Implementação e Análise de Desempenho de um Mecanismo Adaptativo para Tolerância a Falhas em Sistemas Distribuídos com QoS

Sérgio Gorender[1], Raimundo J. A. Macêdo, Matheus Cunha

Laboratório de Sistemas Distribuídos (LaSiD)
Departamento de Ciência da Computação (DCC)
Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros S/N, Ondina,
Salvador - BA, CEP: 40.170-11

{gorender, macedo, matheusc}@ufba.br

Resumo. Tolerância a falhas e adaptabilidade são requisitos importantes para sistemas distribuídos modernos, especialmente aqueles que precisam se adaptar dinamicamente para diferentes níveis de qualidade de serviço (*QoS*). Neste artigo, apresentamos a implementação de uma infraestrutura de comunicação e gerenciamento de recursos de *QoS*, implementada sobre uma rede de estações LINUX equipadas com um pacote de controle de trafego (*DiffServ*). Baseados nesta estrutura, mostramos a implementação de um mecanismo de tolerância a falhas adaptável em tempo de execução (*run-time*) a diferentes níveis de *QoS*, o qual é composto de um detector de defeitos e um módulo de consenso. O protocolo de consenso apresentado goza de uma propriedade bastante interessante: a de poder operar concomitantemente, numa mesma execução, com diferentes níveis de *QoS* para processos distintos, caracterizando um modelo híbrido de tolerância a falhas. Também apresentamos dados de desempenho coletados a partir de vários experimentos onde foram medidos os tempos de obtenção de consenso para cenários diversos.

Abstract. Fault tolerance and adaptability are important issues for modern distributed systems. The capability of dynamically adapting to distinct run-time conditions is especially relevant for environments where negotiated QoS conditions cannot always be delivered between communicating processes. In this paper, we present an implementation of a framework for communication and resource management, using a network of LINUX workstations, which were equipped with a traffic control packet. We also implemented a fault tolerant mechanism, composed of a failure detector and a consensus protocol, which dynamic adapts to distinct QoS levels. The consensus protocol has the property that in the same consensus distinct processes can have different QoS, which characterize a hybrid fault tolerant model. We also measure the execution of the consensus, and present the result of these experiments.

[1] Aluno de doutorado do CIN/UFPE, pesquisador do LaSiD/UFBA e professor do DCC/UFBA e da Faculdade Ruy Barbosa.

1. Introdução

Tolerância a falhas adaptativa é um requisito importante para sistemas distribuídos modernos, especialmente em sistemas que executam em ambientes com diferentes níveis de qualidade de serviço (*QoS*) [20]. Os mecanismos de tolerância à falhas, em geral, assumem um modelo de execução específico. No modelo síncrono existem limites temporais conhecidos e garantidos para os serviços de execução (execução de passos dos processos e transferência de mensagens), enquanto que no modelo assíncrono estes limites temporais não existem. Para o modelo assíncrono foi provado em [10] a impossibilidade de se obter o consenso tolerando falhas. Devido a esta impossibilidade foram criados os modelos chamados parcialmente síncronos, que estendem o modelo assíncrono com características síncronas [9, 8, 7, 19]. O consenso distribuído é freqüentemente utilizado como uma forma de testar a tolerância à falhas destes modelos, além de ser utilizado em diversas aplicações distribuídas.

Por outro lado, utilizando as novas tecnologias para prover um ambiente de execução com diferentes níveis de serviço [3, 5, 20], podemos obter um ambiente híbrido, no qual convivem serviços de execução isócronos (serviços realizados dentro de limites de tempo previamente definidos) e serviços de execução não isócronos (sem limites de tempo). Além disso, os serviços de execução podem ter o seu nível de qualidade alterado dinamicamente, o que exige dos sistemas a capacidade de se adaptar a estas alterações. Ou seja, além das aplicações, os mecanismos de tolerância a falhas precisam ser adaptativos nesses ambientes com QoS.

Neste artigo, apresentamos a implementação de uma infraestrutura de comunicação e gerenciamento de recursos de *QoS*, implementada sobre uma rede de estações LINUX equipadas com pacotes de controle de tráfego funcionando segundo a arquitetura Serviços Diferenciados (*DiffServ*) [5]. Baseados nessa estrutura, mostramos a implementação de um mecanismo de tolerância a falhas, composto de um detector de defeitos e um módulo de consenso, que são adaptáveis em tempo de execução (*run-time*) a diferentes níveis de *QoS* [11, 12]. O protocolo de consenso descrito goza de uma propriedade bastante interessante: a de poder operar concomitantemente, numa mesma execução, com diferentes níveis de *QoS* para os processos distintos, caracterizando um modelo híbrido de tolerância a falhas. Também apresentamos dados de desempenho coletados a partir de vários experimentos onde foram medidos os tempos de obtenção de consenso para cenários diversos.

Na seção 2, a seguir, são apresentadas uma infraestrutura de comunicação e gerenciamento de recursos e sua implementação. Na seção 3 são apresentados os conceitos básicos de um detector de defeitos adaptativo e de um algoritmo de consenso híbrido e adaptativo, assim como a implementação destes dois mecanismos. Na seção 4 são apresentados resultados obtidos com a execução do protótipo implementado e uma análise de desempenho. Os trabalhos correlatos são apresentados na seção 6 e na seção 7 apresentamos algumas conclusões.

2. Infraestrutura de comunicação e gerenciamento de recursos

O ambiente de comunicação é composto por canais de comunicação, os quais são providos com diferentes níveis de serviço, segundo a arquitetura Serviços Diferenciados (*DiffServ*) para prover QoS [5]. A arquitetura DiffServ determina que os fluxos de informação (pacotes de dados) a ser transferidos sejam atribuídos a classes de serviço para o encaminhamento de pacotes. Estas classes de serviço determinam o nível do

serviço de comunicação que será provido ao fluxo de pacotes. Cada classe é definida baseada em requisitos, tais como um atraso limitado ou não limitado para a transferência de pacotes, um *jitter* (variação no atraso) alto ou restrito, uma maior ou menor prioridade para o descarte de pacotes e uma maior ou menor largura de banda. Cada classe de serviço na arquitetura *DiffServ* é identificada através de um valor especial, chamado *DiffServ Code Point* (DSCP), o qual é armazenado no campo *Type of Service* (ToS) do cabeçalho IP de cada pacote. Quando um pacote entra em uma rede *DiffServ*, diversos campos de seu cabeçalho IP são analisados com o objetivo de se identificar a que classe de encaminhamento o pacote deve ser associado (classificação do pacotes), e o pacote tem um valor DSCP armazenado no seu campo ToS (marcação do pacote). O roteador que marca os pacotes é chamado de roteador de borda. Os demais roteadores no caminho do pacote até o seu destino (rota), os quais são chamados de roteadores de núcleo, apenas analisam o valor DSCP armazenado mo cabeçalho do pacote e encaminham o pacote de acordo com a classe de serviço associada a este valor.

Existem diversas classes de serviço definidas na arquitetura *DiffServ*. Entre estas, os Serviços Expressos fornecem uma comunicação isócrona, garantindo limites máximos para a transferência de pacotes. Esta garantia é fornecida através de altas prioridades para o encaminhamento, e de uma alta taxa de transmissão, suficiente para transmitir imediatamente todo pacote que chegue a um roteador. As demais classes fornecem serviços de comunicação não isócronos, sem garantias temporais. Estas classes atribuem diferentes prioridades para o descarte de pacotes, no caso de haver congestionamentos nos roteadores. Chamamos os canais de comunicação criados com serviços isócronos (Serviço Expresso) de canais de comunicação *timely*, e os canais de comunicação criados com serviços não isócronos de canais *untimely*.

Implementamos classes de serviço isócrono e não isócrono para o encaminhamento de fluxos de pacotes foram implementadas no ambiente de execução, o qual é composto por em estações LINUX configuradas para funcionar como roteadores. Para criar estas classes de serviço utilizamos um pacote de controle de tráfego para o LINUX, chamado TC. Este pacote define uma linguagem, a qual é utilizada para a definição das classes de serviço para o encaminhamento de pacotes, através da construção de *scripts*. As classes de serviço são criadas a partir da definição de filas para o armazenamento dos pacotes a serem encaminhados, uma para cada classe. Para cada fila é definida uma prioridade para o encaminhamento de pacotes, além de um algoritmo específico para o gerenciamento da fila e disciplinas de filas adequadas à definição da classe. A disciplina de fila DSMark [20] implementa a marcação de pacotes com um valor DSCP nos roteadores de borda, antes de estes roteadores encaminharem os pacotes. Esta mesma disciplina de fila é utilizada pelos roteadores de núcleo para classificarem cada pacote para a classe de serviço apropriada, dependendo apenas do conteúdo do campo ToS (valor DSCP). Na definição das filas para o encaminhamento de pacotes, o TC permite definir a largura de banda aplicada a cada fila, a prioridade para o encaminhamento de pacotes, o algoritmo de fila a ser utilizado e outros requisitos, permitindo assim a definição de uma classe de encaminhamento isócrona, com uma alta largura de banda e prioridade máxima para o encaminhamento de pacotes.

A infraestrutura de gerenciamento de comunicação (QoS Provider) e o mecanismo para sistemas distribuídos tolerantes a falhas (detector de defeitos e consenso) foram implementados na linguagem Java e testados sobre o ambiente de

comunicação criado com o LINUX e o pacote TC.

2.1 QoS Provider

Denominamos de *QoS Provider* o mecanismo que desenvolvemos e implementamos cuja atribuição é criar canais de comunicação para a aplicação e gerenciar o nível do serviço de comunicação fornecido a cada canal. Este mecanismo executa distribuído em módulos associados aos processos do sistema. Gerenciar a QoS dos canais de comunicação implica em negociar o nível do serviço a ser provido a cada canal e monitorar este serviço durante a sua execução. As funções *CreateChannel(), DefineQoS(), Delay()* e *QoS()* são chamadas pelos processos das aplicações distribuídas para criar canais de comunicação entre processos, negociar a *QoS* para um determinado canal de comunicação, determinar o atraso na transferência de mensagens para um canal de comunicação e monitorar a *QoS* corrente atribuída a um canal de comunicação. É importante notar que a função *Delay()* retorna um valor probabilístico se o canal de comunicação for *untimely*, mas retorna um valor determinado caso este canal de comunicação seja *timely*.

Além destas funções, o QoS Provider monitora todos os canais de comunicação *timely*, para verificar se estes canais continuam sendo providos com serviços isócronos. Esta monitoração é feita utilizando mensagens de um protocolo de sinalização, que permitam a troca de informações entre os módulos do QoS Provider e os roteadores e provedores de serviços de comunicação. Sempre que for identificada alguma alteração na qualidade de serviço provida a um canal, fazendo este canal passar a *untimely*, esta alteração é sinalizada aos processos comunicantes.

Os conceitos que foram utilizados no desenvolvimento do QoS Provider foram baseados em diversas arquiteturas desenvolvidas para prover *QoS* fim-a-fim a aplicações distribuídas [3], como por exemplo a arquitetura Omega e o dispositivo QoS Broker [18].

2.2 Implementação do QoS Provider

Foi desenvolvida uma implementação inicial do QoS Provider com o objetivo básico de criar canais de comunicação e fornecer estes canais para os processos solicitantes. Nesta implementação as funções *CreateChannel()* e *Delay()* foram implementadas. As demais funcionalidades do QoS Provider, tais como negociar a *QoS* dos canais de comunicação e monitorar esta *QoS* ainda estão sendo desenvolvidas. Nesta implementação inicial as classes de serviço isócrono e não isócrono são criadas diretamente nos roteadores LINUX, através da execução de *scripts* criados com a linguagem TC. Desta forma, a cada teste realizado, pudemos estabelecer os canais de comunicação como sendo *timely* ou *untimely*.

O QoS Provider possui duas classes principais, *Provider* e *Negotiator*. A classe *Provider* faz a interface entre o processo que está requisitando o canal de comunicação e o provedor de serviço da rede, e utiliza a classe *Negotiator* para acessar o provedor de serviço de comunicação. Quando um processo solicita a criação de um canal de comunicação com um outro processo, a instância da classe *Negotiator* pertencente ao processo solicitante envia uma mensagem *BeginNegotiation* ao outro processo. Ao receber esta mensagem, a instância da classe *Negotiator* deste outro processo registra a requisição e envia uma mensagem *EndNegotiation* para o processo solicitante, finalizando assim o estabelecimento do canal de comunicação. O canal de comunicação

criado é do tipo *UDP/IP*.

3. Implementação de um mecanismo adaptativo tolerante a falhas para sistemas distribuídos

Apresentamos uma visão geral do mecanismo adaptativo tolerante a falhas para sistemas distribuídos [11, 12], o qual foi implementado em Java/LINUX sobre a infraestrutura de comunicação apresentada na seção 2.

Um sistema distribuído é composto por um conjunto $\Pi=\{p_1, ..., p_n\}$ de processos, conectados por canais de comunicação pertencentes ao conjunto $C=\{c_{1/2}, ..., c_{1/n}, ..., c_{n-1/n}\}$. Os conjuntos Π e C formam o grafo completo $DS(\Pi, C\}$.

Assumimos no nosso sistema o modelo assíncrono aumentado com detectores de defeitos não confiáveis, como apresentado por Chandra & Toueg em [6]. Assumimos adicionalmente que os canais de comunicação podem ser fornecidos com *QoS*, de acordo com a arquitetura Serviços Diferenciados, como apresentado na seção anterior.

Assumimos também que os processos falham por *crash*.

3.1 Detector de defeitos adaptativo

O detector de defeitos adaptativo executa distribuído em módulos, um para cada processo do sistema. Os módulos do detector de defeitos dos processos em funcionamento enviam solicitações periódicas de mensagens de *heartbeat* para todos os demais módulos através dos canais de comunicação, e esperam por mensagens de *heartbeat* em resposta, até a ocorrência de um *timeout*, previamente calculado. O *timeout* para cada mensagem de *heartbeat* é calculado a partir do atraso para a transferência de mensagens, o qual é determinado para cada canal de comunicação. Por sua vez, este atraso é calculado pela função *Delay()* do QoS Provider. Para canais de comunicação *timely* o atraso informado pela função *Delay()* é garantido, enquanto o canal permanecer *timely*, enquanto que para canais *untimely* o atraso informado é probabilístico. Considerando as classes de serviço dos canais de comunicação existem dois tipos de detecção de defeitos: suspeitas, no caso de o canal de comunicação ser *untimely*, e notificações, se o canal de comunicação for *timely*. Como os módulos divulgam as notificações realizadas, um processo que possua ao menos um canal de comunicação *timely* será notificado por todos os processos corretos, caso venha a falhar.

Conseqüentemente, podemos classificar os processos operacionais como suspeitáveis ou notificáveis, inserindo a identificação destes processos, respectivamente, nos conjuntos *suspeitáveis* e *notificáveis*. Esta classificação é efetuada e atualizada por cada módulo do detector de defeitos, a partir da análise de uma representação local do grafo DS com informações sobre a *QoS* de cada canal de comunicação, obtidas através da função *QoS()* do QoS Provider, e de informações que são trocadas entre os módulos. Apenas processos cuja identificação pertence ao conjunto *notificáveis* podem ser notificados pelos módulos do detector de defeitos. Os demais processos só podem ser suspeitos de terem falhado. Processos notificáveis não podem ser erroneamente suspeitos por nenhum módulo do detector de defeitos, uma vez que um módulo que não possua canal de comunicação *timely* com este processo não irá monitorá-lo, apenas notificando uma falha deste processo, se receber uma mensagem, de outro módulo, comunicando esta notificação.

Processos que são notificados têm a sua identificação retirada do conjunto

notificáveis e inserida no conjunto *faltosos*, enquanto processos suspeitos têm a sua identificação inserida no conjunto *suspeitos*, apesar de continuarem processos suspeitáveis (a identificação destes processos não é retirada do conjunto *suspeitáveis*). Todos estes conjuntos são locais a cada módulo do detector de defeitos. Os módulos do detector de defeitos atualizam seus conjuntos quando notificam processos ou suspeitam de processos, ou quando percebem alterações na *QoS* dos canais de comunicação. Os processos obtêm informações do seu módulo do detector de defeitos a respeito do estado do sistema através dos conjuntos fornecidos pelo módulo.

Um detector de defeitos com as características descritas acima possui as propriedades *strong completeness* (todas as falhas são detectadas, em algum momento, por todos os processos operacionais), *eventual weak accuracy* (a partir de algum momento haverá um processo correto que não será suspeito erroneamente por qualquer processo operacional) e Conditional QoS (processos notificáveis não são detectados erroneamente). Definimos que um detector de defeitos com estas propriedades pertence à classe de detectores $\Diamond S^{Adapt}$. As propriedades *strong completeness* e *eventual weak accuracy* são características de detectores de defeitos da classe $\Diamond S$, conforme definidas por Chandra & Toueg em [6]. Na seção 5 apresentamos uma definição resumida dos detectores de defeitos não confiáveis.

3.2 Consenso híbrido adaptativo

O algoritmo de consenso híbrido e adaptativo implementado resolve o problema do consenso uniforme, caracterizado pelas propriedades validade (se um processo decidir por um valor *v*, *v* deve ter sido proposto por algum processo), terminação (todos os processos corretos devem decidir por algum valor) e acordo uniforme (dois processos corretos não devem decidir de forma diferente). Este algoritmo executa em rodadas assíncronas com coordenadores circulantes, adotando um padrão de troca de mensagens descentralizado, no qual o coordenador, se não falhar, envia seu valor estimado para todos os processos, e estes respondem, também a todos, com uma mensagem *ack*, se a estimativa do coordenador foi recebida, ou com uma mensagem *nack*, se o coordenador foi notificado ou suspeitado. O número de mensagens que cada processo deve receber, em cada rodada, para poder progredir, decidindo ou não, é chamado de quorum. Se um processo recebe apenas mensagens *ack* de um quorum de processos ele decide pelo valor proposto pelo coordenador.

No nosso algoritmo, o quorum é composto por todos os processos notificáveis do sistema mais uma maioria dos processos suspeitáveis, sendo, portanto um quorum híbrido. Como os conjuntos notificáveis e suspeitáveis podem ser alterados durante a execução do consenso, o quorum também será alterado dinamicamente, durante o consenso, caracterizando uma adaptação do consenso à capacidade do detector de defeitos de realizar detecções.

3.3 Implementação do Detector de Defeitos Adaptativo

O detector de defeitos foi implementado segundo o modelo *pull*, no qual uma mensagem é gerada por um módulo do detector de defeitos para cada outro módulo, solicitando um *heartbeat* de resposta.

Todos os processos possuem um módulo do detector de defeitos, o qual é composto por diversas classes, as quais são instanciadas como *threads* do processo em execução. A classe *HeartBeatCenter* envia periodicamente mensagens

HeartBeatMessage para todos os processos em execução, caracterizando uma solicitação de uma mensagem de *heartbeat*. Ao receber uma mensagem *HeartBeatMessage*, um *HeartBeatCenter* envia em resposta uma mensagem *HeartBeatAnswer*.

A *thread* detector, também instanciada para cada processo, verifica o atraso nas respostas dos *heartbeats* enviados e recebidos pelo *HeartBeatCenter* (mensagens *HeartBeatMessage* e *HeartBeatAnswear*). A *thread* detector verifica o atraso no recebimento de mensagens *HeartBeatAnswear* de cada processo, chamando o método *isLate*(p_i, p_j), sendo p_i o processo que está executando o método e p_j o processo a ser monitorado. Este método verifica se o *timeout* definido para o recebimento de uma mensagem *HeartBeatAnswear* do processo monitorado foi alcançado sem o recebimento da mensagem, e se o processo monitorado é notificável ou suspeitável. Dependendo se houve um atraso e da situação do processo, o módulo do detector pode suspeitar do processo monitorado, notificar uma falha deste processo ou nada fazer. O *timeout* para o recebimento de cada mensagem *HearBeatAnswear* é determinado a partir do atraso máximo definido para cada canal de comunicação, o qual é calculado e informado pela função *Delay()* do QoS Provider.

3.4 Implementação do Consenso Híbrido Adaptativo

O algoritmo de consenso é instanciado como uma *thread*, pertencendo a cada processo. Esta *thread* possui um método que a cada mensagem recebida verifica se o quorum foi alcançado, e se a mensagem recebida foi do tipo *ack* ou *nack*. Se o quorum for alcançado com mensagens *ack* o consenso decide.

4. Análise de resultados

Os testes foram realizados utilizando três computadores com processadores Pentium III de 900 Mhz, 128 Mb de memória principal, executando o sistema operacional LINUX Conectiva 8. O *kernel* utilizado foi o 2.4.18. Os computadores estão conectados em rede, através de segmentos de 10 *megabits*. O *kernel* do LINUX foi recompilado para que as máquinas funcionassem como roteadores *DiffServ*.

Em todos os computadores foram executados *scripts* do pacote TC para a criação de filas para o encaminhamento de pacotes provendo serviços isócronos e serviços não isócronos. De acordo com cada teste realizado o *DiffServ* foi executado um *script* específico em cada roteador, associando fluxos de dados de cada processo às classes de encaminhamento adequadas para o teste.

Nesta primeira implementação não foi nosso objetivo obter tempos expressivos na execução do consenso, por isso não dedicamos tempo demasiado em sua otimização. Espera-se que, após a otimização desta implementação, os tempos obtidos serão melhorados.

4.4 Resultados e análise de desempenho

Foram realizados dois tipos básicos de testes. O primeiro teste foi feito com o objetivo de verificar o quanto o aumento no número de processos notificáveis no sistema causaria um aumento no tempo de execução do consenso. Para tanto realizamos testes do consenso com 3 processos, com 6 processos e com 9 processos, todos eles notificáveis, e sem ocorrer falhas. Foram executados 10 testes para cada situação. Em todos estes testes o consenso foi obtido na primeira rodada.

Nos testes realizados com 3 processos obtivemos um tempo médio de execução do consenso de 97,3 milissegundos, enquanto que com 6 processos obtivemos o tempo médio de 444,4 milissegundos, e com 9 processos foi obtido o tempo médio 505,9 milissegundos. Como a implementação do detector de defeitos e do consenso ainda está sendo otimizada acreditamos que estes tempos podem ser melhorados. Entretanto, a cada mensagem recebida, o algoritmo de consenso efetua um processamento para verificar se o quorum foi obtido, e se foi possível chegar a uma decisão. Este processamento faz com que o tempo necessário para o tratamento de cada mensagem cresça, aumentando por conseqüência o tempo do consenso. Na figura 1 apresentamos um gráfico contendo os tempos médios de consenso obtidos em cada conjunto de experimentos.

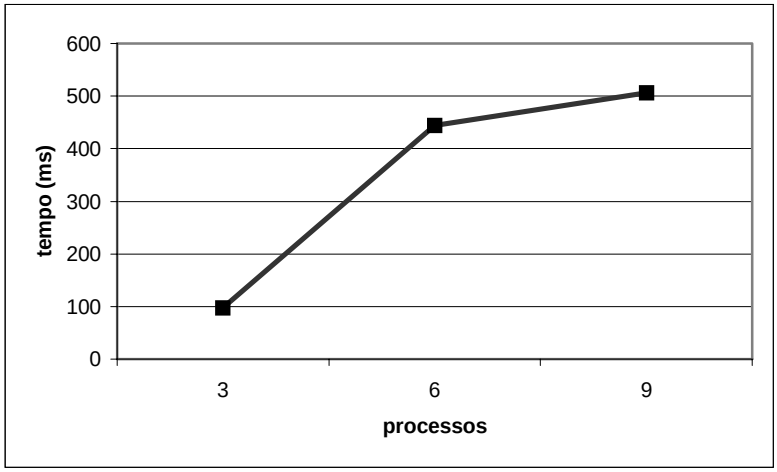

Figura 1: Tempos médios do consenso com 3, 6 e 9 processos.

Realizamos um segundo teste, com o objetivo de comparar os tempos de execução do consenso com um conjunto de processos notificáveis, e com um conjunto de processos suspeitáveis. Os resultados destes testes são apresentados na figura 2. Foram realizados 10 execuções de cada consenso, sempre com 6 processos, sem a ocorrência de falhas. No caso do consenso com processos suspeitáveis o quorum é formado por 4 processos, enquanto que no consenso com processos notificáveis o quorum passa a ser formado por todos os 6 processos. Devido ao *overhead* gerado pelo processamento das mensagens para verificar o quorum, o consenso executado com processos notificáveis apresenta tempos de execução superiores ao consenso com processos suspeitáveis.

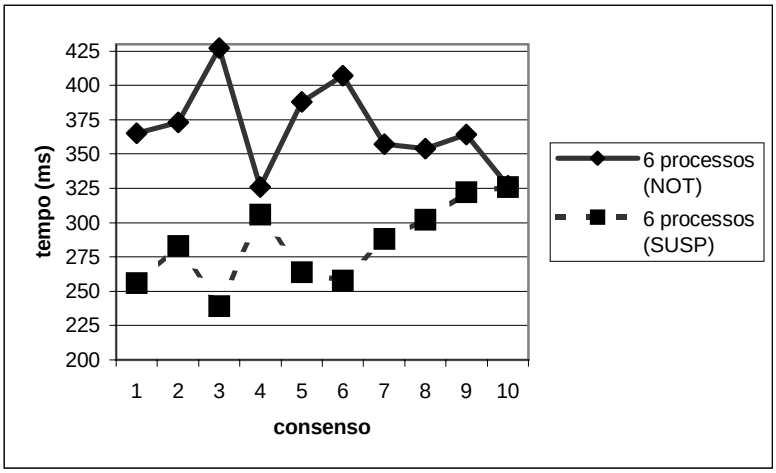

Figura 2: Consenso com 6 processos, todos notificáveis ou todos suspeitáveis.

Um outro resultado interessante que obtivemos com alguns testes realizados e com o processo de otimização da implementação, é que a necessidade de se policiar o tráfego gerado pelos processos, para que este tráfego não venha a exceder a capacidade de transmissão negociada com a rede, pode vir a aumentar o atraso na transferência de mensagens, também aumentando o tempo de execução do consenso. Em uma implementação anterior utilizamos um dispositivo para controlar o envio de pacotes, de acordo com uma taxa máxima previamente negociada, e ao retirarmos este dispositivo, os tempos obtidos para a execução do consenso diminuíram consideravelmente.

5. Trabalhos Correlatos

Tolerância à falhas em sistemas distribuídos assíncronos tem sido enfocada, de uma maneira geral, estendendo o ambiente assíncrono com alguma característica síncrona, uma vez que, como provado em [10], em ambientes assíncronos é impossível executar aplicações, como o consenso, tolerando qualquer número de falhas. Os modelos parcialmente síncronos, criados segundo este ponto de vista, permitem que aplicações sejam executadas, tolerando um número determinado de falhas. As soluções desenvolvidas dependem, em geral, do ambiente de execução e modelo de falhas adotado.

Dolev et al estenderam os resultados da impossibilidade do consenso, apresentados em [10], combinando 5 parâmetros, que podem ser favoráveis (síncronos) ou desfavoráveis (assíncronos) [8]. Dwork et al apresentam dois modelos parcialmente síncronos [9], um no qual os limites de tempo para a execução de ações e a transferência de mensagens existem, mas são desconhecidos, e outro no qual estes limites são conhecidos, mas só passam a valer depois de um tempo não determinado. Cristian e Feltzer apresentam o modelo Assíncrono Temporizado (*Timed Asynchronous*) [7], no qual os processos acessam relógios locais, e se assume que os ambientes de execução apresentam períodos de estabilidade, nos quais as ações dos processos são executadas com limites de tempo conhecidos e determinados, e apenas a um número limitado de mensagens é permitido ser entregue após um tempo limitado. Neste modelo foi criado o conceito de *fail awareness*, o qual determina que as ações que tenham excedido seu limite de tempo para executar, as mensagens que tenham sido entregues após o seu *timeout* e os relógios que apresentem um desvio de tempo acima de um limite pré-definido, sejam marcados como apresentando falha de performance. Casimiro e

Veríssimo apresentaram a Base de Computação Temporizada (*Timely Computing Base*) [19]. A TCB executa em um ambiente síncrono, monitorando a aplicação, sinalizando falhas de performance e executando ações temporizadas para a aplicação. Chandra e Toueg apresentaram o modelo assíncrono equipado com detectores de defeitos não confiáveis, os quais são distribuídos em 8 classes distintas, definidas pelas propriedades *accuracy* e *completeness* [6]. A propriedade *accuracy* diz respeito à capacidade de o detector não efetuar detecções incorretas e a propriedade *completeness* diz respeito à capacidade de o detector efetuar detecções corretas de todos os processos que falharam.

Algoritmos de consenso foram desenvolvidos para todos estes modelos, sendo que diversos protocolos de consenso, baseados em diferentes conceitos e paradigmas, têm sido desenvolvidos para executar com os detectores de defeitos não confiáveis, em especial os detectores da classe ◊S, caracterizados como os mais fracos a permitir a execução do consenso tolerando falhas. Além de Chandra e Toueg, algoritmos de consenso foram desenvolvidos por Mostefaoui e Raynal [16] e Hurfin et al [13], entre outros. Lamport desenvolveu o algoritmo de consenso Paxos para o sistema assíncrono [15], estendido com um detector de defeitos para eleição de líderes. Algoritmos de consenso híbridos, utilizando detectores de defeitos e oráculos para a geração de números aleatórios foram desenvolvidos em [1, 17]. Algoritmos de consenso que acessam oráculos para geração de números aleatórios foram introduzidos por Ben Or em [4].

De uma maneira geral, detectores de defeitos têm sido implementados utilizando o conceito de *timeout*, o qual determina limites no tempo para que mensagens sejam entregues ou ações sejam executadas. Entretanto, algumas implementações de detectores de defeitos não têm utilizado *timeouts*, monitorando os processos com base na análise das mensagens transferidas pela própria aplicação, e detectando falhas se um número demasiado de mensagens da aplicação não tiver sido entregue [2].

Existem diferentes formas para prover adaptação a sistemas distribuídos tolerando falhas. Os algoritmos de consenso descritos em [16, 13, 1, 17] possuem características de adaptabilidade, no sentido em que podem utilizar diferentes oráculos em sua execução, como os algoritmos descritos em [1, 17], os quais utilizam um oráculo detector de defeitos e um oráculo gerador de números aleatórios, quando o primeiro não funciona. Nós adotamos como objetivo do nosso mecanismo a adaptação a um ambiente de execução com QoS, que possa prover serviços de execução (em especial comunicação) com diferentes níveis de qualidade, o que difere das soluções conhecidas,

Utilizamos em nosso trabalho os detectores de falhas não confiáveis de Chandra & Toueg, porém em um ambiente de execução que altera dinamicamente o nível dos serviços fornecidos, exigindo de nosso mecanismo a capacidade de se adaptar dinamicamente a estas alterações. O nosso algoritmo de consenso é também adaptativo, sendo, além disto, híbrido, por considerar a existência de componentes isócronos e de componentes não isócronos, no ambiente de execução.

6. Conclusão

Apresentamos neste artigo uma implementação de um ambiente de comunicação que cria canais de comunicação e gerencia o nível de serviço provido a cada canal. Este ambiente de comunicação foi implementado sobre uma rede de estações LINUX executando um sistema para controle de tráfego, o qual permite definir disciplinas de enfileiramento de pacotes baseadas na arquitetura Serviços Diferenciados, para prover

QoS a serviços de comunicação. Também apresentamos a implementação em Java de uma infraestrutura tolerante a falhas para sistemas distribuídos, a qual é composta por um detector de defeitos adaptativo, e por um algoritmo de consenso híbrido, também adaptativo. As detecções de defeitos obtidas de um módulo do detector de defeitos podem ser suspeitas ou notificações, dependendo do nível de serviço provido a cada canal de comunicação. O consenso é obtido utilizando um quorum híbrido, o qual é composto por processos notificáveis e por processos suspeitáveis.

Os resultados obtidos mostraram haver um aumento no tempo de execução do consenso à medida que aumenta o número de mensagens a serem processadas. Isto se deve ao fato de que a cada mensagem recebida o consenso necessita verificar se o quorum foi alcançado e o consenso obtido. Estamos trabalhando para otimizar este processamento, e com isto, reduzir o tempo do consenso. Por outro lado, verificamos que, com um número maior de processos, o tempo total para a obtenção do consenso por todos os processos operacionais não será tão elevado, uma vez que, o primeiro processo a obter o consenso irá comunicar sua decisão aos demais processos, os quais poderão decidir ao receber a mensagem de decisão, antes de completarem o seu quorum.

Na próxima versão de nossa implementação o QoS Provider executará de forma a negociar e monitorar a *QoS* provida a cada canal de comunicação.

Referências

[1] Aguilera, M. K. e Toueg, S., *Failure Detection and Randomization: A Hybrid Approach to Solve Consensus*, SIAM Journal of Computing, 28(3), 1998, pp. 890-903, junho, 1999.

[2] Aguilera, M. K., Chen, W. e Toueg, S., *Heartbeat: a timeout-free failure detector for quiescent reliable communication*, *Proceedings of the 11th International Workshop on Distributed Algorithms*, Lecture Notes on Computer Science. Springer-Verlag, setembro, 1997.

[3] Aurrecoechea, C., Cambell, A. T. e Hauw, L., *A Survey of QoS architectures*, Multimedia Systems 6(3), Springer-Verlag, pp – 138-151, maio, 1998.

[4] Ben Or, M., *Another Advantage of Free-Choice: Completely Asynchronous Agreement Protocols*, Proceedings of the 2nd Annual ACM Symposium on Principles of distributed Computing (PODC 1983), pp. 27-30 , agosto, 1983.

[5] Blake, S. at al, "An Architecture for Differentiated Services", RFC 2475, dezembro, 1998.

[6] Chandra, T. D. e Toueg, S., *Unreliable Failure Detectors for Reliable Distributed Systems*, Journal of the ACM, Vol. 43 (2), pp. 225-267, março, 1996.

[7] Cristian, F. e Fetzer C., *The Timed Asynchronous Distributed System Model*, IEEE Transactions on Parallel and Distributed Systems, Vol. 10 (6), pp. 642-657, junho, 1999.

[8] Dolev, D., Dwork, C. e Stockmeyer, L., *On the Minimal Synchronism Needed for Distributed Consensus*, Journal of the ACM, Vol. 34 (1), pp. 77-97, janeiro, 1987.

[9] Dwork, C., Lynch, N. e Stockmeyer, L., *Consensus in the Presence of Partial Synchrony*, Journal of the ACM, Vol. 35 (2), pp. 288-323, abril, 1988.

[10] Fisher, M. J., Lynch, N. A. e Paterson, M. S., *Impossibility of Distributed Consensus with One Faulty Process*, Journal of the ACM, vol. 32 (2), pp. 374-382, abril, 1985.

[11] Gorender, S. e Macêdo, R. *Um Modelo para Tolerância a Falhas em Sistemas Distribuídos com QoS*, Anais do Simpósio Brasileiro de Redes de Computadores (SBRC2002), pp. 277-292, maio, 2002.

[12] Gorender, S. e Macêdo, R., *A Dynamically QoS Adaptable Consensus and Failure Detector*, The IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2002 – Fast Abstract Track, pp. B80-B81, junho, 2002.

[13] Hurfin M., Macêdo R., Mostefaoui A. e Raynal M., *A Consensus Protocol based on a Weak Failure Detector and a Sliding Round Window*, Proceedings of the 20th IEEE Int. Symposium on Reliable Distributed Systems (SRDS'01). New Orleans, USA, pp. 120-129, outubro, 2001.

[14] Keidar, I. e Rajsbaum, S., *On the Cost of Fault-Tolerant Consensus When There Are No Faults – A Tutorial*, SIGACT News 32(2), Distributed Computing column, pp. 45-63, junho,2001.

[15] Lamport, L., *The Part Time Parliament*, ACM Transactions on Computer Systems, 16(2), pp. 133-169, maio, 1998.

[16] Mostefaoui, A. e Raynal, M., *Solving Consensus Using Chandra-Toueg's Unreliable Failure detectors: a General Quorum-Based Approach*, in Proceedings of the 13th International Symposium on Distributed Computing (Disc1999), pp. 49-63, setembro, 1999.

[17] Mostefaoui, A., Raynal, M. e Tronel, F., *The Best of Both Worlds: a Hybrid Approach to Solve Consensus*, Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000), pp. 513-522, junho, 2000.

[18] Nahrstedt, K. e Smith, J. M., *The QoS Broker*, IEEE Multimedia, 2(1), pp – 53-67, março, 1995.

[19] Veríssimo, P., Casimiro, A. e Fetzer, C., *The Timely Computing Base: Timely Actions in the Presence of Uncertain Timeliness*, Proceedings of the International Conference on Dependable Systems and Networks, pp. 533-542, junho, 2000.

[20] Xiao, X. e Ni, L. M., "Internet QoS: A Big Picture", IEEE Network, pp. 8 – 18, março/abril, 1999.