilidade de perda de mensagens no número de enganos cometidos pelo detector. Os resultados demonstram que a estratégia *Gossip* é mais resistente à perda de mensagens. Para os valores de 20% e 30% de mensagens perdidas, a quantidade de enganos é aproximadamente 10% menor em comparação com a estratégia *Fanout*.

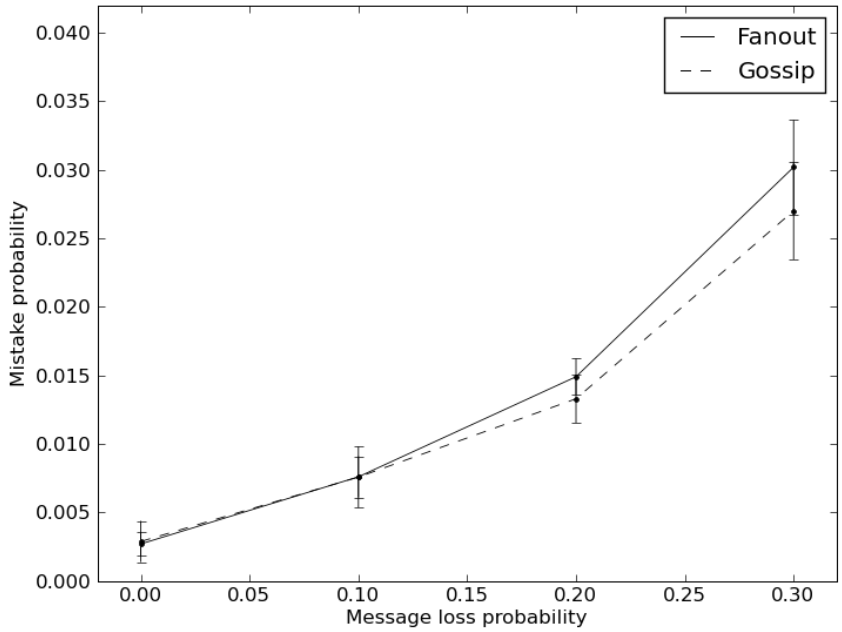

Figura 4.6: Variação no número de enganos de acordo com a probabilidade de perda de mensagens. O valor de banda utilizada é fixado em 25.

Tempo de Detecção e Recuperação

Estes experimentos têm o objetivo de verificar a diferença nos tempos de detecção e recuperação para as duas estratégias. Tempo de detecção é o tempo médio entre a falha de um *peer* e sua suspeita por outro *peer*. Tempo de recuperação é o tempo médio entre a recuperação de um *peer* e o tempo que otro *peer* deixa de suspeitar do mesmo. O tempo de recuperação também pode ser visto como o tempo médio que um novo *peer* leva para ser descoberto por outro *peer*.

Os testes realizados são semelhantes aos das seções anteriores, 10 *peers*, com os mesmos valores para os parâmetros. Porém, em um dado instante, um *peer* é parado. Este *peer* volta a executar 10 segundos depois de parar. Os testes são executados sem perda de mensagens.

A Figura 4.7 mostra o tempo de detecção para diferentes valores de banda utilizada. Pode-se observar pelos resultados uma grande variação nos tempos de detecção. A variação para os

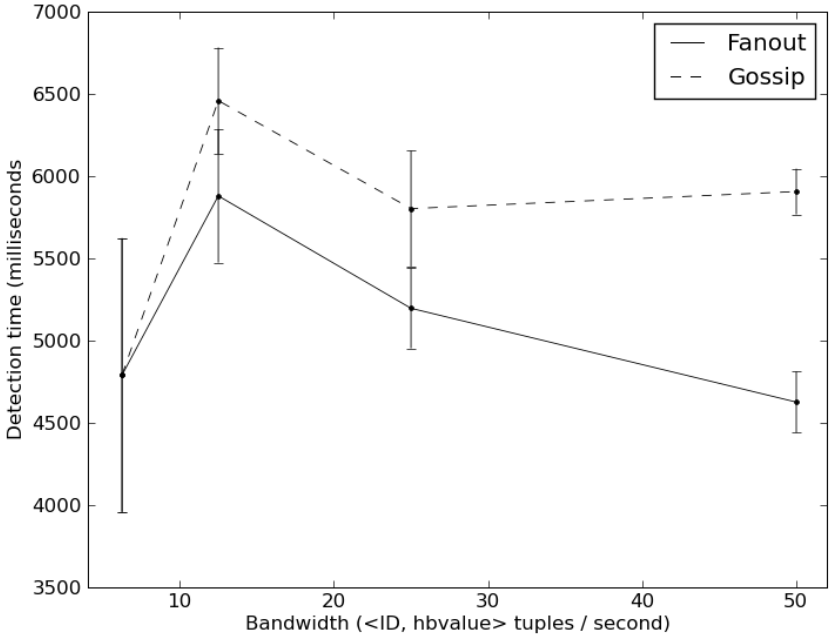

Figura 4.7: Tempo de detecção de uma falha, para diferentes valores de banda utilizada.

valores baixos de uso de banda se devem provavelmente ao maior número de enganos, pois um *peer* tem mais chance de estar suspeito, mesmo que ainda não tenha falhado. O gráfico também mostra que o tempo de detecção para a estratégia de *Fanout* é consideravelmente menor, sendo aproximadamente 20% menor para o valor de banda 50. Isto se deve ao fato de um *peer* ser considerado suspeito quando informações sobre o mesmo não são recebidas a mais de $SUSPECT\_TIME$ unidades de tempo. Como a frequência de atualizações da estratégia *Fanout* é menor, o tempo da última atualização sobre um dado *peer* é, em média, menor que o da estratégia *Gossip*, provocando um tempo menor de detecção.

Na Figura 4.8, é mostrado o tempo de recuperação para diferentes valores de banda. Neste caso, a estratégia de *Gossip* é superior, possuindo um tempo de recuperação aproximadamente 55% inferior à estratégia de *Fanout* para o valor de banda 50. Esta diferença se deve, novamente, à diferença na frequência de atualizações. Também pode-se observar que o impacto no aumento do uso de banda afeta diretamente o tempo de recuperação.

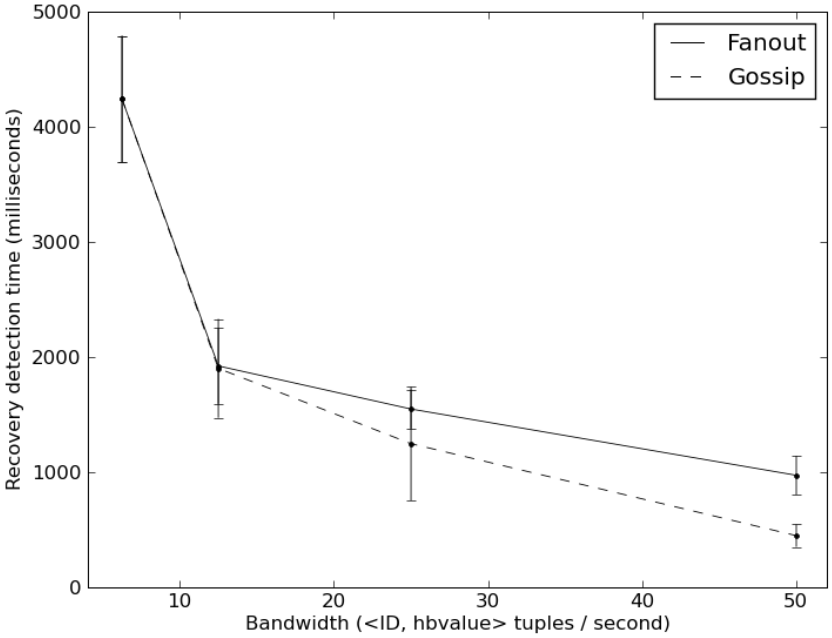

Figura 4.8: Tempo de detecção da recuperação, para diferentes valores de banda utilizada.

4.4.2 Resultados da Simulação

Os experimentos de simulação têm como objetivo avaliar o comportamento do algoritmo de detecção para um grupo maior de *peers*, sem a interferência da plataforma JXTA nos resultados.

A Figura 4.9 mostra o resultado da simulação de um experimento semelhante ao executado com a implementação JXTA, mostrado na figura Figura 4.5. O experimento tem como objetivo a comparação com os resultados obtidos com a implementação JXTA para verificação do funcionamento do simulador. Os testes foram executados com 10 *peers*, e com 30% das mensagens sendo descartadas. Como esperado, o resultado obtido é bastante semelhante ao da implementação JXTA.

Os seguintes experimentos de simulação foram realizados com um grupo de 200 peers. Alguns parâmetros são mantidos em todos os experimentos, para facilitar a interpretação dos resultados. O $SUSPECT\_TIME$ é igual a 5 unidades de tempo e o $REMOVE\_TIME$ é igual

a 20 unidades de tempo. O detector é consultado a cada 0.25 unidades de tempo. A rotina *BroadcastTask* é executada a cada 1 unidade de tempo, sendo o $BCAST\_MAX\_PERIOD$ igual a 20 unidades de tempo e o $BCAST\_FACTOR$ igual a 8.2, fazendo com que um *broadcast* seja feito a cada 10 unidades de tempo, aproximadamente. A estratégia *Gossip* mantém o $FANOUT$ em 1 e varia o $GOSSIP\_INTERVAL$, enquanto a estratégia $FANOUT$ mantém o $GOSSIP\_INTERVAL$ em 2 unidades de tempo e varia o valor $FANOUT$.

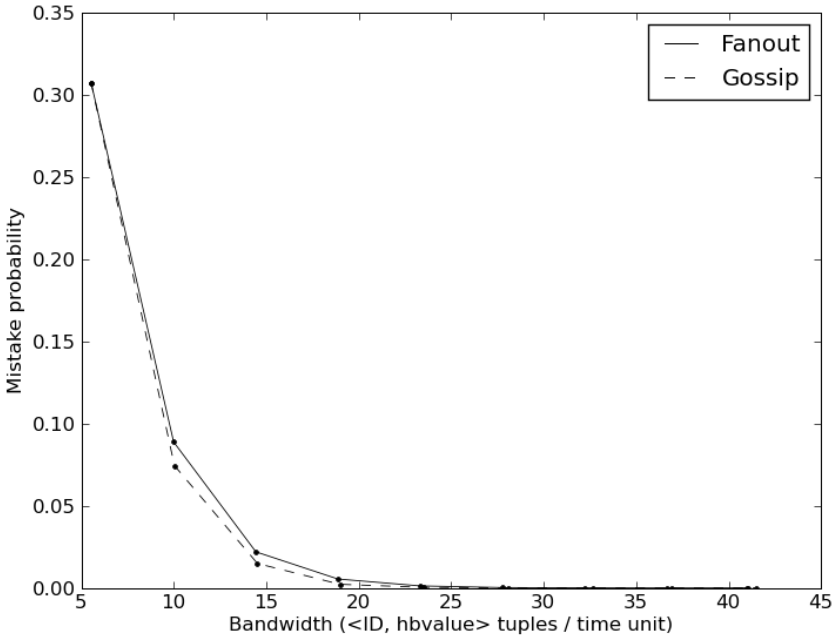

Figura 4.9: Probabilidade de engano do algoritmo. Experimento de simulação realizado com 30% das mensagens sendo descartadas.

Probabilidade de Enganos

Este experimento teve como objetivo avaliar o impacto do aumento da banda utilizada na exatidão do detector. Nenhum *peer* falha durante o experimento, logo, qualquer suspeita é considerada um engano.

A Figura 4.10 mostra a variação na quantidade de enganos do detector, de acordo com a banda utilizada pelo algoritmo. A perda de mensagens é de 50%. Pode-se observar que o

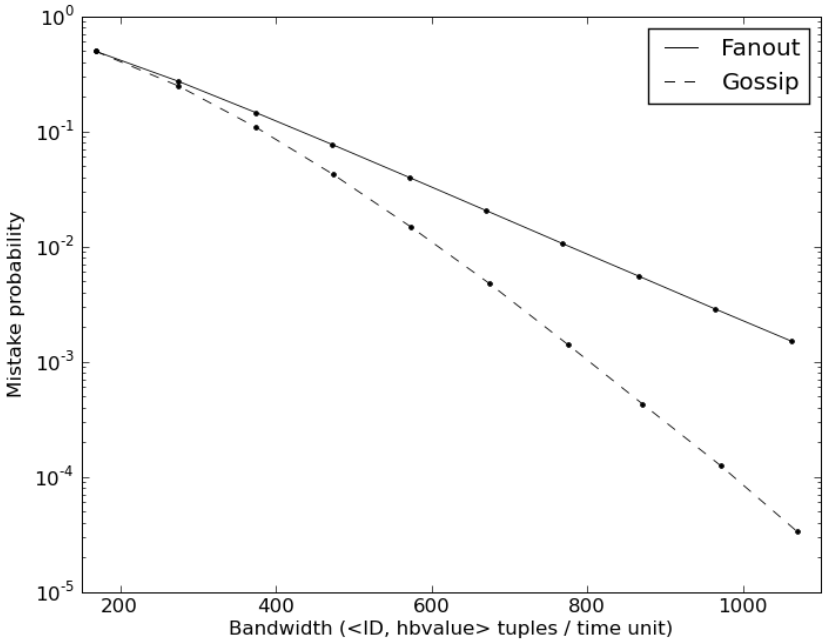

Figura 4.10: Probabilidade de engano do algoritmo. Experimento realizado com 50% das mensagens sendo descartadas.

aumento da frequência das mensagens *gossips* tem um impacto muito maior na diminuição dos enganos do detector, tendo, após certo ponto, uma vantagem de mais de uma ordem de magnitude em relação ao aumento no $FANOUT$.

Na Figura 4.11 pode-se observar o impacto da perda de mensagens no número de enganos do detector. A banda utilizada é fixada em aproximadamente 550 ($FANOUT$ igual a 5 e $GOSSIP\_INTERVAL$ de 0.4 unidades de tempo). O gráfico mostra que a estratégia *Gossip* tem uma probabilidade muito menor de cometer enganos, para todos os valores de probabilidade de falha utilizados.

Estes resultados, juntamente com os resultados apresentados para a implementação JXTA, mostram que a estratégia *Gossip* é bastante superior à *Fanout* em relação à exatidão do detector, ou seja, a probabilidade de enganos. Além disso, a diferença entre as duas estratégias se torna ainda maior com o aumento no número de *peers* do grupo.

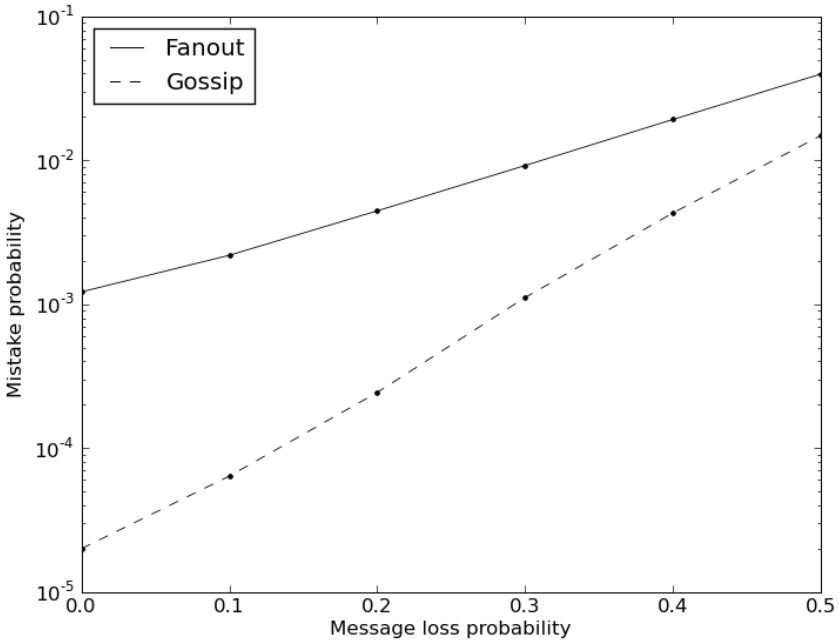

Figura 4.11: Probabilidade de engano do algoritmo para diferentes probabilidades de perda de mensagens.

Eleição

Este experimento tem o objetivo de verificar a viabilidade do uso do detector proposto para a solução do problema da eleição. São utilizados os mesmos valores dos experimentos anteriores para os parâmetros do detector. Cada *peer* considera como líder o *peer* de menor identificador considerado correto pelo seu detector. Desse modo, a cada 1 unidade de tempo aproximadamente, o detector de todos os *peers* são consultados simultaneamente. Para que a eleição tenta sucesso, todas estas consultas devem indicar o mesmo líder. Nenhum *peer* falha durante estes experimentos.

A Figura 4.12 apresenta a probabilidade de uma eleição ser correta para diversos valores de banda utilizada. Mensagens são perdidas com 50% de chance. Este gráfico mostra que a estratégia de *Gossip* é mais eficaz para a execução do algoritmo de eleição. Alem disso,

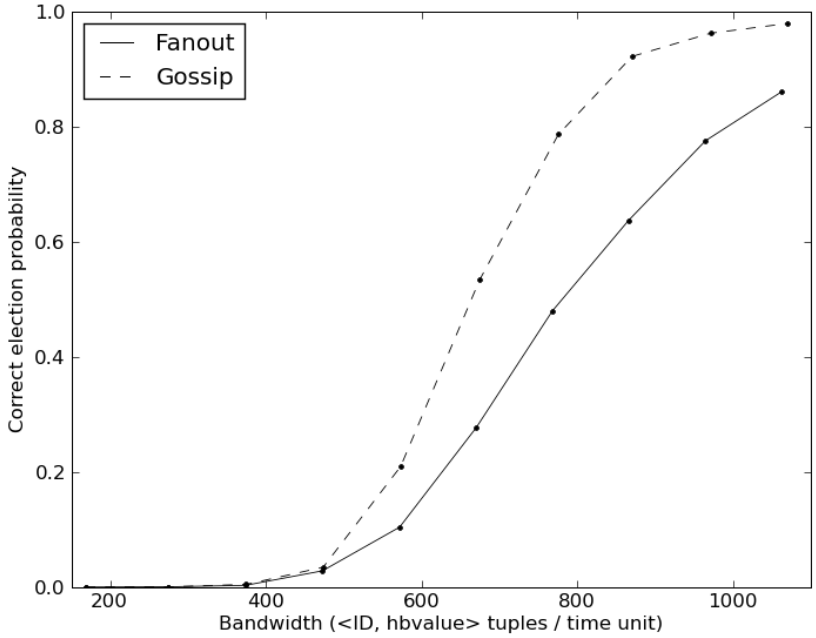

Figura 4.12: Probabilidade de eleição correta. Experimento com 50% de mensagens perdidas.

pode-se verificar que resultados razoáveis somente são obtidos com valores maiores de banda, ou seja, baixa probabilidade de enganos.

Na Figura 4.13 podemos verificar o impacto da perda de mensagens na eleição. A banda utilizada é fixada em aproximadamente 650 (*FANOUT* igual a 6 e *GOSSIP_INTERVAL* de 0.332 unidades de tempo). O gráfico mostra que a perda de mensagens causa grande impacto no resultado da eleição. Pode-se observar também que a eleição utilizando a estratégia *Gossip* se mostra consideravelmente mais resistente a falhas.

Estes resultados mostram que a eleição é probabilística, tendo maior chance de sucesso quanto maior a banda utilizada pelo algoritmo de detecção. Fica claro também que a estratégia *Gossip* obtém melhores resultados que a *Fanout* para um mesmo valor de banda utilizada.

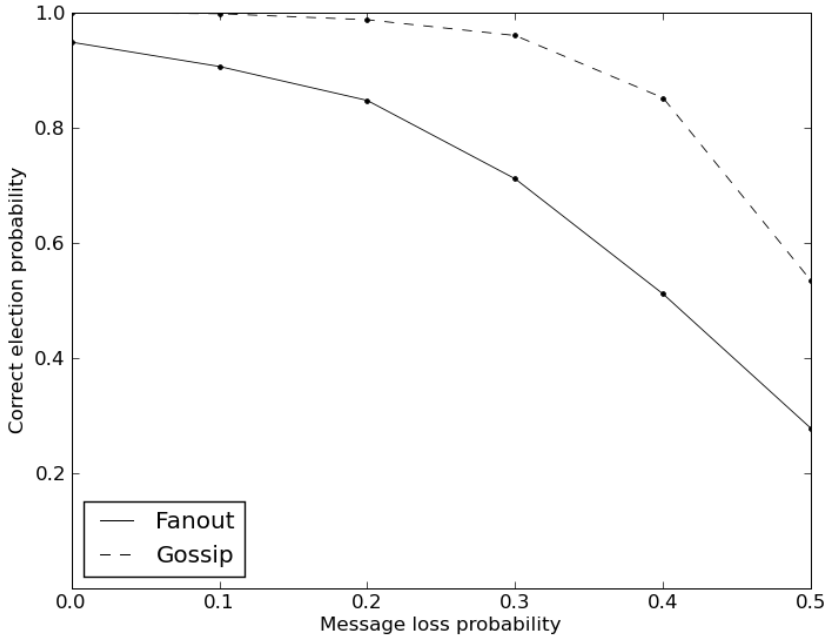

Figura 4.13: Probabilidade de eleição correta para diferentes valores de perda de mensagens. A banda utilizada é de aproximadamente 650.

Capítulo 5

Conclusão

Este trabalho apresentou a especificação, implementação e avaliação de um serviço de detecção de falhas baseado em disseminação epidêmica. O serviço foi implementado na plataforma JXTA e, para permitir a avaliação do detector com um número maior de *peers* e sem a interferência do JXTA, foi implementado um simulador com a biblioteca SMPL. *Peers* que implementam o serviço e participam de um grupo de monitoração podem consultar o detector para obter informações sobre o estado dos outros *peers* do grupo.

O serviço de detecção foi avaliado através de experimentos para a plataforma JXTA e simulador. Os resultados obtidos demonstram que o algoritmo de disseminação epidêmica é escalável para o número de *peers* participantes e, utilizando o valor de banda necessário, robusto. Os resultados também mostram que a estratégia de disseminação epidêmica resulta em um número consideravelmente menor de enganos do detector em comparação com uma estratégia equivalente mas baseada em aumento do *fanout*. A diferença entre as estratégias se torna ainda maior com o aumento do número de *peers*. O único resultado desfavorável para a estratégia de disseminação foi relacionado ao tempo de detecção de falhas, em média maior do que o tempo da estratégia de *fanout*.

A escolha da plataforma JXTA para a implementação do serviço foi motivada pela disponibilidade de mecanismos interessantes, como o mecanismo de *relay* e de *multicast* (*propa-*

gated pipes). Porém, diversas dificuldades foram encontradas com a plataforma. A principal delas sendo a falta de documentação. Os livros tratando da plataforma são antigos e já não refletem o estado atual da implementação. Os guias e tutoriais disponíveis tratam somente do básico. O canal que se mostrou mais eficiente para obter informações sobre o JXTA foi o fórum do site oficial. Apesar disso, nem sempre são obtidas as respostas desejadas.

Inicialmente, estava previsto o uso do mecanismo de *relays* para avaliação do serviço de detecção com *peers* executando em diferentes *hosts*, localizados em redes distintas. Testes iniciais, realizados com o uso de *relays* públicos fornecidos pela comunidade JXTA apontaram a possibilidade dos experimentos. Porém, estes servidores públicos se tornaram indisponíveis e as tentativas de uso de *relays* próprios não foram bem sucedidas. Por este motivo, os testes JXTA foram executados em um único *host*. Por fim, cada instância do JXTA executa somente um único *peer*, e a quantidade de processamento necessária para cada instância é alta. Isto fez com que os testes tivessem de ser realizados com um pequeno grupo de *peers*, para que os resultados não fossem impactados pela falta de recursos de processamento.

Para trabalhos futuros, alguns caminhos podem ser apontados. O primeiro, seria a execução do serviço JXTA-FD em um número maior de *hosts*, utilizando os mecanismos de *rendezvous* e *relay* quando necessário. Mais testes também poderiam ser feitos para verificar o impacto do JXTA na exatidão do algoritmo de detecção. Outro caminho, seria a implementação do serviço de detecção para outras plataformas e tecnologias. Estas implementações poderiam então ser comparadas entre si e com a implementação original na plataforma JXTA. Por fim, algum algoritmo para solução de problema de acordo poderia ser implementado utilizando como base o serviço de detecção. Um exemplo seria o algoritmo Paxos[37] para a resolução do consenso.

Referências Bibliográficas

[1] *JXTA Java Standard Edition v2.5: Programmers Guide*, Setembro 2007.

[2] Extensible markup language (xml). Disponível no endereço: http://www.w3.org/XML/, Acessado em agosto de 2008.

[3] Jxta community website. Disponível no endereço: https://jxta.dev.java.net/, Acessado em agosto de 2008.

[4] Jxta v2.0 protocols specification. Disponível no endereço: https://jxta-spec.dev.java.net/nonav/JXTAProtocols.html, Acessado em agosto de 2008.

[5] M. K. Aguilera, W. Chen, and S. Toueg. Heartbeat: A timeout-free failure detector for quiescent reliable communication. In *WDAG '97: Proceedings of the 11th International Workshop on Distributed Algorithms*, pages 126–140, London, UK, 1997. Springer-Verlag.

[6] M. K. Aguilera, W. Chen, and S. Toueg. On quiescent reliable communication. *SIAM J. Comput.*, 29(6):2040–2073, 2000.

[7] M. K. Aguilera, S. Toueg, and B. Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 19–33, London, UK, 1999. Springer-Verlag.

[8] S. Androutsellis-Theotokis and D. Spinellis. A survey of peer-to-peer content distribution technologies. *ACM Comput. Surv.*, 36(4):335–371, 2004.

[9] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM.

[10] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifier (uri): Generic syntax. Request For Comments (RFC) 3986, 2005.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, 1996.

[12] T. D. Chandra, V. Hadzilacos, S. Toueg, and B. Charron-Bost. On the impossibility of group membership. In *PODC '96: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 322–330, New York, NY, USA, 1996. ACM.

[13] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *J. ACM*, 43(2):225–267, 1996.

[14] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. *IEEE Trans. Comput.*, 51(1):13–32, 2002.

[15] F. Chu. Reducing Ω to $\diamondsuit\mathcal{W}$. *Inf. Process. Lett.*, 67(6):289–293, 1998.

[16] V. F. Costa and F. Greve. Aspectos práticos da realização do consenso ft-cup em redes móveis ad-hoc. *Workshop de Testes e Tolerância a Falhas (WTF'07)*, pages 189–202, 2007.

[17] A. Das, I. Gupta, and A. Motivala. Swim: scalable weakly-consistent infection-style process group membership protocol. In *Proc. International Conference on Dependable Systems and Networks DSN 2002*, pages 303–312, 23–26 June 2002.

[18] R. De Prisco, D. Malkhi, and M. K. Reiter. On k-set consensus problems in asynchronous systems. In *PODC '99: Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing*, pages 257–265, New York, NY, USA, 1999. ACM.

[19] X. Défago. *Agreement-Related Problems: From Semi-Passive Replication to Totally Ordered Broadcast*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 2000.

[20] X. Défago, A. Schiper, and P. Urbán. Totally ordered broadcast and multicast algorithms: A comprehensive survey. Technical report, Swiss Federal Institute of Technology, 2000.

[21] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[22] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, New York, NY, USA, 2004. ACM.

[23] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *J. ACM*, 34(1):77–97, 1987.

[24] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl. Reaching approximate agreement in the presence of faults. *J. ACM*, 33(3):499–516, 1986.

[25] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[26] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *J. ACM*, 32(2):374–382, 1985.

[27] J. Gray. Notes on data base operating systems. In *Operating Systems, An Advanced Course*, pages 393–481, London, UK, 1978. Springer-Verlag.

[28] F. G. P. Greve. Protocolos fundamentais para o desenvolvimento de aplicações robustas. *SBRC'05*, 2005.

[29] F. G. P. Greve, M. Hurfin, M. Raynal, and F. Tronel. Primary component asynchronous group membership as an instance of a generic agreement framework. In *ISADS '01: Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems*, page 93, Washington, DC, USA, 2001. IEEE Computer Society.

[30] R. Guerraoui. Nonblocking atomic commit in asynchronous distributed systems with failure detectors. *Distrib. Comput.*, 15(1):17–25, 2002.

[31] R. Guerraoui, M. Herlihy, P. Kouznetsov, N. Lynch, and C. Newport. On the weakest failure detector ever. In *PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*, pages 235–243, New York, NY, USA, 2007. ACM.

[32] R. Guerraoui and P. Kouznetsov. On the weakest failure detector for non-blocking atomic commit. In *TCS '02: Proceedings of the IFIP 17th World Computer Congress - TC1 Stream / 2nd IFIP International Conference on Theoretical Computer Science*, pages 461–473, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V.

[33] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[34] I. Gupta, K. P. Birman, and R. van Renesse. Fighting fire with fire: using randomized gossip to combat stochastic scalability limits. *Quality and Reliability Engineering International*, 18(3):165–184, 2002.

[35] I. Gupta, T. D. Chandra, and G. S. Goldszmidt. On scalable and efficient distributed failure detectors. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*, pages 170–179, New York, NY, USA, 2001. ACM.

[36] V. Hadzilacos and S. Toueg. A modular approach to fault-tolerant broadcasts and related problems. Technical report, Cornell University, Ithaca, NY, USA, 1994.

[37] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[38] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[39] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *SRDS '00: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS'00)*, page 52, Washington, DC, USA, 2000. IEEE Computer Society.

[40] M. Larrea, A. Fernández, and S. Arévalo. On the implementation of unreliable failure detectors in partially synchronous systems. *IEEE Trans. Comput.*, 53(7):815–828, 2004.

[41] N. A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[42] M. H. MacDougall. *Simulating Computer Systems, Techniques and Tools*. The MIT Press, 1997.

[43] R. Moats. Urn syntax. Request For Comments (RFC) 2141, 1997.

[44] A. Mostefaoui, E. Mourgaya, and M. Raynal. Asynchronous implementation of failure detectors. In *Proc. International Conference on Dependable Systems and Networks*, pages 351–360, Los Alamitos, CA, USA, 22–25 June 2003. IEEE Computer Society.

[45] A. Mostefaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[46] A. Oram, editor. *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2001.

[47] M. Raynal. A short introduction to failure detectors for asynchronous distributed systems. *SIGACT News*, 36(1):53–70, 2005.

[48] D. Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM.

[49] J. Turek and D. Shasha. The many faces of consensus in distributed systems. *Computer*, 25(6):8–17, 1992.

[50] R. van Renesse, Y. Minsky, and M. Hayden. A gossip-style failure detection service. Technical report, Cornell University, Ithaca, NY, USA, 1998.

[51] B. J. Wilson. *JXTA*. New Riders, 2002.