Projeto e Implementação de um Suporte para a Execução de Serviços Replicados Orientados a Prioridade

Marcelo Jorge Aragão

Dissertação de Mestrado submetida à Coordenação dos Cursos de Pós-Gradução em Engenharia Elétrica da Universidade Federal de Campina Grande - Campus I como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

Área de Concentração: Processamento de Informação

Francisco Vilar Brasileiro, Dr.
Orientador

Campina Grande, Paraíba, Brasil

A659p 2003

Aragão Marcelo Jorge

Projeto e implementação de um suporte para a execução de serviços replicados orientados a prioridade/ Marcelo Jorge Aragão Campina Grande: UFCG, 2003

73 p.: il.

Inclui bibliografia

Dissertação (mestrado em Eng. Elétrica) - UFCG/CCT/DEE

1. Sistemas distribuidos 2.Tolerância a falhas
3. Inversão de prioridade 4. Protocolos de acordo
5.Ponte de Wheatstone

I. Título

CDU:681.3.069:621.3

PROJETO E IMPLEMENTAÇÃO DE UM SUPORTE PARA A EXECUÇÃO DE SERVIÇOS REPLICADOS ORIENTADOS A PRIORIDADE

MARCELO JORGE ARAGÃO

Dissertação Aprovada em 23.12.2003

Francisco Vilar Brasileiro

FRANCISCO VILAR BRASILEIRO, Dr., UFCG
Orientador

ANGELO PERKUSICH, D.Sc., UFCG
Componente da Banca

Walfredo da Costa Cirne Filho

WALFREDO DA COSTA CIRNE FILHO, Dr., UFCG
Componente da Banca

CAMPINA GRANDE - PB
Dezembro - 2003

Projeto e Implementação de um Suporte para a Execução de Serviços Replicados Orientados a Prioridade

Marcelo Jorge Aragão

Dissertação de Mestrado apresentada em dezembro de 2003

Francisco Vilar Brasileiro, Dr.
Orientador

Walfredo Cirne, Dr.
Componente da Banca

Angelo Perkusich, Dr.
Componente da Banca

Campina Grande, Paraíba, Brasil, dezembro de 2003

"Sou o poeta a se alçar às nuvens, o escritor a se devanear, o historiador a reviver o passado, o cronista a estilizar o cotidiano, o contista a dramatizar sem sofrer, o homem simples de poucos amigos, o filho agradecido que dos pais não esquece o desvelo, o esposo que não espera ficar viúvo, o pai que tem nos filhos o estuário retemperante do amor, o ente, enfim, que nas horas afáveis da vida transforma-se em criança e, no reverso das emoções, em que do mundo somente ingratidão recebe, readquire da caverna os extremos da ferocidade. Se, por enquanto, não me identifico, devo assim proceder por razões que se justificam. Ainda não nasci. Poder-me-iam inquinar de contraditório, visto como o verbo SER é preludiado, mas acima desse desejo está a cronologia dos fatos, regra sem a qual nenhum historiador conduzirá o seu raciocínio de acordo com o ordenamento estrutural dos acontecimentos."

Raimundo Batista Aragão (1921-2003)

Dedicatória

Aos meus avós, R. Batista Aragão, Maria Luzia Pereira, Domingos Jorge e Anastácia Melo. Embora que alguns não estejam mais fisicamente presentes, todos permanecem no coração.

Agradecimentos

A Deus, por me impulsionar e iluminar em todos os instantes de minha vida, me presenteando com força, perseverança, discernimento, equilíbrio e firmeza para enfrentar os desafios e atingir meu objetivos.

À Patricia, minha esposa, pelo apoio e carinho que sempre pude contar ao longo de meus intermináveis "*vou já...*".

Aos meus pais, Darlan e Fátima, irmãos, Cristiani & Colares, Marcio, Mônica, Lucas, Renato e Carleuda que apesar da distância sempre me apoiaram. Grato pelo carinho e amor. Como eu já dizia antes, somos um time inseparável.

A todos da minha família, tios, tias, primos, primas, sobrinhas, Sr.Robson, Dna Gilma e Kelly.

Um agradecimento especial ao meu orientador Fubica pela credibilidade depositada para meu ingresso no curso de mestrado, pela paciência, apoio, orientação, dedicação e pelo valioso aprendizado obtido.

Aos meus amigos Milton e Pedro, grandes amigos, que sempre contei com o apoio, incentivo e acima de tudo, prontidão. Grato pela co-orientação e pela amizade de dez anos.

Aos professores das disciplinas das quais cursei (Angelo, Walfredo, Fátima, Elmar, Berto) que ajudaram seja de forma específica no trabalho, seja com a ampliação dos meus conhecimentos na área, ou incentivando-me sempre que possível.

À KNAPP Systemintegration GmbH/KNAPP Sudamerica LTDA pela flexibilidade disponibilizada no trabalho e pelo suporte financeiro.

A todos os meus grandes amigos, de longas datas, que tive que me afastar por motivos profissionais e de estudo: Pedrão, Arthur, Eduardo, Charles, Alisson, Henrique, Sérgio, Marcos, Enildo, Camilo e Marcio Silveira.

A todos os que não constam nesta nota, mas que de alguma forma contribuíram para a realização deste trabalho.

Essa dissertação é um presente para todos vocês.

Resumo

A rápida disseminação de microcomputadores e estações de trabalho, o aumento nas suas capacidades de processamento, e o surgimento de redes de comunicação com grande largura de banda, têm levado ao desenvolvimento cada vez maior de aplicações distribuídas. Nesse contexto, muitas organizações passaram a depender exclusivamente de seus sistemas computacionais para o provimento de seus serviços. Essa dependência exige que os sistemas computacionais estejam sempre em funcionamento.

A indisponibilidade de serviços é normalmente causada pela escassez de recursos ou pela ocorrência de falhas. Uma maneira para evitar a escassez de recurso em momentos de pico, pode ser realizada através de uma *política de alocação de recursos baseada em prioridade*. A ocorrência de falhas é inevitável, contudo, suas conseqüências podem ser evitadas quando técnicas de tolerância a falhas são usadas. A união dessas duas estratégias, uso de prioridades e uso de técnicas de tolerância a falhas, propiciam o desenvolvimento de aplicações com alta disponibilidade.

A replicação ativa é uma técnica de tolerância a falhas que quando aliada a uma política adaptativa de alocação de recursos baseada em prioridade, precisa de um tratamento especial quando a chegada e o processamento de requisições ocorrem assincronamente nas réplicas do servidor. Esse tratamento consiste em evitar no contexto de grupo, casos de *inversão de prioridade*, ou seja, evitar que uma requisição de alta prioridade que acabou de chegar em um servidor fique à espera da liberação de um recurso utilizado por uma requisição de baixa prioridade. Como as réplicas do servidor podem não observar os mesmos casos de *inversão de prioridade*, o tratamento do problema em uma réplica deve considerar o estado global das réplicas de modo a garantir consistência entre elas.

Nesse trabalho são propostos o projeto e a implementação de um *middleware* que viabilize a 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.

Abstract

The fast dissemination of microcomputers and workstations, together with their increasing processing capacity and the advent of communication networks with high bandwidth, have promoted a progressive development of distributed applications. Following this trend, several enterprises became entirely dependent on their computational systems to provide their services. This dependence requires that the computational systems remain always in operation.

The lack of availability of services are usually caused by the shortage of resources or the occurrence of failure. A strategy to avoid resource shortage can be accomplished through an adaptive priority-based policy of resource allocation. Failures are inevitable, however, their consequences can be avoided with proper use of fault tolerance techniques. The combination of these two strategies, use of priorities and use of fault tolerance techniques, enables the development of applications with high availability.

Active replication is a technique for fault tolerance that when combined with adaptive priority-based resource allocation, needs a special treatment if the arrival and the processing of request occurs asynchronously in the replicas of the server. This treatment consists of avoiding, within the context of a group of replicas, cases of *Priority Inversion*, i.e to avoid that a high priority request that has just arrived in a server waits for a low priority request to release a resource. As the replicas of a replicated server might not observe the same cases of *Priority Inversion*, the solution to this problem must be based in the global state of the replicated server in order to guarantee consistency among all replicas.

We propose in this work the design and implementation of a *middleware* that enables 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.

Conteúdo

Lista de Símbolos e Abreviaturas

$\diamond S$	: Diamante S
Δ	: Intervalo de Tempo
$\prod$	: Conjunto de Servidores
ϑ	: Conjunto de Réplicas que Retornam Valores Corretos
Ψ	: Subconjunto Contido em ϑ
f	: Número de Réplicas Falhas
$\perp$	: Processador Inativo
$\prec$	: Precedência
$\mathcal{F}$	: Função
EDF	: *Earliest Deadline First* (Prazo Mais Cedo Primeiro)
E-GAF	: *Extended General Agreement Framework* (Extensão da Estrutura Genérica de Acordo)
EROPSAR	: Escalonador de Requisição Orientado a Prioridade para Serviços Ativamente Replicados
FIFO	: *First In First Out* (Primeiro a Entrar Primeiro a Sair)
GAF	: *General Agreement Framework* (Estrutura Genérica de Acordo)
GEP	: *Group Execution Progress* (Progresso de Execução de Grupo)
GPI	: *Group Priority Inversion* (Inversão de Prioridade em Grupo)
JDK	: *Java Developer Kit* (Kit de ferramentas JAVA)
LEP	: *Local Execution Progress* (Progresso de Execução Local)
LPI	: *Local Priority Inversion* (Inversão de Prioridade Local)
RPC	: *Remote Procedural Call* (Chamada a Procedimento Remoto)

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Alta disponibilidade é extremamente importante para um número crescente de aplicações distribuídas. Por exemplo, a falta de disponibilidade dos serviços de comércio eletrônico de uma loja virtual pode trazer enormes prejuízos.

Por sua vez, a indisponibilidade de serviços é normalmente causada pela escassez de recursos ou pela ocorrência de falhas. A *superprovisão* e a *alocação de recursos baseada em prioridade* 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, os recursos do sistema podem ser dimensionados de modo a suportar essa carga. Esse método é relativamente caro, pois o sistema pode permanecer a maior parte do tempo submetido a uma baixa carga, subutilizando os recursos dimensionados. A dificuldade desse método é 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 [18, 19]. Essa é uma estratégia mais econômica que a *superprovisão*, pois a quantidade de recursos permanece inalterada, não sendo necessária a adição de componentes de hardware ao sistema. Contudo, o resultado não é o mesmo da *superprovisão*, que evita de uma maneira total a escassez de recursos. A idéia dessa estratégia de alocação de recursos é evitar, em momentos de pico, o máximo possível situações que possa ocasionar perdas computacionais ou financeiras. 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* [27].

De um modo geral, o problema de inversão de prioridade ocorre quando uma atividade de alta prioridade é atrasada por uma atividade de baixa prioridade. O termo *atividade* é

um termo genérico que pode corresponder a um processo, a uma linha de controle (*thread*), a uma transação ou até mesmo a um processamento de uma requisição [29]. Algumas aplicações podem tolerar esse problema, dentro de um certo limite de atraso, desde que não possuam requisitos de previsibilidade. Contudo, quando o tempo de execução de uma atividade de baixa prioridade pode ser muito maior que o tempo de execução de uma atividade de alta prioridade, esse limite pode ser ultrapassado e o problema precisa ser detectado e tratado para que o serviço disponibilizado cumpra o que se propõe a fazer. O tratamento do 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 software denominado escalonador de requisições. Dependendo de como o processamento da requisição é interrompido, a requisição interrompida poderá continuar ou reiniciar sua execução quando todas as requisições de mais alta prioridade tiverem sido concluídas.

Os resultados do estudo apresentado em [18, 19] demonstraram a viabilidade de políticas adaptativas para alocação de recursos para sítios 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. Com base nesses parâmetros, três classes de prioridades são assumidas: alta, média e baixa. A associação de prioridades a um determinado cliente funciona da seguinte forma: todo usuário entra com alta prioridade no sistema. A idéia é permitir que usuários em estágios iniciais não desistam até que encontrem os produtos desejados. Após um tempo *m1*, este usuário tem sua prioridade reduzida de alta para média caso não tenha realizado nenhuma compra ou não tenha adicionado nenhum novo ítem ao seu pedido de compra. Se a prioridade for reduzida para média, um limite *m1+m2* é usado como novo limite para reduzir a prioridade do consumidor de média para baixa se nenhuma compra ou intenção de compra foi realizada. Caso alguma adição de ítens seja efetuada, a prioridade do consumidor é alterada para alta, de onde não é mais modificada. Através dessas políticas adaptativas, o trabalho conclui em seus resultados que há um potencial de ganho em faturamento de até 29% sobre sistemas convencionais nos momentos de pico [19].

Além da escassez de recursos uma outra causa para a indisponibilidade de serviços é a 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 utilizadas nos sistemas. As técnicas de tolerância a falhas caracterizam-se pela introdução de redundância de componentes (hardware e/ou software) [16]. Redundância é normalmente introduzida pela replicação

de componentes ou serviços [3, 26]. Na replicação de componentes ou serviços, 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* [3], 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 (passiva fria [22]), ou salvo nas réplicas secundárias (passiva quente [22]). 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 [26] 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. Se um resultado poder ser inválido, as saídas de todas as réplicas são submetidas a um elemento votador, que realiza votação majoritária para decidir o resultado final do processamento. A consistência entre as réplicas é garantida desde que as réplicas recebam as mesmas requisições na mesma ordem e produzam as mesmas saídas. Dependendo da semântica de falha das réplicas, a replicação ativa pode ser utilizada sem votação, pois qualquer saída produzida poderá ser um valor correto, como no caso de *falha por parada* [16]. Para implementar essa técnica são necessários protocolos que garantam os requisitos de *ordem* e de *acordo*, como especificado em [26]. A principal vantagem dessa técnica está em oferecer tempos de resposta aceitáveis em caso de falhas, contudo, essa técnica exige protocolos mais complexos.

Em muitas organizações os sistemas computacionais são considerados críticos para o negócio da empresa. Por exemplo, sítios de comércio eletrônico funcionam como uma loja virtual, aberta 24 horas por dia, durante todos os dias do ano, para clientes de todo o planeta

e, portanto, dependem exclusivamente de seus sistemas computacionais para o provimento de seus serviços. Essa dependência exige que os sistemas computacionais estejam sempre em funcionamento. Para que isso seja possível, além de evitar a escassez de recursos, esses sistemas devem tolerar falhas e não podem estar sujeitos a um único ponto de falha.

1.1 Motivação

Com base no que foi descrito, observa-se que 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 [29]. 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.

De modo resumido, os autores em [29] propõem um escalonador de requisições que, apoiado por um protocolo de acordo e um protocolo de ordenação total *sensível a prioridade*, evita inversão de prioridade em grupo. O protocolo de ordenação total *sensível a prioridade* garante que as réplicas de um servidor replicado processem as requisições dos clientes em uma mesma ordem. A obtenção dessa ordem e o tratamento de inversão de prioridade fundamentam-se no *progresso de execução do grupo replicado. O progresso de execução do grupo replicado* é baseado no *progresso de execução individual de cada réplica* e é calculado através de um protocolo de acordo. Dessa forma, o tratamento de inversão de prioridade deixa de ser isolado e passa a ser realizado conforme a evolução do grupo, o que garante consistência no processamento de requisições. O escalonador segue uma abordagem otimista, permitindo portanto realizar antecipadamente algum processamento, antes de se obter o resultado final do escalonamento. Como conseqüência, o processamento de

algumas requisições poderá eventualmente sofrer retrocesso (*rollback*).

No contexto de comércio eletrônico, a alta disponibilidade permite que maiores ganhos financeiros possam ser obtidos em situações de falhas ou picos. Como esses ganhos são desejáveis a uma variedade de aplicações, seria ideal que essas funcionalidades fossem encapsuladas em uma camada de software, permitindo que as aplicações incorporassem esses benefícios de um modo prático e simplificado com custo de desenvolvimento mínimo. Essa camada de software pode ser vista como um *middleware*.

1.2 Objetivo do Trabalho

Nesse trabalho são propostos o projeto e a implementação de um *middleware* que viabilize a 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. O *middleware* deve solucionar o problema de inversão de prioridade em grupo e se baseia no estudo realizado em [29].

Mais especificamente, no nosso contexto o *middleware* é uma camada de software que possibilita a comunicação entre aplicações clientes e aplicações servidoras (servidor de aplicação e banco de dados). O *middleware* deve possuir um protocolo escalonador que permita o processamento ativamente replicado de requisições segundo as prioridade dos usuários do sistema. Considerando que nas aplicações de comércio eletrônico é possível receber vários pedidos de diferentes clientes enquanto o servidor realiza o processamento de um ou mais pedidos, esse *middleware* implementa um protocolo escalonador que evita e trata casos de inversão de prioridade.

1.3 Relevância

A relevância desse trabalho está em disponibilizar pela primeira vez um software que trata do problema de inversão de prioridade dentro de um grupo de processadores que realizam um processamento ativamente replicado. Conforme foi mostrado em outros trabalhos, a funcionalidade provida por esse software permitirá a continuidade de serviços em caso de falhas e permitirá a obtenção de maiores lucros em aplicações de comércio eletrônico em situações de pico.

1.4 Organização da Dissertação

Essa dissertação está organizada da seguinte forma. O Capítulo 2 contém a fundamentação teórica do trabalho, onde apresentamos o problema de inversão de prioridade em um grupo [29]. Nesse capítulo, o problema de inversão de prioridade em grupo é definido e as dificuldades encontradas na solução do problema são apresentadas. No Capítulo 3 descrevemos o projeto do *middleware* mostrando a estrutura do *middleware*, o modelo funcional, o modelo de replicação e os serviços de suportes exigidos no projeto. Em seguida, no Capítulo 4, mostramos como o *middleware* foi implementado. Realizamos alguns testes e fizemos uma avaliação de desempenho de nossa implementação. Finalmente, no Capítulo 5 apresentamos nossas conclusões e discutimos sobre possíveis trabalhos futuros.

Capítulo 2

Inversão de Prioridade em Grupo

Neste capítulo apresentamos o problema de inversão de prioridade em um grupo de processadores que realizam um processamento ativamente replicado. Inicialmente apresentamos o modelo do sistema considerado. Em seguida, apresentamos o problema de inversão de prioridade em um sistema com processamento não replicado e em um sistema com processamento ativamente replicado. Especificamos então o problema de inversão de prioridade em grupo [29], apresentando alguns conceitos, definições e as dificuldades que precisam ser vencidas na solução para o problema.

2.1 Introdução

Em sistemas multitarefa é comum que mais de uma atividade utilize os mesmos recursos, como discos, impressoras, processadores, linhas de comunicação etc. O termo *atividade* é um termo genérico que pode corresponder a um processo, a uma linha de controle, a uma transação ou até mesmo ao processamento de uma requisição [29]. Para implementar o compartilhamento de recursos, é necessário um procedimento gerenciador de recursos. Esse procedimento é realizado por um componente de software denominado escalonador[1]. O critério utilizado por um escalonador para determinar a ordem que um recurso é disponibilizado pode ser baseado em diversas políticas, como por exemplo: *first-in-first-out* (FIFO), *shortest-job-first*, *round-robin* e orientado a prioridade [28].

Nesse capítulo, consideramos uma política de escalonamento baseada em prioridade, onde o recurso disponibilizado é um processador que executa requisições, atividades solicitadas por aplicações clientes. A concessão ao recurso é disponibilizada por ordem de

[1] O escalonador implementa políticas para alocação que uma função gerencia, mas há outras necessidades no gerenciamento, como sincronização, exclusão mútua, resolução e detecção de impasses, etc.

prioridade, ou seja, da requisição de maior prioridade para a requisição de menor prioridade. As requisições possuem prioridade fixa [21], ou seja, uma vez definida a prioridade de uma requisição, esta não é alterada durante seu tempo de vida. A prioridade é definida antes do envio da requisição. Admitimos um escalonamento preemptivo [28], onde o processamento de uma requisição pode ser interrompido ou abortado para iniciar ou retomar o processamento de um outra requisição. A preempção poderá ocorrer para tratar casos de inversão de prioridade, pois é possível que o processamento de uma requisição seja atrasado por uma outra requisição de menor prioridade que detenha o direito de acesso a um recurso compartilhado.

Abordamos o problema de inversão de prioridade em um grupo de processadores que realizam um processamento ativamente replicado em um sistema assíncrono no qual processos estão sujeitos a falhas por parada. Como será visto no decorrer do capítulo, o tratamento desse problema é bem mais complexo que em um processamento não replicado.

2.2 Modelo do Sistema

Nas seções seguintes, o problema de *inversão de prioridade em grupo* é discutido considerando o modelo de sistema distribuído assíncrono equipado com *detectores de falhas não confiáveis*. Um sistema distribuído assíncrono é formado por diversos nodos autônomos que são conectados por uma rede de comunicação. Cada nodo consiste em um processador, uma memória local inacessível a todos os outros processadores e um relógio local. Os nodos não possuem memória compartilhada, se comunicam por troca de mensagens [16] e não há limites nos tempos de transmissão de mensagens nem nas velocidades de processos. O interesse nesse modelo de sistema é justificado por sua aplicação prática (diversas aplicações possuem oscilação de carga de processamento e de rede que ocasionam assincronia), pois é o mais realista para modelar os sistemas comumente disponíveis hoje. Além da ampla área de aplicação, sistemas desenvolvidos sob esse modelo são mais portáveis que outros que incorporam alguma concepção de tempo [6].

Considera-se que os processos podem falhar por parada [16]. Para que falhas sejam detectadas é necessário que exista um mecanismo encarregado de detectá-las. Contudo no modelo de sistema assíncrono, é impossível detectar a falha de um processo, pois não há como diferenciar se um processo parou ou se ele está apenas muito lento [11]. A impossibilidade de alcançar consenso em um sistema distribuído assíncrono, motivou a definição de outros modelos de sistemas, tais como: parcialmente síncrono [8], assíncrono temporizado [10] e assíncrono equipado com detectores de falhas [6]. Todos esses modelos possuem

um propósito em comum, que é o de introduzir algum grau de sincronismo no sistema. Nesse trabalho, o modelo de sistema assíncrono é equipado com *detectores de falhas não confiáveis*, conforme [6].

2.3 Inversão de Prioridade em um Sistema com Processamento não Replicado

O escalonamento de requisições orientado a prioridade define a ordem em que um conjunto de requisições é processado. Essa ordem é definida de acordo com o nível de prioridade das requisições, obedecendo uma hierarquia que varia da requisição de maior prioridade à requisição de menor prioridade. Esse tipo de escalonamento é ilustrado na Figura 2.1.

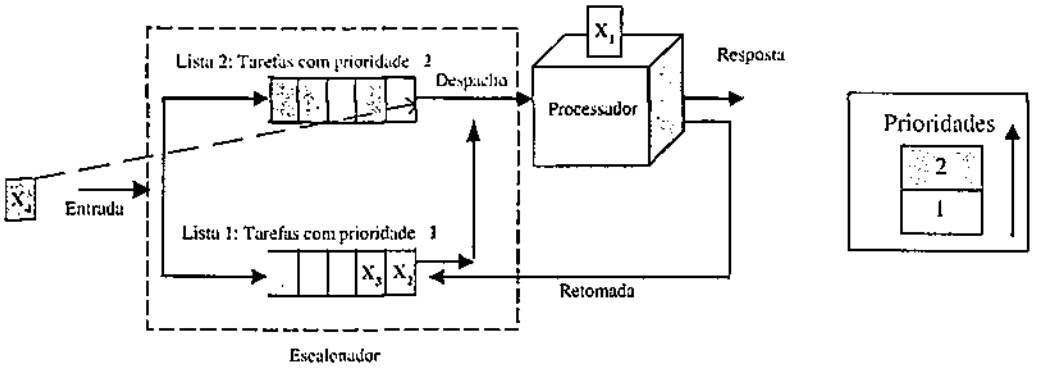


Figura 2.1: Inversão de prioridade em um servidor não replicado

Na Figura 2.1 é apresentado um diagrama 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, pode 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 escalonador passa a escolher as requisições da lista seguinte.

Quando a entrada de requisições no escalonador ocorre dinamicamente, pode acontecer que uma requisição de prioridade alta ficar esperando a conclusão do processamento de uma outra requisição de prioridade mais baixa. Verifica-se através da Figura 2.1 que o processa-

dor está ocupado com requisição X_1 de prioridade *1* quando a requisição X_4 de prioridade *2* é recebida. Como *Prioridade* (X_1) < *Prioridade* (X_4), observa-se que o processamento da requisição X_1 está atrasando o processamento da requisição X_4, evidenciando portanto, um caso de inversão de prioridade. Ao contrário de sistemas em tempo-real, não consideramos aqui, perda de prazo, consideramos apenas a impossibilidade de uma atividade de alta prioridade iniciar seu processamento por conta do processador encontrar-se processando uma atividade com prioridade inferior.

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 suspenso ou abortado, deve ser reordenada e ter seu processamento reiniciado a partir do último estado anteriormente salvo. O momento do reinício irá ocorrer quando o processamento de todas as requisições de prioridade mais alta houver sido concluído.

2.4 Inversão de Prioridade em um Sistema com Processamento Ativamente Replicado

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.

Na Figura 2.2 ilustra-se o problema de inversão de prioridade em um sistema com processamento ativamente replicado. Duas réplicas processam independentemente suas requisição utilizando os mesmos critérios ilustrados na Figura 2.1. No momento que a requisição 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* 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 X_1 em processamento na réplica *B* possui prioridade menor

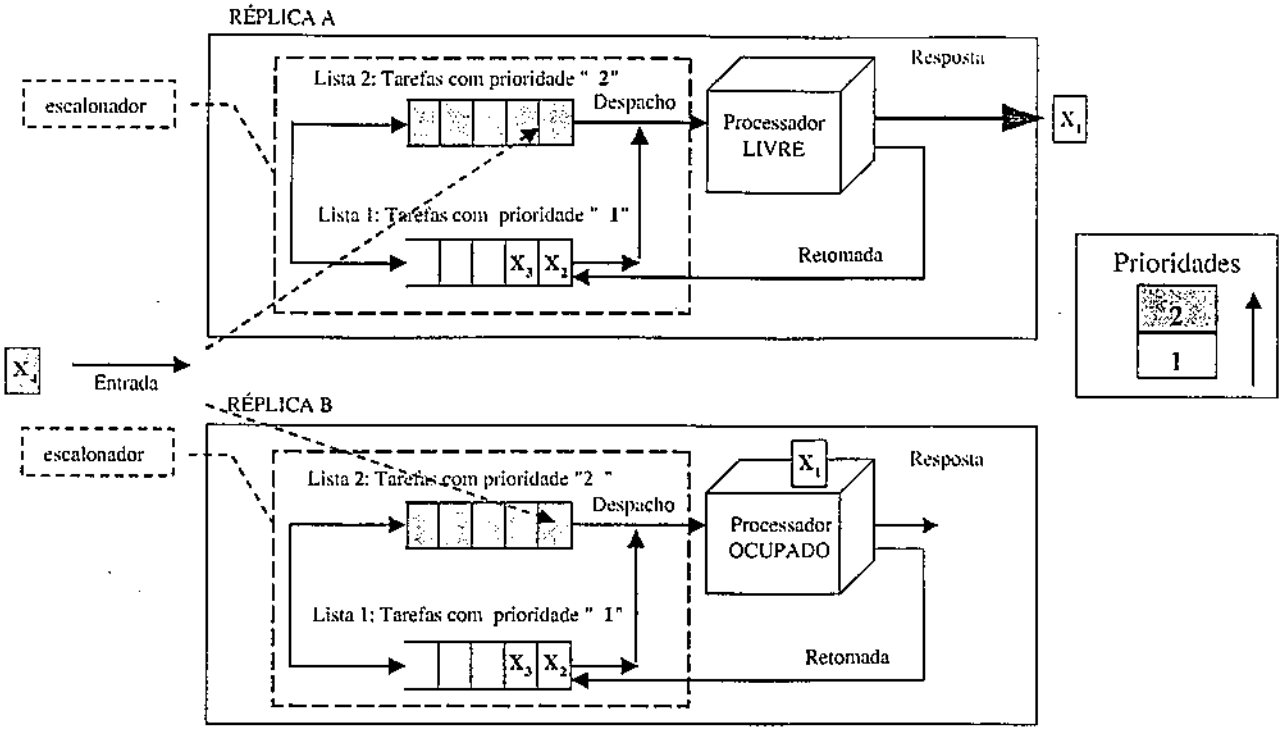


Figura 2.2: Inversão de prioridade em um servidor replicado

que a nova requisição X_4, ou seja, *Prioridade* (X_1) < *Prioridade* (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 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 na relação entre o estado local e global das réplicas. Em [29] apresenta-se duas alternativas para tratar o problema. 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, como será visto nas subseções a seguir.

O conceito de inversão de prioridade em grupo (*Group Priority Invertion - GPI*) foi introduzido pela primeira vez em [29], 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 [29] foi proposto um protocolo escalonador de requisição, orientado a prioridade, que garante ordenação total e ao mesmo tempo evita inversão de prioridade em grupo. As subseções seguintes apresentam o problema de inversão de prioridade em grupo apresentado em [29].

2.4.1 Escalonando Requisições: Terminologia e Funcionamento

Uma aplicação que faz uso do escalonador é formada por clientes que enviam requisições para serem processadas em um servidor replicado S. O grupo de processadores associados a um servidor S é representado por $\prod$. O número de elementos no conjunto $\prod$ é igual a n e chamado de grau de replicação do servidor replicado S. Dentre n processadores, temos que no máximo f, $0 \leq f < n$, podem falhar enquanto que n - f são corretos, isto é, nunca falharão. O termo *requisição* se refere a uma tarefa computacional. A cada requisição enviada pelo cliente existe uma prioridade associada. Uma requisição é representada por r. Por definição, $Prioridade(r) < Prioridade(r')$, se e somente se, a requisição r tiver o nível de prioridade menor que r'. Cada réplica p do serviço S, identificada como S_p, executa um módulo de escalonamento de requisição denominado como RS_p. Esse módulo deve desempenhar o seguinte papel:

- receber as requisições enviadas pelos clientes;
- escalonar as requisições de acordo com suas prioridades;
- evitar a ocorrência de inversão de prioridade em grupo; e
- enviar a resposta de cada requisição aos clientes correspondentes.

Após o processamento de uma requisição, os clientes podem receber até n respostas. Denotamos $\vartheta(a)$, onde $\vartheta(a) \subseteq S$, o conjunto de todas as réplicas que retornam uma resposta válida para uma requisição a. Esse conjunto contém todos os processos que entregam suas respostas. Somente uma resposta válida é utilizada, pois como assumimos falhas por parada, todas as respostas geradas são corretas. Desse modo as demais são descartadas.

A interação entre uma réplica S_p e seu módulo RS_p (Figura 2.3 [29]) é realizado da seguinte maneira: sempre que S_p tiver completado o processamento de uma requisição, S_p solicita novas requisições a RS_p. Caso não existam requisições prontas para serem executadas, S_p bloqueia até que RS_p disponibilize uma nova requisição. A preempção de uma tarefa só ocorre em casos de inversão de prioridade. Nesses casos, o processamento é interrompido, as requisições são reordenadas ou desfeitas e o estado da réplica é atualizado para um estado anteriormente salvo. O estado recuperado é igual a um estado salvo antes do processamento da última requisição completada que não sofreu retrocesso.

Em um dado momento t, cada módulo RS_p mantém uma história local de suas requisições denotada como $História_{p,t}$. A seqüência $História_{p,t}$ é formada por duas listas: $Completada_{p,t}$ e $Pronta_{p,t}$. A lista $Completada_{p,t}$ é formada por todas as requisições no

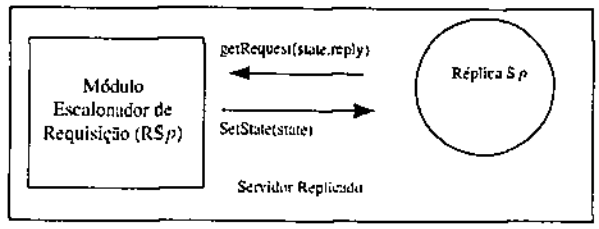


Figura 2.3: Interação entre S_p e RS_p

tempo t que já foram processadas por S_p. A lista $Pronta_{p,t}$ contém todas as requisições no tempo t que estão prontas para serem processadas. $t_p^{História}$ se refere ao tempo que $História_p$ possui uma determinada configuração. $t_p^{envio}(r)$, se refere ao tempo que uma requisição r foi submetida por um cliente ao módulo RS_p. O tempo $t_p^{Completada}(r)$ se refere ao tempo em que a réplica S_p terminou o processamento de r. O número de elementos de $Completada_{p,t}$ representa o progresso de execução local de uma réplica S_p no tempo t e é denotado por $LEP_{p,t}$. Sempre que u ma requisição é completada por S_p, o valor de LEP_p é atualizado por RS_p. LEP representa a divisão entre $Completada_{p,t}$ e $Pronta_{p,t}$ em $História_{p,t}$.

Para referenciar a requisição que está em processamento em S_p no tempo t, utilizamos a notação $Executando_{p,t}$. Quando a réplica S_p encontra-se inativa, dizemos que $Executando_{p,t}=\perp$.

Com base na terminologia apresentada, definimos, formalmente inversão de prioridade local (LPI) e inversão de prioridade em grupo (GPI):

Inversão de prioridade local: Para qualquer processador p e requisição r, r sofre inversão de prioridade em p (i.e., $\mathrm{LPI}_p(r)$=verdade), se e somente se, a expressão abaixo for satisfeita:

$\exists t \mid Executando_{p,t} \neq \perp \wedge r \in Pronta_{p,t} \wedge Prioridade(Executando_{p,t}) < Prioridade(r)$.

Inversão de prioridade em grupo: Para qualquer requisição r, r sofre uma inversão de prioridade em grupo (i.e., GPI(r)=verdade), se e somente se, a seguinte expressão for satisfeita:

$\exists \Psi \mid \Psi \subseteq \vartheta(r) \wedge (\forall p \in \Psi : \mathrm{LPI}_p(r)) \wedge |\vartheta(r)| - |\Psi| < 1$.

Uma inversão de prioridade em grupo irá ocorrer sempre que o processamento de uma requisição replicada r de alta prioridade for atrasado por atividades de menor prioridade. O processamento de uma requisição replicada r requer que no mínimo 1 processo gere uma

resposta válida. Isso quer dizer que uma requisição r não sofrerá inversão de prioridade em grupo, se for possível garantir que pelo menos *1* resposta a essa requisição será gerada sem qualquer atraso. Portanto, r sofrerá inversão de prioridade em grupo quando o número de processadores em $\vartheta(r)$, que não sofreram inversão de prioridade local em uma atividade r, for menor que *1*.

2.4.2 Especificação do Problema de Inversão de Prioridade em Grupo

Assim como no contexto não replicado, quando uma inversão de prioridade é detectada, a requisição em execução poderá ser suspensa ou abortada. O custo do mecanismo de sincronização necessário para garantir que todas as réplicas sejam suspensas no mesmo ponto de execução (réplicas precisam passar pelas mesmas mudanças de estado) é muito alto (troca de mensagens e variáveis de controle para sincronização) sendo portanto mais viável que a requisição seja abortada.

Para possibilitar essa preempção, quando uma réplica p solicita ao módulo RS_p por novas requisições (invocação getRequest(state, reply) na Figura 2.3), a réplica S_p fornece ao módulo o seu novo estado local e o resultado da atividade recém processada. Antes de liberar uma atividade para processamento, RS_p utiliza o estado local anteriormente recebido para armazenar um ponto de salvaguarda (*checkpoint*), desse modo, quando necessário, o módulo RS_p pode realizar um retrocesso na réplica S_p para restaurar um estado previamente consistente.

A possibilidade de retrocessos pode causar inconsistências, particularmente quando existe um relação causal [17] entre as respostas das requisições e um cliente recebe o resultado de uma requisição cujo processamento sofreu retrocesso. Para evitar esse problema, o módulo escalonador RS_p armazena a resposta de r. Esta resposta será enviada ao cliente somente quando RS_p garantir que a réplica S_p não sofrerá retrocesso para um estado que anteceda o estado salvo após o processamento de r. Na necessidade de um retrocesso, caso S_p esteja em processamento, o módulo RS_p interrompe seu processamento e, em seguida, utiliza uma salvaguarda para reinicializar S_p para o último estado consistente salvo (invocação setState(state) na Figura 2.3). Finalmente, a requisição que causou a alteração de $História_p$ é iniciada em S_p.

Verifica-se que, para evitar inversão de prioridade, uma atividade pode ser processada várias vezes devido à ocorrência de retrocesso. Para que uma atividade não seja submetida a infinitos retrocessos, [29] introduziu os conceitos de *irreversibilidade* e *conflito*.

Requisições irreversíveis: Uma requisição irreversível é aquela que, uma vez processada e pertencente à lista *Completada*, nunca estará presente na lista *Pronta*, conseqüentemente, nunca sofrerá retrocesso. Segue a expressão que define uma requisição irreversível:

$\exists S_p \in \prod, \exists t : r \in Completada_{p,t} \wedge r \notin Pronta_{p,t'} \forall t' > t.$

Requisições livres de conflito: Uma requisição r é livre de conflito se todas as requisições r', com prioridade mais alta que r, satisfazem as seguintes condições:

- O envio de r' ocorre antes que o envio de r em todas as réplicas.
- O envio de r' ocorre após a conclusão do processamento de r em todas as réplicas.

Formalmente, temos que r é livre de conflito se para todo r' com prioridade maior que prioridade de r for possível garantir a expressão:

$(\forall S_p\ t_p{}^{envio}(r') < t_p{}^{envio}(r)) \vee (\forall\ S_p\ t_p{}^{envio}(r') > t_p{}^{Completada}(r))$.

Para evitar inversão de prioridade em grupo em um sistema ativamente replicado, é preciso que todas as réplicas corretas processem a mesma seqüência de requisições e, ao mesmo tempo, evitem inversão de prioridade em grupo. Portanto, a especificação do problema de inversão de prioridade em grupo (GPI) resulta na combinação entre as propriedades da difusão atômica e as definições das requisições *irreversíveis* e *livres de conflito*.

A especificação do problema GPI em um sistema ativamente replicado, conforme [29], é apresentada a seguir:

1. Ordem total: Para quaisquer duas requisições r e r' irreversivelmente processadas por quaisquer duas réplicas $S_p \in \prod$ e $S_q \in \prod$, se r' foi irreversivelmente processada por S_p antes de r, então r' também foi irreversivelmente processada por S_q antes de r.

2. Ordem de prioridade: Para qualquer requisição r, GPI(r)=falso.

3. Acordo uniforme: Qualquer requisição r que foi irreversivelmente processada por uma réplica $S_p \in \prod$, é, em um dado momento, irreversivelmente processada por todas as réplicas corretas $S_q \in \prod$.

4. Terminação: Para qualquer requisição r, se r é recebida por uma réplica correta $S_p \in \prod$, então r é, em um dado momento, processada por S_p.

5. Não trivialidade: Para qualquer requisição r, se r é livre de conflito, então o processamento de r é irreversível e r é processada somente uma vez.

2.4.3 Dificuldades

O esquema de replicação ativa requer que cada réplica dentro do conjunto $\prod$ receba as requisições dos clientes na mesma ordem. Como, no modelo de sistema considerado, as réplicas estão sujeitas a falhas por *parada*, é preciso que seus módulos escalonadores entrem em um acordo que garanta que todas as réplicas possuam as mesmas requisições e as ordenem da mesma maneira. Este acordo é obtido através de um protocolo de ordenação total executado pelos módulos escalonadores sempre que novas requisições são enviadas pelos clientes. O resultado do protocolo, ou *valor de decisão* do protocolo, representa um conjunto ordenado de requisições que se encontram presentes em todas as réplicas. Sempre que uma nova decisão é obtida, cada módulo escalonador atualiza sua lista *Pronta*. A ordem em que essas requisições serão processadas deve ser atualizada de modo a evitar inversão de prioridade.

Como vimos na Subseção 2.4.1, uma inversão de prioridade local em uma requisição r pode ser detectada testando se $Prioridade(r) > Prioridade(Executando_{p,t})$, ou seja, se a prioridade de r for maior que a prioridade da requisição em execução na réplica p e no tempo t, temos então um caso de inversão de prioridade local. Esse procedimento não é suficiente para detectar inversão de prioridade em grupo, pois o recebimento e o escalonamento de requisições ocorrem gradualmente e assincronamente em cada um dos módulos escalonadores RS das réplicas. Assim, uma inversão de prioridade local, detectada por um módulo RS_p, pode nunca ocorrer em um módulo RS_q, pois a adição e remoção dos elementos de suas listas *Pronta* ocorrem de uma maneira não sincronizada.

A dificuldade causada pela ausência de sincronismo entre os módulos escalonadores das réplicas é apresentada em dois cenários ilustrados pelas Figuras 2.4 e 2.5. Em ambos os cenários, r e r' são duas requisições tal que $Prioridade(r) < Prioridade(r')$. C e C' são clientes que enviam as requisições r e r' para as réplicas S_p e S_q. Uma requisição pode assumir 3 estados: *pronta para execução*, *em processamento* e *completada*. Cada um desses estados é representado nas figuras por uma tonalidade de cor. Ao longo do tempo, as requisições mudam de cor de acordo com seu estado.

Cenário 1: Chegada de requisições nas réplicas não é sincronizada

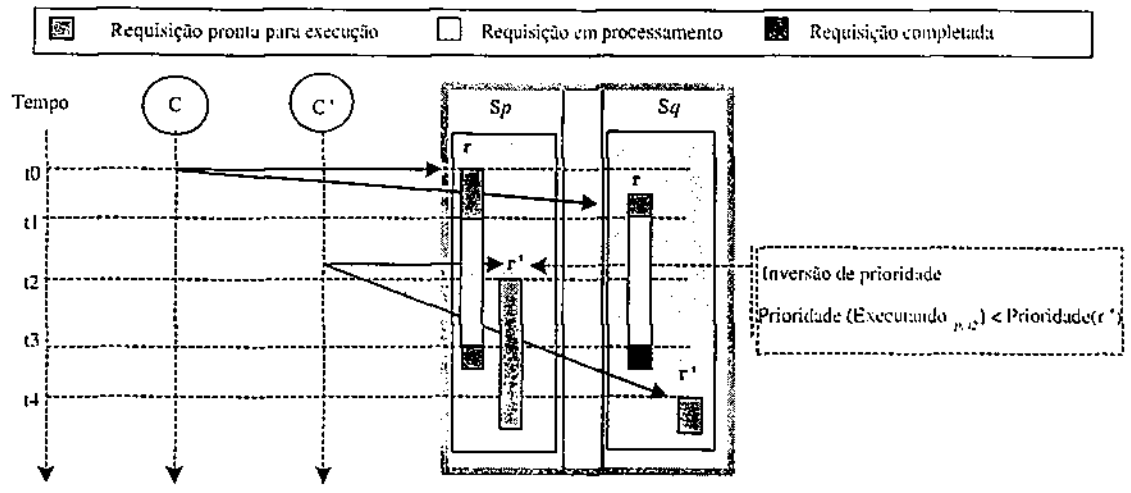


Figura 2.4: Chegada não sincronizada de requisições nas réplicas

1. Em t1, S_p e S_q iniciam o processamento de r;
2. Durante o processamento de r, r' é enviado para S_p;
3. No tempo t2, o estado de r é atualizado para "pronta para execução";
5. Em t3, S_p e Sq concluem o processamento de r;
6. Em t4, S_q recebe r' e o estado de r' é atualizado para "pronta para execução";

Considerando que t1<t2<t3<t4, o módulo escalonador de S_p detecta uma inversão de prioridade em t2, pois $Prioridade(\text{Executando}_{p,t2}) < Prioridade(r')$. Contudo, observa-se que o módulo escalonador de S_q não detecta inversão de prioridade em t4 após receber r', pois, nesse caso, r já havia sido completada em t3.

Esse cenário mostra que, mesmo que o processamento das requisições seja perfeitamente sincronizado em todas as réplicas, a lista *História* não será a mesma em todas as réplicas caso o tratamento de inversão de prioridade seja realizado em S_p. Após o tratamento, as seguintes configurações podem ser observadas: $História_{p,t3}=\{r', r\}$ e $História_{q,t3}=\{r, r'\}$. Os mesmos casos de inversão de prioridade não serão observados em todas as réplicas, pois a entrada das requisições nas réplicas ocorre de maneira não sincronizada.

Cenário 2: Processamento das requisições nas réplicas não é sincronizado

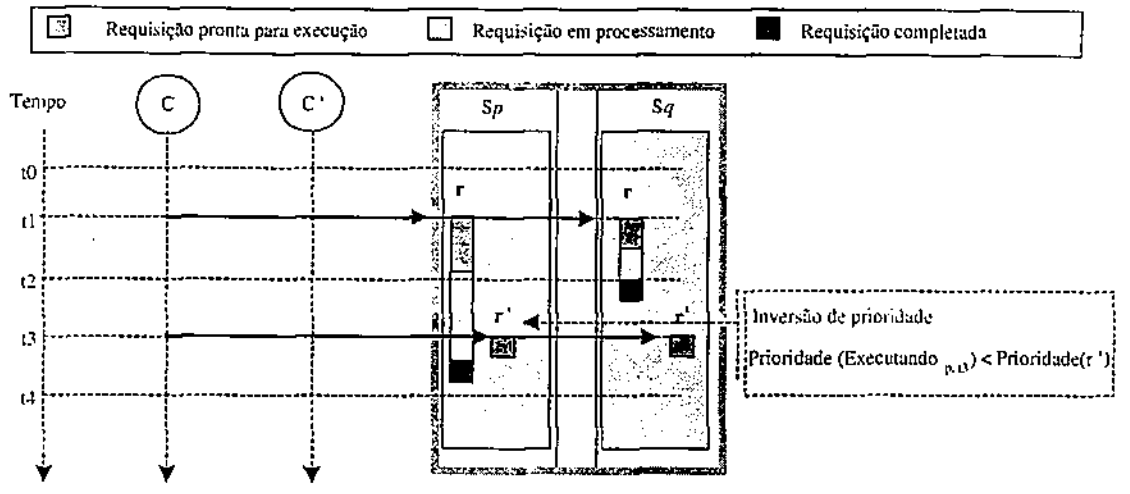


Figura 2.5: Processamento não sincronizado das requisições nas réplicas

1. Em t1, S_p e S_q recebem simultaneamente r;
2. Em t2, S_q completa o processamento de r;
3. Em t3, S_p e S_q recebem simultaneamente r';
4. Em t3, S_p permanece processando r.

Considerando que t1<t2<t3<t4, o módulo escalonador de S_p detecta uma inversão de prioridade em t3, pois $Prioridade(\text{Executando}_{p,t3}) < Prioridade(r')$. Contudo, observa-se que o módulo escalonador de Sq não detecta inversão de prioridade em t3 após receber r', pois, nesse caso, r já havia sido completada em t3.

Esse cenário mostra que, mesmo que a entrada de requisições seja perfeitamente sincronizadas em todas as réplicas, a lista *História* não será a mesma em todas as réplicas após o tratamento de inversão de prioridade em S_p ($História_{p,t3}=\{r', r\}$ e $História_{q,t3}=\{r, r'\}$). Os mesmos casos de inversão de prioridade não serão observados em todas as réplicas, pois as réplicas processam suas requisições de maneira não sincronizada. Os dois cenários mostram que, para detectar e evitar inversão de prioridade em grupo, os módulos escalonadores precisam ter uma visão consistente do grupo quanto às suas *histórias* e aos seus progressos de execução. Caso contrário, algumas ocorrências de inversão de prioridade de grupo poderão não ser detectadas, enquanto que outras poderão ser detectadas sem terem ocorrido. Através de um acordo que define um referencial de progresso de execução do grupo de réplicas, como será visto na Subseção 3.4, pode-se adicionar um grau de sincronia inexistente entre as réplicas, de modo a contornar os problemas apresentados.

Capítulo 3

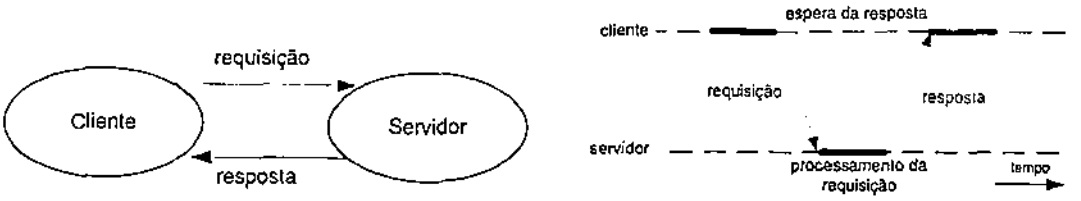
O Middleware EROPSAR

Nesse capítulo apresentamos o projeto de um *middleware* que denominamos *EROPSAR* (Escalonador de Requisição Orientado a Prioridade para Serviços Ativamente Replicados). O *EROPSAR* viabiliza a construção de aplicações de comércio eletrônico que precisam tolerar falhas e utilizar políticas adaptativas de alocação de recursos baseadas em prioridade. A técnica de replicação ativa é empregada para tolerar falhas. O *middleware* soluciona o problema de inversão de prioridade em grupo conforme estudo realizado em [29] e apresentado no Capítulo 2.

Inicialmente é apresentado de modo simplificado o funcionamento do *middleware* e a estrutura do *EROPSAR*. Em seguida, apresentamos uma solução do problema de inversão de prioridade em grupo. Apresentamos então o *modelo de replicação* e o *modelo funcional*. Prosseguimos com algumas considerações sobre o escalonamento de requisições baseado em prioridade. Por fim mostrado os *serviços de suporte* necessários no projeto.

3.1 Introdução

Conforme discutido no Capítulo 1, muitas organizações dependem exclusivamente de seus sistemas computacionais para o provimento de seus serviços. Essa dependência exige que os sistemas computacionais estejam sempre em funcionamento. Para viabilizar a construção de aplicações com alta disponibilidade, projetamos um *middleware* utiliza uma política de alocação de recursos baseada em prioridades e que adiciona transparentemente um processamento ativamente replicado às aplicações. Esse *middleware* contém protocolos que possibilitam a *comunicação cliente-servidor* (*Remote Procedure Call - RPC*), o *compartilhamento de recursos com tratamento de inversão de prioridade* (escalonador) e a *redundância de componentes* (replicação ativa).



(a) Interação requisição-resposta (b) Modo de comunicação síncrono

Figura 3.1: Comunicação cliente-servidor

Na *interação requisição-resposta* ilustrada na Figura 3.1, um processo cliente envia uma mensagem com uma requisição de serviço para um processo servidor que realiza o serviço e envia uma mensagem com a resposta para o cliente. Como pode ser visto na Figura 3.1(b), o modo de comunicação nessa interação é síncrono. Nesse modo de comunicação, o cliente envia uma mensagem e suspende o seu processamento até que o servidor receba a mensagem, execute o serviço e envie a mensagem de resposta ou a confirmação indicando que o serviço foi realizado ou não. Quando o cliente recebe a resposta, ele retoma seu processamento.

Para implementar esse tipo de comunicação, o *middleware EROPSAR* se baseia no paradigma de chamada de procedimento remoto (RPC) [23]. RPC se fundamenta no mecanismo de chamadas a procedimentos existente nas linguagens de programação convencionais. A idéia chave desse paradigma é auxiliar a construção de sistemas distribuídos, de modo que os serviços remotos sejam tão fáceis de acessar quanto os procedimentos ou serviços operacionais locais. O modo de comunicação da RPC é normalmente síncrono, no entanto, prevendo situações em que não é necessário que a chamada a um procedimento aguarde a resposta, o *EROPSAR* fornece também facilidades de funcionamento assíncrono, de modo que a invocação retorna antes de seu processamento e o resultado da procedimento invocado pode ser consultado posteriormente.

3.2 Aplicações Baseadas no *middleware EROPSAR*

O *middleware EROPSAR* pode ser utilizado por aplicações onde os clientes enviam concorrentemente requisições a um servidor. Estas requisições devem ser processadas priorizando os clientes mais importantes segundo um critério particular da aplicação. A Figura 3.2 ilustra o exemplo de uma aplicação utilizando o *middleware EROPSAR*.

Como pode ser visto, por exemplo, no contexto de uma aplicação de comércio eletrô-

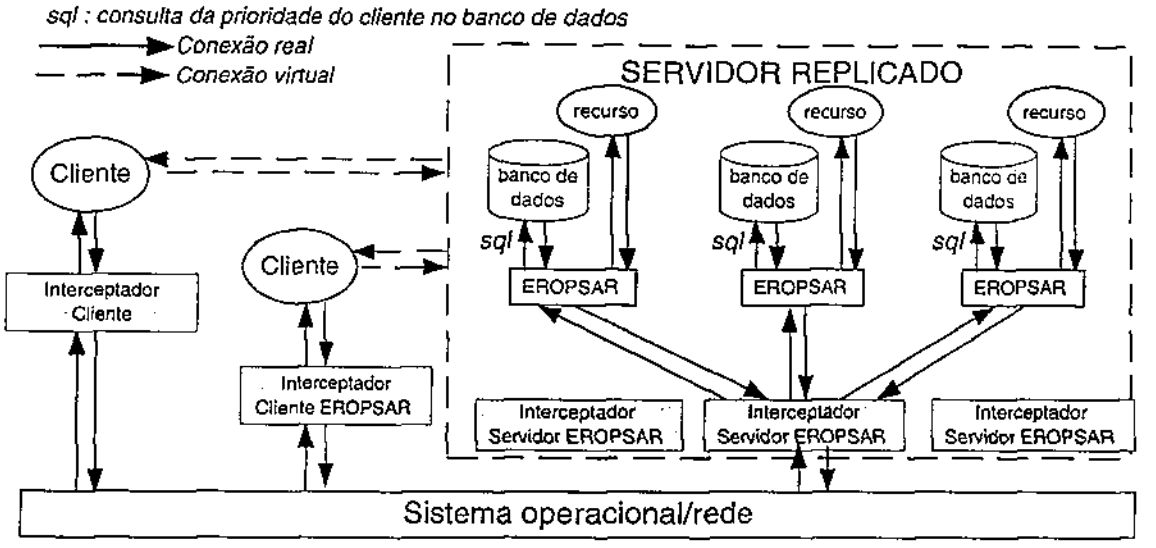


Figura 3.2: Transparência utilizando o *middleware EROPSAR*

nico ilustrado na Figura 3.2, os clientes acessam *transparentemente* (linhas pontilhadas) os recursos de um servidor replicado utilizando dois componentes *interceptadores* (interceptador cliente e interceptador servidor). Esses componentes se baseiam no *padrão arquitetural interceptador* (*Interceptor Architectural Pattern*) [25]. Esse padrão permite a adição de serviços a um determinado *framework*, de forma transparente, tornando-o extensível. Nesse exemplo, os bancos de dados da aplicação, ilustrados em cada réplica, são utilizados pelo *EROPSAR* para consultar a prioridade de um cliente e devem estar sincronizados em todas as réplicas. A prioridade de uma requisição é inicialmente igual ao identificador do cliente. Quando a requisição é recebida no servidor, sua prioridade é decodificada conforme um valor armazenado no banco de dados. A chave utilizada para consultar a prioridade no banco de dados é o identificador do cliente. Essas operações são realizadas internamente no *EROPSAR* por um protocolo escalonador de requisição. O protocolo escalonador das réplicas está relacionado a um protocolo de replicação ativa que deve ordenar e processar as requisições na mesma ordem em todas as réplicas. Como a chegada e o processamento de requisições ocorrem assincronamente, problemas de *inversão de prioridade em grupo* são possíveis de ocorrer e são evitados. Após o processamento de uma requisição, o protocolo escalonador armazena temporariamente seu resultado e verifica por requisições *estáveis* (requisições que jamais terão sua posição na lista de requisições alterada). Quando uma requisição torna-se *estável*, ela é enviada a um mecanismo de validação, para que seu resultado seja finalmente enviado ao cliente.

3.3 Estrutura Interna

O estrutura do *EROPSAR* é formada por 6 componentes identificados na Figura 3.3 pelos retângulos em cinza. Descrevemos a seguir cada um desses componentes e como eles interagem uns com os outros para acessar um recurso remoto a partir de uma requisição enviada por uma aplicação cliente.

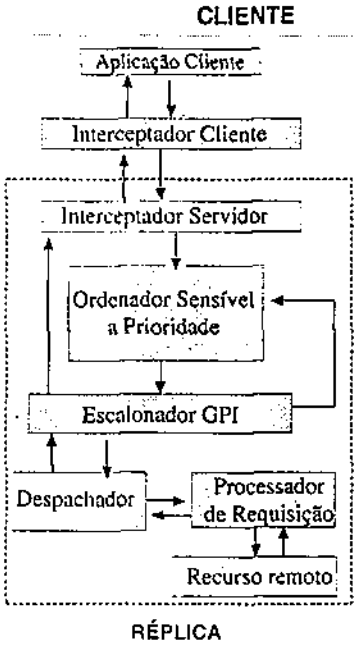


Figura 3.3: Estrutura interna do EROPSAR

Uma *aplicação cliente* envia suas requisições e recebe seus resultados através de um componente *interceptador*. O interceptador pode ser do tipo cliente ou servidor. O *interceptador cliente* é carregado transparentemente no lado cliente da aplicação e é utilizado para enviar a requisição a um dos *interceptadores servidores* corretos. O interceptador servidor reside no lado servidor da aplicação, recebe a requisição do interceptador cliente e a difunde nas réplicas de um servidor replicado. Após receber o resultado do processamento, o interceptador servidor envia a primeira resposta de uma requisição ao cliente e descarta as respostas já recebidas.

O componente *Ordenador Sensível a Prioridade* reside no lado servidor da aplicação e recebe as requisições dos *interceptadores servidores* e as armazenam localmente. Esse componente garante que todos os outros componentes ordenadores não falhos observem a mesma 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 inversão de prioridade em grupo. Conforme será detalhado mais adiante, esses dois componentes trabalham em estrita cooperação.

O componente *despachador* recebe do *escalonador GPI* a requisição pronta de maior prioridade. Uma vez recebida, utilizando a referência do *processador de requisição* da aplicação, 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, devido a possibilidade de interrupções e retrocessos. Dessa forma um 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.*

3.4 Solucionando o Problema de Inversão de Prioridade em Grupo

Utilizamos no nosso projeto a mesma terminologia introduzida no Capítulo 2. Desse modo, consideramos também uma lista *História* formada por duas sublistas: *Completada* e *Pronta.* A sublista *Pronta* armazena requisições prontas para execução e a sublista *Completada* armazena requisições que já foram processadas. O módulo escalonador, *escalonador GPI*, apresentado na seção anterior evita inversão de prioridade em grupo e mantém a lista *História* atualizada.

Casos de inversão de prioridade em grupo poderiam ser evitados sem nenhum custo de comunicação entre as réplicas se seus módulos escalonadores, após a execução do protocolo de ordenação total, ordenassem as novas requisições na lista *História* por ordem de prioridade sem considerar os estados *prontas para execução, em processamento* e *completadas* das requisições nas sublistas *Pronta* e *Completada.* Desse modo, considerando que o processamento de requisições segue da direita para a esquerda da sublista *Pronta*, quando

uma nova requisição r é escalonada, esta será adicionada em uma posição de *História* onde as requisições à esquerda têm menor prioridade de execução e as requisições à direita têm prioridade de execução maior ou igual à prioridade de r. *História* seria então dividida em 2 partes e, em seguida, estas partes seriam unidas através da nova requisição r. No pior caso, todas as requisições completadas à esquerda de r em *História* sofreriam retrocesso.

tempo	evento	*Lista História = Lista pronta + Lista completada*
t0	adicione r1, prioridade=4	
t1	adicione r2, prioridade=2	**r1**
t2	adicione r3, prioridade=3	r2,**r1**
t3	adicione r4, prioridade=1	r2,r3,**r1**
t4	adicione r5, prioridade=5	r4,**r2,r3,r1**
t5		r4,r2,r3,r1,r5

Tabela 3.1: Evolução da lista *História* de acordo com as prioridades das requisições

A Tabela 3.1 mostra a evolução da lista *História*, união da sublista *Pronta* (requisições não sublinhadas) com a sublista *Completada* (requisições sublinhadas e em negrito). Considerando $Prioridade(r4)<Prioridade(r2)<Prioridade(r3)<Prioridade(r1)<Prioridade(r5)$ e t0<t5, verifica-se que, em t5, todas as requisições em $Completada_{t4}$ sofreram retrocesso.

Observa-se que essa solução não satisfaz a propriedade de *não-trivialidade* da especificação do problema GPI em um sistema ativamente replicado. Essa propriedade requer que para qualquer requisição r, se r é livre de conflito, o processamento de r é irreversível e r é processada somente uma vez. Verifica-se que no tempo t5, as requisições *r2*, *r3* e *r1* sofreram retrocesso. Contudo, todas as requisições já eram livres de conflito, pois, em t4, essas três requisições já haviam sido processadas.

Para que essa solução satisfaça todas as propriedades da especificação do problema GPI, é preciso que as réplicas tenham um referencial em comum capaz de garantir que algumas requisições não sofram retrocesso. Na nossa terminologia, esse referencial pode ser visto como o progresso de execução do grupo (GEP) dentro de $\prod$. Esse referencial, computado entre todos os módulos escalonadores corretos das réplicas, está relacionado ao progresso de execução local (LEP) de cada réplica em $\prod$, ou seja, o número de requisições já processadas. Através de GEP, todos os módulos escalonadores são capazes de detectar se uma determinada requisição pode ou não ser reordenada. GEP define dentro de *História*, uma separação entre requisições *estáveis* (requisições cujas posições em *História* nunca serão modificadas) e requisições *não estáveis* (requisições cujas posições em *História* poderão ser modificadas e, quando completadas, poderão sofrer retrocesso para evitar inversão de prioridade em grupo). Uma requisição *estável* quando processada é *irreversível*.

Para o cálculo de GEP, as réplicas executam um protocolo de acordo onde cada réplica

propõe um valor EP (*Execution Progress*) que representa o maior valor entre LEP e GEP, ou seja, EP = max(LEP, GEP). Esse cálculo garante que o GEP jamais diminua em nenhuma das réplicas. O protocolo aplica uma função a todos os EPs retornados pelas réplicas. Essa função é calculada em dois passos: primeiro, o subconjunto contendo os $f + 1$ maiores valores EPs é determinado. Em seguida, o valor de GEP é obtido, extraindo o menor valor contido dentro do 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.

A Figura 3.4 ilustra o problema de respostas inconsistentes. Nesse problema os clientes poderão observar respostas inconsistentes bastando que a réplica mais rápida falhe logo após enviar suas resposta para um cliente. A réplica que falhou não poderá participar de um acordo caso uma atividade de maior prioridade seja requisitada. Dessa forma a visão local da réplica que falhou, não será incluída no próximo acordo que ocorrer entre as réplicas.

A Figura 3.4 mostra três réplicas: r1, r2 e r3 que modificam seu estado conforme a chegada e o processamento de requisições nos tempos t1, t2, t3, t4 e t5. No tempo *t1*, uma determinada requisição 6 de prioridade 6 é iniciada e completada. No tempo *t2*, a réplica r1 retorna o resultado da requisição 6 para o cliente. No tempo *t3*, a réplica r1 falha e uma requisição de prioridade 7 entra no buffer das replicas r2 e r3. No tempo *t4*, o módulo escalonador das réplicas r2 e r3 adiciona essa requisição de prioridade 7 na lista *pronta* e reordena a lista conforme as prioridades das requisições. Para finalizar, no tempo *t5*, as réplicas r2 e r3 completam a requisição de prioridade 7. Observa-se que quando as réplicas r2 e r3 forem processar a requisição de prioridade 6, ela não será iniciada a partir do mesmo estado que foi iniciada pela réplica r1, pois em t1 a requisição 7 não chegou a ser processada na réplica r1. Desse modo o processamento da requisição 6 pelas réplicas r2 e r3 poderá gerar um resultado diferente do que foi gerado pela réplica r1.

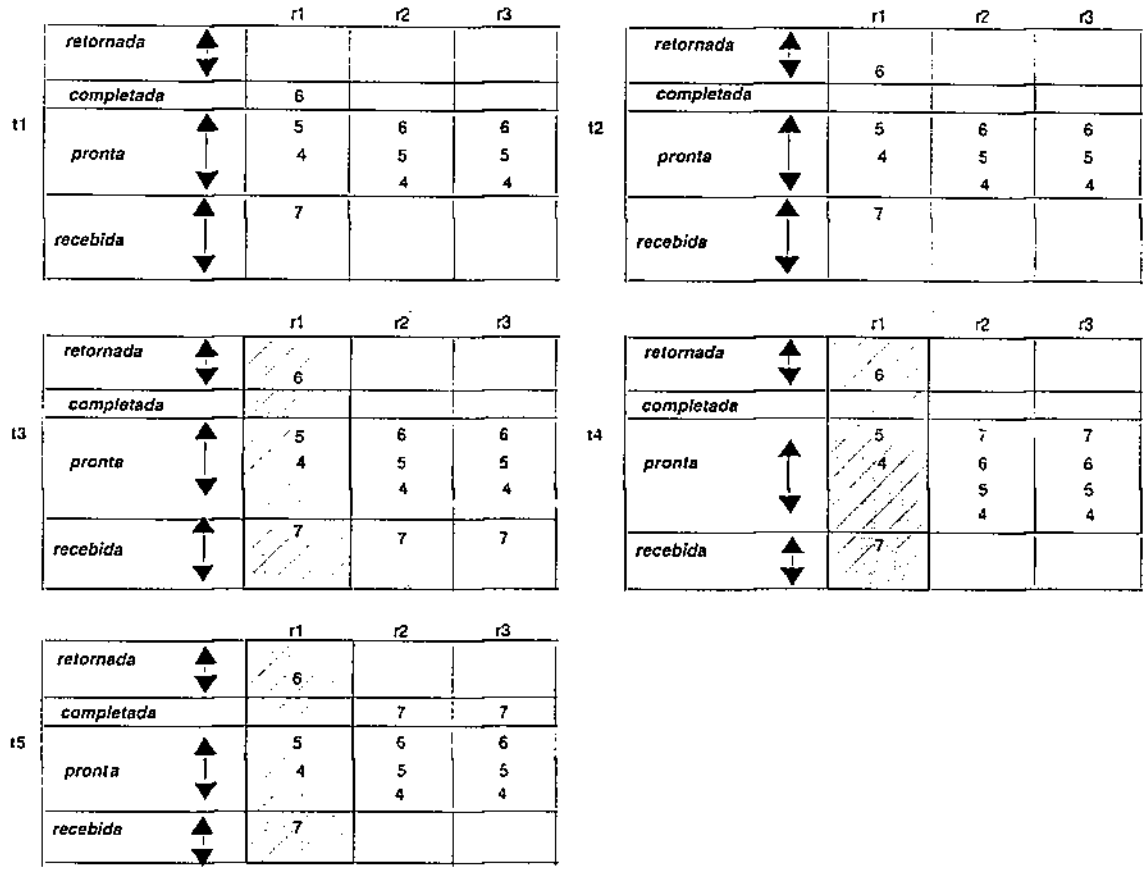


Figura 3.4: Respostas inconsistentes para uma mesma requisição

3.5 Modelo

3.5.1 Modelo de Replicação e Abstrações

Assumimos um servidor replicado formado por um conjunto de réplicas que devem executar requisições emitidas por *clientes*. As requisições emitidas por clientes deverão ser executadas segundo um critério de prioridade. No nosso caso, esse critério se refere à importância que determinados clientes possuem perante a aplicação. O acesso a um recurso na réplica e a comunicação entre componentes (clientes e réplicas) é realizada por chamadas a procedimentos remoto. Internamente as réplicas se comunicam por troca de mensagens.

Por utilizar a técnica de replicação ativa como estratégia de redundância, falhas são mascaradas dos clientes, pois as réplicas falham independentemente uma das outras, passando pelas mesmas mudanças de estado. Portanto, na falha de uma ou mais réplicas, uma invocação ao servidor tolerante a falhas irá ter sucesso mesmo que algumas das réplicas falhem antes de ter completado a tarefa.

Para atenuar a complexidade da replicação, utilizamos a noção de *grupo* [2] (de servi-

dores) e a noção de *primitivas de comunicação em grupo* [2]. A noção de *grupo* funciona como um mecanismo de endereçamento lógico, permitindo que clientes ignorem o grau de replicação e a identidade de cada umas das réplicas envolvidas na computação. *Primitivas de comunicação em grupo* oferecem comunicação um-para-muitos com várias semânticas. Essas semânticas escondem grande parte da complexidade de manter um servidor replicado consistente. Uma primitiva essencial nesse projeto é a *difusão atômica*. Segundo Schneider [14], formalmente, uma *difusão atômica* é uma *difusão confiável*[1] que satisfaz o seguinte requisito de *ordenação total*: dados dois processos corretos p e q onde ambos entregam as mensagens m e m', se p entrega m antes que m' então q também entrega m antes que m'.

3.5.2 Modelo Funcional

Segundo os autores em [30], um protocolo de replicação pode ser descrito utilizando cinco fases genéricas mostradas na Figura 3.5. Essas fases representam passos importantes que podem ser utilizados para caracterizar diferentes abordagens de replicação. Dependendo da técnica de replicação utilizada, as fases podem estar ordenadas de diferentes formas, mescladas ou unidas em uma outra seqüência, ou com alguns passos removidos. No caso da replicação ativa, as cinco fases são descritas a seguir:

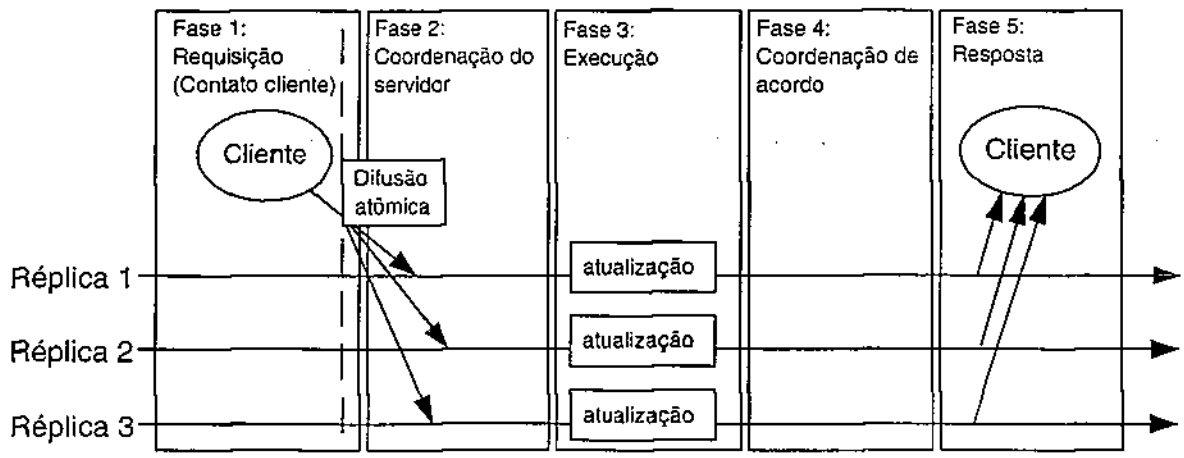


Figura 3.5: Fases genéricas da replicação ativa

Fase 1 (Requisição): o cliente envia uma requisição aos servidores utilizando *difusão atômica*. Isso pode ser realizado de duas formas: o cliente pode enviar a requisição para todas as replicas ou

[1] Se um processo dissemina uma mensagem para os demais processos no sistema, essa difusão deve ser confiável, ou seja, a mensagem enviada deve ser recebida por todos os nodos operacionais.

o cliente pode enviar a requisição para umas das replicas que por sua vez deverá enviar a requisição para as demais réplicas.

Fase 2 (Coordenação do servidor): as réplicas do servidor replicado tentam definir a ordem em que as requisições precisam ser processadas. Essa ordem se baseia na propriedade de *ordenação total* da *difusão atômica* para sincronizar a execução das requisições.

Fase 3 (Execução): a requisição é executada nas réplicas na ordem definida na fase 2.

Fase 4 (Coordenação de acordo): não há coordenação, pois todas as réplicas processam as mesmas requisições na mesma ordem e produzem as mesmas saídas.

Fase 5 (Resposta): todas as réplicas enviam suas respostas para o cliente. Se uma resposta poder ser inválida (por exemplo, quando falhas por valor [7] são consideradas), as respostas de todas as réplicas são submetidas a um elemento votador, que realiza votação majoritária para decidir o resultado final do processamento.

No nosso projeto utilizamos essas mesmas fases, porém, como precisamos evitar inversão de prioridade em grupo segundo [29] e alocar recursos segundo critérios de prioridades [18], as fases 2, 3 e 4 são modificadas e podem ocorrer várias vezes para cada requisição. Essas fases são projetadas de modo a oferecer transparência *de localização* (mascarando a localização física de serviços), *de acesso* (mascarando qualquer diferença na representação e mecanismo de invocação de uma operação), *de replicação* (mascarando redundância) e *de falha* (mascarando recuperação de serviços após falhas). Esses níveis de transparência são alcançados através de um ambiente com as seguintes funcionalidades:

- **Difusão atômica**: envio de requisições a um grupo de réplicas, de forma que sejam entregues e ordenadas atomicamente;
- **Reordenação de requisições por prioridade**: reordenação da lista de requisições resultantes da difusão atômica segundo um critério de prioridade;
- **Transformação de prioridades**: prioridades definidas pelos clientes são decodificadas para uma prioridade associada no lado servidor;
- **Reordenação de requisições evitando inversão de prioridade em grupo**: reordenação das requisições de modo a evitar inversão de prioridade em grupo e garantir consistência no grupo replicado;
- **Seleção de requisição para execução**: seleção da requisição de maior prioridade para execução;
- **Despacho de requisições**: despacho da requisição para processamento;
- **Processamento de requisições**: interpretação das informações contidas na requisição e iniciação da operação solicitada;

- **Acesso a recursos remotos**: obtenção da referência de recursos remotos e execução de operações utilizando a referência obtida;
- **Atualização do valor de *progresso de execução do grupo*** [29]: troca de informações com todas as réplicas a fim de entrar em acordo sobre um valor de *progresso de execução do grupo*;
- **Envio do resultado de requisições *estáveis***: envio da resposta de cada requisição a um mecanismo validador sempre que elas se tornarem *estáveis*;
- **Validação dos resultados**: eliminação de duplicidades das respostas para os clientes correspondentes.

Essas funcionalidades foram agrupadas da seguinte forma em nossa abordagem: na fase 1 (Requisição) temos a *difusão atômica*; na fase 2 (Coordenação do servidor) temos a *reordenação de requisições por prioridade*, a *transformação de prioridades*, a *reordenação de requisições evitando inversão de prioridade em grupo*, a *seleção de requisição para execução* e o *despacho de requisições*; na fase 3 (Execução) temos o *acesso a objetos remotos* e o *processamento de requisições*; na fase 4 (Coordenação de acordo) temos a *atualização do valor de progresso de execução do grupo* e o *envio do resultado de requisições estáveis* e na fase 5 (Resposta) temos a *validação de resultados.*
Essa nova abordagem está ilustrada na Figura 3.6. Observa-se que as fases 2,3 e 4 podem ocorrer n vezes para uma requisição. O valor de n é igual ao número de *interrupções* mais o número de retrocessos ocorridos antes da requisição se tornar *estável.*

3.6 Escalonando Requisições no *EROPSAR*

3.6.1 Política de Prioridade

Conforme descrito na seção 3.5.1, as requisições emitidas por clientes deverão ser executadas segundo um critério de prioridade. Através de interfaces que deverão ser implementadas pela aplicação, a prioridade da requisição é inicialmente igual ao identificador do cliente e quando a requisição é recebida no servidor, sua prioridade é atualizada para um valor armazenado em um banco de dados. Uma vez que o identificador do cliente é utilizado como chave para consultar a prioridade de uma requisição no servidor, assumimos que o *middleware EROPSAR* utiliza o modelo de política de prioridade *client priority propagation* (prioridade propagada pelo cliente) da especificação RT-CORBA [21]. Nesse modelo de política de prioridade, a requisição é executada na prioridade requisitada pelo cliente e

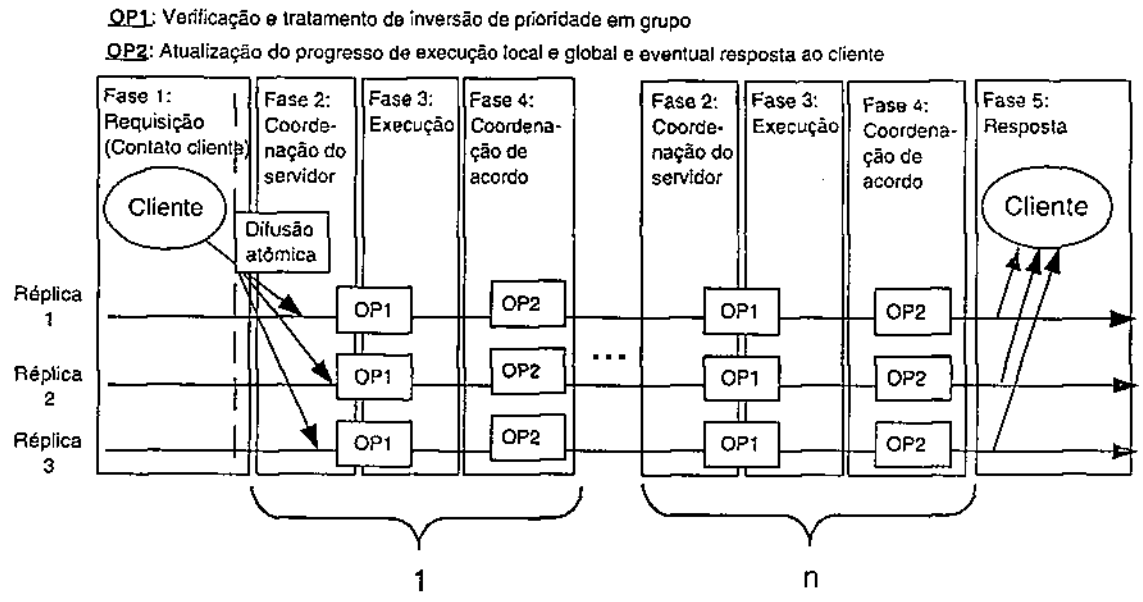


Figura 3.6: Fases da replicação ativa no projeto EROPSAR

é codificada como parte da requisição do cliente. A decodificação da prioridade no lado servidor se baseia na interface *PriorityTransform* da especificação RT-CORBA.

Quando duas requisições tiverem a mesma prioridade, a requisição que tiver sido enviada primeiro será a escolhida, seguindo portanto, uma ordenação *FIFO* (First-In-First-Out) [28].

3.6.2 Estados de uma Requisição

Uma requisição, durante seu tempo de vida, entre o envio do cliente ao servidor até o envio do seu resultado ao cliente, deve passar pelos seguintes estados: *não recebida, recebida, pronta, despachada, completada e retornada.*

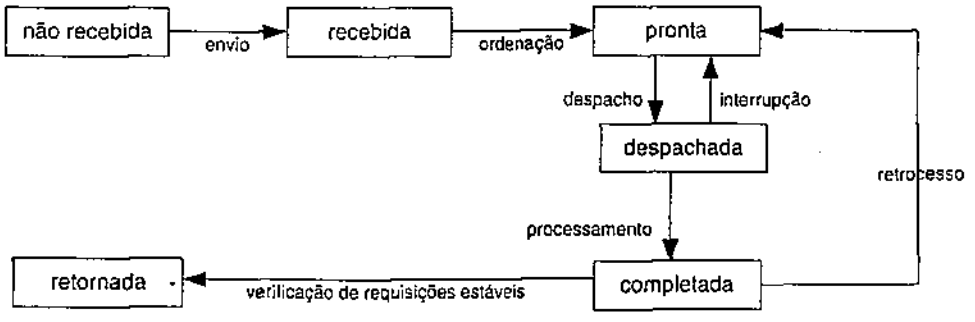


Figura 3.7: Estados das requisições no EROPSAR

Para que uma requisição passe de uma estado para outro, é necessário que ocorra uma

transição (*operação interna do middleware*). A transição por sua vez é disparada por um evento particular como ilustrado na Figura 3.7. A Figura mostra as sete transições que modificam o estado das requisições: *envio, ordenação, despacho, processamento, interrupção, retrocesso e verificação de requisições estáveis.* O *EROPSAR* utiliza um escalonamento preemptivo onde o processamento de requisições pode ser interrompido ou até mesmo sofrer retrocesso para o tratamento de *inversão de prioridade em grupo.* Isso quer dizer que uma requisição pode passar pelo mesmo estado mais que uma vez.

3.7 Serviços de Suporte

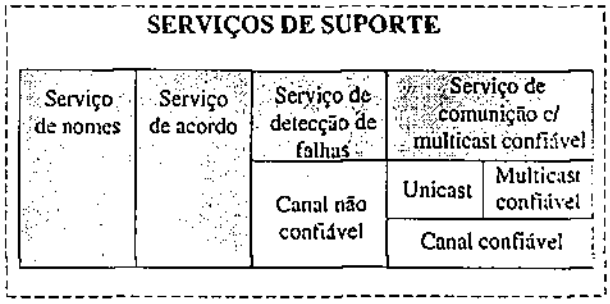


Figura 3.8: Serviços de suporte

Os serviços de suporte oferecem um nível de abstração que facilitam 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, ilustrados na Figura 3.8, 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 necessários para a construção do nosso *middleware*.

3.7.1 Serviço de Nomes

O *serviço de nomes* permite a localização de recursos em um sistema distribuído para que possam ser utilizados a partir de uma referência. A utilização do *serviço de nomes* envolve normalmente duas fases: registro de um nome que fica atribuído a um dado recurso (*binding*) e obtenção da referência para o recurso correspondente a um dado nome (*lookup* e *resolve*) [20]. A primeira fase é normalmente efetuada pelo próprio servidor e a segunda por clientes que pretendem usar os serviços oferecidos pelo servidor com um dado nome.

No nosso contexto, o servidor deve registrar no *serviço de nomes* os recursos que 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*.

3.7.2 Serviço de Acordo

Em várias situações em um sistema distribuído, problemas de acordo (*agreement problems*) precisam ser resolvidos. Eles constituem uma classe fundamental de problemas que seguem um mesmo padrão, onde os processos envolvidos em uma dada computação, precisam entrar em acordo para realizar uma determinada tarefa. A natureza da decisão do acordo depende do tipo de problema a ser solucionado. Essa classe de problema abrange os problemas de *difusão ordenada* [24], *eleição de líder* [12] e *comprometimento atômico* [13].

O problema de consenso oferece uma abstração para a maioria dos problemas de acordo. Nele, cada processo propõe um valor, e todos os processos não falhos devem entrar em acordo sobre uma única decisão. Esta precisa ser igual a um dos valores propostos. Diversas soluções para os problemas de acordo utilizam o problema de consenso como um elemento fundamental, pois em algum momento, todas as soluções dependem de uma decisão unânime, no caso, o consenso. Uma maneira de solucionar os problemas de acordo, é primeiro prover uma solução para o problema de consenso e então reduzir cada problema de acordo a um consenso. Essa redução foi demonstrada em [6], onde se provou que o problema de difusão ordenada e o de consenso são problemas equivalentes em sistemas distribuídos assíncronos sujeitos a falhas por parada. [17] e [26], por sua vez, mostraram que a solução para replicação ativa precisa de difusão ordenada.

O projeto do *EROPSAR* se baseia em um *serviço de acordo* para realização dos acordos de *ordem* (na Subseção 3.3) e *progresso de execução do grupo* (na Seção 3.4). Como a chegada de requisições no servidor dá-se de forma contínua[2], 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, conforme descrito na Subseção 3.4, 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.

[2]Novas requisições são recebidas enquanto as réplicas ainda estão em acordo na ordenação de requisições anteriormente recebidas

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 *middleware* poderá ser mais eficiente que se os acordos fossem executados um após o outro.

O *serviço de acordo* necessário ao nosso projeto deverá utilizar as informações providas por um *serviço de detecção de falha não confiável* [6] descrito na Subseção 3.7.3. Caso contrário, os processos envolvidos em um acordo podem ficar indefinidamente à espera das mensagens de um outro que está muito lento ou que falhou, impossibilitando a decisão do acordo [11].

3.7.3 Serviço de Detecção de Falhas

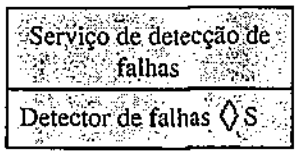


Figura 3.9: Serviço de detecção de falhas

O *serviço de detecção de falhas* necessário para o *EROPSAR* é baseado em detectores de falha não confiáveis [6]. Esses detectores fornecem informações de todos os processos suspeitos de terem sofrido uma falha. O modelo de Chandra e Toueg [6] 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 ◇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 [5], desde que a maioria dos processos não falhem. O detector ◇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.

3.7.4 Serviço de Comunicação com *Multicast* Confiável

O *serviço de acordo* precisa de um *serviço de comunicação* para que os processos envolvidos em um acordo se comuniquem. A comunicação é realizada através das primitivas *unicast* e *multicast* (difusão confiável). Os processos enviam mensagens para um outro processo através da primitiva *unicast* e difundem mensagens para todos os processos através da primitiva *multicast*. Essas primitivas são descritas a seguir.

Primitiva *unicast*: É uma operação não bloqueante, ou seja, o processo emissor, pode prosseguir sua execução assim que o conteúdo (dados a enviar) for copiado para o *buffer*[3] de envio. Trata-se de um *unicast* não confiável, ou seja, a mensagem possui um único destinatário e não existe garantia em sua entrega.

Primitiva *multicast*: É uma operação não bloqueante, ou seja, o processo emissor, pode prosseguir sua execução assim que a mensagem for copiada para o *buffer* de envio. Trata-se de uma difusão confiável sem noção de ordem, onde uma mensagem m é *entregue* em múltiplos destinos. Informalmente uma difusão confiável garante que: (1) todos os processos corretos *entregam* o mesmo conjunto de mensagens, (2) todas as mensagens difundidas por um processo correto são *entregues* e (3) uma mensagem adulterada jamais será *entregue*.

[3]Dispositivo ou área de armazenamento temporário de informações/dados durante a transferência desses dados de uma parte do sistema para outra.

Capítulo 4

Implementação do EROPSAR

Nesse capítulo apresentamos como implementamos o *middleware EROPSAR*. Inicialmente descrevemos a implementação dos serviços de suporte. Mostramos em seguida como as funcionalidades apresentadas no Capítulo 3 foram implementadas. Prosseguimos com alguns testes de funcionalidade, testes de tolerância a falhas e uma avaliação de desempenho. Concluímos o capítulo discutindo como uma aplicação deve ser configurada para utilizar o *middleware EROPSAR*

4.1 Introdução

Apresentamos a seguir alguns aspectos relacionados com a implementação do *middleware EROPSAR*. Os serviços de suporte, com exceção do *serviço de nomes*, foram também implementados. O *serviço de acordo* baseou-se em GAF (General Agreement Framework), uma estrutura genérica de acordo proposta em [15]. A fim de tornar o *middleware* mais eficiente, estendemos GAF para que dois acordos possam ser realizados em conjunto através de uma única execução do protocolo. A extensão de GAF (E-GAF) se baseia no protocolo de consenso proposto por Chandra e Toueg [6]. Esse protocolo utiliza um serviço de comunicação construído sobre canais confiáveis (oferecido na implementação do *serviço de comunicação com multicast confiável*) e um detector de falha $\diamond$S (oferecido na implementação do *serviço de detecção de falha*).

As cinco fases funcionais apresentadas no Capítulo 3 foram implementadas através de três camadas: *camada de requisição e resposta, camada de coordenação e acordo do servidor* e *camada de execução.*

Todo o projeto foi implementado em JAVA (JDK 1.4.1) e o ambiente utilizado para desenvolvimento foi o JBuilder Personal 7.0. A avaliação experimental foi realizada em

computadores com processador 1.2GHz com 256Mb de RAM, com sistemas operacionais Windows XP e Linux (SuSE 8.0) conectados por uma rede Ethernet 10Mbps.

4.2 Serviços de Suporte

4.2.1 Serviço de Nomes

Como nosso projeto é implementado em JAVA, o *serviço de nomes* oferecido no JDK 1.4.1 atendeu às nossas necessidade. Utilizamos, portanto, a ferramenta *rmic* para gerar *stubs* e *skeletons* dos objetos remotos e a ferramenta *rmiregistry* para serviço de registro de objetos remotos. Através dessas ferramentas e da biblioteca *java.rmi* foi possível registrar objetos e localizar referências de objetos remotos.

4.2.2 Serviço de Acordo

Nosso serviço de acordo que implementamos se baseia no algoritmo em [15]. Uma grande vantagem desse serviço é que não se modifica o protocolo de consenso para adequá-lo a um determinado protocolo de acordo, pois GAF permite que as particularidades de um acordo sejam configuradas através de seis parâmetros versáteis (GET, $\prec$, $\mathcal{F}$, $ACCEPTABLE$, $EXCUSED$ e $EARLY$) mostrados adiante, enquanto que a estrutura do consenso permanece inalterada. O uso dessa estrutura permite que muitas requisições possam ser ordenadas em uma única execução do protocolo de acordo e permite também que uma decisão seja obtida aplicando uma função a diversos valores de entrada (necessários para cálculo do *progresso de execução do grupo*). Estendemos GAF para que dois acordos possam ser realizados em conjunto através de uma única execução do protocolo de acordo e denominamos essa extensão de E-GAF.

Assim como GAF, E-GAF é baseada no paradigma do coordenador rotativo, que avança em rodadas consecutivas assíncronas até que uma decisão seja alcançada [6]. Cada rodada r é coordenada por um processo Pc predeterminado, definido por $c = (r \bmod n) + 1$. Durante uma rodada r, um processo comunica-se exclusivamente com o coordenador Pc. O número de rodadas realizadas por cada processador é arbitrário: ele depende da ocorrência de falhas e também do comportamento dos módulos detectores de falha.

Como o protocolo avança de acordo com as mensagens enviadas e recebidas pelo coordenador, o protocolo utiliza o detector de falha para evitar que os processos envolvidos em um acordo fiquem indefinidamente à espera das mensagens de um coordenador que está muito lento ou que falhou. Quando um processo, através de um detector de falhas,

verifica que o coordenador é suspeito de ter falhado, ele avança para uma nova rodada e elege um novo coordenador, dessa maneira o protocolo avança uma rodada de decisão onde o coordenador possui um valor de decisão e não é suspeito por nenhum processo correto.

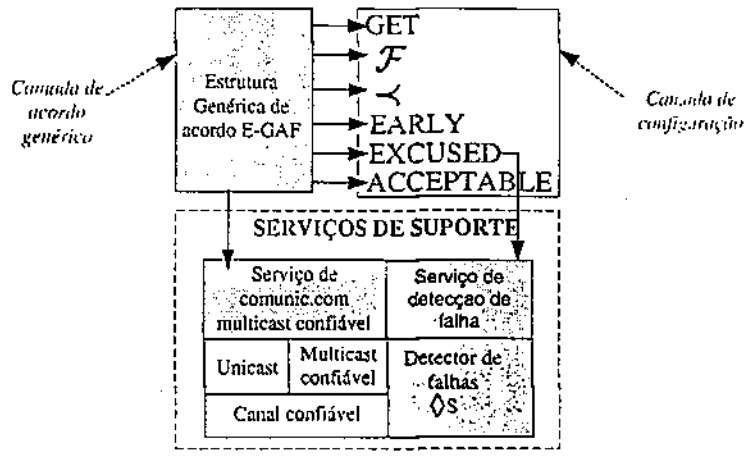


Figura 4.1: Estrutura genérica de acordo E-GAF

E-GAF é formada por uma *camada de configuração* e por uma *camada de acordo genérico*. A *camada de configuração* implementa os parâmetros versáteis e *camada de acordo genérico* o protocolo de consenso. E-GAF utiliza o *serviço de comunicação com multicast confiável* para comunicação com as réplicas de um grupo e o *serviço de detecção de falha* (detector de falha ◇S) para verificar se um processo é suspeito de falha. A estrutura de E-GAF pode ser visualizada na Figura 4.1.

A *camada de acordo genérico* mostrada no Algoritmo 1 é uma adaptação do Algoritmo apresentado pelos autores em [15]. A principal diferença, está em informar como argumento, uma lista contendo um ou mais acordos. Na linha 1, armazenamos em β a lista informada; nas linha 9,10,18,27,28,29,31, adicionamos β como argumento para os parâmetros versáteis *GET*, $\prec$, $\mathcal{F}$, *ACCEPTABLE*, *EXCUSED* e *EARLY* respectivamente. Descrevemos a seguir a função de cada parâmetro versátil.

- **Função GET**: quando o protocolo E-GAF é iniciado, a função GET (linha 9) solicita continuamente à camada superior que novas propostas sejam fornecidas (linhas 8-11). Essa função retorna as propostas relacionadas a um determinado problema de acordo. À medida que o protocolo avança, existirá um momento (linha 19) em que os processos não poderão mais enviar suas propostas ao coordenador e portanto, o valor da proposta não poderá ser mais alterado e GET não é mais invocado.

- **Função $\prec$**: quando a função GET é chamada duas ou mais vezes, a camada superior pode fornecer o mesmo valor ou então um outro valor mais significante (linha 10). A

função $\prec$ expressa a relação de significância entre 2 valores. Na linha 27, essa função verifica se a nova estimativa recebida "*est*" é menos significante que uma outra, do mesmo processo, previamente armazenada "$est\_from_i[j]$". A relação entre esses valores, quanto a sua significância, está intrinsecamente relacionada ao problema de acordo. Essa função implementada pela camada superior, utiliza duas propostas como argumento, selecionando a de maior significância.

- **Função $\mathcal{F}$**: no protocolo E-GAF, um processo coordenador deverá em algum momento, apoiado por um serviço de detecção de falha (linha 45), receber a maioria de estimativas (linha 29), para poder definir um valor de decisão. O valor de decisão é calculado pela camada superior que implementa a função $\mathcal{F}$ (linha 28). A função $\mathcal{F}$ possui como argumento: *i)* os valores que foram propostos por processos; *ii)* um valor identificado como $\perp$ para os processos que não enviaram suas propostas (linha 1).

  O valor retornado por $\mathcal{F}$ pertence a um conjunto predefinido de possíveis valores de saída.

- **Função ACCEPTABLE**: o conjunto dos possíveis valores de saída é formado por dois subconjuntos: um subconjunto contém valores que são considerados aceitáveis, enquanto o outro contém somente valores não aceitáveis. Essa função aplicada a um valor de saída, retorna *verdade* quando o valor de saída pertence ao subconjunto de valores aceitáveis (linhas 18 e 19), retornando falso para o outro subconjunto (linha 20). Essa função é implementada na camada superior, sendo chamada quando um processo recebe do coordenador da rodada atual uma nova estimativa. A nova estimativa é aplicada à função e dependendo de seu resultado, o processo envia uma confirmação positiva (ACK) ou negativa (NACK) ao coordenador. Se a maioria dos processos retornarem ACKs ao coordenador, ele então decide (linhas 37 e 38). Quando a quantidade das confirmações (ACKs e NACKs) for igual a maioria, o coordenador inicia uma nova rodada (linha 42).

- **Função EXCUSED**: o coordenador só poderá enviar uma nova estimativa a todos os processos no momento que tiver recebido a maioria das estimativas dos processos envolvidos em um acordo (linha 29). Nesse momento, dependendo do problema de acordo a ser resolvido, pode ser que as estimativas ainda pendentes sejam essenciais para resolução do acordo. A função EXCUSED, implementada pela camada superior, é chamada pelo coordenador, retornando *verdade* ou *falso*. A cada estimativa recebida, o coordenador invoca EXCUSED, enviando o processo emissor como argumento. O retorno de EXCUSED como *falso*, indica que o coordenador deve continuar a receber as estimativas pendentes, e como *verdade*, indica que o coordenador já pode enviar a nova estimativa caso tenha obtido estimativas suficientes. Essa função pode utilizar a informação retornada por um módulo detector de falha ou utilizar mecanismos de temporizarão para retornar seu valor.

- **Função EARLY**: alguns problemas de acordo são caracterizados pela possibilidade de

decidir antecipadamente. Dependendo do problema de acordo a ser resolvido, é possível que, em um dado momento, a coleta de estimativas adicionais pelo coordenador não altere o retorno da função $\mathcal{F}$. Portanto a função EARLY (linha 31), é utilizada pelo coordenador da rodada para verifica se as estimativas recebidas já são suficientes para suspender a coleta das estimativas dos processos restantes. Se este for o caso, o coordenador pode enviar uma nova estimativa a todos os processos (linha 32). A função EARLY é implementada pela camada superior, e utiliza as estimativas dos processos como argumento.

As funções **EXCUSED** e **EARLY** funcionam de forma semelhante: ambas possibilitam que um acordo possa ser decidido antecipadamente conforme os valores previamente recebidos. Contudo uma função não substitui a outra. Pode ocorrer que a função **EARLY** retorne *verdade* sem que o protocolo possa ser finalizado, pois o problema sendo resolvido pode necessitar que no mínimo uma quantidade especifica de valores sejam recebidos. Da mesma maneira, pode ocorrer que a função **EXCUSED** retorne *verdade* sem que o protocolo possa ser finalizado, caso o problema não possa determinar um resultado final com os valores até então recebidos (comprometimento atômico).

Algoritmo 1 *camada de acordo genérico*

```
consenso(lista de acordos a serem resolvidos)
início
(01) β ← acordos a serem resolvidos; r_i←0; new_round_i←verdade; est_i←⊥; ts←0; est_from_i←{⊥,⊥,...,⊥};
(02)  enqto (verdade) faça %O loop vai da linha 2 até a linha 48%
(03)    se (new_round_i) então %Inicializa as variáveis da rodada de p_i%
(04)        new_round ← falso; r_i ← r_i +1; c←(r_i mod n) + 1; phase1_begin←verdade;
(05)        se (i=c) então %Inicializa as variáveis da rodada do coordenador%
(06)          received_from_i←0; tsm_i←0; phase2_end_i←falso; accept_i←0; reject_i←0;
(07)    fim-se fim-se;
(08)    se (ts_i=0) então %O valor proposta pela camada superior pode ser modificado%
(09)        est_from_i[i]← GET (β); %Coleta uma nova proposta%
(10)        se(≺(est, est_from_i[i], β)) então est_i←est_from_i[i]; phase1_begin_i←verdade; fim-se;
(11)    fim-se;
(12)    se(phase1_begin_i) então envie(ESTIMATE<p_i,t_i,est_i,ts_i>) para p_c; phase1_begin←falso fim-se;
(13)    se uma mensagem m (como definida abaixo) foi recebida então
(14)     caso m seja
(15)      m=DECISION<j,est> {m vem de qualquer p_j}
(16)        envie(DECISION<i,est>) para todos exceto p_i,p_j; retorna(est);
(17)      m=NEW_ESTIMATE<c,t,new_est> tal que r←r_i {m vem de p_c}
(18)        se (ACCEPTABLE(new_est, β)) então
(19)          est_i←new_est; ts_i←r_i; envie(VOTE<i,r_i,ack>) para p_c %p_i aceita new_est%
(20)        senão envie (VOTE<i,r_i,nack>) para p_c %p_i rejeita new_est%
(21)        fim-se;
(22)        se (i≠c) então new_round_i←verdade fim-se
(23)      m=ESTIMATE<j,r,est,ts> tal que r=r_i {m vem de qualquer p_j para p_c (i=c)}
(24)        se ¬(phase2_end_i) então
(25)          received_from_i←received_from_i∪j;
(26)          se (tsm_i<ts) então tsm_i←ts; new_est_i←est fim-se;
(27)          se ((tsm_i=0) e ¬(≺(est. est_from_i[j], β) )) então
(28)            est_from_i[j]←est; new_est_i← F(est_from_i, β) fim-se;
(29)          se ((|received_from_i| ≥ ⌈(n+1)/2⌉) e (∀ p_j: j ∈ received_from_i or EXCUSED(j, β)))
(30)              or
(31)              (EARLY(new_est_i, 3) e (tsm_i=0) ) então
(32)              envie (NEW_ESTIMATE<i,r_i,new_est_i>) para todos; phase2_end_i ←verdade;
(33)          fim-se fim-se;
(34)        m=VOTE<j,r,answer> tal que r=r_i m vem de qualquer p_j para p_c (i=c)
(35)          se (answer=ack) então
(36)            accept_i←accept_i ∪ {j}; %O coordenador p_i conta o número de confirmações positiva %
(37)            se (|accept_i|= ⌈(n+1)/2⌉) então
(38)              envie(DECISION<i,est_i>) para todos exceto {p_i}; retorna(est_i);
(39)            fim-se;
(40)          senão reject_i←reject_i ∪ {j} %O coordenador p_i conta o número de rejeições%
(41)          fim-se;
(42)          se (|accept_i ∪reject_i|=⌈(n+1)/2⌉) então new_round_i← verdade fim-se; %Previne Deadlock%
(43)      endcase
(44)    fim-se;
(45)    se (¬(new_round_i)) e (i≠c) e (p_c ∈ suspected_i)) então
(46)      envie(VOTE<p_i,r_i,nack>) para p_c; new_round_i←verdade;
(47)    fim-se
(48)  loop
fim
```

Para gerar o protocolo de acordo com as nossas necessidades, utilizamos a *camada de*

configuração. Os parâmetros dessa camada devem ser configurados para um acordo de ordem (*Acordo_ Ordem*) e um acordo de progresso de execução do grupo (*acordo_ GEP*). Apresentamos a seguir a configuração desses parâmetros. Os parâmetros apresentados utilizam um conjunto β como argumento. β está contido ou é igual ao conjunto {*Acordo_ Ordem, acordo_ GEP*}.

1. β = {**Acordo_Ordem**}
   (a) **GET**(β): se $EP \leq GEP$, **GET** retorna as requisições existentes em US, caso contrário retorna $\perp$ (valor não significativo);
   (b) $\prec(v, v', \beta)$: se $|v| <= |v'|$, $\prec$ retorna *verdadeiro*, caso contrário, retorna falso;
   (c) $\mathcal{F}$(est_from$_i$, β): retorna os valores em est_from$_i$ sem que exista repetições de um mesmo elemento;
   (d) **ACCEPTABLE**(new_est, β): sempre retorna *verdadeiro*;
   (e) **EXCUSED**(j, β): após n ms do início do consenso ou se j não for coordenador ou se |lista de suspeitos|>0, EXCUSED retorna *verdadeiro*, caso contrário, retorna falso;
   (f) **EARLY**(new_est$_i$, β): sempre retorna falso.
2. β = {**acordo_GEP**}
   (a) **GET**(β): retorna *EP*, ou seja, se $LEP < GEP$, **GET** retorna *GEP*, caso contrário, retorna LEP;
   (b) $\prec(v, v', \beta)$: se $v <= v'$, $\prec$ retorna *verdadeiro*, caso contrário, retorna *falso*;
   (c) $\mathcal{F}$(est_from$_i$, β): seleciona os $f + 1$ maiores valores retornados por *GET*, em seguida, retorna o menor valor do conjunto selecionado;
   (d) **ACCEPTABLE**(new_est, β): sempre retorna *verdadeiro*;
   (e) **EXCUSED**(j, β): após n ms do início do consenso ou se |lista de suspeitos|>0, EXCUSED retorna *verdadeiro*, caso contrário, retorna falso;
   (f) **EARLY**(new_est$_i$, β): sempre retorna falso.
3. β = {**Acordo_Ordem , acordo_GEP**}
   (a) **GET**(β): retorna os items 1(a) e 2(a) ao mesmo tempo;
   (b) $\prec(v, v', \beta)$: sempre retorna *verdade*;
   (c) $\mathcal{F}$(est_from$_i$, β): retorna os items 1(c) e 2(c) ao mesmo tempo;
   (d) **ACCEPTABLE**(new_est, β): sempre retorna *verdade*;
   (e) **EXCUSED**(j, β): após n ms do início do consenso, EXCUSED retorna *verdade* caso contrário, retorna *falso*;
   (f) EARLY(new_est$_i$, β): sempre retorna falso.

Em nosso projeto o *framework* E-GAF é implementado pela classe *E_ Gaf_ FW*. Os principais métodos dessa classe são:

- *void consensus(double pAgreements,String pFD,double pConsensusID):*
  Implementa a *camada de acordo genérico* de E-GAF e utiliza o método *teleInterpreter* para que uma mensagem do consenso seja devidamente interpretada.

- *void teleInterpreter(Object pTelegram):*
  Acessa os métodos da implementação da interface ***E_ GafConfig_ interface***, mostrada a seguir, através de um *callback*.

```
public interface E_GafConfig_interface {
   public  Vector _getProposal (double pAgreements);
   public boolean _LeftIsMoreSignificant (Object pProposal,
                                          Object pPreviousProposal,
                                          double pAgreements);
   public  Vector _F_custom_function_over_estimates (Vector pProposalList,
                                                     double pAgreements);
   public boolean _Early (Object pProposal, double pAgreements);
   public boolean _Excused (Object pReplica, double pAgreements);
      public boolean _Acceptable (Object pNewEstimate, double pAgreements);
   public void _decisionReaction (Vector pDecision, double pAgreements);
}
```

E_ GafConfig_ interface representa a camada de configuração de E-GAF. Essa interface é implementada pela camada superior, no nosso caso no *middleware*. Com exceção do método *_decisionReaction*, todos os métodos representam um dos parâmetros versáteis apresentados anteriormente. O método *_decisionReaction* permite que a invocação do acordo seja realizada de forma assíncrona. Desse modo, o solicitador do consenso poderá escolher em esperar o resultado do acordo ou em ser notificado que o consenso foi concluído.

4.2.3 Serviço de Detecção de Falhas

Para implementar nosso serviço de detecção de falha utilizamos como base um detector adaptativo para sincronismo parcial [6]. Este detector é provido de um mecanismo que procura melhorar a precisão das suspeitas dos detectores, realizando adaptações dinâmicas sobre o limite específico de tempo utilizado para suspeita de processos. Conforme a nomenclatura utilizada em [9], esse detector segue o modelo dos detectores *Push*. Nesse modelo, o fluxo de controle segue a mesma direção do fluxo das informações, ou seja, dos processos monitorados em direção ao detector. Para isso, em um detector *Push*, os processos monitorados necessitam estar ativos. Eles enviam periodicamente mensagens identificadas (conhecidas como *EuEstouOperacional*) que têm como função principal demonstrar que se

o processo está enviando mensagens, ele ainda está operacional. Se um detector de falhas não receber tais mensagens dentro de um limite específico de tempo, ele passa a suspeitar do processo monitorado.

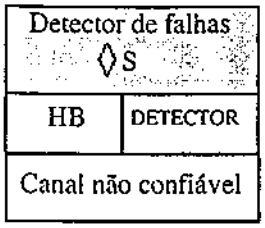


Figura 4.2: Estrutura do detector de falhas ◇S

Nesse contexto, nosso detector de falha ◇S (ilustrado na Figura 4.2) é formado por um HB (*heartBeat*) e um detector. O HB envia as mensagens *EuEstouOperacional* para os detectores, que mantém atualizada uma lista de processos suspeitos. Esta lista é disponibilizada à camada superior que utiliza o serviço. Por se tratar de um detector não confiável, onde falsas suspeitas são permitidas, ele é construído sobre canais não confiáveis (UDP).

O Algoritmo 2 descreve o funcionamento de nossa implementação. Inicialmente, a lista de processos suspeitos do detector é inicializada como vazia (linha 2), o tempo limite para suspeita de cada processo monitorado é ajustado para um valor padrão configurado pela camada superior (linha 4) e o tempo do último recebimento de *EuEstouOperacional* de cada processo monitorado é inicializado com a marca de tempo atual do relógio (linha 5). Os processos monitorados enviam a cada ΔHEARTBEAT unidades de tempo a mensagem *EuEstouOperacional* para os processos detectores (linhas 7 e 8). Os processos detectores por sua vez, podem atualizar a lista de suspeitos em duas ocasiões: na primeira, a cada ΔDETECTOR segundos é verificado, utilizando a última marca de tempo armazenada de um processo monitorado não pertencente a lista de suspeitos, se seu limite de tempo foi expirado, quando esse for o caso, o processo é adicionado à lista de suspeitos (linhas 11 e 12); na segunda ocasião, sempre que o detector recebe a mensagem *EuEstouOperacional* (linha 17) a marca de tempo do recebimento é armazenada (linha 18) e é verificado se o emissor pertence à lista de suspeitos, se esse for o caso, o emissor é removido da lista e seu tempo limite é incrementado (linhas 19-21).

Aplicações que utilizarem o detector de falha poderão verificar se um processo é suspeito, através de uma função que possui como argumento o nome do processo monitorado ou quando necessário poderão também obter toda a lista de processos suspeitos. O uso do detector de falha pelo serviço de acordo pode ser visualizado na linha 45 do Algoritmo 1.

Comm_Layer(). O envio de uma mensagem a um único destinatário é realizado através do método *send(InetAddress pDestinatario, Object pMsg)*. Para enviar uma mensagem a múltiplos destinatários, utiliza-se o método *multicast(Vector pDestinatarios, Object pMsg)*. Quando esses métodos são invocados pela primeira vez, *Comm_Layer* cria uma conexão com o(s) destinatário(s) e armazena as informações da conexão construída em um vetor para posterior reutilização. Se a conexão for perdida, ela será recriada na próxima invocação.

Para que um objeto possa receber e, opcionalmente, enviar mensagens, ele precisa informar ao construtor a *porta de escuta*. Isso é realizado através dos seguintes contrutores: *Comm_Layer(int pPortaDeEscuta)* e *Comm_Layer(int pPortaDeEscuta, Object pCallback)*. O construtor *Comm_Layer(int pPortaDeEscuta)* permite que as mensagens sejam recebidas através do método *receive(long pTimeout)* que deve ser invocado a critério do objeto instanciador. O construtor *Comm_Layer(int pPortaDeEscuta, Object pCallback)* faz com que as mensagens sejam recebidas através de um *callback*, para isso o objeto instanciador deve passar sua referência como parâmetro e implementar a interface *RequestHandler_interface* descrita a seguir:

```
public interface RequestHandler_interface  {
       public void handle(String pFromHost, String pFromPort,  Object pMsg) ;
}
```

Um objeto emissor de mensagens pode estabelecer quantas conexões desejar com um objeto receptor de mensagens. Para que isso seja possível, basta que o receptor utilize diferentes *portas de escuta* na máquina. Para evitar o desperdício de múltiplas conexões, *Comm_Layer* permite que uma mesma conexão seja compartilhada por diferentes módulos de uma mesma aplicação. Isso é possível através dos métodos *use_shared_channel(int pPortaDeEscutaVirtual)* e *use_shared_channel(int pPortaDeEscutaVirtual, Object pCallback)*. Esse método permite a criação de *portas de escutas virtuais* a partir de uma porta de escuta real pré-existente. Para enviar mensagens utilizando uma conexão compartilhada o emissor deve adicionar o parâmetro *pPortaDeEscutaVirtual* aos métodos *send* e *multicast* anteriormente mencionados.

A Figura 4.3 ilustra o funcionamento de uma conexão compartilhada. O objeto 1 da máquina A acessa o objeto 1 da máquina B sem *portas de escuta virtual*. O objeto 2 da máquina A acessa o objeto 2 da máquina B através da *porta de escuta virtual* 1 e acessa o objeto 3 da máquina B através da *porta de escuta virtual* 2.

Os principais métodos dessa classe são:

- *Comm_Layer(int pPortaDeEscuta)*:
  Inicia escuta de mensagens na porta *pPortaDeEscuta*;

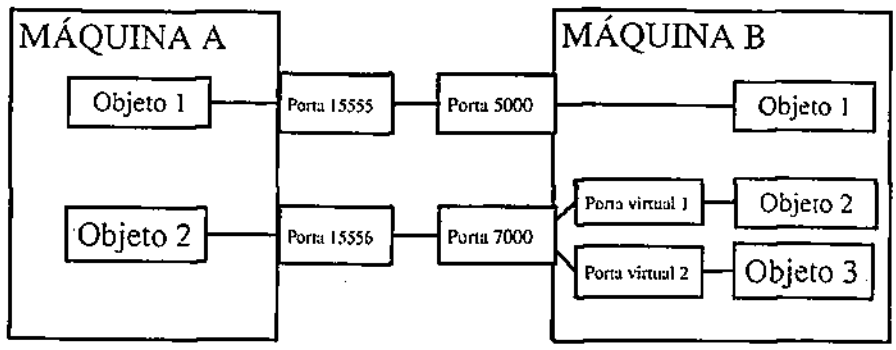


Figura 4.3: Compartilhamento de conexão através de porta de escuta virtual

- *Comm_Layer(int pPortaDeEscuta, Object pCallback)*:
  Inicia escuta de mensagens na porta *pPortaDeEscuta* e registra *pCallback* para notificar a chegada de novas mensagens;

- *void use_shared_channel(int pPortaDeEscutaVirtual, Object pCallback)*:
  Cria um *porta de escuta virtual* caso não exista e registra *pCallback* para notificar a chegada de novas mensagens;

- *boolean send(InetAddress pDestinatario, Object pMsg)*:
  Implementa a primitiva *unicast* e se baseia nas propriedades do canal confiável TCP. Envia a mensagem *pMsg* em uma *porta de escuta* existente no destinatário *pDestinatario*;

- *boolean send(InetAddress pDestinatario, int pPortaDeEscutaVirtual, Object pMsg)*:
  Envia a mensagem *pMsg* em uma *porta de escuta virtual pPortaDeEscutaVirtual* existente no destinatário *pDestinatario*;

- *Vector multicast(Vector pDestinatarios, Object pMsg)*:
  Implementa a primitiva *multicast*, onde um processo p envia uma mensagem *pMsg* para todas os processos de um grupo (*pTargets*) incluindo p. Quando essa mensagem é recebida pela primeira vez no destino, se o emissor de *pMsg* for diferente de p então *pMsg* é enviado para todos os processos do grupo e em seguida a mensagem *pMsg* é entregue. Quando um processo recebe mais de uma vez uma mensagem *pMsg*, *pMsg* é então descartado. Esse método não garante atomicidade. A atomicidade é garantida por um protocolo de acordo, apresentado na Subseção 4.2.2, que utiliza o método *multicast*.

- *Vector multicast(Vector pDestinatarios, int pPortaDeEscutaVirtual, Object pMsg)*:
  Mesmo método *multicast* descrito no item anterior, porém as mensagens *pMsg* são enviadas em uma *porta de escuta virtual* existente no destinatário *pDestinatario*;

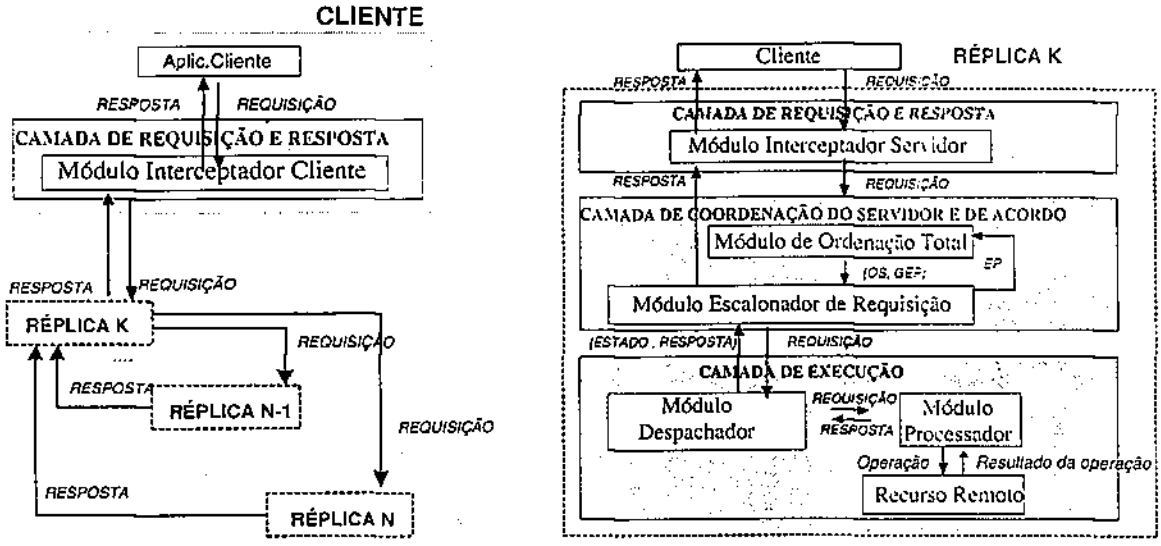
- *Object receive(int pTimeout):*
  Permite a coleta das mensagens entregues pelo protocolo de comunicação. Trata-se de uma operação bloqueante, isto é, o processo receptor fica bloqueado até a chegada da mensagem ou até a expiração de um timeout especificado;

- *Object receive(String pPortaDeEscutaVirtual, int pTimeout):*
  Mesmo método *receive* descrito no ítem anterior, porém a coleta de mensagens ocorre através de uma *porta de escuta virtual*;

- *void handle(String pFromHost, String pFromPort, Object pMsg):*
  Para que a recepção de uma mensagem não cause obrigatoriamente o bloqueio do processo, o serviço de comunicação oferece um serviço que alerta a chegada de uma nova mensagem. O método *handle* permite que as mensagens sejam recebidas de modo não bloqueante. O processo receptor no início de sua execução informa ao protocolo de comunicação como deve ser notificado sobre novas mensagens. Uma vez, informado, o processo passa a receber as mensagens através de um *callback*;

4.3 Arquitetura do EROPSAR

As cinco fases funcionais apresentadas no Capítulo 3 foram implementadas através de uma arquitetura em três camadas: *camada de requisição e resposta*, *camada de coordenação e acordo do servidor* e *camada de execução*. Essas camadas são apoiadas pelos serviços de suporte apresentados na Subseção 3.7. A Figura 4.4(a) ilustra a *camada de requisição e resposta* do lado cliente e sua interação com n réplicas. A Figura 4.4(b) mostra a *camada de coordenação e acordo do servidor* e a *camada de execução*, existentes em uma réplica K, onde $0<K\leq n$. Descrevemos a seguir os módulos internos dessas camadas, esses módulos estão também ilustrados na Figura 4.4.



(a) Interação cliente-réplicas (b) Visualização interna da réplica

Figura 4.4: Arquitetura *EROPSAR*

4.3.1 Camada de Requisição e Resposta

A camada de requisição e resposta implementa as funcionalidades da fase 1 e da fase 5 apresentadas na Seção 3.5.2. Essa camada é formada pelo módulo interceptador cliente (reside no lado cliente) e pelo módulo interceptador servidor (reside no lado servidor).

Módulo interceptador Cliente e Módulo Interceptador Servidor

Esses módulos, apresentado na Figura 4.4, conectam a aplicação cliente ao *middleware EROPSAR*. O Algoritmo 3 mostra o pseudo-código do módulo *interceptador cliente*. O

interceptador cliente, disponibilizado através de um *Java Applet* (Applet_interceptor), inicialmente localiza um dos *interceptadores servidores* corretos (linhas 02-05) para que as requisições da aplicação cliente sejam difundidas nas réplicas (linha 12-15). O resultado pode ser obtido através do método *getReply* (linhas 16-19), esse método obtém o resultado através de um outro método *getReply* do interceptador servidor descrito a seguir.

Algoritmo 3 *Módulo Interceptador Cliente*

```
connectReconnect()
(01) início
(02)   Obtém a partir do host onde a página foi carregada a lista de todos os
.       interceptadores servidores disponíveis (IntList)
(03)   Para todo interceptador servidor IntSrv em IntList faça
(04)     %Localiza a referência de um interceptador servidor e a armazena em IntSrvRef.%
(05)     IntSrvRef = lookup(IntSrv);
(06)     se IntSrvRef ≠ NULO então
(07)       retorna IntSrvRef;
(08)     fim-se;
(09)   loop;
(10)   retorna NULO;
(11) fim;

schedule(Requisição requisição)
(12) início
(13)   %Invoca o método schedule a partir da referência IntSrvRef%
(14)   IntSrvRef.schedule(requisição);
(15) fim

getReply(Requisição requisição, Inteiro TempoDeEspera)
(16) início
(17)   %Invoca o método getReply a partir da referência IntSrvRef%
(18)   IntSrvRef.getReply(requisição, TempoDeEspera)
(19) fim
```

O interceptador servidor pode ser instalado no lado cliente e conseqüentemente, acessado diretamente sem a necessidade do interceptador cliente. Essa alternativa não se adequa às aplicações via internet, sendo mais apropriada quando os clientes e servidores fazem parte de uma mesma rede local.

O Algoritmo 4 mostra o pseudo-código do módulo interceptador servidor. Após receber a requisição do *interceptador cliente*, esse módulo armazena temporariamente em um repositório (linha 2) e realiza uma difusão confiável para entrega da requisição em todas as réplicas operacionais (linha 3). Uma requisição permanece nesse repositório até que uma resposta para ela seja retornada pelo servidor e depois lida pelo cliente.

O módulo interceptador servidor permite que as requisições sejam transparentemente enviadas de modo assíncrono ou síncrono para todas as réplicas (linhas 4-13). Quando o cliente quiser consultar o resultado de uma requisição enviada de modo assíncrono, é preciso

especificar um prazo de espera pois a requisição pode não ter sido ainda concluída (linhas 15-23). Caso ela tenha sido concluída, a consulta retorna a requisição com um resultado não nulo. Caso contrário, a consulta retorna a requisição com um resultado nulo.

Algoritmo 4 *Módulo Interceptador Servidor*

```
schedule(Requisição requisição)
(01) início
(02)   add(repositorioTMP, requisição);%adicionar requisição em um repositório temporário%
(03)   multicast(requisição); %enviar requisição para todos (difusão confiável)%
(04)   %modo de comunicação é definido na requisição pelo cliente: síncrono, assíncrono %
(05)   se (modo de comunicação é síncrono) então
(06)     enqto (verdade) %Espera até que a requisição seja retornada pelo servidor%
(08)       req ← find(repositorioTMP, requisição);
(09)       se status de req igual retornada então
(10)         retorna req; fim-se; %retorna a primeira retornada%
(11)     loop
(12)   senão
(13)     retorna null; fim-se; %retorna nulo para o cliente%
(14) fim;

getReply(Requisição requisição, Inteiro TempoDeEspera)
(15) início
(16) req ← find(repositorioTMP, requisição);
(17)   enqto (verdade)
(18)     se (status da requisição igual a retornada) então
(19)       retorna req; fim-se;%retorna a requisição já processada%
(20)     se (tempo decorrido desde a invocação de getReply for igual a TempoDeEspera) então
(21)       retorna req; fim-se;
(22)   loop;
(23) fim

handle(Message m)
(24) início
(25)   se (uma msg m contendo uma requisição foi recebida) então
(26)     requisição ← corpo da mensagem m contendo a requisicao
(27)     se (status da requisição igual a escalonada) então
(28)       %atualiza status da requisição no repositorioTMP caso não tenha sido escalonada%
(29)       update(repositorioTMP, requisição); fim-se;
(30)     se (status da requisição igual a retornada) então
(31)       %atualiza status da requisição no repositorioTMP caso não tenha sido ainda retornada%
(32)       se (requisição não foi retornada ainda) então update(repositorioTMP, requisição); fim-se;
(33)       senão
(34)         descarta msg; fim-se
(35)   fim-se;
(36) fim
```

Esse módulo poderá receber das réplicas mais de uma resposta de uma mesma requisição, porém somente a primeira será entregue ao interceptador cliente (linha 30-34) e em seguida ao cliente, as demais respostas serão descartadas. Portanto, o módulo contém um mecanismo de filtro que realiza a eliminação de duplicidades das respostas das requisições.

4.3.2 Camada de Coordenação e Acordo do Servidor

A *camada de coordenação e acordo do servidor*, ilustrada na Figura 4.4(b), implementa as funcionalidades da fase 2 e 4 apresentadas na Seção 3.5.2. Essa camada é formada pelo *Módulo de Ordenação Total* e pelo *Módulo de Escalonamento de Requisição*. O *Módulo de Ordenação Total* e o *Módulo de Escalonamento de Requisições* possuem, respectivamente, as mesmas características do componente *Ordenador Sensível a Prioridade* e do componente *Escalonador GPI* descritos no Capítulo 3.

Módulo de Ordenação Total

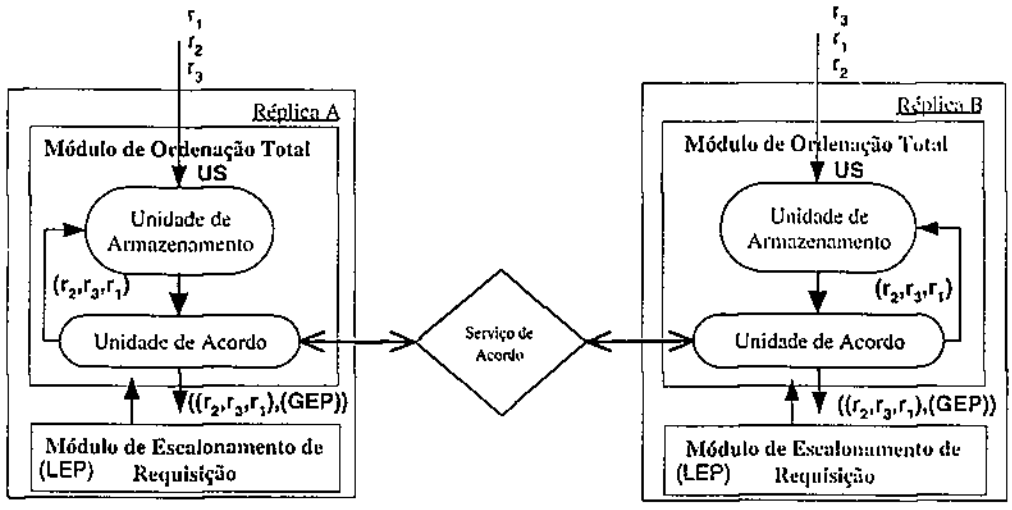


Figura 4.5: Módulo de ordenação total

O módulo de ordenação total ilustrado na Figura 4.5, permite que todas as réplicas de um servidor ativamente replicado processem as mesmas requisições na mesma ordem. Esse módulo é formado pelas unidades: *unidade de armazenamento* e *unidade de acordo*. O Algoritmo 5 mostra o pseudo-código dessas unidades.

A *unidade de armazenamento* armazena, em uma lista *US* (linha 4), as requisições enviadas pelo *módulo interceptador servidor* ainda não escalonadas e utiliza a *unidade de acordo* para realizar um consenso juntamente com as demais réplicas sobre quais requisições devem ser escalonadas (linha 5 e 6). O acordo de ordem precisa de um GEP atualizado, para que mais requisições possam ser ordenadas, dessa forma o consenso realiza dois acordos (Acordo_Ordem e Acordo_GEP) em uma única execução do protocolo fazendo com que a nova ordem obtida se baseie em um GEP atualizado. A *unidade de acordo* faz uso de um *serviço de acordo* que oferece as seguintes capacidades: *i*) capacidade de decidir um valor

que é o resultado de uma dada função aplicada aos valores de entrada; *ii*) capacidade de permitir que vários valores sejam propostos em uma mesma execução do protocolo; e *iii*) capacidade de realizar dois acordos em uma única execução do protocolo de acordo.

Algoritmo 5 *Módulo de Ordenação Total*

```
Unidade de Armazenamento
(01) início
(02)  enqto (verdade) faça
(03)   se (uma requisição r enviada pelo módulo interceptador servidor foi recebida ) então
(04)     add(r, US); %adicionar requisição r na lista não ordenada US%
(05)     se (Consensus_{k-1}(Acordo_Ordem, Acordo_GEP) na unidade de acordo foi concluído)
(06)     então inicia Consensus_k(Acordo_Ordem, Acordo_GEP) através da Unidade de Acordo;
(07)     fim-se
(08)   fim-se
(09)   se (uma msg m contendo um conjunto de requisições ordenadas (newOS) foi recebida) então
(10)     remove da lista US, todas as requisições existentes em newOS;
(11)   fim-se
(12)  loop;
(13) fim;

Unidade de Acordo
(14) início
(15)  enqto (verdade) faça
(16)   se (Consensus_k(Acordo_Ordem, Acordo_GEP)) foi enviado pela Unidade de Armazenamento ou
.       pelo Módulo de Escalonamento de Requisição) então
(17)     iniciar Serviço de Acordo;
(18)     Unidade de Acordo disponibiliza regras do acordo (funções) ao Serviço de Acordo via callback;
.       %As funções callback fornecem EP a partir do mód.de escalonam. e calculam newGEP%
.       %As funções callback fornecem US a partir da unid. de armazenam. e calculam newOS%
(19)     recebe resultado do Consensus_k(Acordo_Ordem, Acordo_GEP): newOS e newGEP;
(20)     %atualiza as prioridades da requisições em newOS para uma prioridade associada no
.        banco de dados (BD). Utiliza a chave CLIENT_ID_PRIORITY_MAPPING para localizar
.        prioridade no BD%
(21)     transform_priority("CLIENT_ID_PRIORITY_MAPPING", newOS);
(22)     newOS ← Ordenar(lista decidida,"PRIORIDADE"); %ordena por prioridade a lista decidida)%
(23)     envia {newOS}à Unid. de Armazenam. e {newOS;newGEP} ao mód.de escalonam.%
(24)   fim-se;
(25)  loop;
(26) fim;
```

O *serviço de acordo* que foi implementado nesse trabalho foi descrito na Subseção 4.2.2. Através desse serviço (linhas 17-18), a *unidade de acordo* atende a todas as solicitações de acordo do *middleware* e interage com as outras réplicas para obtenção de um resultado contendo uma lista de requisições ordenadas *OS* e um valor de *progresso de execução do grupo GEP* (linhas 19). Como o cliente define na requisição uma prioridade que precisa ser decodificada pelo servidor, a *unidade de acordo* interpreta e atualiza as prioridades das requisições da lista *OS* e, em seguida reordena essa lista (linha 20-23). Para que a prioridade de uma requisição seja atualizada para um valor associado em um banco de dados, o *módulo de ordenação total* utiliza o método *tranforma_priority* da interface *PriorityTransform* do

EROPSAR mostrada a seguir.

A interface *PriorityTransform* deve ser implementada pela camada superior da aplicação. O método *tranform_priority* possui dois parâmetros: *pScheduleP olicy* e *pUS*. O parâmetro *pScheduleP olicy* indica qual política deve ser utilizada para transformar as prioridades. No nosso caso, CLIENT_ID_PRIORITY_MAPPING (prioridade em um repositório de dados baseado no identificador do cliente). Outras políticas poderiam ser utilizadas, de modo a transformar as prioridades segundo um outro critério ou até mesmo segundo as regras de um escalonamento *FIFO*, *EDF (Earliest Deadline First)* [28], etc. O parâmetro *pUS* deve conter todas as requisições que precisam ser transformadas.

```
interface PriorityTransform {
     Object transform_priority (in string pSchedulePolicy,
     in Object pUS);
};
```

O pseudo-código do método *transform_priority* da interface *PriorityTransform* pode ser visto no Algoritmo 6. Inicialmente, verifica-se qual política de prioridade é utilizada (linha 1), em seguida, as prioridades das requisições passadas através do parâmetro *pUS* são atualizadas de acordo com as prioridades associadas no banco de dados.

Algoritmo 6 *PriorityTransform*

```
transform_priority(string pSchedulePolicy, Array pUS)
(01)   se (pSchedulePolicy ="CLIENT_ID_PRIORITY_MAPPING") então
(02)     para toda requisicao em pUS
(03)       início % Localiza a prioridade utilizando o identificador do cliente e a atualiza em pUS%
(04)         SELECT prioridade
(05)         FROM clientes
(06)         INTO [requisição em pUS] %Atualiza prioridade encontrada
(07)         WHERE identificador do cliente no banco é igual ao código da prioridade na requisicao
(08)       fim
(09)     retorna pUS
(10)   fim-se
```

Módulo de Escalonamento de Requisição

O *Módulo de Escalonamento de Requisição* recebe da *Unidade de Acordo* um conjunto de requisições e o adiciona a uma lista contendo todas as requisições previamente adicionadas (mesma lista *História* introduzida no Capítulo 2). Essa adição é realizada de modo a evitar o problema de *inversão de prioridade em grupo*. Para evitar e tratar esse problema, esse módulo utiliza as seguintes variáveis: GEP (*Progresso de Execução do Grupo*), LEP (*Progresso de Execução Local*), *História* (lista de requisições prontas, em processamento e completadas), *Ativo* (Estado do Módulo Processador de Requisições) e *Max* (número de requisições em *História*).

Como o *módulo processador de requisição* em cada réplica está a princípio inativo, a variável *Ativo* é igual a falso. GEP e LEP iniciam com 0. Como nenhuma requisição está a princípio escalonada, a lista *História* está vazia e, portanto, o número de elementos existente em *História* (*Max*) é igual a 0.

Como pode ser visto no Algoritmo 7, sempre que uma nova decisão da *Unidade de Acordo* é obtida, as seguintes ações devem ser realizadas: atualização das variáveis internas, reordenação, retrocesso. Essas ações são descritas a seguir:

Algoritmo 7 *Módulo de Escalonamento de Requisição - Reação a uma decisão*

```
(01) GEP ← 0; LEP ← 0; História ← ∅; Ativo ← falso; Max ← 0;
(02) enqto (verdade) faça
(03)    após a recepção do resultado de Consensus_k(Acordo_Ordem, Acordo_GEP)
                          através Unidade de Acordo com a decisão {newOS;newGEP}
(04)      GEP ← newGEP; m ← |newOS|;
(05)      Retorna as respostas armazenadas em log para o Cliente até newGEP;
(06)      Max = Coleta de lixo(); %Exclui de História as requisições retornadas na linha anterior e
                                   atualiza Max para o novo tamanho de História%
(07)      História[Max+ 1..Max + m] ← newOS[1..m];
(08)      SR ← Ordenar (História[GEP+1::Max + m]); %SR retorna a posição em História em que a
                                                   primeira requisição de newOS foi inserida%
(09)      Max ← Max+ m;
(10)      se (LEP ≥ GEP) então
(11)        se (LEP = GEP ∧ Ativo ∧ SR = LEP+1) então
(12)          Recupera do log o estado da LEP-ésima requisição processada;
(13)          envia setState(state) a camada de execução; % retrocesso para um estado anterior %
(14)          Ativo ← falso; fim-se
(15)        se (LEP > GEP ∧ Ativo ∧ SR ≤ LEP) então
(16)          Recupera do log o estado da (SR-1)-ésima requisição processada;
(17)          envia setState(state) à camada de execução; % retrocesso para um estado anterior %
(18)          LEP ← SR - 1; Ativo ← falso; fim-se
(19)        se (LEP > GEP) então
(20)        inicia Consensus_k(Acordo_Ordem, Acordo_GEP) através Unidade de Acordo para cálculo
                                                         de newOS e progresso de GEP;
(21)          se (Ativo = falso) então
(22)            Ativo ← verdade;
(23)            Inicia o processamento da requisição História[SR]; fim-se
(24)      fim-se
(25) loop
```

1. Atualização das variáveis:
   Inicialmente, o número de elementos existentes no conjunto *newOS* decidido pela *Unidade de Acordo* é armazenado na variável *m* (linha 4). As requisições desse conjunto são adicionadas no final da lista *História*. *Max* representa o número de elementos existente em *História* antes da adição de *newOS*. Desse modo essa lista é representada por *História*[*1..Max+m*]. O valor de GEP é atualizado conforme o valor de decisão da unidade de acordo (linha 4). O novo valor de GEP permite que as respostas das requisições com índice inferior a GEP em *História*, que foram processadas e armazenadas, sejam finalmente entregues aos seus respectivos clientes requisitantes. Os pontos de salvaguarda relacionados às requisições entregues aos clientes são removidos.

2. Reordenação:
   A reordenação (linha 8) é uma tarefa fundamental para evitar inversão de prioridade em grupo. Ela adiciona o conjunto de requisições OS, retornado pelo módulo de ordenação total, na posição em *História* onde se encontra a requisição não estável com prioridade inferior a prioridade da primeira requisição em OS, formando o conjunto *História*[*GEP+1..Max+m*]. Após a adição, esse conjunto é ordenado por prioridade pelo procedimento *Ordenar*, que retorna a posição em *História*, onde a primeira requisição de OS foi inserida. Por fim, o número de elementos em *História* é atualizado em *Max*.

3. Retrocesso:
   O valor SR (posição em História em que a primeira requisição de newOS foi inserida) retornado pelo procedimento de reordenação (linha 8) é útil para detectar se um retrocesso é necessário. Os seguintes casos precisam ser analisados:

   - LEP < GEP: retrocesso não são necessários, pois, por definição, todas as requisições são estáveis.
   - LEP = GEP: se LEP igual a GEP, então todas as requisições até LEP são estáveis. Se o processador estiver ativo processando uma requisição com prioridade inferior a SR, no caso LEP+1, então um retrocesso deverá ser realizado até o último ponto de salvaguarda.
   - LEP > GEP: se LEP maior que GEP então algumas requisições processadas ainda não são estáveis. Desse modo, se o processador estiver ativo processando uma requisição com prioridade inferior a SR e se SR for menor ou igual LEP, então um retrocesso deverá ser realizado em todas as requisições completadas até a posição (SR-1) de *História*.

Uma outra funcionalidade do *Módulo de Escalonamento de Requisição* é a de enviar a requisição de estado *pronto* de maior prioridade à *Camada de Execução* e receber o resultado do processamento para que possa atualizar sua lista *História*. Como a lista *História* não pode crescer indefinidamente todas as requisições que já foram retornadas à *camada de requisição e resposta* podem ser excluídas. Isso é realizado através de um procedimento de coleta de lixo realizado após o envio de um resultado à *camada de requisição e resposta* (linha 6). O procedimento exclui de *História* todas as requisições *estáveis* em História[1..GEP] que foram retornadas e que portanto, ocupam uma posição em *História* menor ou igual a GEP. Esse procedimento retorna o novo número de elementos em *História*, atualizando desse modo a variável *Max*.

A interação entre o *Módulo de Escalonamento de Requisição* e a *Camada de Execução* pode ser vista no Algoritmo 8. Sempre que a *camada de execução* tiver completado o processamento de uma requisição, o *módulo escalonador de requisição* incrementa o valor de LEP. Se LEP $\leq$ GEP, a requisição é considerada *estável* e seu resultado é enviado ao cliente requisitante (linha 5). Caso contrário, o resultado da requisição é armazenado e o valor de GEP é atualizado. Por fim, o módulo verifica se existem novas requisições prontas para serem processadas e retorna ao módulo despachador da *camada de execução* a requisição de estado *pronto* de maior prioridade (linha 11). Caso não existam novas requisições (linha 9), o *Módulo de Escalonamento de Requisição* não retornará o controle para a *Camada de Execução* até que receba novas requisições.

Algoritmo 8 *Módulo de Escalonamento de Requisição - Interação com a camada de execução*

```
(01) enqto (verdade) faça
(02)    após a recepção de getRequest(state, reply) da camada de execução ou Ativo igual a falso:
(03)      Ativo ← falso; LEP ← LEP +1;
(04)      se (LEP ≤ GEP) então
(05)        envia replyRequest(resposta) ao Cliente;
(06)      senão log ← log ∪ reply; % armazena em log a resposta e o estado %
(07)        inicia Consensus_k(Acordo_Ordem, Acordo_GEP) através da Unidade de Acordo para cálculo
                                                          de newOS e para progresso de GEP;
(08)      fim-se
(09)      se (Max > LEP) então
(10)        Ativo ← verdade;
(11)        Retorna o controle para camada de execução enviando a requisição História[LEP+1];
(12)      fim-se
(13) loop
```

4.3.3 Camada de Execução

A *Camada de execução* implementa as funcionalidades da fase 3, apresentadas na Seção 3.5.2. Essa camada é formada pelo *módulo despachador* e pelo *módulo processador de requisição.*

Módulo despachador

O módulo despachador *solicita* requisições ao *módulo escalonador de requisições*, *recebe* a requisição de estado *pronto* de maior prioridade e invoca o *processador de requisição.* Uma vez processada a requisição, o resultado do processamento é enviado ao *módulo escalonador de requisições* juntamente com o estado do recurso.

Módulo processador de requisição

O módulo processador de requisição deve ser implementado pela camada superior da aplicação através da interface *Processor* mostrada a seguir. Essa interface contém 3 métodos: *dispatch*, *getState* e *setState.* O método *dispatch* recebe a requisição como parâmetro e deve interpretá-la para que possa iniciar o processamento de um operação descrita na requisição. Como esse módulo é implementado na camada superior da aplicação, o programador poderá implementar as operações solicitadas na requisição diretamente nesse módulo. Caso contrário, poderá utilizar o *serviço de nomes* presente nos *serviços de suporte* para localizar a referência do recurso remoto solicitado e, em seguida, executar a operação solicitada. A operação *getState* armazena o estado de um recurso e a operação *setState* atualiza em um recurso um estado previamente salvo através de *getState.*

```
interface Processor {
     Object dispatch(in Object requisição)
     Object getState();
     Boolean setState(in Object pState);
};
```

Um exemplo de como implementar o método *dispatch* da interface *Processor* pode ser visto no Algoritmo 9. Observa-se através do algoritmo que a requisição despachada deve ser interpretada de modo a obter o nome do recurso e a operação solicitada (linhas 1,2 e 8). Se o nome da interface for igual a *Processor_interface* (linha 1), a operação solicitada encontra-se disponível e pode ser executada sem a necessidade de obter referência do recurso, assim, os parâmetros são lidos (linhas 3 e 4) e a operação é executada (linha 5). Se o nome da interface for diferente de *Processor_interface*, então o recurso deverá ser localizado

através de uma referência obtida a partir de um *serviço de nomes* (linha 9), em seguida os parâmetros da requisição devem ser lidos (linhas 10 e 11) e utilizando a referência do recurso obtido, a operação é executada (linha 12).

Algoritmo 9 *Processor*

```
dispatch(Object request)
início
(01)    se (nome da interface da requisição request ="Processor_interface")
(02)      se (nome do método remoto contido na requisição request="SOMA")
(03)        p1 ← getInParam(1) %Ler primeiro parâmetro de entrada da requisição%
(04)        p2 ← getInParam(2) %Ler segundo parâmetro de entrada da requisição%
(05)        setOutParam(1,soma(p1, p2)) %Atualiza valor de saída%
(06)      fim-se
(07)    senão
(08)      se (nome do método remoto contido na requisição request="SOMA")
(09)        ref ← lookup(request.InterfaceName) %Obtém referencia do objeto remoto%
(10)        p1 ← getInParam(1) %Ler primeiro parâmetro de entrada da requisição%
(11)        p2 ← getInParam(2) %Ler segundo parâmetro de entrada da requisição%
(12)        setOutParam(1,ref.soma(p1, p2)) %Atualiza valor de saída%
(13)      fim-se
(14)    fim-se
(15)    fim-se
fim
```

4.4 Avaliação Experimental

Para analisar o *middleware EROPSAR* criamos uma aplicação onde clientes enviam continuamente requisições com prioridade variando de 1 a 9. As requisições que devem ser processadas são definidas em um arquivo que deve ser lido pela aplicação cliente, desse modo, vários testes podem ser realizados utilizando as mesmas requisições. Armazenamos em um arquivo de log, para uma posterior análise, todas as transações envolvidas no *middleware* desde o envio da requisição pelo cliente até o recebimento do resultado do processamento no servidor replicado. Foram realizados testes com diferentes graus de replicação. Em cada teste, foram processadas cem requisições.

Utilizando o arquivo gerado, analisamos os casos de *inversão de prioridade em grupo* e o custo envolvido no processamento ativamente replicado. Para testar o tratamento de inversão de prioridade em grupo e a ordenação total das mensagens, foram observados três casos: *i) Adição de novas requisições sem casos de inversão de prioridade*; *ii) Adição de novas requisições com casos de inversão de prioridade*; e *iii) Adição de novas requisições com casos de retrocessos*. Para verificar o suporte a tolerância a falhas foram realizados 3 testes: *I) Falha por parada de uma das réplicas*; *II) Suspeita de falha de uma das réplicas*; e *III) Suspeita de falha de (n-⌈(n+1)/2)⌉) réplicas, com n igual ao grau de replicação utilizado*.

4.4.1 Testes de Funcionalidade

Nos exemplos a seguir, ilustrados pelas Figuras 4.6, 4.7 e 4.8, identificamos uma requisição *50/R_002/P_2-100* da seguinte forma: *50* indica que a requisição foi enviada por um cliente identificado por *50*; *R_002* indica que esta é a segunda requisição enviada pelo cliente *50*; *P_2* indica que a prioridade da requisição *R_002* tem prioridade *2*; *100* indica o status da requisição, no caso completada (100). Os seguintes status são considerados: pronta (20), em processamento (50), completada (100) e retornada (110). Assim como no sistema operacional Linux, consideramos aqui que, quanto menor for o número identificador de prioridade, maior será a prioridade de uma requisição. Dessa forma prioridade(1) > prioridade(9).

Consideramos duas réplicas: réplica 1 e réplica 2. Cada umas das réplicas possuem um estado inicial identificado por EG1. Quando uma requisição é adicionada às réplicas 1 e 2, elas evitam inversão de prioridade em grupo e passam para um estado final identificado por EG2. Vale salientar que cada réplica possui um EG1 e EG2 distinto até que elas tenham processado as mesmas requisições.

Adição de novas requisições sem casos de inversão de prioridade:

Ao enviar uma requisição *50/R_002/P_2* ao conjunto de réplicas verifica-se que não ocorre inversão de prioridade em nenhuma das réplicas. O processador encontrava-se livre na réplica 2, pois a última requisição estava pronta para processamento (status=20). Nesse contexto, jamais poderia ocorrer inversão de prioridade.

Para definir a posição onde a requisição deve ser inserida, utiliza-se o valor de GEP+1 como tentativa inicial. Se a requisição existente nessa posição tiver mais prioridade ou mesma prioridade que *50/R_002/P_2-50*, segue-se adiante, caso contrário, se a requisição nesse ponto tiver menos prioridade e estiver pronta (status=20), as requisições desse ponto em diante são deslocadas para dar espaço ao ingresso da nova requisição *50/R_002/P_2-50*. Se a requisição existente tiver menos prioridade e estiver completada(status=100), ela sofrerá retrocesso e as requisições desse ponto em diante são deslocadas para dar espaço ao ingresso da nova requisição *50/R_002/P_2-50*.

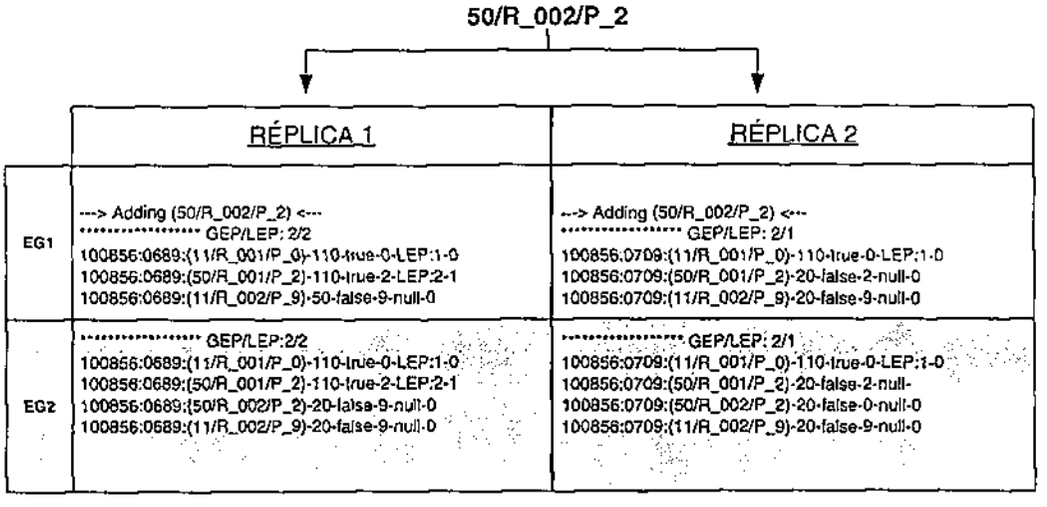


Figura 4.6: Adição de novas requisições sem casos de inversão de prioridade

Adição de novas requisições com casos de inversão de prioridade:

Ao enviar uma requisição *50/R_003/P_1* ao conjunto de réplicas ocorre inversão de prioridade na réplica 1. A requisição *11/R_002/P_9* estava em processamento (status=50) e portanto foi interrompida. A interrupção ocorreu por 2 fatores: por prioridade de *50/R_003/P_1* ser maior que prioridade de *11/R_002/P_9* e pelo fato de *11/R_002/P_9* não ser estável, pois sua posição na lista(idx=4) em EG1 está além de GEP (igual a 3).

Na réplica 2, a requisição *50/R_002/P_2*, embora não processada, já era estável, sua posição (idx=3) era igual ao GEP, portanto, mesmo com a prioridade de *50/R_003/P_1* sendo maior que a prioridade de *11/R_002/P_9*, não houve a necessidade de reordenação das requisições.

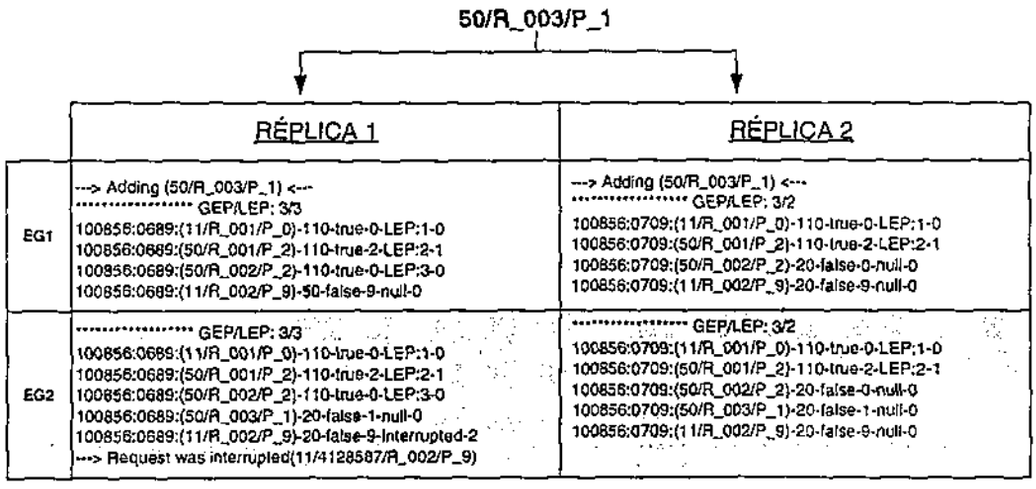


Figura 4.7: Adição de novas requisições com casos de inversão de prioridade

Adição de novas requisições com casos de retrocesso:

Ao enviar uma requisição *50/R_003/P_1* ao conjunto de réplicas ocorre retrocesso da requisição *11/R_002/P_9* na réplica 1. Embora *11/R_002/P_9* estivesse completada (status=100), ainda não era estável (LEP>GEP). Na réplica 2, a requisição *50/R_002/P_2*, embora não completada, já era estável, sua posição (idx=3) era igual ao GEP, portanto, mesmo com a prioridade de *50/R_003/P_1* sendo maior que a prioridade de *11/R_002/P_9*, não houve a necessidade de reordenação das requisições.

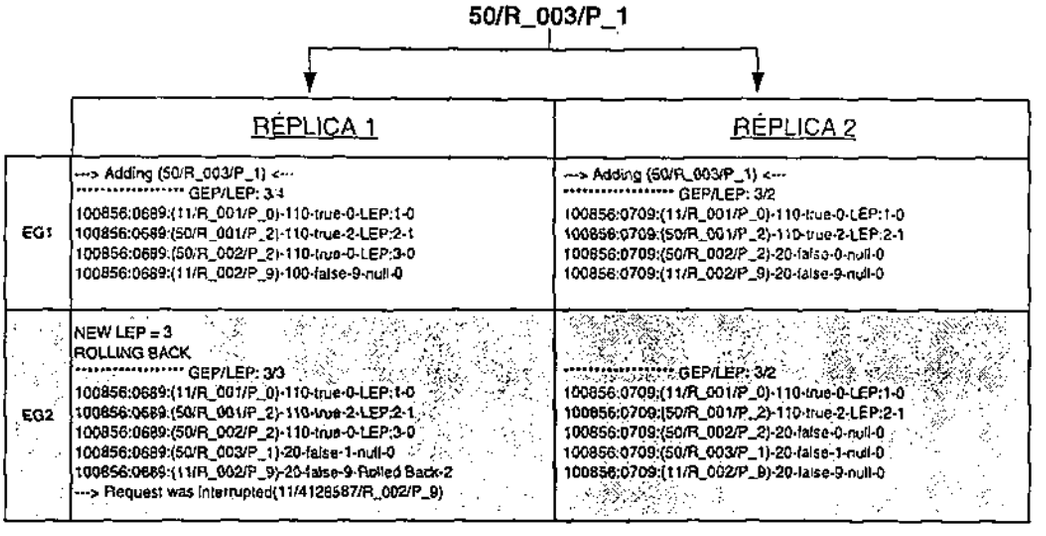


Figura 4.8: Adição de novas requisições com casos de retrocesso

4.4.2 Suporte a Tolerância a Falhas

Para verificar o suporte a tolerância a falhas do *EROPSAR*, testamos uma aplicação com cinco réplicas e provocamos situações onde uma réplica sofre uma falha ou fica impossibilitada de se comunicar com as outras réplicas temporariamente.

Para simular a suspeita de falha de uma réplica, é enviada uma mensagem que, quando recebida por uma réplica, bloqueia o protocolo de acordo da mesma, desse modo, essa réplica passa a não responder às mensagens dos acordos de ordenação total promovidos pelas outras réplicas até que seja notificado com nossa mensagem para desbloqueá-la. Como a réplica passa a não responder, o serviço de detecção de falha das outras réplicas adiciona a réplica bloqueada a sua lista de suspeitos. Uma vez adicionada na lista de suspeitos, as demais réplicas prosseguem com a ordenação das mensagens.

- **Falha por parada de uma das réplicas:**
  Para realizar esse teste, simplesmente finaliza-se uma das réplicas através das teclas *CTRL+C* ou do comando *kill -2 [número do processo no sistema].* Após a falha de uma réplica, as outras réplicas continuam normalmente o processamento de requisições. O serviço de detecção de falha leva alguns milisegundos ou até segundos para suspeitar a falha de uma réplica, desse modo, um pequeno atraso é gerado no protocolo de acordo até que a réplica seja adicionada à lista de suspeitos. Como o serviço de detecção de falha é configurável, esse atraso pode ser controlado pelo usuário.

- **Suspeita de falha de uma das réplicas:**
  Implementamos o utilitário *sendSignal* para simular falha de uma réplica. Esse utilitário possui a sintaxe: *sendSignal [maquina] [porta] [CONGELA|DESCONGELA].* Através do commando *sendSignal maquina porta CONGELA*, a réplica receptora suspende temporariamente e não participa de nenhum protocolo de acordo até que o comando *sendSignal maquina porta DESCONGELA* seja executado. Enquanto a réplica estiver *CONGELADA*, ela armazenará as mensagens enviadas e não acionará a unidade de acordo para ordenação de requisições. Uma vez que a mensagem DESCONGELA é recebida, a réplica iniciará a ordenação das mensagens previamente recebidas e processará todas as decisões de consenso na qual esteve bloqueada.

  Assim como no teste anterior, o serviço de detecção de falha leva alguns segundos para suspeitar que a réplica falhou, gerando também um pequeno atraso no protocolo de acordo até que a réplica seja adicionada à lista de suspeitos.

- **Suspeita de falha de (n-⌈(n+1)/2)⌉) réplicas, com n igual ao grau de replicação utilizado:**
  A suspeita de falha nesse teste pode ser realizada utilizando o comando *kill* ou o utilitário *sendSignal.* Esse teste teve como objetivo mostrar a exigência do protocolo de acordo (maioria dos processos corretos). Como utilizamos um detector de falhas $\diamond$S, o protocolo de acordo só finaliza (*decide*) se a maioria dos processos não estiverem presentes na lista de suspeitos. Nesse contexto, foram iniciadas cinco réplicas. Para adicionar as réplicas na lista de suspeitos, utilizamos *kill* ou *sendSignal.* Nesse contexto, paramos duas réplicas, todas as outras continuaram a ordenar requisições normalmente, quando paramos a terceira réplica, o protocolo de acordo das duas réplicas restantes ficaram bloqueados esperando uma estimativa ou um voto de uma réplica suspeita de falha. Desse modo, o acordo só pode ser concluído quando a maioria das réplicas ficaram novamente fora da lista de suspeitos. Verifica-se que o

servidor replicado foi tolerante a (n-⌈(n÷1)/2)⌉) falhas.

4.4.3 Avaliação de Desempenho

Para avaliar o desempenho do *middleware*, nossa aplicação sincroniza o relógio da máquina cliente com os relógios das réplicas do servidor replicado. Desse modo, a aplicação cliente armazena, em uma tabela interna, a diferença entre os relógios, para que possa calcular a duração entre o envio de uma mensagem e o recebimento da primeira resposta enviada por uma das réplicas.

Descrevemos a seguir as marcas de tempo armazenadas no arquivo de log:

- **tempo no envio**: instante de tempo que uma requisição é enviada pelo cliente;
- **tempo na recepção**: instante de tempo que uma requisição é recebida pela réplica;
- **tempo no escalonamento**: instante de tempo que uma requisição leva para ficar pronta para processamento;
- **tempo no despacho**: instante de tempo que uma requisição é enviada para processamento;
- **tempo no término de processamento**: instante de tempo que é concluído o processamento de uma requisição;
- **tempo no envio do resultado ao cliente**: instante de tempo que uma requisição é retornada 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.

A operação de envio modifica o estado da requisição de *não recebida* para *recebida*. A operação de ordenação modifica o estado da requisição de *recebida* para *pronta*. A operação de despacho modifica o estado da requisição de *pronta* para *despachada*. A operação de processamento modifica o estado da requisição de *despachada* para *completada*. A operação de retorno modifica o estado da requisição de *completada* para *retornada*.

O grau de replicação (GR) variou de 3 a 5, com o cliente e as réplicas na mesma rede local. 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 4.9 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.

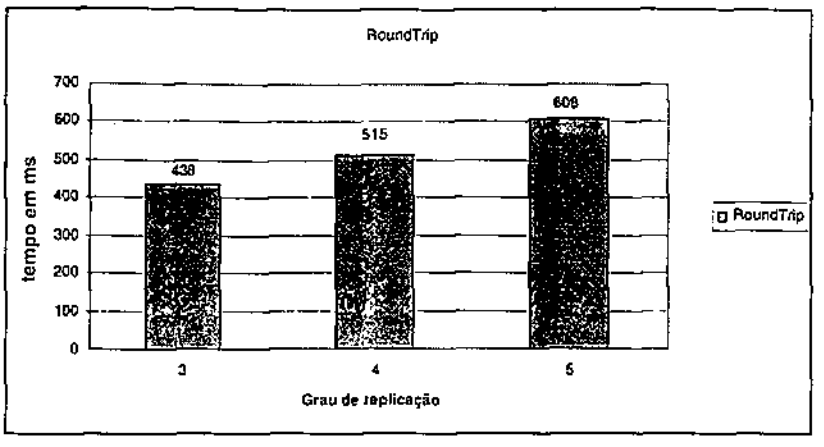


Figura 4.9: *RoundTrip* no *EROPSAR*

O gráfico da Figura 4.10 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 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.

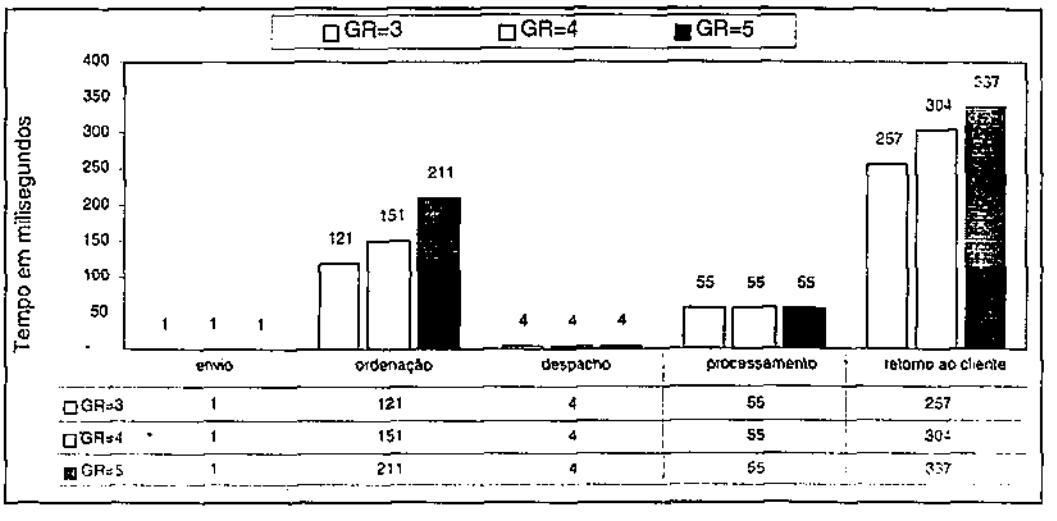


Figura 4.10: Duração das operações de transição de estado

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

4.5 Desenvolvendo Aplicações Utilizando EROPSAR

O *EROPSAR* disponibiliza quatro 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, *ClientInterceptador* permite a comunicação transparente entre cliente (aplicações em JAVA) e servidor replicado e finalmente *Applet_Interceptor* permite a comunicação transparente entre cliente (*browsers*) e servidor replicado.

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

Configurando uma Aplicação Cliente

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

1. Instanciar a classe *Applet_interceptor* (*interceptador cliente*) e obter a referência de um dos objetos da classe *ClientInterceptor* (*interceptador servidor*) localizados remotamente através da interface *ClientInterceptador_interface*.

   Caso a aplicação cliente deseje acessar o *interceptador servidor* diretamente através de uma aplicação escrita em JAVA, *ClientInterceptor* deverá está instalada no lado cliente e deverá ser utilizada da seguinte forma:

   (a) Criar uma instância da classe *ClientInterceptor* (*interceptador servidor*), enviando como parâmetro as propriedades do módulo interceptador servidor (ilustrado na Figura 4.11 com a descrição de cada parâmetro). O parâmetro *args* contém essas propriedades.

      i. ClientInterceptor ci = new ClientInterceptor(args);

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: request.priority = "5";

   (b) nome da interface. Ex: request.objInterfaceName = "Obj_interface";

   (c) nome do método. Ex: request.objTask = "MEU_METODO";

   (d) invocação síncrona ou assíncrona. Ex: request.synchronous =true;

```
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.DELAY_SIMULATION_OF_REQUEST=200     # Simula um processamento de 200ms.
                                              Útil para testa inversão de prioridade
EROPSAR.E_GAF_UNIT.NumberOfProcess=2        # Grau de replicação do servidor replicado
EROPSAR.E_GAF_UNIT.Process1=austin:7000     # Nome e porta de escuta da réplica 1
EROPSAR.E_GAF_UNIT.Process2=austin:7100     # Nome e porta de escuta da réplica 2
EROPSAR.E_GAF_UNIT.Process3=austin:7200     # Nome e porta de escuta da réplica 3
EROPSAR.E_GAF_UNIT.PortToListen=0           # Porta de escuta. Definida dinamicamente
                                            # quando igual a 0
```

Figura 4.11: Propriedades para configuração do Interceptador Servidor

(e) definir os parâmetros de entrada e saída. 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 objeto da classe *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.

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 *middleware* (ilustrado na Figura 4.12 com a descrição de cada parâmetro). O parâmetro *args* contém essas propriedades.

   (a) RSM rsm = new RSM(args);

```
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 4.12: Propriedades para configuração do *middleware* - Lado Servidor

2. Criar uma instância dos objetos que serão acessadas remotamente.

   (a) ClientInterceptor_Impl ci_Impl = new ClientInterceptor_Impl();

   (b) PriorityTransform_Impl pt_Impl = new PriorityTransform_Impl();

   (c) Processor_Impl p_Impl = new Processor_Impl(Host, RMIPort, SrvPort);

   (d) Criação dos objetos da aplicação;

3. Registrar no serviço de nomes o objeto implementado

   (a) rebind("//"+ Host + ":"+ sRMIPort
   + "/ClientInterceptor_interface_"+ SrvPort , ci_Impl);

   (b) rebind("//"+ Host + ":"+ sRMIPort
   + "/PriorityTransform_interface_"+ SrvPort , pt_Impl);

   (c) rebind("//"+ Host + ":"+ sRMIPort
   + "/Processor_interface_"+ SrvPort , p_Impl);

Capítulo 5

Conclusão e Trabalhos Futuros

Nesse trabalho projetamos e implementamos um *middleware*, denominado *EROPSAR*, que viabiliza a construção de aplicações com alta disponibilidade de serviço. A indisponibilidade de um serviço provido por uma aplicação pode ocasionar substanciais perdas financeiras. Podemos citar como exemplo, o sítio de leilões virtuais *eBay* que, em julho de 1999, ficou cerca de 22 horas fora do ar por problemas de infra-estrutura. Isso representou uma perda no seu faturamento estimada entre US$ 3 milhões e US$ 5 milhões [4]. A alta disponibilidade pode ser alcançada através da técnica de replicação ativa [26]. A técnica de replicação ativa associada a uma política de alocação de recursos baseada em prioridade [18, 19] permite que, mesmo com a eventual perda de algumas requisições de baixa prioridade, o sistema permaneça disponível em momentos de pico. Nesse contexto, o *middleware EROPSAR* implementa 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*.

O uso da técnica de replicação ativa, aliada a uma política adaptativa de alocação de recursos baseada em prioridade, exige um tratamento especial para garantir que o processamento de requisições seja realizado satisfazendo os critério de prioridades. Desse modo, um servidor ativamente replicado precisa evitar o problema de inversão de prioridade em grupo. Esse problema ocorre quando casos de inversão de prioridade local são detectados em muitas réplicas. O *middleware EROPSAR* soluciona o problema de inversão de prioridade em servidores ativamente replicados e se baseia nos resultados dos estudos realizados em [29]. Os autores em [29] propõem um escalonador de requisições que, apoiado por um protocolo de acordo e por um protocolo de ordenação total *sensível a prioridade*, evita inversão de prioridade em grupo. O *EROPSAR* implementa tais protocolos, bem como um escalonador que segue uma abordagem otimista, permitindo portanto realizar antecipada-

mente algum processamento, antes de se obter o resultado final do escalonamento. Como conseqüência, o processamento de algumas requisições pode eventualmente ser desfeito.

Para resolver os problemas de acordo do *EROPSAR*, foi projetado e implementado um protocolo de acordo que estende a estrutura genérica de acordo GAF proposta em [15]. GAF [15] permite, através de seis parâmetros versáteis, a construção de protocolos de acordo especializados. GAF define um protocolo de consenso genérico capaz de atender às diferentes necessidades dos problemas de acordo encontrados em sistemas distribuídos. GAF permite que muitas requisições possam ser ordenadas em uma única execução do protocolo de acordo e que uma decisão seja obtida aplicando uma função a diversos valores de entrada. A fim de tornar o *middleware* mais eficiente, estendemos GAF para que dois acordos possam ser realizados em conjunto através de uma única execução do protocolo. Denominamos essa extensão de E-GAF. Assim como GAF, E-GAF se baseia no protocolo de consenso de introduzido em [6]. A grande vantagem de E-GAF é que não se modifica o protocolo de consenso para adequá-lo a um determinado protocolo de acordo, pois esta estrutura oferece uma solução em camadas que deixa as particularidades de um acordo a cargo dos seis parâmetros versáteis, enquanto que a estrutura do consenso permanece inalterada.

O serviço de comunicação utilizado para construção desse *middleware* foi também implementado nesse projeto. Este possibilita a comunicação de processos, oferecendo de uma maneira simplificada as primitivas *unicast* e *multicast* confiável. Um outra facilidade oferecida nesse serviço é a possibilidade de compartilhar conexões através de *portas virtuais*. As portas virtuais permitem reduzir o uso de portas do sistema operacional.

Os benefícios oferecidos no *middleware 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.

Como proposta de trabalhos futuros, esse *middleware* poderia ser adaptado para adicionar a um sistema de gerenciamento de banco de dados (SGBD) a capacidade de executar consultas *SQL* de acordo com a prioridade das requisições definidas por aplicações clientes dentro de um tempo específico. Todas as consultas *SQL* passariam antes pelo *EROPSAR* que as escalonariam de acordo por ordem de suas prioridades. Essas consultas após um tempo pré-determinado seriam canceladas caso não tivessem sido ainda concluídas.

Bibliografia

[1] J. Almeida, M. Dabu, A. Manikutty, and P. Cao. Providing differentiated levels of service in web content hosting. In *Proceedings of the ACM SIGMETRICS Workshop on Internet Server Performance (WISP)*, pages 91–102, Madison, WI, June 1998.

[2] K. P. Birman. The process group approach to reliable distributed computing. *Communications of the ACM*, 36(12):37–53, 1993.

[3] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. In S. Mullender, editor, *Distributed Systems*, chapter 8, pages 199–216. 2nd edition, 1993.

[4] D. F. Carr. Don't get spiked. *Internet World*, 5(34):59, January 1999. In http://www.acm.org/technews/articles/1999-1/1208w.html (5/Nov/2003).

[5] T. Chandra. V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.

[6] T. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Communications of the ACM*, 43(2):225–267, 1996.

[7] F. Cristian. Understanding fault-tolerant distributed systems. *Communications of the ACM*, 34(2):56–78, 1991.

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

[9] P. Felber. *The CORBA Object Group Service: A Service Approach to Object Groups in CORBA*. PhD thesis, École Polytechnique Fédérale de Lausanne, Switzerland, 1998. Number 1867.

[10] C. Fetzer and F. Cristian. On the possibility of consensus in asynchronous systems. In *Proceedings of the 1995 Pacific Rim International Symposium on Fault-Tolerant Systems*, Newport Beach, CA, 1995.

[11] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2), April 1985.

[12] H. Garcia-Molina. Elections in a distributed system. *IEEE Transaction on Computers*, 31(1):47–59, January 1982.

[13] V. Hadzilacos. On the relationship between the atomic commitment and consensus problems. In *Fault-tolerant distributed computing*, B. Simons and A. Specotr editors, LNCS 448, pages 201–208. Springer Verlag, 1990.

[14] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In S. Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. 2nd edition, 1993.

[15] M. Hurfin, R. Macêdo, M. Raynal, and F. Tronel. A general framework to solve agreement problems. In *Proceedings of 18th IEEE Int Symposium on Reliable Distributed Systems (SRDS'99)*, pages 55–65, Lausanne, Switzerland, October 1999.

[16] P. Jalote. *Fault Tolerance in Distributed Systems*. Prentice Hall, NJ, 1994.

[17] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[18] D. Menasce, V. Almeida, R. Fonseca, and M. A. Mendes. A methodology for workload characterization of e-commerce sites. In *Proceedings of the 1st ACM Conference on Electronic Commerce*, pages 119–128, Denver, Colorado, United States, 1999. ACM Press.

[19] D. Menasce, V. Almeida, R. Fonseca, and M. A. Mendes. Resource management policies for e-commerce servers. In *Proceedings of the Second Workshop on Internet Server Performance*, Atlanta, Georgia, United States, May 1999.

[20] T. J. Mowbray and R. Zahavi. *The Essential CORBA - Systems Integration Using Distributed Objects*, chapter 3: An Introduction to CORBA, pages 48–49. J. Wiley & Sons, New York, USA, 1995.

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

[22] Object Management Group. *Fault-Tolerant CORBA Specification*, document ptc/2000-04-04 edition, April 2000. URL: http://www.omg.org.

[23] The Open Group. *DCE 1.1: Remote Procedure Call*, cae specification c706 edition, 1997. http://www.opengroup.org/products/publications/catalog/c706.htm.

[24] L. Rodrigues and M. Raynal. Atomic broadcast in asynchronous crash-recovery distributed systems. In *Proceedings of the 20th IEEE International Conference on Distributed Computing Systems*, Taipem Taiwan, April 2000.

[25] D. Schmidt, M. Stal, H. Rohnert, and F. Buschmann. *Pattern-Oriented Software Architecture*, volume Vol. 2: Patterns for Concurrent and Networked Objects. John-Wiley & Sons, 2000.

[26] F. Schneider. Replication management using the state-machine approach. In S. Mullender, editor, *Distributed Systems*, chapter 7, pages 169–197. 2nd edition, 1993.

[27] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: an approach to real-time synchronisation. *IEEE Transaction on Computers*, 39(9):1175–1185, 1990.

[28] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, 3rd edition, 1998.

[29] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin. Solving the group priority inversion problem in a timed asynchronous system. *IEEE Transactions on Computers*, 51(8):900–915, 2002.

[30] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings of 20th International Conference on Distributed Computing Systems (ICDCS'2000)*, pages 264–274, Taipei, Taiwan, R.O.C., 2000. IEEE Computer Society Technical Commitee on Distributed Processing.