Suporte à Execução Replicada de Serviços Orientados a Prioridade

Marcelo Jorge Aragão[1], **Francisco Brasileiro**[1,2]

[1] Universidade Federal de Campina Grande
[1]Coordenação de Pós-Graduação em Engenharia Elétrica
[2]Coordenação de Pós-Graduação em Informática
Av. Aprígio Veloso, 882, Campina Grande, Paraíba, 58.109-970, Brazil
Tel: (+55) 83 310 1433 Fax: (+55) 83 310 1365

aragao@dee.ufcg.edu.br fubica@dsc.ufcg.edu.br

Abstract. *This paper presents the design and implementation of a service that suports the construction of fault tolerant e-commerce applications that use an adaptive priority-based policy for resource allocation. Active replication is implemented to tolerate failures and* Group Priority Inversions *are avoided.*

Resumo. *Esse artigo apresenta o projeto e a implementação de um* framework *de suporte à construção de aplicações de comércio eletrônico tolerantes a falhas, que utilizam políticas adaptativas de alocação de recursos baseadas em prioridade. A técnica de replicação ativa é implementada para tolerar falhas e o problema de inversão de prioridade em grupo é tratado.*

1. Introdução

Alta disponibilidade é extremamente importante para um número crescente de aplicações distribuídas. Por exemplo, em julho de 1999, o site de leilões virtuais *eBay* ficou cerca de 22 horas fora do ar por problemas de infra-estrutura, o que representou uma perda no seu faturamento estimada entre US$ 3 milhões e US$ 5 milhões [3].

A indisponibilidade de serviços é normalmente causada pela escassez de recursos ou pela ocorrência de falhas. A *superprovisão* [7] e a *alocação de recursos baseada em prioridade* [1, 9, 10] são estratégias utilizadas para contornar o problema de falta de disponibilidade devido à escassez de recursos. A *superprovisão* consiste em estimar a maior carga que pode ser submetida a um sistema e, com base nessa estimativa, dimensionar os recursos do sistema de modo a suportar essa carga. Esse método é caro, pois o sistema pode permanecer a maior parte do tempo submetido a uma baixa carga, subutilizando os recursos dimensionados. Além disso, é difícil identificar corretamente essas estimativas, pois, muitas vezes, a carga submetida a um sistema é imprevisível.

A *alocação de recursos baseada em prioridade* permite que o sistema priorize o acesso e o uso de recursos conforme métricas que objetivam ganhos computacionais [1] ou ganhos financeiros [9, 10]. Essa é uma estratégia mais econômica que a *superprovisão*, pois a quantidade de recursos permanece inalterada. Um estudo realizado em [9, 10] demonstrou a viabilidade de políticas adaptativas de alocação de recursos para *sites* de comércio eletrônico. As políticas analisadas estabelecem prioridades baseadas no perfil do consumidor (freqüente ou ocasional), na duração da sessão e no valor total de compras acumuladas até o momento. O trabalho conclui em seus resultados que há um potencial de ganho em faturamento de até 29% sobre sistemas convencionais

nos momentos de pico [10]. No entanto, essa solução traz consigo um problema encontrado em sistemas com escalonamento baseado em prioridade. Esse problema é conhecido na literatura como problema de *inversão de prioridade* [14].

No nosso contexto, o problema de inversão de prioridade ocorre quando o processamento de uma requisição de alta prioridade é atrasado pelo processamento de uma requisição de baixa prioridade. O tratamento desse problema exige que uma requisição em execução seja suspensa ou abortada, para que o processamento da requisição de mais alta prioridade possa ser iniciado. Esse tratamento é normalmente realizado por um componente de software denominado escalonador de requisições. Dependendo do modo que o processamento da requisição é interrompido, este poderá continuar ou reiniciar sua execução após a conclusão do processamento da requisição de mais alta prioridade.

Além da escassez de recursos, a indisponibilidade de serviços pode também ser causada pela ocorrência de falhas. Falhas são inevitáveis, mas suas conseqüências, ou seja, o colapso do sistema, a interrupção no fornecimento do serviço ou a perda de dados, podem ser evitadas quando técnicas de tolerância a falhas são usadas de forma adequada. As técnicas de tolerância a falhas caracterizam-se pela introdução de redundância de componentes (hardware e/ou software) [8]. Redundância é normalmente introduzida pela replicação de componentes ou serviços [2, 13]. Na replicação, se uma das réplicas não está operacional, outra réplica garante que um determinado serviço seja oferecido. No entanto, replicação requer protocolos que assegurem consistência de estado entre as réplicas. Problemas de inconsistência podem acontecer devido à concorrência de operações de atualização do estado das réplicas (quando dois ou mais clientes concorrentes fazem diferentes requisições ao serviço replicado) ou devido a falhas em nodos.

Duas estratégias bastante utilizadas para se obter redundância são as técnicas de replicação passiva e ativa. Na técnica de replicação passiva, também chamada de *primary/backup* [2], existe uma réplica primária e uma ou mais secundárias. A réplica primária está sempre em execução, pronta para o processamento de requisições, e tem seu estado interno periodicamente salvo em memória estável, como por exemplo em disco. As réplicas secundárias, por sua vez, permanecem inativas e, periodicamente ou na ocorrência de falha, uma delas é promovida a primária e tem seu estado interno atualizado para o último estado interno salvo pela réplica primária anterior. Em casos de falhas, isto significa um retrocesso no estado do sistema, pois as ações realizadas pela réplica primária depois do último ponto de salvaguarda serão executadas novamente. Apesar de possibilitar a continuidade do serviço, o atraso existente nessa técnica pode não ser aceitável em alguns sistemas.

A técnica de replicação ativa [13] é uma técnica de replicação não centralizada em que todas as réplicas de um componente recebem e executam independentemente a mesma seqüência de requisições enviada por seus clientes. Desta forma, se uma réplica falhar, as outras réplicas produzirão as mesmas respostas requeridas sem o atraso de recuperação de estado existente na replicação passiva. A consistência entre as réplicas é garantida desde que as réplicas recebam as mesmas requisições na mesma ordem e produzam as mesmas saídas. Para implementar essa técnica são necessários protocolos que garantam os requisitos de *ordem* e de *acordo*, como especificado em [13]. A principal vantagem dessa técnica está em oferecer tempos de resposta aceitáveis em caso de falhas, em contra partida, essa técnica exige protocolos mais complexos.

Em resumo, o uso de prioridades, aliado à técnica de replicação ativa, pode oferecer condições a muitas aplicações para o provimento de um serviço de alta disponibilidade. Entretanto, a união dessas duas estratégias exige um tratamento especial quanto ao problema de inversão de prioridade em um processamento ativamente replicado. No contexto replicado, o tratamento é bem mais complexo que no contexto não replicado. Considerando que a chegada e o processamento de requisições ocorrem assíncronamente nas réplicas, uma réplica pode não observar os mesmos

casos de inversão de prioridade que outra. Verifica-se nesse contexto que os casos de inversão de prioridade não poderão ser tratados isoladamente nas réplicas, caso contrário, as requisições poderão ser processadas em ordem diferentes e inconsistências poderão ocorrer nos resultados das mesmas.

A solução do problema de inversão de prioridades no contexto de um grupo de processos realizando um processamento ativamente replicado foi proposta em [15]. O conceito de *inversão de prioridade em grupo* foi introduzido pela primeira vez no mesmo trabalho. Informalmente, uma *inversão de prioridade em grupo* ocorre quando casos de inversão de prioridade (*local*) são detectados em muitas réplicas.

Nesse artigo nós apresentamos o projeto e a implementação de um *framework* para suportar o desenvolvimento de aplicações de comércio eletrônico tolerantes a falhas que utilizam políticas adaptativas de alocação de recursos baseadas em prioridade. A técnica de replicação ativa é implementada para tolerar falhas. O *framework* soluciona o problema de inversão de prioridade em grupo e se baseia no estudo realizado em [15]. Mais especificamente, no nosso contexto o *framework* é uma camada de software que possibilita a comunicação entre aplicações clientes e aplicações servidoras (servidor de aplicação e banco de dados). Esse *framework* deve possuir um protocolo escalonador que permita o processamento ativamente replicado de requisições segundo as prioridade dos usuários do sistema.

O restante do artigo está organizado da seguinte forma. A Seção 2 apresenta o modelo de sistema considerado e define o problema de inversão de prioridade em grupo. A Seção 3 apresenta a arquitetura do *framework*, enquanto que a Seção 4 discute aspectos da utilização do mesmo pelas aplicações. Na Seção 5 é feita uma avaliação de desempenho do protótipo implementado. A Seção 6 conclui o artigo com os nossos comentários finais.

2. Modelo do Sistema e Definições

2.1. Modelo do Sistema

Consideramos o modelo de sistema distribuído assíncrono, onde não há limites conhecidos nos tempos de transmissão de mensagens nem nas velocidades de processos. Um sistema distribuído é formado por diversos nodos autônomos que são conectados por uma rede de comunicação. Os nodos não possuem memória compartilhada e se comunicam por troca de mensagens [8]. Adotamos o modelo de falha em que os processos podem falhar por parada [8]. Além disso assumimos que o sistema é equipado com *detectores de falhas não confiáveis* da classe $\diamond S$, conforme definidos em [5].

2.2. Inversão de Prioridade em um Grupo de Réplicas

O escalonamento de requisições orientado a prioridade define a ordem em que um conjunto de requisições é processado. Esse tipo de escalonamento é ilustrado na Figura 1.

Na Figura 1 é apresentado um escalonador com 2 listas (*Lista 1 e Lista 2*). Cada lista armazena requisições que possuem uma determinada prioridade. As requisições são representadas por $\mathbf{X_n}$, onde *n* representa um identificador único da requisição, e suas prioridades são representadas por um retângulo que, dependendo da prioridade da requisição, podem ser cinza (*prioridade 2*) ou branco (*prioridade 1*). O armazenamento de novas requisições em cada lista segue uma ordem FIFO segundo suas prioridades. O processamento ocorre da seguinte maneira: o escalonador, representado por linhas tracejadas, escolhe a lista de maior prioridade e despacha a primeira requisição da direita para esquerda da lista. Quando não houver mais requisições na lista, o

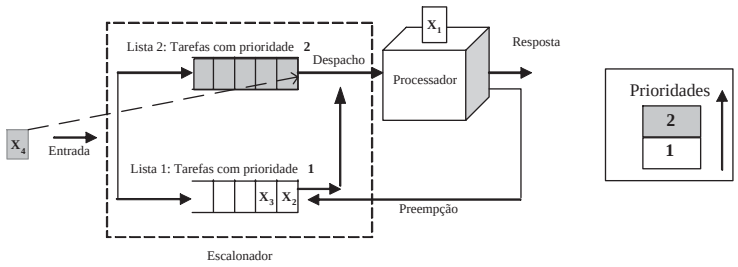

Figura 1: Inversão de prioridade

escalonador passa a escolher as requisições da lista seguinte por ordem de prioridade, e assim por diante.

Quando a entrada de requisições no escalonador ocorre dinamicamente, pode acontecer de uma requisição de prioridade alta ficar esperando a conclusão do processamento de uma outra requisição de prioridade mais baixa. No exemplo da Figura 1 vê-se que o processador está ocupado com requisição $\mathbf{X_1}$ de prioridade ***1*** quando a requisição $\mathbf{X_4}$ de prioridade ***2*** é recebida. Como *Prioridade* ($\mathbf{X_1}$) < *Prioridade* ($\mathbf{X_4}$), observa-se que o processamento da requisição $\mathbf{X_1}$ está atrasando o processamento da requisição $\mathbf{X_4}$, evidenciando portanto, um caso de inversão de prioridade.

Para tratar esse tipo de problema, é preciso um mecanismo capaz de detectar as ocorrências de inversão de prioridade, de forma a suspender ou abortar o processamento de uma requisição de baixa prioridade para dar lugar ao processamento de uma requisição de alta prioridade. Uma requisição que teve seu processamento interrompido deve ser reordenada e ter seu processamento reiniciado a partir do último estado anteriormente salvo. Seu reinício irá depender de sua posição em uma das listas de requisições pendentes.

O escalonamento em um sistema com processamento ativamente replicado exige que todas as réplicas possuam uma mesma visão sobre um conjunto de requisições. No escalonamento orientado a prioridade, essa lista precisa além de respeitar as prioridades das requisições, estar ordenada da mesma maneira em todas as réplicas. No modelo considerado não existe sincronização entre os instantes de tempo em que as requisições são recebidas, tornadas prontas para processamento ou processadas nas diferentes réplicas. Desse modo, uma réplica não pode considerar casos de inversão de prioridade local sem antes analisar o estado global das outras réplicas, pois os mesmos casos de inversão de prioridade podem não ocorrer em todas as réplicas.

A Figura 2 ilustra o problema de inversão de prioridade em um sistema com processamento ativamente replicado. Duas réplicas processam independentemente suas requisições utilizando os mesmos critérios explanados na Figura 1. No momento que a requisição $\mathbf{X_4}$ de prioridade ***2*** é escalonada na lista *2* de ambas as réplicas, o processador da réplica *B* encontra-se ocupado com o processamento de uma requisição que já foi concluído na réplica *A* (requisição $\mathbf{X_1}$ de prioridade ***1***). Embora as duas réplicas agora possuam as mesmas listas de requisições, elas não observam os mesmos casos de inversão de prioridade. A requisição $\mathbf{X_1}$ em processamento na réplica *B* possui prioridade menor que a nova requisição $\mathbf{X_4}$, ou seja, *Prioridade* ($\mathbf{X_1}$) < *Prioridade* ($\mathbf{X_4}$). Por sua vez, a réplica *A* encontra-se com o processador livre para processamento. Assim, o problema de inversão de prioridade é identificado somente na réplica *B*.

Para evitar e tratar esse problema no contexto de grupo, existem duas alternativas: realizar um retrocesso (*rollback*) da requisição já completada pela réplica *A* e em seguida reordenar as listas nas duas réplicas; ou ignorar o caso de inversão de prioridade na réplica *B* baseando-se em algumas premissas nas réplicas.

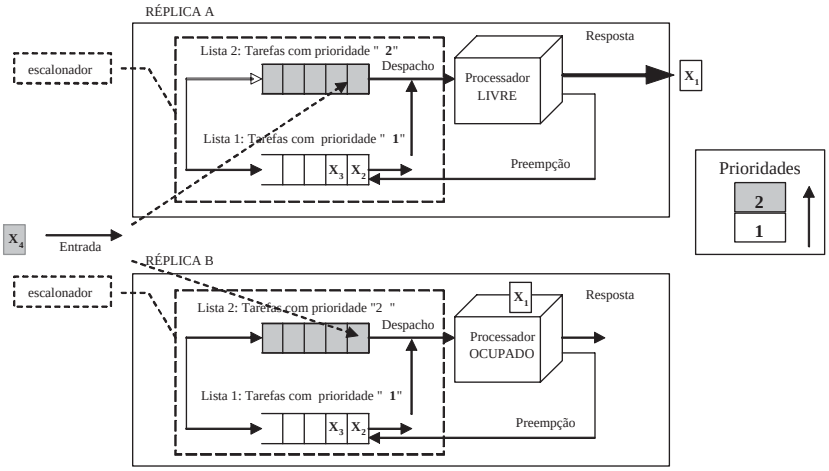

Figura 2: Inversão de prioridade em grupo

A escolha da alternativa mais apropriada irá depender de um referencial em comum entre as réplicas (*Group Execution Progress - GEP*). Esse referencial definirá como as novas requisições deverão ser ordenadas e será usado para decidir se uma requisição deverá ou não sofrer retrocesso. GEP é calculado a partir do progresso de execução individual de cada réplica (*Local Execution Progress - LEP*), ou seja, da última requisição já processada por uma determinada réplica.

O escalonamento de requisições pode seguir uma abordagem pessimista ou otimista. Na abordagem pessimista, mesmo que não ocorra casos de inversão de prioridade, ocorre um bloqueio na lista de requisições até que o acordo (ordenação total ou cálculo de GEP) seja concluído. Dessa forma mesmo que um processador esteja livre, ele precisa esperar pelo fim do acordo. Dessa forma retrocessos jamais ocorrem. Na abordagem otimista, é possível realizar antecipadamente algum processamento, antes de se obter o resultado final do escalonamento. Contudo o processamento de uma requisição poderá ser eventualmente desfeito em casos de inversão de prioridade. Como conseqüência, para garantir consistência entre as réplicas, os estados e as respostas das requisições sujeitas a retrocesso devem ser armazenados para serem entregues aos respectivos clientes quando o processamento das requisições não puder ser mais desfeitos.

O conceito de inversão de prioridade em grupo (*Group Priority Invertion - GPI*) foi introduzido pela primeira vez em [15], onde o problema de inversão de prioridade em uma única réplica (*Local Priority Inversion - LPI*) foi estendido a um grupo de processadores que realizam um processamento ativamente replicado. Em [15] foi proposto um protocolo escalonador de requisição, orientado a prioridade, que garante ordenação total, evita inversão de prioridade em grupo e segue uma abordagem otimista. Uma inversão de prioridade em grupo irá ocorrer sempre que o processamento de uma requisição replicada de alta prioridade for atrasado por atividades de menor prioridade. Assumindo um modelo de falhas por parada, o processamento de uma requisição replicada requer que no mínimo um processo gere uma resposta válida. Isso quer dizer que uma requisição não sofrerá inversão de prioridade em grupo, se for possível garantir que pelo menos uma resposta a essa requisição será gerada sem qualquer atraso.

Para o cálculo de GEP em [15], as réplicas executam um protocolo de acordo onde cada réplica propõe um valor EP (*Execution Progress*), onde EP = max(LEP, GEP). Esse protocolo utiliza uma função que é aplicada a todos os EPs obtidos. Considerando *f*, como o número máximo de réplicas que podem falhar sem afetar a disponibilidade do serviço, essa função é calculada em dois passos: primeiro, o subconjunto contendo os $f + 1$ maiores valores é determinado. Em seguida, o valor de GEP é obtido, extraindo o menor valor contido no conjunto de $f + 1$ valores. A seleção dos $f +$

1 valores oferece um avanço mais rápido de GEP e a seleção do menor valor contido no conjunto $f + 1$ garante que, no mínimo, um desses valores foi fornecido pelo módulo escalonador de uma réplica que não falhará. Dessa forma, o valor de GEP assegura que o envio de respostas aos clientes só ocorrerá quando LEP $\leq$ GEP. Assim, mesmo que ocorra uma falha em uma réplica, os clientes jamais receberão respostas inconsistentes de uma mesma requisição.

3. Arquitetura do *framework*

3.1. Estrutura Interna

A arquitetura do EROPSAR (**E**scalonador de **R**equisição **O**rientado a **P**rioridade para **S**erviços **A**tivamente **R**eplicados) é formada por 5 componentes identificados na Figura 3 pelos retângulos em cinza. Descrevemos como eles interagem uns com os outros para acessar um recurso remoto a partir de uma requisição enviada por uma aplicação cliente.

Uma *aplicação cliente* envia suas requisições e recebe seus resultados através do componente *interceptador*[1]. Antes do envio da requisição ao *interceptador*, a requisição deverá conter as seguintes informações: prioridade, recurso remoto, operações e parâmetros. O componente *interceptador* reside no lado cliente da aplicação e recebe a requisição cliente e a difunde nas réplicas de um servidor replicado (a Figura 3 ilustra apenas uma réplica, contudo a mesma estrutura se aplica às outras réplicas com setas partindo do mesmo *interceptador*). Após receber o resultado do processamento, o *interceptador* envia a primeira resposta de uma requisição ao cliente e descarta as respostas duplicadas que possam ser recebidas.

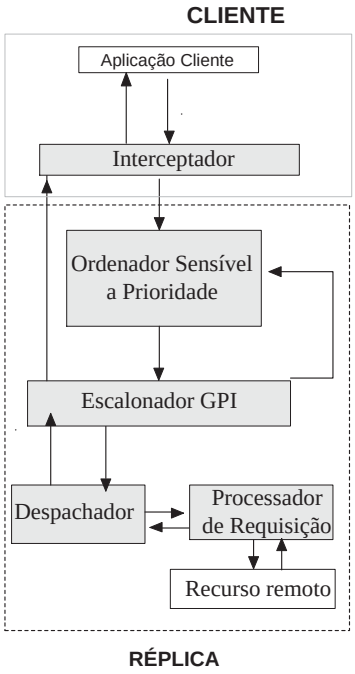

Figura 3: Modelo Estrutural do EROPSAR

O componente *Ordenador Sensível a Prioridade* reside no lado servidor da aplicação e recebe as requisições dos *interceptadores* associados aos clientes e as armazena localmente. Esse componente garante que todos os outros componentes ordenadores não falhos observem a mesma

[1]Esse componente baseia-se no *padrão arquitetural interceptador* (*Interceptor Architectural Pattern*) [12]. Esse padrão permite a adição de serviços a determinado *framework*, de forma transparente, tornando-o extensível.

seqüência de requisições. Ao receber requisições, esse componente decodifica as prioridades das requisições a partir do identificador do cliente e as ordena de acordo com uma prioridade associada no servidor. Uma vez decodificadas e reordenadas, as requisições são enviadas ao *escalonador GPI*.

O componente *Escalonador GPI* recebe do *Ordenador Sensível a Prioridade* as requisições ordenadas e as escalona de modo a evitar o problema de inversão de prioridade em grupo. Esses componentes trabalham em estrita cooperação para tratar esse problema.

O componente *despachador* recebe do *escalonador GPI* a requisição pronta de maior prioridade. Uma vez recebida, ele localiza o *processador de requisição* da aplicação e invoca um procedimento padrão no *processador de requisição* que deverá interpretar o conteúdo da requisição. O *despachador* salva o estado do *processador de requisição* antes do processamento. Esse estado poderá ser utilizado se eventualmente o processamento for interrompido pelo *escalonador GPI* quando uma inversão de prioridade em grupo for detectada. Se esse for o caso, o estado do *processador de requisição* deve ser recuperado após interrupções e eventuais retrocessos. Dessa forma uma requisição que foi interrompida será novamente iniciada a partir de um estado que antecedeu sua interrupção.

A primeira ação do componente *processador de requisição* é localizar o recurso remoto contido na requisição para então invocar as operações contidas na requisição. Em seguida o *recurso remoto* informado na requisição é acessado e realiza a operação solicitada, retornando o resultado ao *processador de requisições*. Por fim esse resultado é retornado ao *despachador*, que por sua vez envia o resultado juntamente com o estado salvo para o *escalonador GPI*.

Conforme colocado anteriormente, uma vez que o identificador do cliente é utilizado como chave para consultar a prioridade de uma requisição no servidor, assumimos que o EROPSAR utiliza o modelo de política de prioridade *client priority propagation* (prioridade propagada pelo cliente) da especificação RT-CORBA [11]. Nesse modelo de política de prioridade, a requisição é executada na prioridade requisitada pelo cliente e é codificada como parte da requisição do cliente. A atualização da prioridade no lado servidor se baseia na interface *PriorityTransform* de RT-CORBA.

3.2. Serviços de Suporte

Os serviços de suporte oferecem um nível de abstração que facilita a interação com o grupo de réplicas. Eles adicionam uma camada de distribuição ao *EROPSAR*, oferecendo acesso a recursos remotos, visão de grupo e primitivas como difusão. Esses serviços funcionam como blocos básicos na construção do protocolo de replicação ativa e do protocolo de acordo (ordenação total e valor do progresso de execução do grupo) necessários no tratamento do problema de inversão de prioridade em grupo. Descrevemos a seguir os serviços de suporte necessários para o *EROPSAR*.

O *EROPSAR* utiliza um *serviço de nomes* para registrar os recursos que o servidor deseja disponibilizar a um cliente. O cliente, por sua vez, adiciona em uma requisição o nome do recurso e a operação desejada com seus respectivos parâmetros, para que possa ser enviada ao servidor. Quando a requisição for despachada, ela deverá ser interpretada de modo a obter o nome do recurso e a operação. Em seguida, de posse dessas informações, o recurso poderá ser acessado através de uma referência obtida a partir de um *serviço de nomes*.

O projeto do *EROPSAR* se baseia em um *serviço de acordo* que precisa realizar a ordenação total de requisições e precisa calcular o *progresso de execução do grupo* para que casos de inversão de prioridade em grupo sejam detectados e tratados. Como chegada de requisições no servidor dá-se de forma contínua, o *serviço de acordo* deve permitir que várias propostas de requisições sejam realizadas mesmo após um consenso ter sido iniciado. Desse modo, muitas requisições podem ser

ordenadas em uma única execução do protocolo. Para que o valor do *progresso de execução do grupo* seja calculado, o *serviço de acordo* deverá prover um protocolo capaz de decidir um valor que é o resultado de uma dada função aplicada aos diversos valores de entrada. Esses dois acordos estão diretamente relacionados para garantir consistência nas réplicas. Devido a essa relação, se o *serviço de acordo* permitir que dois acordos sejam realizados em conjunto (em uma única execução do protocolo de consenso), o serviço poderá ser mais eficiente que se os acordos fossem executados um após o outro.

Para evitar que os processos envolvidos em um acordo fiquem indefinidamente à espera das mensagens de um outro que está muito lento ou que falhou, impossibilitando a decisão do acordo [6], o serviço de acordo é baseado em detectores de falha não confiáveis [5]. Esses detectores fornecem informações de todos os processos suspeitos de terem sofrido uma falha. [5] define várias classes de detectores, todas elas especificadas por duas propriedades básicas: abrangência (*completeness*): que define em que situações os componentes falhos serão detectados; e exatidão (*accuracy*): que limita os erros que o detector pode fazer, ou seja, limita as falsas suspeitas dos processos que não falharam. Suspeitas são implementadas utilizando-se mecanismos de temporizarão (*timeout*) assim: *i)* a detecção de uma falha real pode ficar retardada, e *ii)* um detector de falhas pode cometer erros suspeitando incorretamente de um processo que não falhou.

Dentre as várias classes de detectores, a classe chamada de $\diamond S$ é muito atrativa. Essa é a classe mais fraca de detectores de falhas que permite a solução determinística do problema de consenso em sistemas assíncronos [4], desde que a maioria dos processos não falhem. O detector $\diamond S$ tem as seguintes propriedades:

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

Estas propriedades garantem que, após um determinado intervalo de tempo (intervalo finito, mas desconhecido), todos processos falhos serão considerados suspeitos, mas ao menos um processo correto não será suspeito por nenhum detector. O *serviço de detecção de falha* do *EROPSAR* utiliza essa classe de detector.

4. Desenvolvendo Aplicações Utilizando o *Framework* EROPSAR

O EROPSAR disponibiliza três classes às aplicações clientes e servidores: *RSM* implementa o módulo escalonador de requisição da camada de requisição e resposta, *Request* contém métodos e propriedades úteis para a configuração da requisição e *ClientInterceptador* permite a comunicação transparente entre cliente e servidor replicado através da interface *ClientInterceptador_interface*

Para utilizar o EROPSAR é preciso instanciar essas três classes e implementar as interfaces *PriorityTransform_interface* e *Processor_interface*, descritas anteriormente.

4.1. Configurando uma Aplicação Cliente

Os seguintes passos devem ser seguidos na construção de uma aplicação cliente:

1. Localizar a referência de um dos objetos *ClientInterceptor* disponibilizados remotamente através da interface *ClientInterceptador_interface*.
2. Criar uma instância da classe *Request*
   (a) Request request = new Request();
3. Preencher as propriedades exigidas de uma requisição:

(a) Prioridade da requisição. Ex:
    i. request.priority = ”5”;
(b) nome da interface contendo o método desejado. Ex:
    i. request.objInterfaceName = ”ObjectImpl_interface”;
(c) nome do método remoto desejado.
    i. request.objTask = ”SUM”;
(d) definir se a invocação deverá ser síncrona ou assíncrona. Ex:
    i. request.synchronous =true;
(e) definir os parâmetros de entrada e saída do método invoca. A classe *request* permite a passagem de parâmetros do tipo: *int*, *String* e *Object*. Para definir um parâmetro é preciso especificar o *nome*, o *valor* e a *direção* (parâmetro de entrada ou parâmetro de saída). Ex:
    i. request.addIntParam(”SAIDA_1”,0,request.OUT);
    ii. request.addIntParam(”Parcela1”,5,request.IN);
    iii. request.addIntParam(”Parcela2”,6,request.IN);
    iv. request.addIntParam(”SimulatedDelay”,2000,request.IN);

4. Enviar requisição para o escalonador e receber o resultado do processamento.
   (a) request = clientInterceptor.schedule(request, InvocationTimeout);
   (b) Receber o resultado da requisição.
   Se a invocação for síncrona (*request.synchronous=true*), o método retornará assim que o servidor tiver executado a requisição. O resultado da requisição poderá ser obtido de duas maneiras: *replyRequest.Reply*, para métodos com um único valor de saída e *request.getParam(”Saída_1”)*, para métodos com 1 ou mais valores de saída.
   Se a invocação for assíncrona (*request.synchronous=false*), o método retornará imediatamente após o recebimento da requisição pelas réplicas. O resultado da requisição poderá ser verificado através de *clientInterceptor.requestsPoll* que contém um *Vector* com as últimas requisições processadas e em processamento ou através de *clientInterceptor.getReply(request.ID,pTimeout)*. Quando a requisição não tiver sido ainda processada, *clientInterceptor.getReply(request.ID,pTimeout)*, retorna após *pTimeout* milisegundos, caso contrário, retorna imediatamente com o resultado do processamento.

4.2. Configurando uma Aplicação Servidor

O seguintes passos devem ser seguidos na construção de uma aplicação servidor:

1. Criar uma instância do módulo escalonador de requisição (RSM) enviando como parâmetro as propriedades do serviço (ilustrado na Figura 4 com a descrição de cada parâmetro). O parâmetro *args* contém essas propriedades.
   (a) RSM rsm = new RSM(args);
2. Criar uma instância da implementação das interfaces do *EROPSAR* e também as interfaces dos objetos da aplicação que serão acessadas remotamente.
   (a) ClientInterceptor_Impl clientInterceptor_Impl = new ClientInterceptor_Impl();
   (b) PriorityTransform_Impl priorityTransform_Impl = new PriorityTransform_Impl();
   (c) Processor_Impl processor_Impl = new Processor_Impl(sHostname, sRMIPort, sServerPort);
   (d) Criação dos objetos específicos da aplicação que devem ser acessados remotamente;
3. Registrar a referência do objeto implementado no serviço de nomes

```
EROPSAR.REGISTRY_PORT=10999                # Porta de escuta do serviço de nomes
EROPSAR.DEBUG_LEVEL=6                      # Nível de depuração para escrita no arquivo de log
EROPSAR.ReplicaDelayBeforeProcessing=0     # Gera um atraso de 0ms no processamento de todas as
                                             requisições. Útil para simular uma réplica lenta

# Política de Escalonamento exigida na aplicação, ex: EDF, FIFO, ...
EROPSAR.SchedulePolicy=CLIENT_ID_PRIORITY_MAPPING

EROPSAR.E_GAF_UNIT.NumberOfProcess=2       # Grau de replicação do servidor replicado
EROPSAR.E_GAF_UNIT.Process1=robalo:7000    # Nome e porta de escuta da réplica 1
EROPSAR.E_GAF_UNIT.Process2=traira:7100    # Nome e porta de escuta da réplica 2
EROPSAR.E_GAF_UNIT.PortToListen=7200       # Porta de escuta dessa réplica

EROPSAR.E_GAF_UNIT.FW.NumberOfProcess=2    # Número de processos com serviço de acordo
EROPSAR.E_GAF_UNIT.FW.Process1=robalo:9000 # Nome e porta de escuta do serviço de acordo na réplica 1
EROPSAR.E_GAF_UNIT.FW.Process2=traira:9100 # Nome e porta de escuta do serviço de acordo na réplica 2
EROPSAR.E_GAF_UNIT.FW.PortToListen=9200    # Porta de escuta do serviço de acordo dessa réplica

FD.NumberOfProcess=4                       # Número de detectores de falhas
FD.Process1=robalo:8500                    # Detector de falha 1
FD.Process2=traira:8600                    # Detector de falha 2
FD.PortToListen=8700                       # Porta de escuta do desse DETECTOR
FD.DELTA_DETECTOR_TIMEOUT=9000             # Intervalo inicial em ms p/ que um
                                             processo monitorado seja suspeito
                                             caso não envie "EuEstouOperante"
FD.DELTA_HEARTBEAT=3000                    # Intervalo em ms p/ que HEARTBEAT envie "EuEstouOperante"
```

Figura 4: Propriedades para configuração do serviço - Lado Servidor

(a) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/ClientInterceptor_interface_"+ sServerPort , clientInterceptor_ Impl);
(b) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/PriorityTransform_interface_"+ sServerPort , priorityTransform_ Impl);
(c) Naming.rebind("//"+ sHostname + ":"+ sRMIPort
+ "/Processor_interface_"+ sServerPort , processor_ Impl);

5. Avaliação de Desempenho

Todo o projeto foi implementado em JAVA (JDK 1.4.1) e o ambiente utilizado para desenvolvimento foi o JBuilder Personal 7.0. Os testes foram realizados em máquinas PC's 1.2GHz com 256Mb de RAM, com sistemas operacionais Windows XP e Linux (SuSE 8.0) conectados por uma rede Ethernet 10Mbps.

Para avaliar o desempenho do *framework*, armazenamos em um arquivo de log as seguintes marcas de tempo: do envio, da recepção no servidor, do escalonamento, do despacho, do término de processamento e do envio do resultado ao cliente. Essas marcas são utilizadas por dois experimentos. O primeiro experimento, *RoundTrip*, calcula o tempo que o *EROPSAR* leva para enviar uma requisição do cliente a um grupo de réplicas e receber o resultado do processamento de um dos membros do grupo replicado. O segundo experimento calcula o tempo de duração das operações de envio, ordenação total, despacho, processamento e retorno do resultado ao cliente.

O grau de replicação (GR) variou de 3 a 5. Em cada um dos graus de replicação utilizados foram enviadas cem requisições. Esse experimento teve como objetivo detectar a sobrecarga causada a medida que o sistema incrementa seu grau de replicação.

O gráfico da Figura 5 mostra a média obtida na medição do *RoundTrip* para cada grau de replicação utilizado. Observamos com as medições que à medida que o grau de replicação aumenta, o tempo de *RoundTrip* também aumenta.

O gráfico da Figura 6 mostra a média obtida na medição das operações de transição de estado da requisição. Observamos com as medições que as operações de envio, despacho e processamento se mantiveram constantes, independente do grau de replicação. Nas operações de ordenação, à

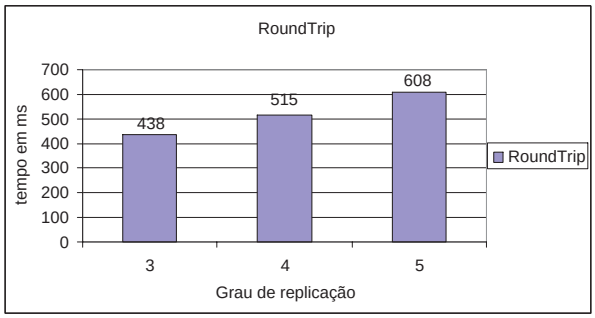

Figura 5: *RoundTrip* no *EROPSAR*

medida que o grau de replicação aumenta, mais tempo é exigido para que a operação seja concluída. Isso ocorre porque essa operação exige a realização de um protocolo de acordo que tende a ser mais lento à medida que mais processos estão envolvidos em um acordo.

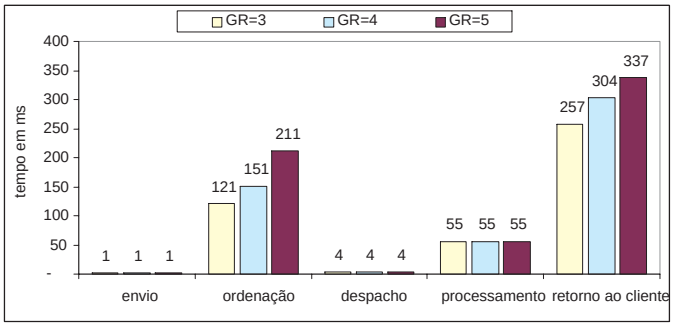

Figura 6: Duração das operações de transição de estado

A operação de retorno também possui uma tendência crescente à medida que o grau de replicação aumenta. Um outro fator que determina essa tendência crescente está relacionado às requisições completadas não entregues por ainda não serem estáveis. Esse fator depende da velocidade de execução das réplicas, portanto do valor de GEP.

6. Conclusões

Nesse trabalho projetamos e implementamos um *framework*, denominado *EROPSAR*, que viabiliza a construção de aplicações com alta disponibilidade de serviço. A alta disponibilidade é alcançada através da técnica de replicação ativa [13] associada a uma política de alocação de recursos baseada em prioridade [9, 10]. Nesse contexto, o EROPSAR contém protocolos que possibilitam: o *compartilhamento de recursos com tratamento de inversão de prioridade*, a *redundância de componentes* e a *comunicação cliente-servidor de modo síncrono e assíncrono*.

Os benefícios oferecidos no EROPSAR para prover alta disponibilidade possuem um custo: quanto mais réplicas forem utilizadas para tolerar falhas mais lento será o escalonamento de requisições. Esse atraso é causado pela necessidade das réplicas realizarem os acordos para definir uma ordem comum de processamento. Um outro custo está relacionado ao uso de prioridades no processamento ativamente replicado, pois como casos de inversão de prioridade em grupo precisam ser tratados, eventuais retrocessos podem ser necessários para garantir a consistência entre as réplicas.

Agradecimentos

Este trabalho foi parcialmente financiado pelo CNPq (processo 300646/1996-8).

Referências

[1] ALMEIDA, J., DABU, M., MANIKUTTY, A., AND CAO, P. Providing differentiated levels of service in web content hosting. In *First Workshop on Internet Server Performance* (Junho 1998), ACM.

[2] BUDHIRAJA, N., MARZULLO, K., SCHNEIDER, F., AND TOUEG, S. *Distributed Systems*, s. mullender ed. Addison Wesley, 1993, ch. 8: The Primary-Backup Approach, pp. 199–216.

[3] CARR, D. F. Don't get spiked. *Internet World 5*, 34 (Janeiro 1999), 59. Disponível por www em http://www.acm.org/technews/articles/1999-1/1208w.html (5 de novembro de 2003).

[4] CHANDRA, T., HADZILACOS, V., AND TOUEG, S. The weakest failure detector for solving consensus. *Journal of the ACM 43*, 4 (Julho 1996), 685–722.

[5] CHANDRA, T., AND TOUEG, S. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM 43*, 2 (1996), 225–267.

[6] FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. Impossibility of distributed consensus with one faulty process. *J. ACM 32*, 2 (1985), 374–382.

[7] GRIBBLE, S. Robustness in complex systems. In *Proceedings of the Eighth International Symposium on Hot Topics in Operating Systems (HotOS-VIII)* (2001).

[8] JALOTE, P. *Fault Tolerance in Distributed Systems*. Prentice Hall, NJ, 1994.

[9] MENASCÉ, D., ALMEIDA, V., FONSECA, R., AND MENDES, M. A. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM conference on Electronic commerce* (1999), ACM Press, pp. 119–128.

[10] MENASCÉ, D., ALMEIDA, V., FONSECA, R., AND MENDES, M. A. Resource management policies for e-commerce servers. In *Proceedings of the Second Workshop on Internet Server Performance* (Maio 1999).

[11] OMG. *Real-Time CORBA - Join Revised Submission*, document orbos/98-12-05 ed. Object Management Group, Dezembro 1998. URL: http://www.omg.org.

[12] SCHMIDT, D., STAL, M., ROHNERT, H., AND BUSCHMANN, F. *Pattern-Oriented Software Architecture*, vol. Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[13] SCHNEIDER, F. *Distributed Systems*, s. mullender ed. Addison Wesley, 1993, ch. 7: Replication Management using the State-Machine Approach, pp. 169–197.

[14] SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers 39*, 9 (1990), 1175–1185.

[15] WANG, Y., ANCEAUME, E., BRASILEIRO, F., GREVE, F., AND HURFIN, M. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers 51*, 8 (2002), 900–915.