Um Serviço Distribuído de Detecção de Falhas Baseado em Disseminação Epidêmica

Leandro P. de Sousa, Elias P. Duarte Jr.

Departamento de Informática – Universidade Federal Paraná (UFPR)
Caixa Postal 19018 – 81531-980 – Curitiba – PR

{leandrops,elias}@inf.ufpr.br

Abstract. *Failure detectors are abstractions that can be used to solve consensus in asynchronous systems. This work presents a failure detection service based on a gossip strategy. The service was implemented on the JXTA platform. A simulator was also implemented so the detector could be evaluated for a larger number of processes. Experimental results show CPU and memory usage, fault and recovery detection time, mistake rate and how the detector performs when used in a simple election algorithm. The results indicate that the service scales well as the number of processes grow.*

Resumo. *Detectores de falhas são abstrações que, dependendo das propriedades que oferecem, permitem a solução do consenso em sistemas distribuídos assíncronos. Este trabalho apresenta um serviço de detecção de falhas baseado em disseminação epidêmica. O serviço foi implementado para a plataforma JXTA. Para permitir a avaliação com um número maior de processos, foi também implementado um simulador. Resultados experimentais são apresentados para o uso de processador e memória, tempo de detecção, taxa de enganos do detector, além do seu uso na execução de eleição de líder. Os resultados experimentais e de simulação indicam que o serviço é escalável com o número de processos e mostram que a estratégia de disseminação epidêmica possui vantagens significativas em grupos com grande número de processos.*

1. Introdução

Grande parte dos problemas relacionados aos sistemas distribuídos requer algum tipo de coordenação entre os diversos componentes [Greve 2005, Turek and Shasha 1992], chamados aqui de *processos*. Tanto os processos quanto os canais de comunicação entre eles podem sofrer falhas. 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* [Fischer et al. 1985]. Este resultado torna impossível a resolução de uma classe de problemas distribuídos de extrema importância, os chamados *problemas de acordo*.

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*

[Chandra and Toueg 1996]. 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 característas temporais do sistema.

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* [Gupta et al. 2002]. 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 [JXTA 2009]. Um simulador também foi implementado, utilizando a biblioteca SMPL [MacDougall 1997]. Resultados experimentais são apresentados tanto para a implementação JXTA quanto para a simulação. São avaliados o uso de recursos do sistema, probabilidade de enganos e velocidade do detector para diferentes configurações de parâmetros do serviço de detecção.

Os resultados da simulação indicam que o serviço proposto é escalável para o número de processos, 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 processos. Porém, os experimentos realizados para a plataforma JXTA apontam para o uso elevado de processamento pela implementação do serviço, mesmo para um grupo pequeno de processos.

Este trabalho está organizado da seguinte maneira. Na seção 2, são apresentados alguns trabalhos relacionados na área de implementação de detectores de falha. Na seção 3, são descritos o serviço proposto juntamente com o algoritmo de deteção utilizado. Na seção 4 são apresentados resultados experimentais e de simulação. A seção 5 conclui o trabalho e apresenta algumas considerações sobre trabalhos futuros.

2. Trabalhos Relacionados

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. Em [Chandra and Toueg 1996], os autores mostram 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 tem-

porais, assim detectores de falhas são possíveis. Se este período for longo o suficiente, problemas como o do consenso podem ser resolvidos nestes sistemas [Raynal 2005].

Para o uso prático de detectores de falhas, aplicações podem ter necessidades além das propriedades eventuais dos detectores, propostas em [Chandra and Toueg 1996]. 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. 2002] propõe 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. Ainda em [Chen et al. 2002], os autores propõem um 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.

Em [Das et al. 2002], é descrito um protocolo de gestão da composição de grupos, chamado SWIM. 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 [Gupta et al. 2001]. O detector utiliza uma estratégia de *ping* randomizado, onde cada processo periodicamente testa outro processo, selecionado aleatoriamente. Informações sobre a composição do grupo e falhas de processos são transmitidas nas próprias mensagens do detector de falhas, através de um mecanismo de *piggybacking*.

Em [van Renesse et al. 1998], é proposto um algoritmo de detecção de falhas que se utiliza de uma estratégia de disseminação epidêmica (*gossip*). De acordo com esta estratégia, processos enviam periodicamente mensagens de *gossip* para um grupo de outros processos, estes escolhidos de maneira aleatória. A detecção falhas é feita por um mecanismo de *heartbeat*. Cada mensagem de *gossip* é composta pelo valor de *heartbeat* do processo emissor juntamente com últimos valores de *heartbeat* que este tenha recebido de outros processos. Processos que não recebem informação nova sobre um outro determinado processo, após um determinado intervalo de tempo, passam a considerar este último como falho. 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. O serviço de deteção proposto neste trabalho se baseia neste algoritmo de deteção.

3. O Serviço de Detecção Proposto

Este trabalho propõe um serviço de deteção de falhas baseado em disseminação epidêmica. O serviço foi implementado para a plataforma JXTA, sendo chamado JXTA-FD. O algoritmo de detecção implementado pelo serviço JXTA-FD é baseado no algoritmo proposto em [van Renesse et al. 1998]. 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 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

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 1. Algoritmo de detecção de falhas do serviço JXTA-FD.

sistema é assíncrono; mais especificamente, o sistema tem propriedades probabilísticas tanto para o atraso dos canais 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.

3.1. 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 IDs 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 última mensagem de difusão. $timeOfLastBcast$ é utilizado na decisão de quando um processo deve fazer uma nova difusão ou não.

3.2. 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.

3.3. 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 $timeOfLastBcast$.

3.4. 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 $heartbeat$.

3.5. 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 em [van Renesse et al. 1998], $p(t) = (t/BCAST\_MAX\_PERIOD)^{BCAST\_FACTOR}$, 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 [van Renesse et al. 1998].

3.6. 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.

3.7. 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. Implementação e Resultados Experimentais

O serviço JXTA-FD foi implementado como um módulo (*Module*) para a plataforma JXTA, versão 2.5, utilizando a linguagem Java. 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 qualquer momento, um *peer* pode consultar seu módulo de detecção pela lista de processos corretos ou suspeitos.

O comportamento do serviço 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. Os parâmetros do serviço para um dado grupo devem ser decididos a priori, antes da inicialização do mesmo.

4.1. 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 [MacDougall 1997].

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.

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 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.

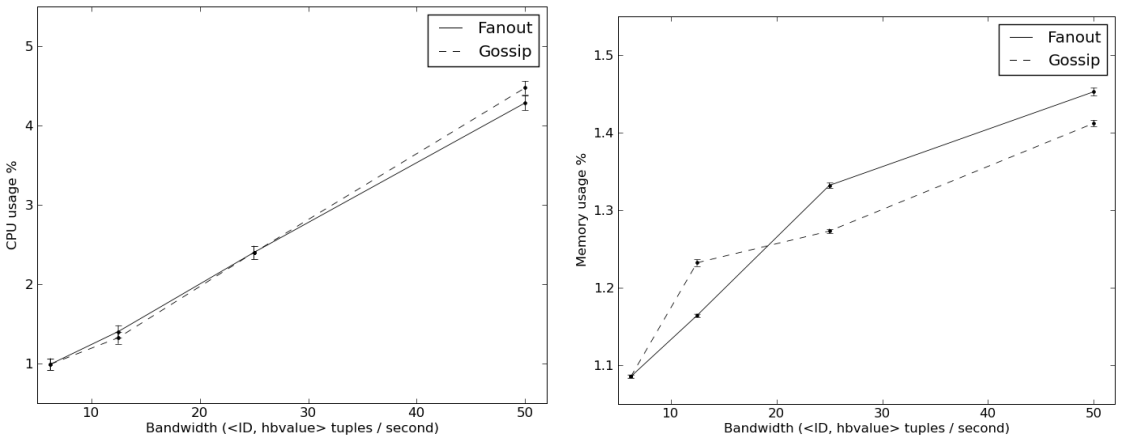
4.2. Resultados da Implementação JXTA

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 com 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. Os experimentos foram realizados 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 2(a) mostra o impacto 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 2(b) que o uso de memória também acompanha o uso de banda do detector.



(a) (b)

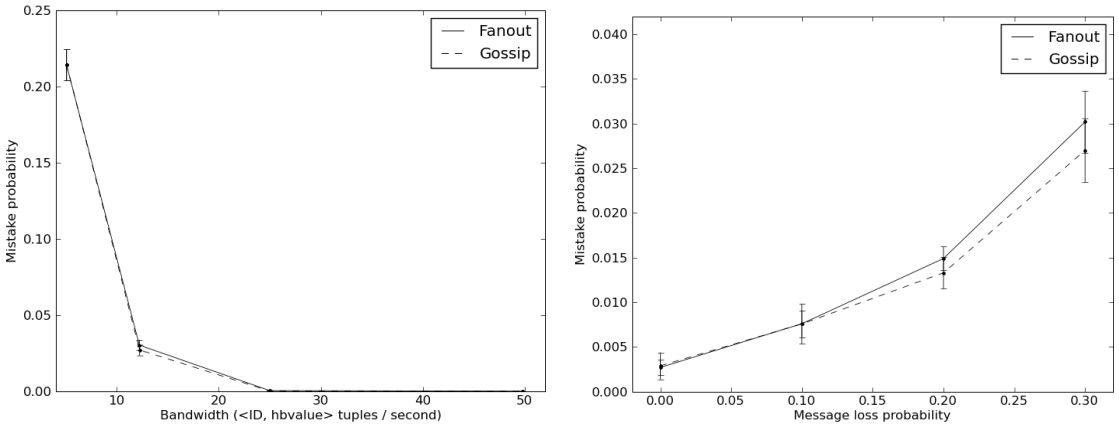
Figura 2. (a) Impacto dos parâmetros do detector no uso de CPU. (b) Impacto dos parâmetros do detector no uso de memória.

Estes resultados demonstram que a plataforma JXTA consome uma quantidade

de recursos considerável, tendo em vista que este é um serviço básico utilizado para a implementação de outros algoritmos. Vale ressaltar que a quantidade de *peers* utilizada é bastante pequena e estes participam de apenas um grupo de detecção.

Probabilidade de Enganos

Estes experimentos buscam verificar 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) (b)

Figura 3. (a) Variação no número de enganos de acordo com a banda utilizada. Experimento realizado com 30% das mensagens sendo descartadas. (b) Variação no número de enganos de acordo com a probabilidade de perda de mensagens. O valor de banda utilizada é fixado em 25.

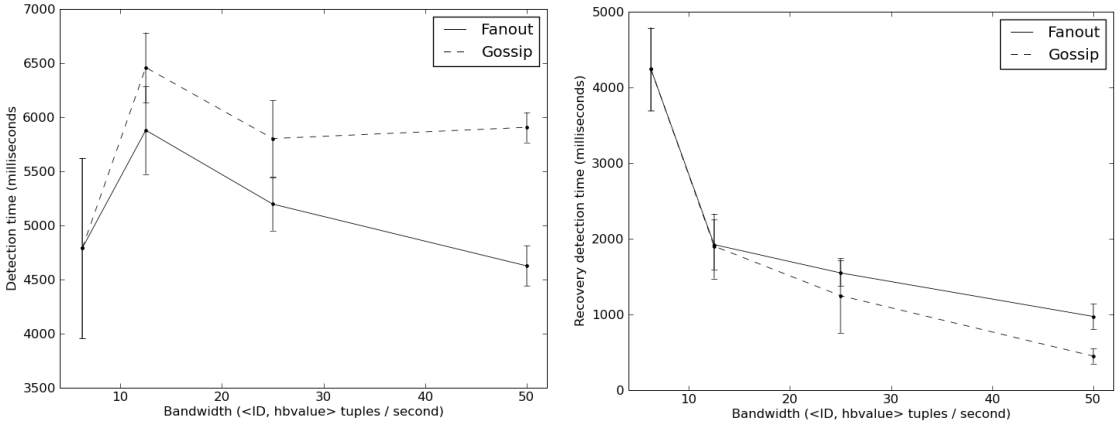
A Figura 3(a) mostra a probabilidade de uma consulta ao detector ser um engano, para uma quantidade de perda de mensagens igual a 30%. É possível observar a melhora na exatidão do detector com o aumento do *fanout* ou na frequência do envio de mensagens de *gossips*. A probabilidade de enganos para o valor de banda 12.5 é aproximadamente 0.02700 para a 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 3(b) 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*.

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 e sem perda de mensagens. Em um dado instante, um *peer* interrompe sua execução. Este *peer* volta a executar 10 segundos depois de parar.



(a) (b)

Figura 4. (a) Tempo de detecção de uma falha, para diferentes valores de banda utilizada. (b) Tempo de detecção da recuperação, para diferentes valores de banda utilizada.

A Figura 4(a) 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 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.

Na Figura 4(b), 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, provavelmente, à maior frequência de atualizações. Também pode-se observar que o impacto no aumento do uso de banda afeta diretamente o tempo de recuperação.

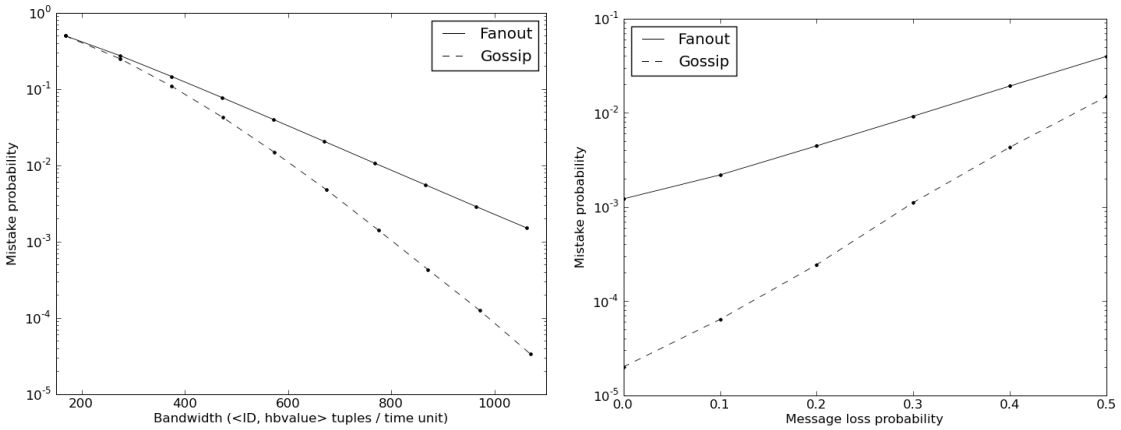
4.3. Resultados da Simulação

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.

Os 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 $FANOUT$.

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) (b)

Figura 5. (a) Probabilidade de engano do algoritmo. Experimento realizado com 50% das mensagens sendo descartadas. (b) Probabilidade de engano do algoritmo para diferentes probabilidades de perda de mensagens.

A Figura 5(a) 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 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 5(b) 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 *Fa-*

nout 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.

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.

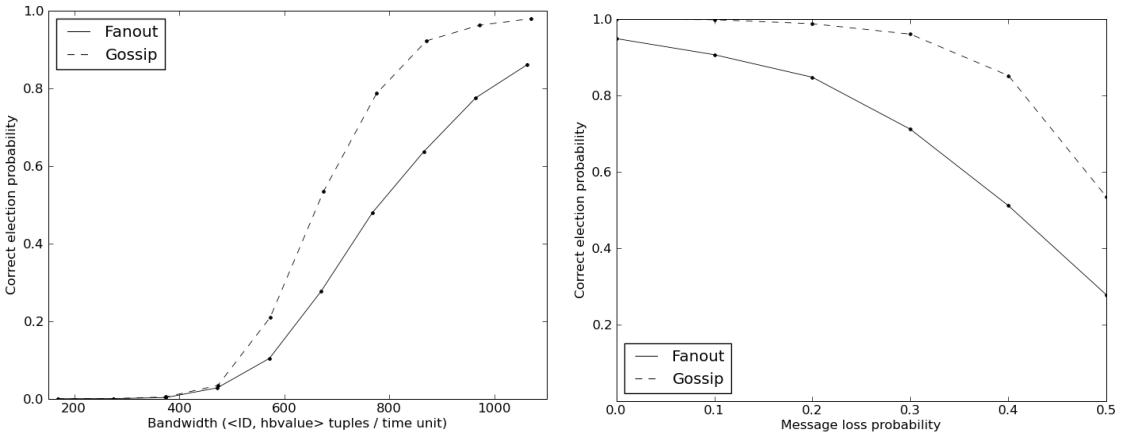


Figura 6. (a) Probabilidade de eleição correta. Experimento com 50% de mensagens perdidas. (b) Probabilidade de eleição correta para diferentes valores de perda de mensagens. A banda utilizada é de aproximadamente 650.

A Figura 6(a) 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, pode-se verificar que resultados razoáveis somente são obtidos com valores maiores de banda, ou seja, baixa probabilidade de enganos.

Na Figura 6(b) 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.

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 decisão pelo uso da plataforma JXTA foi motivada pela disponibilidade de facilidades para o desenvolvimento de aplicações par-a-par, mais especificamente os mecanismos de *relay*, *rendezvous* e *multicast* de mensagens (*propagated pipes*), que deveriam funcionar de maneira transparente. A realidade porém é bastante distinta. Os componentes não funcionaram, impossibilitando testes com máquinas em redes diferentes. A documentação existente era insuficiente ou desatualizada. Uma outra opção seria o uso dos *relays* e *rendezvous* públicos, fornecidos pela comunidade JXTA, mas os mesmos nunca estiveram disponíveis durante a implementação do serviço e realização dos experimentos.

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[Lamport 1998] para a resolução do consenso.

Referências

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

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

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

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

Greve, F. G. P. (2005). Protocolos fundamentais para o desenvolvimento de aplicações robustas. *SBRC'05*.

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

Gupta, I., Chandra, T. D., and Goldszmidt, G. S. (2001). 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. ACM.

JXTA (2009). Jxta community website. `https://jxta.dev.java.net/`, acessado em junho de 2009.

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

MacDougall, M. H. (1997). *Simulating Computer Systems, Techniques and Tools*. The MIT Press.

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

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

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