UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁRTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Implementação de Objetos Replicados usando Java

por

JOÃO CARLOS FERREIRA FILHO

Dissertação submetida à avaliação, como requisito para a obtenção do grau de Mestre em Ciência da Computação

Profª. Maria Lúcia Blanck Lisbôa
Orientadora

Profª. Ingrid E. S. Jansch-Pôrto
Co-orientadora

Porto Alegre, outubro de 2000

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Ferreira Filho, João Carlos
Implementação de objetos replicados usando Java / por João Carlos Ferreira Filho. – Porto Alegre : PPGC da UFRGS, 2000.
119p. : il.
Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2000. Orientadora : Lisbôa, Maria Lúcia Blanck. Co-Orientadora : Jansch-Pôrto, Ingrid E. S.
1. Tolerância a falhas. 2. Replicação de objetos. 3. Java RMI. 4. Primário-backup. I. Lisbôa, Maria Lúcia Blanck. II. Jansch-Pôrto, Ingrid E. S. III. Título.

AGRADECIMENTOS

Agradeço a Deus por ter-me dado forças para concluir este trabalho. Agradeço à Universidade para o Desenvolvimento do Estado e da Região do Pantanal pelo apoio ao curso de mestrado e a esta pesquisa. As Professoras Dra. Maria Lúcia Blank Lisboa e Dra. Ingrid E. S. Jansch-Pôrto que foram sempre presentes nos momentos necessários. Ao Tribunal Regional do Trabalho nas pessoas de Dr. Wilson Barbosa do Rego e Dr. Júlio César Machado que foram decisivos para a concretização deste trabalho. Aos meus colegas de trabalho pela força e pelo companheirismo. Especialmente a minha mãe Terezinha e minha esposa Lucy por terem compreendido os momentos de mau humor nestes dois últimos anos.

Sumário

Lista de Abreviaturas

AC	*Atomic commitment*
API	Application Program Interface
ARG	Argumento
AX	Axioma
COORD	Coordenador
CORBA	*Comum Object Request Broker Architecture*
CPU	*Central Processing Unit*
ID	Identificador
IDL	*Interface Definition Language*
JDK	*Java Development Kit*
INVID	Identificador de invocação
NACK	*Negative Acknowledgment*
OC	Objeto Crítico
OGS	*Object Group Service*
OP	Operação
ORB	*Object Request Broker*
PRIM(x)	Réplica primária do objeto x
REQ	Requisição
RES	Resultado
RMI	*Remote Method Invocation*
UTBR	*Uniform Timed Reliable Broadcast*

Lista de Figuras

Lista de Símbolos

δ	Delta
τ	Tau

Resumo

Este trabalho busca a implementação da replicação de objetos através da linguagem Java e de seu sistema de invocação remota de métodos (Remote Method Invocation - RMI). A partir deste sistema, define-se uma classe de replicação - a máquina de replicação – onde a implementação de grupos de objetos é estruturada de acordo com a arquitetura cliente/servidor, sendo o cliente o representante (a interface) de um grupo de objetos e os servidores representam os demais componentes do grupo. A classe de replicação atende a uma necessidade importante dos sistemas distribuídos - o desenvolvimento de aplicações tolerantes a falhas. Fundamentalmente, a tolerância a falhas é obtida por redundância e, no caso de mecanismos de tolerância a falhas por software, esta redundância significa basicamente replicação de dados, processos ou objetos. A tolerância a falhas para tal tipo de sistema é importante para garantir a transparência do mesmo, visto que, assim como um sistema distribuído pode auxiliar muito o usuário pelas facilidades oferecidas, o não cumprimento de suas atividades de acordo com o esperado pode, em algumas situações, causar-lhe transtornos e erros irrecuperáveis nas aplicações. Finalmente, como principal contribuição, este trabalho descreve e implementa a solução completa para a construção de uma biblioteca de classes que oferece a replicação de forma totalmente transparente para o usuário.

Palavras-Chave : tolerância a falhas, replicação de objetos, Java RMI, primário-*backup*.

***Title** : "The Implementation of Replicated Objects using Java."*

Abstract

The goal of the work here described is the implementation of replicated objects using the Java language and the Remote Method Invocation library, generally referred as RMI. In this work, we have defined a replication class - the replication machine - where the implementation of objects groups has been structured according to the client/server architecture; the client acts as the representative (the interface) of an object group, and the servers represent the other components of the group.

The replication class is a traditional mean used to achieve the needed dependability properties of the distributed systems in view of the development of fault tolerant applications. Basically, fault tolerance is achieved by redundancy; in the case of software fault-tolerant mechanisms, this redundancy may be related to data, process or objects replication. In these systems, both fault tolerance and the associated replication mechanisms transparency are very important. In the same way that a distributed system may help the user due to its facilities, the incorrect or unexpected executions may to cause trouble or data missing in the user applications.

Finally, the main contribution described in this text and implemented is a solution to the construction of a class library that offers completely transparent replication to the users.

***Keywords**: fault tolerance, object replication, Java RMI, primary-backup.*

1 Introdução

1.1 Motivação

As definições de sistemas distribuídos podem apresentar variações, ou níveis de detalhamento diferenciados, se tomados por referência diferentes autores. Porém Babaoglu [BAB98] e Birman [BIR96] concordam que nestes sistemas existe a necessidade de mecanismos de **comunicação** eficientes entre os integrantes do mesmo. De acordo com a natureza do ambiente de aplicação do sistema, a comunicação pode ocorrer entre elementos com funcionalidades semelhantes como, por exemplo, em uma rede onde há compartilhamento de recursos; ou ainda, a comunicação pode ser entre elementos com funcionalidades bastante distintas como um sistema de produção em um fábrica que interliga várias máquinas e mecanismos.

O funcionamento de um sistema distribuído é gerenciado por um sistema operacional apropriado, chamado de **sistema operacional distribuído**. Este sistema operacional, que coordena as atividades de diversos processadores independentes, é diferente dos sistemas operacionais monoprocessados tradicionais, devido às implicações inerentes ao uso de sistemas distribuídos, tais como gerência de recursos compartilhados, concorrência de processos, comunicação e outros mais. A palavra-chave que deve reger um sistema operacional distribuído é **transparência**, ou seja, a complexidade do sistema não deve transparecer para o usuário[1]. Este conceito deve ser disseminado por todos os elementos do sistema fazendo com que aspectos de comunicação, sincronização e gerência não perturbem o usuário, deixando-o concentrar-se sobre os aspectos funcionais desejados. Deve parecer ao usuário que ele interage com uma única máquina, sem precisar preocupar-se com detalhes da arquitetura do sistema.

Assim como o sistema distribuído pode auxiliar muito o usuário pelas facilidades oferecidas, o não cumprimento de suas atividades de acordo com o esperado pode, em algumas situações, causar-lhe transtornos e erros irrecuperáveis nas aplicações. Então, uma necessidade importante desses sistemas é o desenvolvimento **de aplicações tolerantes a falhas**. A tolerância a falhas visa garantir que as propriedades essenciais, ou serviços, sejam preservados mesmo na presença de falhas em alguns componentes físicos do sistema [JAL94]. Sua construção será mais fácil se houver suporte adequado ao seu desenvolvimento.

Fundamentalmente, a tolerância a falhas é obtida por redundância. No caso de mecanismos de tolerância a falhas por software, esta redundância significa basicamente **replicação** de dados ou processos. A replicação permite que, caso a máquina onde uma das réplicas está situada falhe ou torne-se indisponível temporária ou permanentemente, o serviço continue disponível utilizando-se de outra réplica acessível. Em sistemas distribuídos orientados a objetos, a coleção de réplicas é organizada como um grupo de objetos que cooperam para atingir um objetivo comum. Um grupo de objetos pode ser visto como um provedor de serviços para o sistema distribuído como um todo [CAR97]. A transparência proporciona a um cliente que usa os serviços de um grupo de objetos comunicar-se com apenas um objeto representante do grupo, sem estar ciente da composição do grupo.

O uso da programação orientada a grupos com o emprego de ferramentas que forneçam suporte à comunicação *multicast* confiável tem se mostrado

[1] O usuário aqui tem a conotação de quem usa o sistema e seus recursos. Portanto, o programador de aplicações enquadra-se neste perfil.

bastante interessante para diminuir a complexidade de implementação de aplicações distribuídas tolerantes a falhas [LIA90, BIR93, VER92]. Este conjunto de ferramentas tem sido denominado de sistema de comunicação de grupo.

Esta mesma filosofia pode ser aplicada ao desenvolvimento de aplicações distribuídas tolerantes a falhas no modelo de objetos. Uma aplicação no modelo de objetos é estruturada a partir de classes, que definem as estruturas de dados e as funcionalidades esperadas da aplicação, e que determinam os serviços e as interações dos objetos instanciados durante o processo de execução. Alguns desses serviços podem ser considerados críticos pela aplicação, tornando críticos os objetos responsáveis pelos serviços [LIS95, LIS97]. Sob o ponto de vista de propriedades, a replicação estrutural de objetos considerados críticos e sua execução em distintos processadores promete um aumento de confiabilidade.

No processo de desenvolvimento de uma aplicação que utiliza replicação de componentes, existem três preocupações: a implementação dos componentes da aplicação, a obtenção de réplicas dos componentes considerados críticos e a implementação da técnica de gerência das réplicas. Para simplificar a tarefa de obtenção de réplicas, pode-se utilizar componentes específicos para esta finalidade que fornecem à aplicação as réplicas desejadas, no decorrer do processo de execução, juntamente com seu estado [AMA99]. Ou seja, a partir de um ou mais objetos da aplicação considerados críticos, componentes da biblioteca proposta se encarregam da sua replicação e conseqüente distribuição.

Após a obtenção de uma ou mais cópias de um objeto da aplicação, estas cópias devem ser transmitidas para outros processadores, para que formem a base de um sistema de replicação com vistas à tolerância a falhas.

A gerência da replicação, seja pela técnica de cópia primária, de réplicas ativas ou votação, necessita como ponto de partida cópias idênticas de um mesmo componente. Em seguida, as diversas réplicas devem ser mantidas em estado consistente, de forma contínua ou sob demanda. Um estado consistente contínuo pode ser obtido por execução concorrente e monitorizada, ao passo que um estado consistente sob demanda pode ser obtido por atualização do estado da réplica somente quando esta se tornar necessária num determinado momento da execução.

Adicionalmente, busca-se a execução de técnicas de tolerância a falhas usando métodos que possam ser reaproveitados e que tenham o menor custo possível de projeto.

Este trabalho aborda a implementação da replicação através da utilização da linguagem Java e de seu sistema de invocação remota de métodos (*Remote Method Invocation* - RMI). Em trabalho anterior, Filho [FIL99], foi realizado um estudo sobre implementação de replicação usando Java. Usando RMI, a implementação de grupos de objetos pode ser estruturada de acordo com a arquitetura cliente/servidor, sendo o cliente o representante (a interface) de um grupo de objetos e os servidores representando os demais componentes do grupo. Adotando Java como ferramenta de implementação, busca-se um ambiente de desenvolvimento de programas de grande portabilidade, eficiente e que oferece facilidades para a criação e utilização de bibliotecas de componentes.

A facilidade de disponibilizar bibliotecas de componentes para aplicações especiais incentiva a busca de soluções para problemas recorrentes, na forma

de componentes genéricos e reusáveis. O objetivo é claro: liberar o programador de aplicações do esforço de ter que implementar, de forma particular, serviços que podem ser oferecidos de forma independente ao domínio específico da aplicação. Um benefício adicional é propiciar uma solução elegante, explorando recursos da linguagem nem sempre bem conhecidos e dominados pelos seus usuários [LIS98].

1.2 Contribuições

O objeto de estudo deste trabalho é o problema da adição de replicação de componentes em aplicações centralizadas, através da implementação de grupos de objetos compostos por coleções de objetos remotos replicados. O acesso aos objetos remotos é efetuado através da chamada remota de seus métodos e com o mesmo nível de simplicidade de objetos não replicados. A replicação desses objetos exige o controle da replicação de dados [GUE96].

Inicialmente, a implementação da replicação foi feita através de estudos de casos e, progressivamente o nível de abstração foi sendo aumentado, tendo por objetivo final o desenvolvimento de uma biblioteca de componentes genéricos destinados a apoiar o desenvolvimento de aplicações tolerantes a falhas. Este trabalho busca dar continuidade ao trabalho de Amaral et al. [AMA99], que utiliza classes genéricas para a obtenção de cópias de objetos, fazendo a seguir a sua serialização para seu transporte em redes de comunicação.

Para a implementação do serviço de controle da replicação será utilizada a abordagem de replicação primário-*backup*. Esta técnica consiste em determinar um objeto primário que se encarrega da comunicação com todas as réplicas, sendo cada réplica capaz de suportar uma falha. Neste trabalho, o objeto primário atua como uma "máquina de replicação", no sentido que se encarrega de criar as réplicas e distribuí-las em diferente processadores, receber as mensagens de solicitação de serviços, disseminá-las entre as réplicas e assegurar a consistência de estado entre as réplicas.

1.3 Organização do Texto

No capítulo 2, são apresentadas as características fundamentais dos sistemas distribuídos no contexto deste trabalho, bem como o conceito de processo e sua relação com os processadores do sistema. A comunicação e sincronização entre os processos ou processadores de um sistema distribuído é discutida e seus mecanismos são destacados através de uma analogia simples. Ao final, a comunicação de grupos é apresentada como uma ferramenta para a implementação de mecanismos confiáveis de comunicação.

O capítulo 3 apresenta inicialmente uma relação de modelos de falhas e suas características. A seguir são abordadas três técnicas consagradas de replicação: cópia primária (primário-*backup*), replicação ativa e votação. São discutidas características e comportamentos, principalmente das técnicas da cópia primária e da replicação ativa. Por ser uma parte importante no processo de implementação da replicação, a comunicação de grupos é discutida de forma mais específica ao contexto de forma a ressaltar algumas conseqüências da utilização, em conjunto, das técnicas de replicação e de comunicação de grupos.

O capítulo 4 descreve brevemente alguns aspectos da linguagem Java objetivando ilustrar características importantes que são utilizadas neste trabalho, tal como: a arquitetura RMI (*Remote Method Invocation*).

O capítulo 5 apresenta a máquina de replicação. Inicialmente, é oferecido um modelo do sistema de forma a contextualizar e expor as hipóteses assumidas no trabalho. A seguir, um modelo de replicação oferece detalhes do funcionamento do mecanismo de replicação utilizado. Por ser parte importante do mecanismo de replicação, um protocolo de *commitment* atômico é descrito e detalhado de forma a ilustrar o problema da atomicidade na entrega das mensagens. A seguir, para facilitar a compreensão dos componentes da máquina de replicação, é esquematizado um diagrama de classes das classes envolvidas. Finalmente, são mostrados os principais algoritmos utilizados no mecanismo de replicação.

O capítulo 6 descreve os detalhes de implementação da máquina de replicação. A divisão cliente/servidor, característica das aplicações RMI, é ressaltada bem como a criação de interfaces, utilização de objetos como parâmetros e serialização de objetos. Ao final, são mostrados alguns detalhes da execução da máquina de replicação.

O capítulo 7 apresenta algumas constatações feitas a respeito de trabalhos relacionados.

O capítulo 8 apresenta as conclusões deste trabalho.

2. Conceitos Básicos de Sistemas Distribuídos

Este capítulo apresenta as características fundamentais dos sistemas distribuídos, tomadas por base para o desenvolvimento deste trabalho, com o objetivo de contextualizar o estudo no ambiente dos sistemas distribuídos.

2.1 Premissas Básicas

O termo **sistemas distribuídos** é usado neste texto para fazer referência a um sistema computacional com as seguintes características: consiste de muitos computadores que não compartilham memória ou relógio; os computadores comunicam-se uns com os outros por troca de mensagens, através de uma rede de comunicação (FIGURA 1); cada computador tem sua própria memória e executa seu próprio sistema operacional. Os recursos próprios e gerenciados por um computador são ditos locais, enquanto os recursos possuídos por outros computadores, controlados por esses, cujo acesso ocorre apenas através da rede, são ditos remotos. Tipicamente o acesso a recursos remotos é mais caro devido aos atrasos de comunicação que ocorrem em uma rede e ao *overhead* da CPU (*Central Processing Unit*) para processar os protocolos de comunicação [SIN94].

Os sistemas operacionais distribuídos estendem os conceitos de gerência de recursos e interface amigável dos sistemas computacionais tradicionais [SIN94]. Um sistema operacional distribuído aparece para o usuário como um sistema operacional centralizado para uma única máquina, mas ele é executado sobre múltiplos computadores independentes. O conceito chave de sua implementação é a transparência: em outras palavras, o uso de múltiplos processadores e o acesso de dados remotos deveria ser invisível (transparente) para o usuário. Segundo Tanenbaum, apud [SIN94], o usuário vê o sistema como um uniprocessador virtual, e não como uma coleção de máquinas distintas. Ou seja, o usuário submete um trabalho ao sistema operacional distribuído, que gerencia sua execução distribuída. O usuário não sabe em quais computadores o trabalho foi executado, em quais computadores estão armazenados os arquivos necessários para a execução, ou como foi feita a comunicação e sincronização entre os diversos computadores.

Pode-se subdividir os sistemas distribuídos, quanto ao tipo de interação entre os processadores, em acoplamento forte (*tightly coupled*) e acoplamento fraco (*loosely coupled*). No primeiro caso, os elementos têm uma memória comum, o que possibilita implementar a sincronização e comunicação através de variáveis compartilhadas. Nos sistemas fracamente acoplados deve existir troca de mensagens pois não há memória comum. No enfoque de pesquisa, tem sido considerado como parâmetro base os sistemas cujo acoplamento é fraco; também aqui estes serão doravante pressupostos.

FIGURA 1 – Esquema ilustrativo de um sistema distribuído [SIN94]

Segundo Burns [BUR96], um sistema computacional distribuído é um sistema de múltiplos elementos autônomos cooperando em uma tarefa comum ou para atingir um objetivo. Esta definição exclui *pipelines* e *arrays* de processadores porque cada um dos elementos componentes não é autônomo e exclui redes de computadores cujos nodos não trabalhem por um objetivo em comum como, por exemplo, a Internet. Um sistema distribuído aplicado em um ambiente de fabricação é mostrado na FIGURA 2.

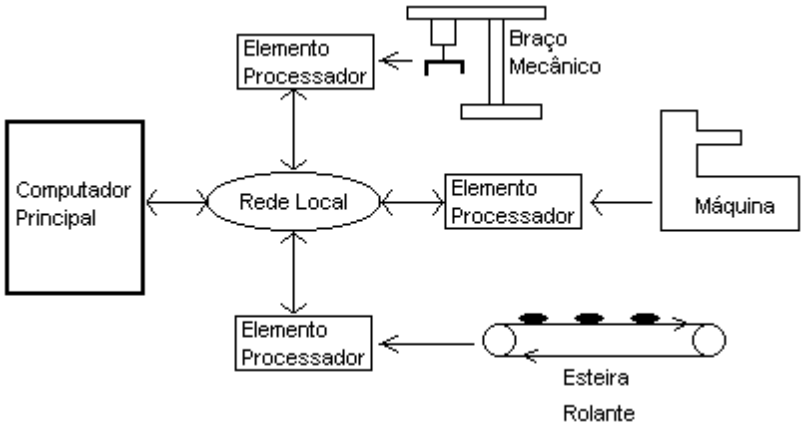

FIGURA 2 - Sistema distribuído em ambiente de fabricação [BUR96]

Em Burns [BUR96] são considerados somente sistemas cujo acoplamento é fraco, sendo assumida comunicação total entre os nodos. Cada nodo ou processador tem seu próprio relógio e estes relógios são fracamente sincronizados, isto é, existe um intervalo dentro do qual eles podem diferir.

2.2 Problemas Inerentes aos Sistemas Distribuídos

É necessário perceber que o não compartilhamento de relógio e memória nos sistemas distribuídos tem implicações no projeto e desenvolvimento dos mesmos.

Nestes sistemas, não existe a noção de tempo global. Este problema não pode ser resolvido através de um relógio único e comum a todos os processos porque dois processos diferentes podem observar, no mesmo instante, valores diferentes de tempo devido aos atrasos imprevisíveis de comunicação. Também não é uma boa idéia que cada processo tenha seu próprio relógio porque a utilização de vários relógios individuais apresenta o problema das diferentes derivas dos mesmos, levando a diferenças significativas com o passar do tempo. Dependendo da precisão dos relógios

individuais, duas solicitações não simultâneas poderiam perceber valores idênticos. Atualmente, existem tentativas bastante desenvolvidas de operação com relógios baseados na hora oficial e medidas obtidas a partir da captação de sinal de satélite. Mas estes sistemas ainda são caros e exigem o desenvolvimento de suporte adequado.

Segundo Singhal e Shivaratri, apud [CEC98], como os processos que formam o sistema distribuído não possuem memória principal compartilhada, para que obtenham dados referentes a todo o sistema devem recorrer à troca de mensagens.

Um processo de um sistema distribuído pode obter uma visão coerente porém parcial ou uma visão total porém incoerente. Uma visão é dita coerente se todas as observações nos diferentes processos foram feitas no mesmo tempo físico.

Um processo é capaz de registrar o seu próprio estado e as mensagens enviadas e recebidas, formando um estado local. Para determinar um estado global do sistema, um processo deve solicitar que os outros processos também registrem seus estados locais e os enviem. Uma vez que os processos não têm acesso a um relógio comum, não conseguem registrar seus estados locais simultaneamente, levando à necessidade de mecanismos que permitam a obtenção de estados globais consistentes.

A palavra-chave para os problemas descritos acima é **sincronização**. O conceito de sincronização será mostrado nas próximas seções.

2.3 Processos Distribuídos

Não obstante existirem muitas definições de processo, de acordo com Singhal e Shivaratri [SIN94], o conceito de processo no contexto de sistemas distribuídos corresponde a um programa, ou a parte de um programa, cuja execução começou e que ainda não foi completada, ou seja, é um programa em execução.

Do ponto de vista de tolerância a falhas, para simplificar a análise, considera-se a premissa de que cada processo deve ser executado, integralmente, em um único processador e que cada processador abriga apenas um processo por vez.

Um processo pode estar em qualquer um dos três estados básicos seguintes: em execução, com o processador executando as instruções do processo correspondente; pronto, quando o processo está pronto para ser executado mas o processador ainda não está disponível para a sua execução; ou bloqueado, quando o processo está aguardando a ocorrência de algum evento.

A FIGURA 3 ilustra estas transições de estado durante o ciclo de vida do processo. Um **processo em execução** torna-se **bloqueado** porque a requisição de um recurso não está disponível ou pode tornar-se **pronto** porque o processador decidiu executar outro processo. Um processo **bloqueado** torna-se **pronto** quando o recurso requisitado torna-se disponível para ele. Um processo **pronto** começa a ser **executado** quando a CPU torna-se disponível para ele.

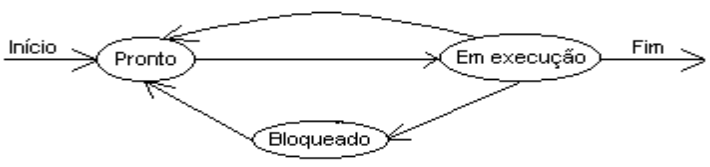

FIGURA 3 - Transições de estado [SIN94]

Dois processos são concorrentes se suas execuções podem potencialmente se sobrepor no tempo, isto é, a execução do segundo processo começa antes do fim da execução do primeiro processo. Em sistemas multiprocessados, onde as CPUs podem executar simultaneamente processos diferentes, é concreto e fácil de visualizar a concorrência. Em sistemas com um única CPU, a concorrência física pode ocorrer devido à execução concorrente da CPU e com uma atividade de entrada/saída. Se a CPU intercala a execução de vários processos então ocorre a concorrência lógica.

Dois processos são **seqüenciais** se a execução de um deve ser completada antes da execução do outro começar. Se dois processos não interagem então suas execuções são transparentes entre eles, isto é, sua execução concorrente é equivalente à sua execução serial.

2.4 Objetos Distribuídos

Um objeto distribuído é um objeto ao qual pode-se ter acesso remotamente, isto é, pode ser usado como um objeto comum mas não precisa estar junto ao processo ou processador que o solicita.

Este conceito pode ser mais facilmente entendido se considerada, como exemplo, uma aplicação distribuída orientada a objetos que faz acesso a um banco de dados. Tradicionalmente este tipo de aplicação divide-se em duas camadas: a aplicação propriamente dita e o banco de dados (FIGURA 4). Quando ocorre uma mudança em alguma regra de negócio da aplicação é necessário redistribuir a alteração por todo o sistema.

Uma abordagem mais atual é a construção de aplicações em três camadas: uma interface, uma camada de regras de negócio e o banco de dados (FIGURA 4). Esta abordagem acrescenta uma camada de regras de negócio onde ficam os objetos aos quais a camada de interface proporciona o acesso remoto. Dessa forma, quando ocorre uma mudança em uma regra de negócio, uma única alteração feita nesta camada é refletida para todo o sistema.

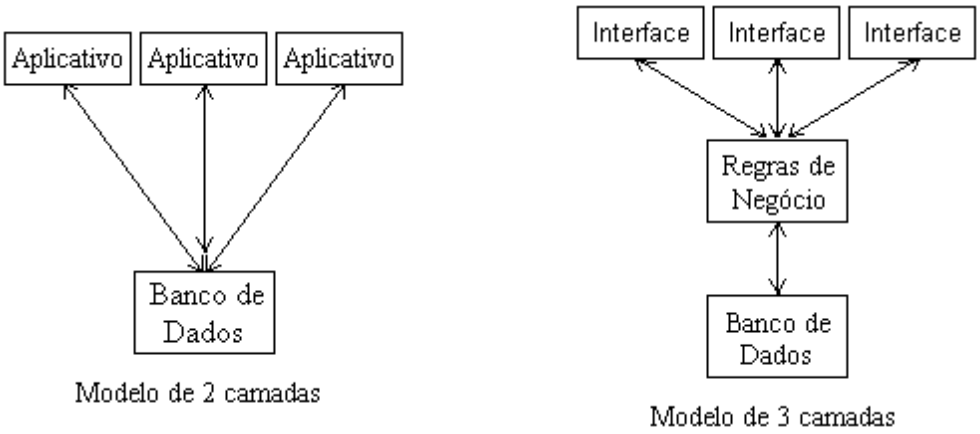

FIGURA 4 - Modelos de aplicações em 2 e 3 camadas [WUT97]

2.5 Comunicação e Sincronização

Para Burns [BUR96], a maior dificuldade associada à programação distribuída e concorrente resulta da necessidade de interação entre os processos. Apesar de se poder definir sistemas distribuídos com nodos ou processadores independentes, este comportamento não corresponde a um modelo prático usual. Portanto deve-se esperar sempre uma dependência crítica de **comunicação** e **sincronização** entre os processos.

Comunicação é a passagem de informações de um processo para outro através de algum meio, conforme visto. Neste trabalho, é considerada apenas a comunicação por troca de mensagens.

Sincronização é a satisfação de requisitos sobre a intercalação de ações de diferentes processos (por exemplo, uma ação particular de um processo só pode ocorrer após uma ação especificada de outro processo). Também pode-se utilizar o termo no sentido de levar dois processos simultaneamente a estados predefinidos. Estes dois conceitos são intimamente ligados, uma vez que, algumas formas de comunicação requerem sincronização, e alguns mecanismos de sincronização requerem comunicação.

2.5.1 Mecanismos de Sincronização

Todo sistema de comunicação baseado em troca de mensagens possui uma sincronização implícita: um processo não pode receber uma mensagem antes dela ter sido enviada para ele. O processo de sincronização pode variar de acordo com a semântica das operações de envio de mensagens e pode ser classificado como segue [BUR96]:

- processo assíncrono (ou sem espera): o remetente continua imediatamente, independentemente da mensagem ter sido recebida.
- processo síncrono: o remetente só continua quando a mensagem enviada for recebida.
- processo de invocação remota: o remetente só continua quando chegar uma resposta do destinatário.

Para esclarecer melhor as idéias expostas, considere-se a seguinte analogia:

- a postagem de uma carta é um envio assíncrono: uma vez que a carta foi enviada, o remetente prossegue com sua vida. Somente a chegada de uma carta resposta, ou seja, um aviso de recebimento, pode garantir ao remetente que sua carta chegou ao destino. Do ponto de vista do destinatário, a carta recebida só informa de fatos passados até o momento de sua postagem, isto é, ela não informa a situação atual do remetente.
- um telefonema é um envio síncrono: o originador da chamada aguarda até que a ligação seja completada e a identidade de quem está do outro lado da linha seja confirmada; só então a mensagem é enviada. Se o destinatário da chamada puder responder imediatamente, isto é, durante a mesma chamada, a sincronização corresponderá a uma invocação remota. Devido ao originador e ao destinatário "reunirem-se" na sincronização da comunicação, este processo é muitas vezes

denominado de ***rendezvous.*** Uma invocação remota é chamada de ***rendezvous* estendido**.

2.6 Comunicação de Grupos

Um grupo é um conjunto de objetos que agem de maneira especificada pelo sistema ou por um usuário. Um objeto que participa de um grupo é chamado membro ou elemento. Cada objeto (estação, processo) possui sua própria memória local e não tem acesso direto aos dados de outro objeto. Em decorrência disto, os objetos trocam informações através de mensagens para realizar a requisição de serviços. Um objeto cliente requisita um serviço para um objeto servidor. O cliente aguarda por uma resposta, enquanto o servidor realiza o serviço [AMA98].

O conceito de grupos permite uma abstração simples, facilitando o desenvolvimento de aplicações distribuídas confiáveis [BIR93]. O modelo de grupos precisa garantir que todos os objetos pertencentes a um grupo recebam e processem as mensagens na mesma ordem.

De acordo com Cosquer e Veríssimo, apud [NUN98], um grupo pode ainda ser conceituado como um conjunto de entidades passivas (dados) ou ativas (processos) relacionadas, e que pode ser endereçado como sendo uma unidade. Por exemplo, um grupo denominado $\mathbf{g_x}$ pode representar de forma abstrata o conjunto das réplicas de **x**: os membros de $\mathbf{g_x}$ correspondem às réplicas de **x**, e $\mathbf{g_x}$ pode ser usado para endereçar o conjunto das réplicas de **x**. Assim enviar uma mensagem para todas as réplicas de **x** pode ser feito sem necessariamente nomear o conjunto de todas as réplicas do objeto **x** (Guerraoui, apud [NUN98]).

Em Montresor [MON99], encontra-se a especificação do paradigma de comunicação de grupos mais voltado ao ponto de vista da replicação. Grupos são abstração chave para a comunicação de grupos e esta, por sua vez, permite o desenvolvimento de aplicações confiáveis e de alta disponibilidade através da replicação. Um grupo é uma coleção de membros (processos ou objetos) que compartilham o mesmo objetivo e mantêm ativamente um estado replicado [MON99]. Ainda que vários serviços de comunicação de grupos tenham aparecido no últimos anos, e entre eles existam várias diferenças, o mecanismo chave destas arquiteturas é o mesmo: um serviço de associação (*membership)* e um serviço de distribuição confiável de mensagens. A tarefa do serviço de *associação* é manter os membros do grupo informados sobre as mudanças que ocorrem na composição do grupo. Esta tarefa é realizada através da instalação de *visões*. A composição do grupo pode variar devido a pedidos voluntários para entrar ou sair do grupo, eventuais falhas ou reparos dos membros do grupo e falhas no sistema de comunicação. As visões consistem de uma coleção de membros e representa a percepção de quem faz parte do grupo que é compartilhada pelos membros do mesmo. A tarefa de um serviço de *multicast* confiável é possibilitar aos membros do grupo a comunicação por mensagens *multicast*. Mensagens entregues são integradas com as visões da seguinte forma: dois membros que instalam o mesmo par de visões na mesma ordem entregam o mesmo conjunto de mensagens entre a instalação dessas visões. Esta semântica, chamada de visão síncrona [GUE96], permite aos membros deduzir sobre o estado de outros membros usando somente informação local.

Duas classes de sistemas de comunicação de grupos têm surgido: partição primária e particionáveis. Um sistema de partição primária procura manter uma única visão dos membros correntes do grupo. Os membros que são excluídos desta visão não podem participar da computação distribuída. Por outro lado, sistemas particionáveis permitem múltiplas visões simultâneas, cada uma representando uma das partições em que a rede esta subdividida. Os membros de uma visão podem continuar a computação distribuída separadamente dos membros não incluídos na visão. Serviços de comunicação de grupo de partição primária são apropriados para sistemas não particionáveis ou para aplicações que precisam manter um único estado através do sistema. Sistemas particionáveis são destinados a aplicações que são capazes de tirar vantagem do seu conhecimento sobre as falhas de partição, a fim de permitir o progresso em partições múltiplas e concorrentes [MON99].

2.6.1 Classificação

Esta classificação, baseada na definição de grupos [BIR87], divide os grupos de acordo com sua estrutura interna:

- Grupos fechados: são grupos onde só existe troca de mensagens entre os integrantes do mesmo. Processos externos não podem enviar mensagens para o grupo como um todo, mas podem fazê-lo individualmente para membros do grupo.
- Grupos abertos: são grupos onde há trocas de mensagens entre o grupo e processos externos ao grupo. Dessa forma, qualquer processo do sistema é capaz de enviar mensagens para um grupo aberto.
- Grupos hierárquicos: são grupos onde há um processo coordenador que gerencia as tarefas dos demais processos do grupo.
- Grupos não hierárquicos: são grupos onde os elementos são homogêneos, não existe um coordenador e todas as decisões são tomadas coletivamente.
- Grupos estáticos: neste tipo de grupo, o conjunto de membros não muda durante o tempo de vida do sistema. Isto não significa que os membros do grupo não sofram falhas. Significa que o conjunto de membros do grupo não muda para refletir a falha de um deles, ou seja, um membro quando sofre um falha continua a pertencer ao grupo.
- Grupos dinâmicos: neste tipo de grupo, pode haver alterações no conjunto de membros durante o tempo de vida do sistema. Por exemplo: um membro que falhou é removido do grupo e quando for recuperado junta-se novamente ao grupo. Este noção de *visão* é usada para modelar o envolvimento entre os membros de um grupo. Uma seqüência de visões de um grupo é dito **histórico** do grupo [GUE96].

É denominada de sobreposição de grupos a situação onde um processo pode ser membro de mais de um grupo ao mesmo tempo.

Outra classificação feita por Liang et al. [LIA90] estendeu a anterior e incluiu o conceito de objetos e grupos de objetos. No entanto, em ambas as definições, o modelo de funcionamento do grupo é o mesmo. Desta forma, a redefinição proposta não invalida a anterior, mas sim, apresenta uma visão obtida a partir da aplicação, o que permite ampliar a classificação dos grupos quanto a sua estrutura interna, e também classificá-los quanto ao seu comportamento externo [NUN98].

Por isso, levando em consideração a presença de homogeneidade dos objetos e operações, a visão da aplicação permite a seguinte classificação [NUN98]:

- Grupos homogêneos: são grupos onde todos os membros mantém réplicas de objetos e/ou operações idênticos.
- Grupos heterogêneos: são grupos onde os membros possuem objetos e/ou operações diferentes.

Finalmente, quanto a seu comportamento externo, os grupos podem ser:

- Grupos determinísticos: são grupos que requerem alta confiabilidade na comunicação para manter a consistência entre os membros.
- Grupos não-determinísticos: são grupos onde a comunicação pode ser não confiável e problemas de consistência são controlados pela aplicação.

2.6.2 Tecnologias de Comunicação

Existem três formas diferentes para distribuir informações para o computadores de um sistema distribuído. A mais simples delas é o *unicast*, onde uma mensagem originada por um computador é endereçada direta e unicamente a outro. Isto é comunicação um-para-um. Por outro lado, quando uma mensagem não é endereçada para um destinatário em particular, mas simplesmente enviada para qualquer um que possa recebê-la, chama-se de *broadcast* [SOR98].

Entre estas duas formas, existe uma terceira, que será a ideal no contexto de grupos: uma mensagem não é simplesmente transmitida para qualquer um, mas também não é endereçada diretamente para um único computador. Esta forma de distribuição chama-se *multicast*. Esta forma pode ser descrita como um *broadcast* endereçado (restrito), para um grupo de destinatários.

Apesar do *multicast* ser o ideal para a comunicação de grupos, seu uso dependerá de características do sistema existente. Se o sistema permitir a criação de endereços especiais que podem ser vistos por várias estações, o *multicast* pode ser implementado atribuindo a cada grupo um endereço especial para a comunicação. O *broadcast* pode ser uma alternativa quando não houver a possibilidade de implementação do *multicast*. No entanto, é preciso ter em vista que o *broadcast* enviará mensagens mesmo para as estações não interessadas, além do que, não há segurança pois mensagens do grupo podem ser vistas por estações que não fazem parte do grupo. O *unicast* também pode ser usado para implementar comunicação de grupo. Desta forma, o transmissor deve enviar uma mensagem - através de múltiplos *unicasts* - para todos os membros do grupo. Esta implementação pode ser bastante onerosa para o sistema visto que, se um grupo possuir **n** membros, então serão necessários **n** envios de mensagens [AMA98].

2.6.3 Comunicação Confiável

No contexto dos sistemas distribuídos, é necessária a comunicação confiável porque isto diminui a complexidade da programação envolvida e também porque geralmente uma aplicação distribuída não tem condições de reconstruir a informação. Portanto, é necessário que as camadas de transporte ofereçam serviços confiáveis [SOR98].

Existem diversos tipos de protocolos usados para assegurar a comunicação confiável. Vários destes protocolos foram estudados e analisados através da monografia desenvolvida por Amaral [AMA98a].

Uma forma simples de implementar a comunicação confiável é descrita a seguir. Considere-se um sistema que requer que os destinatários do *multicast* informem ao remetente, através de uma mensagem de reconhecimento, que o pacote foi recebido. O remetente irá retransmitir a mensagem para cada destinatário que não enviar a mensagem de reconhecimento, em um certo período de tempo. A mensagem de reconhecimento é esperada de todos os membros do grupo.

Considere-se um grupo cujos membros estão localizados fisicamente distantes uns dos outros. Nesta situação, se o membro mais distante do grupo (segundo o ponto de vista do remetente) perde uma mensagem e solicita uma retransmissão, a solicitação deverá viajar por uma longo caminho e a retransmissão da mesma deverá voltar pelo mesmo longo caminho. Além do atraso gerado devido às mensagens perdidas, este método onera consideravelmente o remetente visto que, ele deve processar as solicitações de retransmissão de todos os membros do grupo.

Uma alternativa para reduzir a latência e a carga sobre o remetente seria que a retransmissão pudesse ser solicitada para outros membros do grupo, alguém mais próximo, por exemplo, e não ao remetente.

Outra alternativa seria considerar que pelo menos 50% dos membros do grupo sempre receberão os pacotes e não esperar nenhum tipo de mensagem de reconhecimento por parte dos destinatários. Um destinatário poderia verificar o número seqüencial das mensagens e caso detectasse uma perda ele enviaria um "NACK" (*negative acknowledgment*) para o remetente, ou seja, ao invés de esperar um reconhecimento para cada mensagem de cada membro do grupo, o remetente espera apenas por solicitações de retransmissão. Isto diminui a troca de mensagens e o processamento do remetente.

Porém, mesmo esperando apenas solicitações de retransmissão, podem ocorrer atrasos indesejáveis se estas solicitações vierem de membros muito distantes do remetente. Este problema pode ser resolvido se os destinatários puderem sempre contactar membros mais próximos (seja em distância ou em tempo de transmissão). Isto implica que os destinatários precisam saber sua própria localização em relação a seus vizinhos (outros membros do grupo) e ao remetente. Este método de implementação requer um esquema eficiente que permita a cada membro encontrar seu lugar no grupo. Uma maneira simples seria medir os tempos de transmissão das mensagens enviadas para cada membro do grupo e então montar uma tabela dos vizinhos mais próximos de cada membro. O melhor momento para gerar esta tabela seria quando um novo membro entra no grupo. A FIGURA 5 mostra um grupo de *hosts* (nodos hospedeiros) em diferentes redes, conectados por roteadores. O *host* "E" não faz parte do grupo. O outros quatro são e mantêm, cada um, uma tabela de vizinhos.

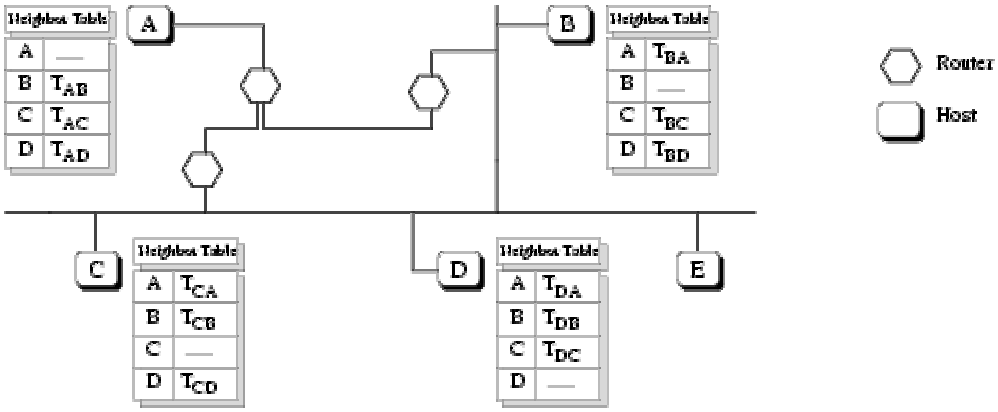

FIGURA 5 - Tabelas de vizinhos [SOR98]

Deve-se considerar os problemas associados a formação de um novo grupo, por um certo número de membros; se todos começam, ao mesmo tempo, a requerer informações para coletar dados sobre seus vizinhos, pode ocorrer um acréscimo de tráfego comprometendo o desempenho da rede e levando a geração de tabelas de vizinhos inconsistentes. Isto pode ser minimizado se os membros possuírem um mecanismo que introduza um atraso pequeno e aleatório, antes de iniciar as transmissões.

2.7 Conclusões

Os conceitos apresentados sobre sistemas distribuídos procuraram inicialmente caracterizar o sistema em estudo. Em seguida, definiu-se a relação entre os elementos do sistema onde destacam-se como pontos chave os **recursos remotos.** O acesso remoto foi apresentado de forma a ressaltar um ponto fundamental na sua implementação: a **transparência.**

Para ilustrar a complexidade intrínseca desses sistemas, foram apresentados alguns aspectos das dificuldades que surgem na distribuição dos mesmos. Não foram considerados problemas técnicos ligados a tecnologias em particular mas sim problemas conceituais, tais como inexistência de tempo global, obtenção de visões coerentes e troca de mensagens. As noções de processos e objetos distribuídos foram apresentadas como partes da solução do mesmo problema: oferecer **recursos remotos**.

A interação entre os elementos do sistema apresenta-se como a maior dificuldade neste contexto. Por isso, a **comunicação** entre os elementos foi apresentada juntamente à **sincronização** para demonstrar a dependência dos mecanismos. Foi apresentado o conceito de **comunicação de grupos** como uma ferramenta para garantir as propriedades da sincronização, tais como **atomicidade** e **ordenação**.

3 Replicação

No passado, era considerado aceitável que um serviço ficasse indisponível devido a falhas. Este idéia evoluiu e, atualmente, empresas como bancos, financeiras, controle industrial e telecomunicações necessitam cada vez mais de aplicações confiáveis e disponíveis.

Inicialmente a tolerância a falhas foi obtida pela replicação de *hardware*. Porém, considerações econômicas, novas aplicações e novas formas de uso dos sistemas, e a necessidade de baratear os custos motivaram a busca de soluções baseadas em *software*. A idéia da replicação por *software* é multiplicar serviços (também chamados de objetos) ou dados, em vários servidores independentes para oferecer tolerância a falhas. Quando uma réplica falha, seja por falha de comunicação ou local, os dados desta réplica devem ficar acessíveis a partir de alguma outra que esteja operacional.

A desvantagem da replicação é que ela introduz complexidade no sistema à medida que exige mecanismos de gerência e consistência de réplicas. Manter a consistência significa garantir que todas as requisições sejam processadas por todas as cópias na mesma ordem, a partir de estados iniciais idênticos [GUE96].

O uso da replicação depende do nível de tolerância a falhas que se deseja obter. Para Schneider [SCH93a], *softwares* distribuídos são muitas vezes estruturados em termos de *clientes* e *serviços*. Cada serviço compreende um ou mais *servidores* e *operações* que são invocadas pelos *clientes* através de *requisições*. Embora o uso de um único servidor centralizado seja a maneira mais fácil de implementar um serviço, o serviço resultante pode ser tão intolerante a falhas quanto o servidor utilizado; ou seja, uma falha no servidor centralizado implica uma falha geral no sistema. Quando esta ausência de tolerância a falhas não for aceitável deve-se utilizar vários servidores que falham independentemente. Neste caso, réplicas de um servidor são, em geral, executadas em diferentes processadores de um sistema distribuído e são utilizados protocolos para coordenar as interações dos clientes com as réplicas.

Assim, este capítulo tem o objetivo de abordar técnicas de replicação consagradas - votação, replicação ativa e primário-*backup* - para oferecer uma análise sobre a sua aplicabilidade neste trabalho.

3.1 Modelos de Falhas

Antes de tratar diretamente as técnicas de replicação, faz-se necessário a introdução de conceitos e terminologias a respeitos de falhas. Um componente é considerado falho quando seu comportamento não está de acordo com sua especificação. Segundo Schneider [SCH93], uma grande variedade de modelos de falhas tem sido propostos para sistemas distribuídos. Todos eles baseiam-se em atribuir os comportamentos indesejados aos componentes do sistema; por isso, um sistema é denominado *t-falhas* tolerante a falhas quando ele continua satisfazendo suas especificações até que não mais que *t* de seus componentes estejam falhos.

A relação a seguir, baseada em Schneider [SCH93], descreve os modelos de falhas mais comumente encontrados na literatura:

- **Parada (***Failstop***)**: um processador falha por parada. Uma vez que ele pára, permanece neste estado. O fato de que o processador falhou é detectável por outros processadores (Schneider [1984]).
- **Colapso (***Crash***)**: um processador falha por parada. Uma vez que ele pára, permanece neste estado. O fato de que o processador falhou pode não ser detectável por outros processadores (Lamport and Fischer [1982]).
- **Colapso + *link* (***Crash+link***)**: um processador falha por parada. Uma vez que ele pára, permanece neste estado. Um *link* falha perdendo algumas mensagens, mas não atrasa, duplica ou corrompe mensagens (Budhiraja *et ali* 92).
- **Omissão no recebimento (***Receive Omission***)**: um processador falha por receber somente um subconjunto das mensagens que foram enviadas para ele ou por parar e permanecer parado (Perry and Toueg [1986]).
- **Omissão no envio (***Send Omission***)**: um processador falha por transmitir somente um subconjunto das mensagens que ele deve enviar ou por parar e permanecer parado (Hadzilacos [1984]).
- **Omissão geral (***General Omission***)**: um processador falha por receber somente um subconjunto das mensagens que foram enviadas para ele, por transmitir somente um subconjunto das mensagens que ele atualmente tenta enviar, ou por parar e permanecer parado (Perry and Toueg [1986]).
- **Falha Bizantina (***Byzantine Failures***)**: um processador falha por exibir um comportamento arbitrário (Lamport, S*host*ak and Pease [1982]).

Nas falhas tipo Parada, um processador nunca realiza operações erradas e as falhas são detectáveis. Assim, outros processadores podem seguramente realizar ações em nome de um processador que falhou por Parada.

A menos que o sistema seja síncrono não é possível distinguir entre um processador que está em execução muita lenta e um que parou devido a uma falha de Colapso. Um processador que falhou por Colapso não pode realizar operações mas um processador que está lento pode. Outros processadores podem realizar operações em nome do processador que falhou por Colapso mas não podem realizar operações em nome do processador lento porque as ações subseqüentes do mesmo podem não ser consistentes com as operações realizadas em seu nome por outros processadores. Assim, falhas de Colapso em sistemas assíncronos são mais difíceis de tratar que as falhas de Parada. Em sistemas síncronos as falhas de Colapso e Parada são equivalentes.

Os quatro modelos seguintes de falhas - Colapso+*link*, Omissão no recebimento, Omissão no envio e Omissão geral - tratam da perda de mensagens e cada um dos modelos determina diferentes causas à perda e atribui esta perda a diferentes componentes. Finalmente, as falhas Bizantinas são as mais complexas de serem tratadas. Um sistema que pode tolerar este tipo de falhas pode tolerar qualquer uma das anteriores.

Para este trabalho não serão consideradas falhas de comunicação e nem falhas Bizantinas e o modelo de sistema adotado será síncrono.

3.2 Técnicas de Replicação

Há basicamente três abordagens pessimistas para o controle de réplicas:

- Votação: exige o controle de quoruns.

- Replicação ativa: não há nodo centralizador e são necessários mecanismos que garantam a coordenação de atividades entre as réplicas.
- Cópia primária: o nodo primário é centralizador e por isso, deve haver uma estratégia para definir um novo primário caso ocorra uma falha no primário atual.

3.2.1 Votação

Segundo Goodman, Skeen e Chan, apud [GUE96], esta técnica utiliza o método de **réplicas disponíveis** que garante a atomicidade através da técnica "**ler de um/escrever em todos**", onde uma operação de leitura pode ser executada sobre qualquer réplica disponível e uma operação de escrita deve ser executada sobre todas as réplicas disponíveis. Esta técnica fornece operações de leitura com um alto grau de disponibilidade a um custo bastante baixo visto que uma operação de leitura só precisa ter acesso a uma réplica. Por outro lado, há uma restrição importante de disponibilidade para as operações de escrita porque elas não podem ser completadas a partir do momento em que apenas uma réplica falhe.

Por isso, as *réplicas disponíveis* são definidas por um mecanismo confiável de detecção de falhas. Sempre que uma réplica $\boldsymbol{x_k}$ de algum objeto $\boldsymbol{x}$ falha, $\boldsymbol{x_k}$ é removida do conjunto de réplicas disponíveis. A necessidade de um mecanismo confiável de detecção de falhas significa que a técnica não previne inconsistências em casos de falhas de comunicação.

A consistência das réplicas pode ser garantida na presença de falhas de comunicação pela introdução de métodos de quorum. Segundo Gifford, apud [GUE96], a idéia é atribuir votos (valores) para cada réplica de um objeto $\boldsymbol{x}$ e definir quoruns de escrita e quoruns de leitura tais que:

1. Quoruns de leitura e quoruns de escrita intersectem-se
2. Dois quoruns de escrita intersectem-se

Desta forma, qualquer operação de leitura é executada sempre sobre pelo menos uma réplica correta.

A técnica acima é chamada de **votação estática** porque os quoruns de leitura e escrita não mudam durante o ciclo de vida do sistema. Um problema sério da **votação estática** é que pode-se tornar impossível obter os quoruns. Para resolver este problema, foi introduzida a votação dinâmica (Davcec e Burkhard, apud [GUE96]) onde, após cada falha, o sistema é reconfigurado e novos quoruns são definidos.

No entanto, a reconfiguração do sistema não resolve por completo o problema porque pode tornar-se muito cara. Por isso, Agrawal e Abbadi [AGR90] propõem uma solução que utiliza uma estrutura lógica em forma de árvore chamada de Árvore de Quoruns. A idéia é evitar qualquer tipo de reconfiguração do sistema e manter baixa a quantidade de acessos necessários para garantir a tolerância a falha.

A árvore de quoruns mantém a principal vantagem da técnica *"ler de um/escrever em todos*" original, isto é, na ausência de falhas, uma operação de leitura acessa somente uma réplica. Na presença de falhas, pode ser necessário acessar mais de uma réplica. As operações de escrita toleram falhas e não precisam de nenhum tipo de reconfiguração. A estrutura em árvore é utilizada para determinar quais réplicas devem ser lidas ou escritas.

A figura 6 apresenta uma árvore ternária com treze cópias de um objeto. A representação é apenas lógica e não tem nenhuma relação com a estrutura de rede utilizada para conectar as réplicas.

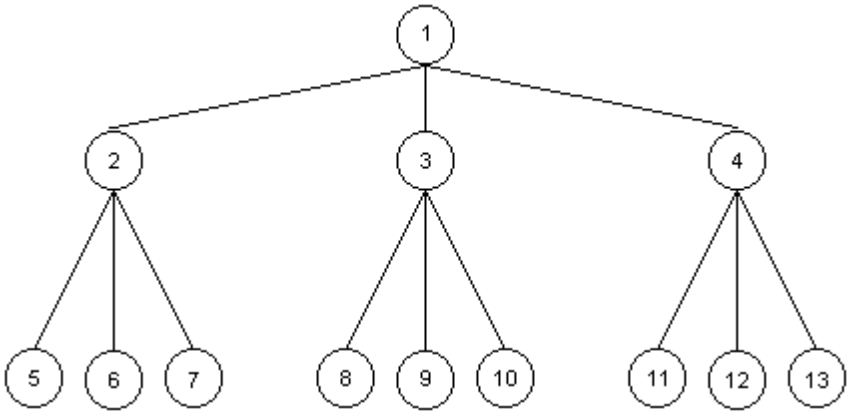

Figura 6 - Árvore de quoruns [AGR90]

Na técnica da árvore de quoruns, uma operação de escrita deve escrever na maioria das cópias de todos os níveis, por exemplo: a raiz, quaisquer duas réplicas do conjunto {2,3,4} e a maioria das réplicas do conjunto {5,6,7,8,9,10,11,12,13}. Neste caso, uma operação de leitura pode ser realizada pelo acesso à maioria das réplicas de qualquer nível. Por exemplo, qualquer um dos seguintes conjuntos pode formar um quorum de leitura: a raiz, quaisquer duas réplicas do conjunto {2,3,4} ou a maioria das réplicas do conjunto {5,6,7,8,9,10,11,12,13}.

A árvore de quoruns tem um desempenho similar à técnica *"ler de um/escrever em todos*" para as operação de leitura e tem uma tolerância a falhas melhor para as operações de escrita. Particularmente, quando a raiz é acessível, uma operação de leitura só precisa realizar um acesso. Além disso, operações de escrita podem ser executadas mesmo depois da falha de várias réplicas em níveis diferentes, por exemplo: a falha das cópias 4,7,8,11 e 12 não impede as operações de escrita.

Embora o mecanismo descrito acima esteja correto, o desempenho da árvore de quoruns pode ser melhorado através de um melhor uso da estrutura da árvore. Em vez de uma operação de escrita ser aplicada na maioria das réplicas de todos os níveis, considere-se uma operação que escreve na raiz, na maioria dos seus filhos, na maioria dos filhos do filhos e assim por diante. Na árvore da figura 6, uma operação de escrita pode então escrever no conjunto de réplicas {1,2,3,5,6,8,9}. Uma operação de leitura deve então tentar acessar a raiz ou a maioria de seus filhos caso ela esteja inacessível. Cada vez que não for possível acessar a maioria das réplicas de um nível, a operação de leitura deve tentar acessar a maioria dos filhos. Por exemplo, considere uma situação onde as réplicas 1, 2 e 3 estão inacessíveis. Neste caso, um quorum de leitura pode ser obtido pelo acesso da réplica 4 e a maioria dos filhos da réplica 2 (5 e 7). Alternativamente, o quorum pode ser formado pela réplica 4 e a maioria dos filhos da réplica 3 (9 e 10). Ambos os quoruns têm uma intersecção não vazia com o quorum de escrita {1,2,3,5,6,8,9}.

Portanto, uma operação de leitura tenta acessar a raiz da árvore; se conseguir, a intersecção com as operações de escrita está garantida. Se a raiz não está acessível, a operação de leitura tenta acessar a maioria de seus filhos. Novamente, se conseguir, e visto que as operações de escrita escrevem na maioria dos filhos da raiz, a intersecção está garantida. Finalmente, se maioria dos filhos da raiz não estão acessíveis a operação de leitura tenta acessar os filhos dos filhos e assim por diante. Uma vez que as operações de escrita devem escrever na maioria dos filhos da raiz, na maioria dos filhos dos filhos da raiz e assim por diante, uma operação de leitura deve ter pelo menos uma réplica em comum com cada operação de escrita.

Assim, a árvore de quoruns é uma variação da técnica de votação por quoruns, é tolerante a falhas e é eficiente para operações de leitura e escrita. Uma descrição completa do protocolo da árvore de quoruns, bem como um estudo de seu custo em relação à técnica de "ler de um/escrever em todos" e a técnica da votação, pode ser encontrada em [AGR90].

3.2.2 Replicação Ativa

Nesta abordagem, também conhecida como máquina de estados [SCH93a], não existe um nodo centralizador e todas as réplicas têm o mesmo papel. Considere um objeto **x**, e a invocação[2] [**x op(arg) p_i**] feita por *pi* (figura 7):

1. A invocação op(arg) é enviada para todas as réplicas de **x**.
2. Cada réplica processa as invocações, altera seu estado e envia uma resposta para o cliente $\mathbf{p_i}$.
3. O cliente aguarda até receber a primeira resposta ou até receber a maioria das respostas idênticas. Se as réplicas não tem comportamento malicioso então o cliente aguarda somente pela primeira resposta, caso contrário, são necessárias **2f+1** réplicas para tolerar **f** réplicas com falhas (Schneider, apud [GUE96]). Assim, o cliente deve aguardar por **f+1** respostas idênticas.

A técnica da replicação ativa requer que as invocações dos processos clientes sejam recebidas pelas réplicas corretas, ou seja, réplicas sem falhas, na mesma ordem. Para isso, é necessário uma primitiva de comunicação que garanta **ordenação** e **atomicidade**. Esta primitiva é chamada de *multicast* de ordenação total e é detalhada em [GUE96].

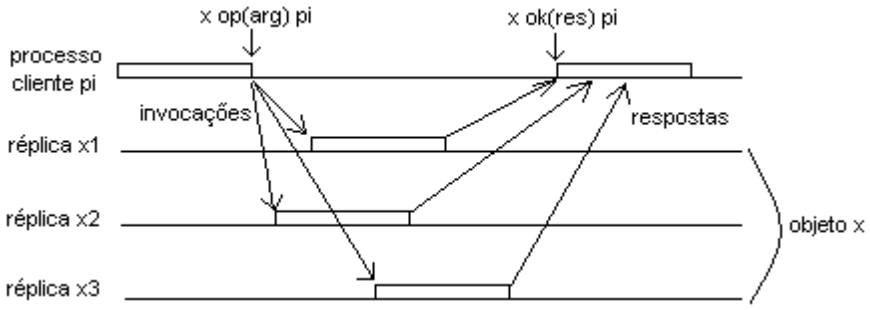

Figura 7 - Técnica da replicação ativa [GUE96]

[2] Uma invocação [*x op(arg) pi*] é uma operação do processo *pi* sobre o objeto ***x***, onde ***arg*** são os argumentos e ***op*** a operação. A resposta à invocação é denotada por [*x ok(res) pi*], onde ***res*** é o resultado retornado.

3.2.3 Primário-*Backup*

Esta abordagem, também conhecida com cópia primária [BUD93], ou replicação passiva, possui várias especificações onde podem ser observadas algumas diferenças. Esta seção apresentará algumas especificações, a fim de ilustrar as diferenças entre elas e permitir uma comparação entre as mesmas. É importante ressaltar que não existe uma especificação mais correta que a outra, mas sim, soluções diferentes que procuram pelo mesmo objetivo em ambientes diversos.

De acordo com Budhiraja [BUD93], um serviço que é implementado utilizando-se esta abordagem de tolerância a falhas consiste de um conjunto de servidores, onde não mais que um deles é denominado de *primário* a cada vez. Um cliente envia uma requisição para o servidor que ele acredita ser o primário. Um cliente é informado pelo protocolo quando o primário muda e assim redireciona suas próximas mensagens. Assume-se que as requisições recebidas por um servidor *s* são organizadas em uma *fila de mensagens de s*.

Desta forma, a figura 8 mostra a execução de um protocolo de cópia primária simples. Este protocolo tolera falha de um único servidor. Assume-se que a comunicação é ponto-a-ponto e não há falhas de *link*. Além disso, cada *link* tem um limite máximo δ para o atraso de mensagens. Há um servidor primário $\mathbf{p_1}$ e um servidor *backup* $\mathbf{p_2}$ conectados por um *link* de comunicação. Um cliente **c** inicialmente envia uma requisição $\mathbf{m_1}$ para $\mathbf{p_1}$. Quando $\mathbf{p_1}$ recebe a requisição $\mathbf{m_1}$, ele:

- processa a requisição e altera seu estado de acordo com a necessidade;
- envia informações $\mathbf{m_2}$ sobre as alterações para $\mathbf{p_2}$. Esta mensagem é denominada **mensagem de atualização de estado**;
- sem aguardar por uma confirmação de $\mathbf{p_2}$, envia a resposta da requisição para o cliente.

A ordem em que as mensagens 2 e 3 são enviadas é importante porque garante que, dada a hipótese feita sobre as falhas, se o cliente recebe uma resposta, então $\mathbf{p_2}$ também receberá a mensagem de atualização de estado em tempo finito ou $\mathbf{p_2}$ estará falho.

Quando o servidor $\mathbf{p_2}$ receber a mensagem $\mathbf{m_2}$, ele atualizará seu estado. Além disso, $\mathbf{p_1}$ envia mensagens de aviso para $\mathbf{p_2}$ a cada τ segundos. Se $\mathbf{p_2}$ não receber tal mensagem por $\tau + \delta$ segundos, ele torna-se o primário. A seguir, $\mathbf{p_2}$ envia a mensagem $\mathbf{m_4}$ para informar aos clientes e começa a processar as próximas requisições.

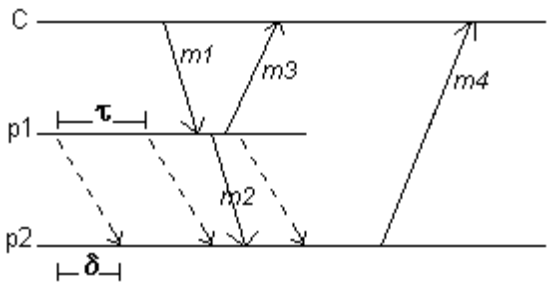

Figura 8 - Técnica do primário-backup [BUD93]

Em outra abordagem, Guerraoui [GUE96] oferece uma solução um pouco diferente para o protocolo do primário-*backup*. Segundo esta abordagem, uma das réplicas, chamada de *primário*, recebe as invocações dos processos clientes, e envia a resposta de volta (figura 9). Dado um objeto ***x***, sua réplica primária é denotada por **prim(x)**. As outras réplicas são chamadas de **backups**. Os backups interagem com o primário e não interagem diretamente com os processos clientes.

Seja a invocação [**x op(arg) p_i**] feita pelo processo p_i. Na ausência de falhas no primário, a invocação é gerenciada da seguinte forma:

- processo p_i envia a invocação **op(arg)** para a réplica **prim(x)**;
- primário **prim(x)** recebe a invocação e executa a tarefa solicitada por ela. No final da operação, a resposta **res** está disponível e o estado do **prim(x)** está alterado. Neste ponto, **prim(x)** envia a mensagem de alteração **(invId, res, state-update)** para os *backups*, onde **invId** identifica a invocação, **res** é a resposta e **state-update** descreve a alteração de estado do primário, resultante da invocação **invId**. Quando os backups recebem as mensagens de alteração eles atualizam seus estados e enviam uma confirmação para o primário (a necessidade dos parâmetros **invId** e **res** é mostrada mais adiante);
- quando o primário recebe a confirmação de todos os backups corretos, isto é, não falhos, ele envia a resposta para p_i;

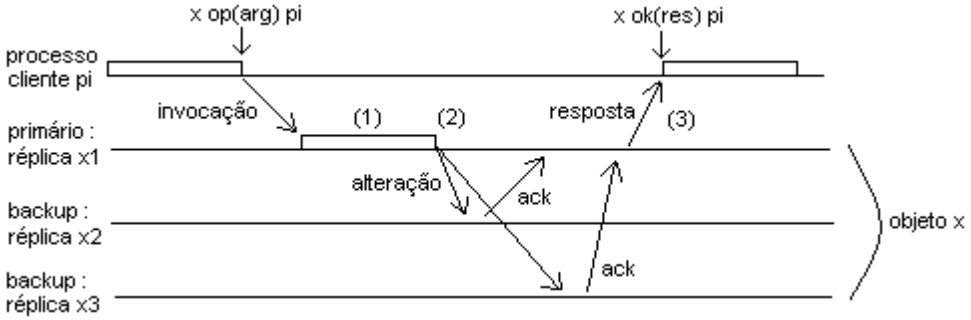

Figura 9 - Técnica do primário-backup [GUE96]

Enquanto o primário não falha, o esquema da figura acima garante a consistência das réplicas, porque a ordem em que o primário recebe as invocações define a ordenação total sobre todas as invocações para os objetos. No entanto, quando o primário falha, a garantia da consistência torna-se mais difícil. Quando isto ocorre, três possibilidades devem ser distinguidas:

1. O primário falha antes de enviar a mensagem de alteração para os backups ((1) na FIGURA 9).
2. O primário falha depois de enviar as mensagens de alteração, mas antes do cliente receber a resposta da invocação ((2) na FIGURA 9).
3. O primário falha depois do cliente ter recebido a resposta da invocação ((3) na FIGURA 9).

Em todas as possibilidades descritas acima um novo primário deve ser selecionado. Nos casos 1 e 2, o cliente não receberá nenhuma resposta para sua invocação e suspeitará de uma falha. Depois de identificar um novo primário ele irá reenviar sua invocação. No caso 1, a invocação é considerada como uma nova invocação pelo novo primário. O caso 2 é o mais difícil de ser gerenciado e a primitiva

de **atomicidade** deve estar presente, isto é, a mensagem de alteração enviada deve ter sido recebida por todos os *backups* ou por nenhum. Quando nenhum *backup* recebe a mensagem o caso 2 torna-se similar ao caso 1. Quando todos os *backups* recebem a mensagem seus estados são alterados pela operação do processo cliente $\mathbf{p_i}$, mas o cliente não recebe uma resposta e irá reenviar a invocação. A informação (**invId, res**) é necessária neste caso para evitar que a mesma invocação seja processada duas vezes. Quando o novo primário recebe a invocação **invId**, antes de processá-la, ele envia a resposta **res** para o cliente.

Basicamente, as abordagens descritas em Budhiraja [BUD93] e Guerraoui [GUE96] diferem quanto ao mecanismo de troca de mensagens entre o primário e os *backups*. Em Budhiraja [BUD93], o primário não espera por mensagens de confirmação dos *backups*, ao passo que, em Guerraoui [GUE96] o primário espera que os *backups* confirmem o recebimento das mensagens. No entanto, dadas algumas hipóteses, há casos em que Guerraoui [GUE96] admite que não é necessário o primário esperar pelas mensagens de confirmação. Assim, nesses casos, as abordagens tornam-se bastante parecidas.

A técnica do primário-*backup* é relativamente fácil de ser implementada na presença de um bom mecanismo de detecção de falhas. A implementação torna-se mais complicada no caso de um sistema assíncrono, onde o mecanismo de detecção de falhas pode não ser confiável. Em Guerraoui [GUE96], é comentada a implementação desta técnica usando por base primitivas de comunicação em grupo, com suporte de visão síncrona, definindo uma semântica que garante a correção do primário-*backup* em casos de mecanismos de detecção de falhas não confiáveis. Uma das grandes vantagens da técnica do primário-*backup* é permitir o uso de operações não determinísticas, garantindo entretanto o determinismo da replicação.

Em Birman [BIR96], encontra-se uma abordagem mais genérica do primário-backup que pode ser aplicada de forma a estruturar sistemas diversos. Neste sentido, em um sistema com processamento replicado o primário e os backups poderiam obter dados em momentos diversos ou de fontes diversas ocasionando indeterminismos. Uma primeira solução a este problema é assumir que a aplicação é completamente determinística, isto é, o comportamento de um servidor pode ser totalmente determinado pela ordem das entradas que ele recebe e por isso é reproduzível através da reexecução das mesmas entradas na mesma ordem em uma segunda cópia. Porém, o autor afirma que uma situação totalmente determinística é pouco provável na realidade porque existem muitas origens para o não-determinismo, tais como interrupção de entregas de mensagens, memória compartilhada e outras. Por outro lado, a utilização de sistemas não-determinísticos é potencialmente difícil de implementar porque as mensagens precisariam ser transmitidas de forma assíncrona levando a algoritmos não-triviais e com isso complicando muito a implementação.

Desta forma, a técnica *Coordinator-cohort* (coordenador-participante[3]) [BIR96] é uma generalização da técnica do primário-*backup* que pode ajudar a superar as limitações descritas acima. Neste método, o trabalho de gerenciar as requisições é compartilhado pelos membros do grupo, e a condição de coordenador-participante é variável ao longo do tempo. O gerente para uma dada requisição é chamado de coordenador para aquela requisição e é responsável por enviar quaisquer informações de alteração ou de rastreamento para outros membros, que são chamados

[3] Os termos coordenador e participante serão usados como tradução para *coordinator/cohorts.*

de participantes para aquela requisição. Como na técnica do primário-*backup*, se o coordenador falha, um dos participantes toma seu lugar. Existem vários coordenadores no mesmo grupo para diferentes requisições. Além disso, a informação de rastreamento para o primário-*backup* normalmente contém as informações necessárias para o *backup* duplicar as ações do primário, enquanto que os dados de rastreamento do coordenador-participante irá muitas vezes consistir de um *log* das alterações feitas pelo coordenador. Neste enfoque, os participantes não replicam ativamente as ações do coordenador, mas meramente atualizam seus estados para refletir as ações do mesmo. Deve-se executar um controle de concorrência através de bloqueios (*locks*). Além disso, o coordenador enviará informações a respeitos das requisições atendidas para que os participantes possam eliminar a requisições pendentes.

A FIGURA 10 ilustra este enfoque. O trabalho de gerenciar as requisições (Req 0 e Req 1) é dividido entre os processos do grupo. Cada coordenador (Coord 0 e Coord 1) replica a requisição e também informa aos membros do grupo que terminou. Neste esquema cada membro do grupo gerencia ativamente algumas requisições enquanto atua passivamente como backup para requisições de outros membros. Este enfoque é mais apropriado para aplicações determinísticas mas pode ser adaptado para aplicações não-determinísticas.

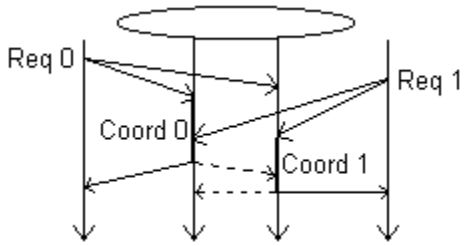

FIGURA 10 - Técnica do Coordinator-Cohort [BIR96]

Basicamente, as linhas gerais do algoritmo apresentado por Birman [BIR96] são as mesmas dos outros autores. Porém, Birman [BIR96] pressupõe um ambiente com características diferentes, "mais realístico", enquanto os outros autores consideram condições ideais de funcionamento. Assim, um ponto importante e que deve ser ressaltado é que a aplicação do algoritmo não é restrita a dados. Além disso, a idéia de processamento paralelo está presente no tratamento das requisições.

Em relação as condições de funcionamento pressupostas por Budhiraja [BUD93], Guerraoui [GUE96] e Birman [BIR96], é importante perceber que as variações da técnica do primário-*backup* dependem principalmente do contexto do sistema onde ela é definida. Há dois pontos fundamentais que interferem no funcionamento da técnica: o tipo de sistema (síncrono ou assíncrono) e o determinismo das operações envolvidas.

O tipo de sistema é importante porque dele depende o comportamento de enviar/receber mensagens das réplicas. Em um sistema síncrono, tal como em Budhiraja [BUD93], o sucesso de uma mensagem enviada pode ser medido em função do recebimento ou não de uma resposta em um intervalo de tempo finito (*timeout*). Esta facilidade também é citada por Guerraoui [GUE96] onde um mecanismo de detecção de falhas é sugerido. Por outro lado, em um sistema assíncrono, encontra-se uma

dificuldade crítica em determinar o sucesso de uma mensagem enviada. Como não há limite de tempo para aguardar uma resposta, é difícil saber se uma mensagem está atrasada, se não chegou ao destino ou se o destinatário falhou.

Para lidar com esta situação, Guerraoui [GUE96] optou por trabalhar com uma semântica de visão síncrona, que utiliza trocas de visões e um detector de falhas.

O determinismo das operações tem implicação direta na aplicabilidade da técnica. Em sistemas onde pode-se garantir o determinismo, a técnica pode ser aplicada como sugere Birman [BIR96]. No entanto, como esta situação é difícil de ser obtida, ou seja, em sistemas reais o indeterminismo é mais provável, Guerraoui [GUE96] resolve o problema centralizando o processamento no primário. Uma vez que há centralização, o primário deve repassar os resultados para as outras réplicas, isto é, atualizar estados.

Este trabalho optou pelo técnica do primário-*backup* sob o ponto de vista dos autores que consideram condições ideais de funcionamento ([GUE96] e [BUD93]). Dessa forma, a complexidade exigida para a implementação da técnica torna-se razoável. A técnica da replicação ativa não foi escolhida porque exige um mecanismo de ordenação total na entrega das mensagens ao passo que, na técnica do primário-*backup* a ordenação pode ser feita pelo primário através uma estrutura simples de fila. Além disso, o determinismo das operações pode ser garantido pelo centralização feita pelo primário. A técnica de votação não foi escolhida porque os mecanismos necessários para garantir sua eficiência, tal como apresentado na seção 3.2.1, tornam sua implementação complexa.

3.3 A Replicação e a Comunicação de Grupos

Independentemente da técnica de replicação, a comunicação de grupos é uma importante abstração para a simplificação da tarefa de construir sistemas confiáveis e disponíveis. No entanto, a concretização desta abstração pode apresentar problemas que merecem ser destacados.

Em Guerraoui [GUE98], estes problemas foram explorados e os parágrafos a seguir resumem aquela experiência que oferece suporte à replicação de objetos em Smalltalk. Para isso, foi utilizada a ferramenta de comunicação de grupos Isis. A mistura entre a semântica de grupos de objetos do Smalltalk e a semântica de grupos de processos do Isis tem muitos efeitos e o mapeamento de uma semântica para a outra foi feito pelos autores utilizando como pano de fundo uma aplicação distribuída e replicada de controle de agendas, onde cada membro do grupo contêm uma agenda usada para armazenar a lista dos compromissos dos membros participantes.

O mapeamento direto de um grupo de objetos para um grupo de processos é uma maneira natural de utilizar, diretamente para objetos, os mecanismos de grupo oferecidos para processos pela ferramenta de grupo. Porém, mapeamento direto, onde um grupo de objetos corresponde a um grupo de processos, tem duas importantes conseqüências relatadas na experiência :

- proliferação de grupos: para a aplicação de controle de agendas, foi criada uma réplica de cada agenda em cada máquina da rede local. Em função disso, foi criado um processo Isis para cada réplica. A FIGURA 11 mostra o cenário onde três

objetos diferentes são replicados sobre as mesmas máquinas (máquina1 e máquina2). As réplicas do mesmo objeto são designadas réplica x.1 e réplica x.2. Dois grupos de processos Isis (processos1 e processos2) são criados para os mesmos objetos e cada um desses grupos determina uma sobrecarga de processamento e de tráfego devido à gerência interna da ordenação de mensagens e controle de membros. Na realidade, um único grupo unindo processos1 e processos2 seria o suficiente.

- super-grupos: a primitiva de *multicast* de ordenação total oferecida pelo Isis garante uma forte consistência em uma interação pura entre cliente-servidor. Porém, existem situações em que um cliente precisa interagir com muitos servidores de forma atômica. No caso do controle de agendas, "agendar um compromisso" requer uma interação com várias agendas replicadas pois um compromisso é agendado, se a data estiver livre para todas as pessoas envolvidas. Por isso, agendar um compromisso implica em alterar agendas. Como as agendas são replicadas, é necessário uma operação atômica que envolve vários objetos replicados. A única maneira de garantir um tipo transacional de atomicidade é criar, para cada requisição envolvendo múltiplos servidores, um super-grupo que contém todas as agendas replicadas. A FIGURA 12 mostra o cenário envolvendo dois objetos replicados. A semântica dos super-grupos resolve o problema, mas gera um aumento de carga considerável devido à gerência do super-grupo temporário.

Além do problema de mapeamento entre as semânticas, também há o problema da duplicação de requisições. Quando um cliente invoca um servidor replicado ativamente, cada réplica do servidor recebe a invocação, executa a invocação e retorna um resultado. Não existe uma coordenação pré-definida entre as réplicas e estas comportam-se como se fossem entidades independentes. Assim, a replicação ativa só pode ser oferecida em associação com uma primitiva de *multicast* de ordenação total quando o servidor replicado executa sempre o papel de servidor. Este não é sempre o caso em uma aplicação baseada em objetos porque os objetos podem alternadamente executar o papel de cliente e servidor. Se um objeto replicado executa o papel de cliente para um outro servidor, o cenário acima irá resultar em pedidos duplicados. Como é mostrado na FIGURA 13, a requisição2 é enviada duas vezes para cada réplica do segundo objeto replicado. Isto introduz uma considerável sobrecarga e pode até mesmo causar inconsistências entre operações.

Segundo Guerraroui [GUE98], apesar de mais sutil, este problema também ocorre com a replicação por primário-*backup*, quando ocorrem algumas falhas. O primário do primeiro objeto replicado pode enviar o pedido e então falhar e quando um novo primário é eleito ele pode refazer o pedido. Deve-se adicionar, sobre o Isis, um mecanismo para filtrar cada invocação de uma réplica a fim de eliminar pedidos duplicados. O mecanismo de filtragem requer um protocolo de sincronização distribuído específico entre as réplicas para garantir que exatamente um pedido seja emitido pelo objeto replicado, mesmo no caso de falha.

Assim, um dos principais resultados do estudo acima é a respeito da complexidade envolvida na comunicação de grupos. Mesmo utilizando-se uma ferramenta pronta, o Isis, percebe-se que há necessidade de mecanismos adicionais para corrigir ou melhorar o mesmo. Por outro lado, a construção de uma ferramenta de comunicação completa deve levar em consideração os problemas mostrados e procurar a sua solução. A utilização da abstração de grupo simplifica o trabalho de

desenvolvimento de sistemas confiáveis e disponíveis, mas o desenvolvimento de ferramentas para implementar esta abstração é uma tarefa bastante complexa.

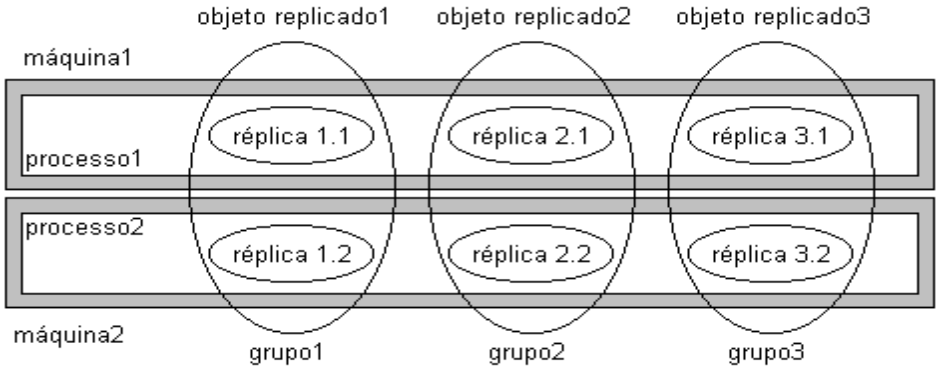

FIGURA 11 - Proliferação de grupos [GUE98]

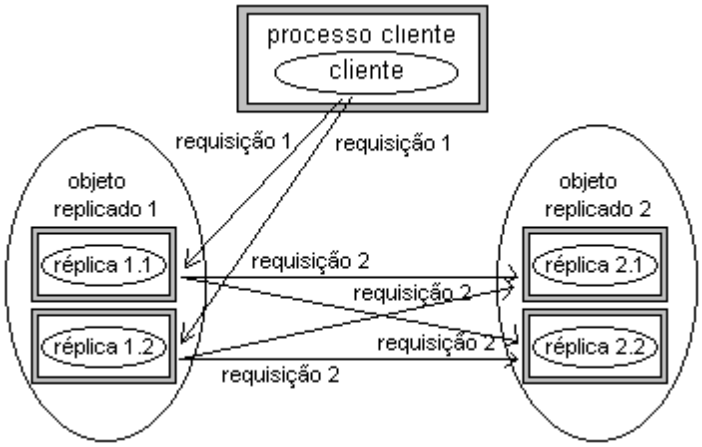

FIGURA 12 - Super grupos [GUE98]

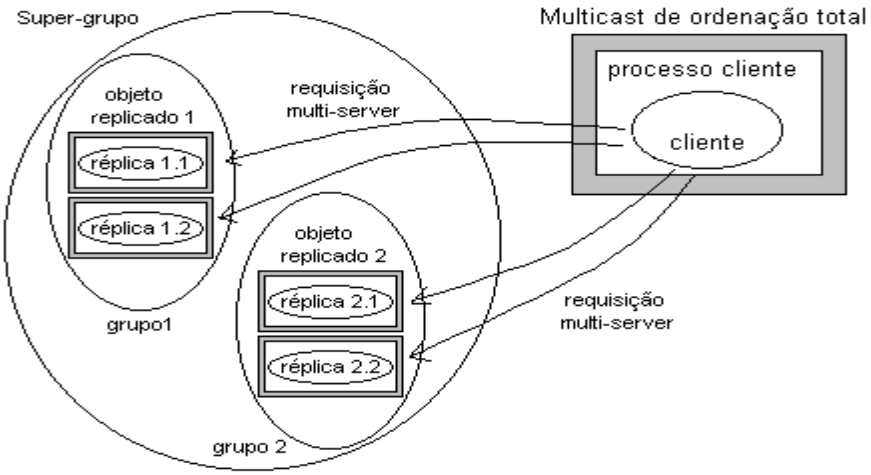

FIGURA 13 - Duplicação de pedidos [GUE98]

4. Aspectos da Linguagem Java

Para ajudar a compreensão de aspectos importantes da linguagem Java no desenvolvimento deste trabalho, este capítulo aborda características gerais da linguagem, características da biblioteca de RMI (*Remote Method Invocation*), detalhes sobre a definição de classes, instanciação de objetos e serialização dos mesmos.

4.1 A Linguagem

Java é uma linguagem de alto nível, com sintaxe similar a C e C++. Ela é simples, orientada a objetos, com tipos de dados estáticos, compilada, independente de arquitetura, *multithreaded*, com coleta de lixo (*garbage collection*), robusta, segura, extensível e bem estruturada.

O Java é parecido com C++, mas muito mais simples. Todos os recursos de linguagens de alto nível que não são absolutamente necessários foram descartados de sua implementação. Ela não tem sobrecarga de operadores, arquivos de cabeçalhos, pré-processadores, aritmética de ponteiros, estruturas, uniões, matrizes multidimensionais, gabaritos (*templates*) ou conversão implícita de tipos [RIT96]. Além disso, não há herança múltipla.

A semelhança com o C++ é proposital e oferece uma aparência com a qual a maioria dos programadores sente-se confortável. A eliminação de elementos como ponteiros, e outros já citados, permite que os programadores empreguem menos tempo preocupando-se com erros de programação e mais tempo desenvolvendo funcionalidades. Alia-se a isso um poderoso conjunto de bibliotecas de classes, que oferecem praticamente toda a funcionalidade básica necessária para se desenvolver um aplicativo de forma rápida e eficiente [RIT96].

A característica mais importante da Java é a **orientação a objetos**. Ela é uma linguagem verdadeiramente orientada a objetos e foi projetada para isso desde o início. Seus projetistas decidiram romper com qualquer linguagem existente e criaram uma, desde o princípio. Seu código é organizado em classes. Cada classe define um conjunto de métodos que formam o comportamento de um objeto. Uma classe pode herdar comportamentos de outra classe. Na raiz da hierarquia de classes está sempre a classe *Object*. Depois que uma classe é criada, o ambiente de execução (*run-time)* de Java permite a carga dinâmica de classes. Isto significa que aplicativos existentes podem adicionar funcionalidades ao adicionarem novas classes que encapsulam os métodos necessários [RIT96]. Ela dá suporte a uma hierarquia de **classes** de herança simples. Isto quer dizer que cada classe só pode herdar de uma outra classe por vez. Porém, nem tudo em Java é um objeto. Num esforço para torná-la mais simples e eficiente, tipos simples como números e variáveis booleanas não são classes. Entretanto, o Java fornece objetos empacotadores (*wrappers*) para todos os tipos simples, de forma que esses tipos podem também ser implementados como objetos [HOF96].

As **interfaces** Java são muito similares às interfaces IDL, o que significa que ela pode ser usada no sistema de objetos CORBA para construção de sistema de objetos distribuídos. Esta compatibilidade é importante, pois tanto interfaces IDL quanto o sistema CORBA são usados em muitos sistemas computacionais.

Quando um programa Java é executado**,** ele é primeiro compilado gerando código binário, ou os chamados *bytecodes*. Este código é bastante similar a instruções de máquina, de forma que programas em Java podem ser bastante eficientes. Entretanto, o *bytecode* não é específico para um modelo de computador em particular, de forma que programas em Java podem ser executados em vários tipos de computadores distintos sem que seja necessário recompilá-los [HOF96].

O *bytecode* é na realidade uma solução do Java ao problema da portabilidade. Duas características fundamentais do mecanismo de portabilidade são [RIT96]:

- para executar um programa Java, o computador deve possuir um programa para converter o código Java em código nativo da máquina;
- a linguagem Java não permite que uma máquina em particular implemente tamanhos diferentes do padrão para tipos fundamentais como inteiros e *bytes*.

Ao ser executado em um ambiente interpretado, o código Java não precisa obedecer a nenhuma plataforma de *hardware* em particular. O compilador Java, que cria os programas executáveis a partir do código fonte, compila para uma máquina cuja contrapartida física não existe - a Máquina Virtual Java. Esta máquina é uma especificação de um processador hipotético que pode executar o código Java.

Além de especificar um código de máquina virtual para garantir a portabilidade, a linguagem Java também assegura-se de que os dados ocupem o mesmo espaço em todas as implementações.

Portanto, pode-se dizer que a **arquitetura independente** de Java deve-se ao fato de que a semântica de execução é a mesma em qualquer computador.

O suporte à **multitarefa** é um ponto importante para o melhor aproveitamento dos recursos dos sistemas operacionais modernos, visto que a maior parte deles , como o Unix, Windows NT e o Windows 2000 dão suporte à multitarefa. Um programa Java pode ter mais de um fluxo de execução concorrente (*threads*). Por exemplo, ele poderia executar algum cálculo extenso em uma linha de execução, enquanto outras linhas de execução interagiriam com o usuário. Isto evita que os usuários parem de trabalhar para esperar que os programas completem operações extensas. Java fornece recursos de **sincronização** que facilitam este estilo de programação (*multithreaded*) [HOF96].

Em linguagens de programação tradicionais, como C e C++, quando uma porção de memória é alocada pelo programa também deve ser liberada por ele. Quando isto não acontece, podem ocorrer erros de programa (perda de endereçamento, extravio de ponteiros, estouro de pilha devido à falta de liberação) e problemas de falta de memória. Em projetos grandes, pode ser bastante difícil manter as ações de alocação ou liberação de memória funcionando sempre corretamente [HOF96]. Em Java, um endereço de memória alocado não pode ser liberado pelo programa e não se pode utilizar ponteiros em programas para acessar a memória. Além disso, para cada acesso a matrizes, é realizada uma verificação de limites para impedir que um programa indexe um espaço não alocado na memória [RIT96]. Para endereços de memória alocados, Java tem um programa embutido chamado **coletor de lixo** (*garbage collector)*, que mantém o registro de todos os objetos e referências a esses objetos em um programa Java. Quando um objeto não possui mais referências, o coletor de lixo o remove.

O coletor de lixo faz parte do mecanismo de consistência da Java que verifica cuidadosamente cada acesso à memória e garante que ele é permitido e não causará nenhum problema.

Estes mecanismos devem-se à necessidade de segurança exigida pelos sistemas de computação distribuída. Para isso, quando o código Java entra inicialmente no interpretador, antes mesmo de ter uma oportunidade de ser executado, ele é verificado quanto à sua adequação à linguagem. Mesmo supondo que o compilador gera somente códigos corretos, o interpretador efetua nova verificação, porque o código pode ter sido intencional ou não intencionalmente alterado entre o momento da compilação e o da execução. A seguir, é determinado um esquema de memória para as classes, onde cada classe é carregada em sua própria área de memória. Além disso, as classes carregadas não acessam o sistema de arquivos, exceto da forma específica em que são permitidos pelo cliente ou usuário [RIT96].

Uma característica importante de Java é que ela foi projetada visando uso em computadores de pequeno porte; por isso, o sistema Java é relativamente pequeno no contexto de uma linguagem de programação. Quanto à velocidade, seu desempenho é muito melhor que linguagens de roteiros (*scripts*) típicas, no entanto, ela é cerca de 20 vezes mais lenta que C. Isto é aceitável para a maioria da aplicações porque, em geral, características como portabilidade, segurança e robustez tem seu custo cobrado em desempenho [HOF96].

4.2 Java RMI

O RMI (*Remote Method Invocation*) é uma maneira de objetos clientes em um computador executarem métodos em um servidor que pode ser o mesmo computador ou um computador remoto em uma rede. Para Montresor [MON99], Java RMI é um modelo de distribuição de objetos que mantém a mesma semântica do modelo de objetos Java, tornando a distribuição de objetos fácil de implementar e usar. Objetos remotos são caracterizados pelo fato de seus métodos poderem ser invocados a partir de outras máquinas virtuais Java, estejam elas no mesmo computador hospedeiro ou não. Dada uma classe qualquer de objetos remotos, o conjunto de seus métodos que podem ser invocados remotamente é definido por uma ou mais *interfaces remotas*.

A FIGURA 14 ilustra a arquitetura Java RMI. Esta arquitetura é composta de três camadas: a camada *stub*/*skeleton*, a camada de referência remota e a camada de transporte.

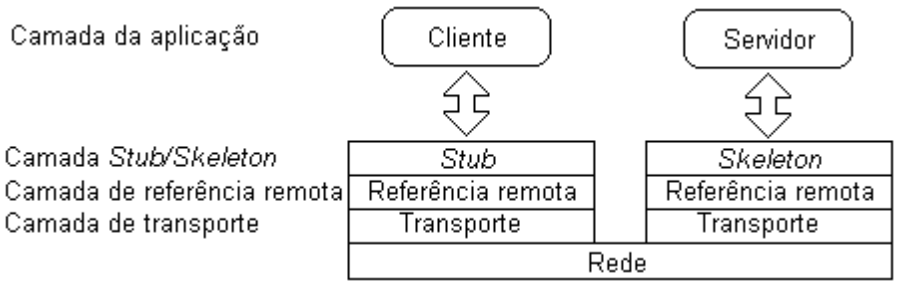

FIGURA 14 - Arquitetura Java RMI [MON99]

Em geral, pode-se dividir a construção de uma aplicação RMI nas seguintes etapas: criação de uma interface remota, definição da classe de implementação da interface remota e definição de uma aplicação que utilize o serviço remoto [JDK99].

As seções a seguir detalham os aspectos mais importantes das etapas de construção de uma aplicação RMI e as principais características da arquitetura Java RMI (FIGURA 14).

4.2.1 Interfaces de Objetos Remotos

Na linguagem Java, um objeto remoto é uma instância de uma classe que implementa uma interface remota. Interfaces remotas tem as seguintes características [JDK99]:

- devem ser declaradas públicas. Caso contrário, um cliente irá obter um erro quando tentar ler um objeto remoto que implementa uma interface remota, a menos que o cliente esteja no mesmo pacote da interface remota;
- estendem a interface **java.rmi.Remote**;
- cada método deve declarar **java.rmi.RemoteException** em sua cláusula **throws**, além de qualquer outra especificação de tratamento de exceções;
- tipos de dados de qualquer objeto remoto que são passados como um argumento ou retornam um valor, devem ser declarados como tipo interface remota (*remote interface*).

4.2.2 Classes de Implementação da Interface Remota (Servidor)

Depois de definida a interface do objeto remoto, um servidor de implementação deve ser escrito. Tipicamente, a implementação da interface do objeto remoto também estende a classe **java.rmi.server.UnicastRemoteObject**.

A classe **UnicastRemoteObject** é uma extensão da classe **RemoteServer**, que atua como uma classe base para a implementação de servidores de objetos no RMI. Subclasses da classe **RemoteServer** podem implementar diferentes tipos de esquemas de distribuição de objetos, tais como: objetos replicados, objetos ***multicast*** ou comunicação ponto-a-ponto. Algumas versões do RMI podem apresentar variações quanto à funcionalidade dos objetos remotos, por exemplo: a versão RMI (1.1) suporta apenas objetos remotos que utilizam comunicação ponto-a-ponto. Nesta versão somente a classe **UnicastRemoteObject** está disponível [FAR98].

4.2.3 Classes para Utilizar o Serviço Remoto (Cliente)

Os clientes são objetos que invocam um método remoto. O cliente precisa de um *stub* para realizar a invocação. Um *stub* de um objeto é uma visão remota do objeto que contém somente os métodos remotos do mesmo. O *stub* roda no lado do cliente e representa o objeto remoto no espaço de dados do cliente. O cliente invoca métodos do *stub* e ele então invoca os métodos no objeto remoto. Isto permite a qualquer cliente invocar métodos remotamente por meio da invocação normal de métodos de Java. Um *stub* é também chamado de ***proxy***. A FIGURA 15 ilustra a relação entre um cliente, um servidor e um *stub***.**

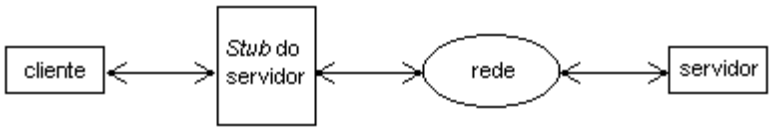

FIGURA 15 - Relação entre cliente, servidor e *stub* do servidor [WUT97]

O servidor também pode invocar métodos do cliente. Neste caso, o cliente passa como parâmetro para o servidor um *stub* dele mesmo. O *stub* que é passado invoca métodos de volta ao objeto original [WUT97]. A FIGURA 16 ilustra como um cliente passa um ***stub*** para o servidor de modo que o servidor possa invocar métodos do cliente.

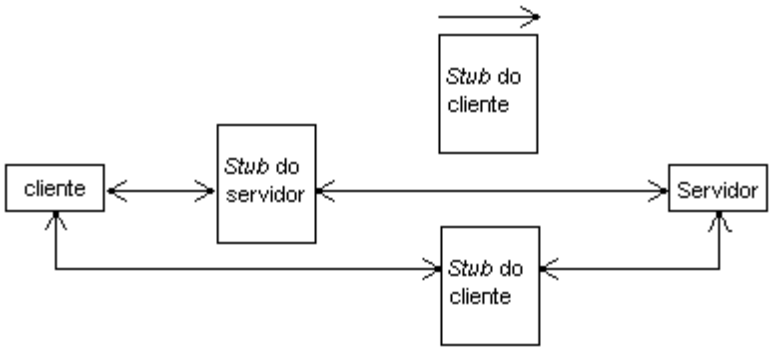

FIGURA 16 - S*tub* do cliente para o servidor [WUT97]

Esta comunicação entre cliente/servidor envolve a capacidade dos clientes de encontrarem os servidores que precisam. Para isso RMI fornece um objeto de busca que permite ao cliente obter um *stub* para um determinado servidor baseando-se no nome do servidor. A FIGURA 17 ilustra como um cliente usa o serviço de nomeação para encontrar um servidor [WUT97].

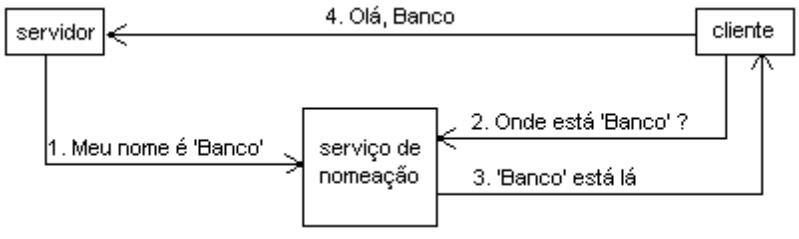

FIGURA 17 - Serviço de nomeação (registro) [WUT97]

O serviço de nomeação, também chamado de ***registry***, é um repositório que pode ser usado para obter *stubs* de objetos remotos simplesmente através de seus nomes. Cada serviço de nomeação mantém um conjunto de registro na forma <nome, objeto remoto>. Novos registros podem ser adicionados utilizando o método `bind`() ou `rebind`(), enquanto o método `lookup`() é usado para obter o *stub* para um objeto remoto a partir de um certo nome.

4.3 Considerações Sobre Objetos Replicados

Para efeitos de replicação de objetos é preciso entender algumas características do Java em relação a definição de classes, instanciação de objetos e declaração de variáveis. Assim, os parágrafos a seguir procuram ilustrar, através de um exemplo, o conceito de **estado de objeto**.

Um objeto x, que representa um conjunto de réplicas $[x_1,x_2,...,x_n]$, deve apresentar um comportamento funcional, sem efeitos colaterais. Por comportamento funcional entende-se um objeto capaz de produzir alteração de estado somente em si mesmo, sem solicitar serviços a outros objetos que possam produzir outras alterações de estado. Esta restrição facilita a tarefa de consistência de estado entre as réplicas, por considerar que somente o objeto alvo da solicitação de serviço seja passível de reconstituição, sem que seja necessária a recuperação de estado de todos os objetos envolvidos em uma seqüência de ativações.

Programas orientados a objetos são estruturados a partir de classes e objetos (instâncias de classe), que encapsulam propriedades: estruturas de dados e métodos. As propriedades de uma classe são acessíveis a todas as suas instâncias (semântica global) enquanto que as estruturas de dados de um objeto são particulares de cada instância (semântica local).

No contexto deste trabalho, todas as réplicas são instâncias de uma mesma classe, ou seja, possuem idênticas propriedades e se distinguem apenas por seu estado. Por estado de um objeto entende-se o valor de suas variáveis de instância, em um determinado momento. Portanto, ao providenciar uma réplica de um componente em um determinado instante, o programador deve estar capacitado a fornecer uma cópia da parte estática (estrutura de dados e código) juntamente com uma cópia da parte dinâmica (variáveis de estado). A obtenção de cópias de objetos é feita num determinado momento da execução e, portanto, cada cópia de objeto contém o estado da computação que pode ser usado pelos algoritmos de controle de consistência de estado de componentes replicados.

Há ainda que se considerar que, na linguagem Java, as variáveis de estado podem estar contidas na classe, sob a denominação de variáveis de classe, e, naturalmente, em cada objeto que representa os dados particulares da instância. Uma variável de classe é global a toda classe, isto é, há apenas uma cópia desta variável para todos os objetos da classe. Isso implica na possibilidade de alteração de estado das variáveis de classe por qualquer instância da mesma.

Os exemplos a seguir, baseados em Newman [NEW97], mostram detalhes da referência a variáveis e métodos em Java. A classe `NumerosSecretos` a seguir armazena três números de um código ou combinação secreta.

```
class Numeros
{
    int primeiro = 0;
    int segundo = 0;
    int terceiro = 0;
}
```

Um segmento de programa pode criar novos números para quem precisa de uma combinação. Por exemplo:

```
Numeros NumerosDoJoao   = new Numeros();
NumerosdoJoao.primeiro  = 1;
NumerosdoJoao.segundo   = 2;
NumerosdoJoao.terceiro  = 3;

Numeros NumerosDaMaria  = new Numeros();
NumerosDaMaria.primeiro = 4;
NumerosDaMaria.segundo  = 5;
NumerosDaMaria.terceiro = 6;
```

Os dois conjuntos de números são distintos. Com os dois comandos `new`, foram criados dois novos objetos em Java. Ao imprimir o conteúdo de `NumerosdoJoao` e `NumerosdaMaria`, será gerada a saída:

```
NumerosdoJoao.primeiro  = 1;
NumerosdoJoao.segundo   = 2;
NumerosdoJoao.terceiro  = 3;
NumerosDaMaria.primeiro = 4;
NumerosDaMaria.segundo  = 5;
NumerosDaMaria.terceiro = 6;
```

As variáveis do exemplo acima são chamadas de variáveis de instância, porque cada objeto da classe possui seu próprio conjunto de valores.

Por outro lado, Java aloca espaço para variáveis de classe apenas uma vez. A palavra-chave `static` em uma declaração indica uma variável de classe. Assim, a modificação mostrada abaixo transforma cada uma das variáveis em variáveis de classe.

```
class Numeros
{
    static primeiro = 0;
    static segundo  = 0;
    static terceiro = 0;
}
```

Como antes, foram atribuídos os números:

```
Numeros NumerosDoJoao   = new Numeros();
NumerosdoJoao.primeiro  = 1;
NumerosdoJoao.segundo   = 2;
NumerosdoJoao.terceiro  = 3;
```

```
Numeros NumerosDaMaria  = new Numeros();
NumerosDaMaria.primeiro = 4;
NumerosDaMaria.segundo  = 5;
NumerosDaMaria.terceiro = 6;
```

No entanto, agora tem-se uma saída diferente:

```
NumerosdoJoao.primeiro  = 4;
NumerosdoJoao.segundo   = 5;
NumerosdoJoao.terceiro  = 6;
NumerosDaMaria.primeiro = 4;
NumerosDaMaria.segundo  = 5;
NumerosDaMaria.terceiro = 6;
```

A referência a métodos depende do tipo de acesso atribuído a eles. Uma descrição completa das vários tipos de acesso e suas implicações pode ser encontrado em [NEW97]. O exemplo a seguir, modifica a classe Numeros e mostra, de forma simplificada, o acesso aos métodos de uma classe.

```
package A
class Numeros
{
    static primeiro = 0;
    static segundo = 0;
    static terceiro = 0;

    public ImprimirNumeros()
    {
         System.out.println("primeiro =" + primeiro);
         System.out.println("segundo =" + segundo);
         System.out.println("terceiro =" + terceiro);
    }

    private protected ImprimirPrimeiro()
    {
         System.out.println("primeiro =" + primeiro);
    }
}
```

O método `ImprimirNumeros` está acessível a qualquer classe do mesmo pacote porque é do tipo *`public`*.

Por outro lado, o método `ImprimirPrimeiro`, definido como *`private protected`*, só está acessível objetos da própria classe ou subclasse. O exemplo a seguir mostra a classe `ObterNumeros` fazendo referência aos métodos da classe `Numeros`.

```
package A
import A.Numeros;
public class ObterNumeros
{
    void Imprimir(Numeros num)
    {
        num.ImprimirNumeros();
        num.ImprimirPrimeiro();
    }
}
```

No código acima, o acesso `num.ImprimirNumeros()` produz a saída esperada. Porém, o acesso `num.ImprimirPrimeiro()` não é válido porque o método `ImprimirPrimeiro` é privativo da própria classe `Numeros` ou de classes derivadas dela.

4.4 Serialização de Objetos

A serialização de objetos é utilizada para disponibilizar a réplica do objeto em um processador remoto. O intuito deste trabalho, ao prover classes para a serialização de objetos, é facilitar a tarefa da obtenção de cópias de objetos, que podem ser armazenadas em memória estável ou transmitidas pela rede [AMA99]. O uso das classes propostas fornece uma abstração de programação específica para a obtenção de réplicas e evita que o programador tenha que aprender detalhes dos métodos de serialização/deserialização da extensa biblioteca disponível na máquina virtual Java. A FIGURA 18 esquematiza o processo de serialização e deserialização.

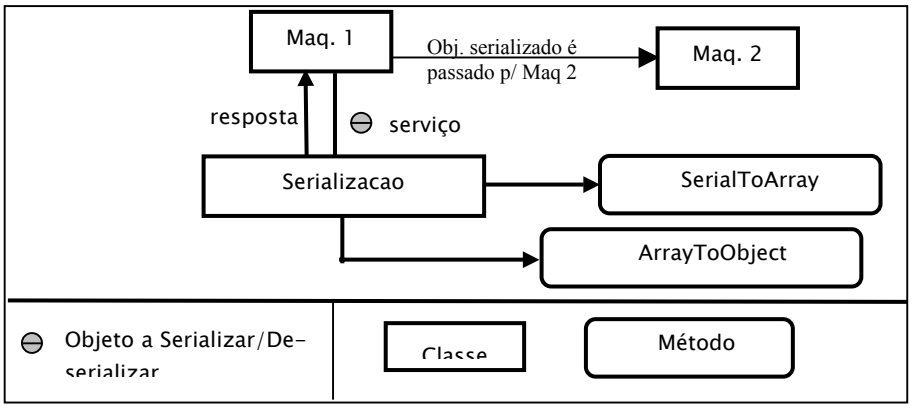

FIGURA 18 – Esquema da Serialização/Deserialização

O método ***SerialToArray*** recebe como argumento um determinado objeto e o serializa no formato de um *array* de *bytes*; o objeto é serializado para este formato para que possa ser enviado posteriormente para outro computador através da rede. O inverso, ou seja, a recuperação de um objeto serializado, é feito através do método ***ArrayToObject***, que recebe como argumento um objeto serializado na forma de *array* de *bytes* e o transforma num objeto *Object*.

4.5 O Java RMI e sua Aplicação neste Trabalho

Para os objetivos pretendidos por este trabalho, Java é uma linguagem bastante apropriada. É preciso ter em mente que este trabalho tem como palavra-chave a **replicação**. Neste sentido, devido à simplicidade, o aprendizado da linguagem Java não necessitará de grande disponibilidade de tempo, permitindo que mais tempo de estudo seja aplicado na implementação da máquina de replicação, que utilizará a invocação de métodos remotos para atualizar o estado das réplicas. Porém, esta simplicidade não significa falta de recursos.

A opção de enquadramento dos servidores e clientes envolvidos como membros de grupos determina a forma básica de funcionamento e os princípios da programação. A implementação da comunicação entre membros de grupos é um ponto fundamental para a replicação e pode ser feita de maneira simples quando se utiliza a linguagem Java.

A gerência de memória conta com recursos como o coletor de lixo (já discutido) e com simplificações estruturais na própria linguagem. Estas características permitem que menos tempo seja gasto com problemas ligados à programação e mais tempo seja alocado para o problema em si.

A utilização da linguagem Java leva ao uso de um paradigma que não é novo, porém é atual: a orientação a objetos. Este paradigma assegura o uso do conceito de modularidade, que, aliada ao encapsulamento, permite que um objeto seja considerado a unidade atômica de replicação e seu estado – representado pelos valores de suas variáveis de estado - seja facilmente preservado e restaurado.

Finalmente, a utilização da linguagem Java permitiu a integração deste trabalho a outro que já estava em desenvolvimento, tal como um ambiente de reflexão computacional que busca adicionar maior amigabilidade ao uso de componentes desenvolvidos em Java [FIL00].

5 Máquina de Replicação

A idéia básica da máquina de replicação é permitir que qualquer programador possa replicar classes consideradas críticas para a sua aplicação. Para isso, a máquina de replicação deve atuar sobre os objetos das classes críticas e fazer a replicação de forma transparente ao programador.

Assim, o objetivo da máquina de replicação é liberar o programador da aplicação do esforço de implementar, de forma particular, serviços de replicação que podem ser oferecidos de forma independente do domínio da aplicação.

Este capítulo tem como objetivo descrever a máquina de replicação através da especificação das hipóteses do trabalho e o modelo de sistema definido.

5.1 Modelo do Sistema

A máquina de replicação baseia-se em um modelo onde um sistema distribuído consiste de um conjunto de nodos interconectados por uma rede de comunicação [BAB93]. Os nodos suportam processos que se comunicam uns com os outros através da troca de mensagens. A máquina de replicação assume que o sistema é **síncrono**, ou seja, existem limites para o atraso das mensagens e para a execução de todas as atividades.

A característica síncrona do sistema determina com clareza a técnica de replicação a ser utilizada, a técnica do primário-backup, visto que esta técnica é relativamente fácil de ser implementada na presença de um mecanismo confiável de detecção de falhas. A implementação torna-se mais complicada no caso de um sistema assíncrono, onde o mecanismo de detecção de falhas pode não ser confiável [GUE96].

Do ponto de vista dos processos, o modelo admite dois estados: operante e inoperante. Enquanto está operante, um processo segue exatamente as ações especificadas pelo programa que ele executa. O único tipo de falha admitido por este modelo são as falhas de **Parada** e **Colapso**, tal como definidas na seção 3.1. Assim, um processo pode sofrer alguma falha, tornar-se inoperante e não executar mais nenhuma ação. Um processo pode também, após sofrer uma falha e perder seu estado local, recuperar-se através de um protocolo de recuperação.

Do ponto de vista da comunicação, o modelo não admite falhas, ou seja, por hipótese, a comunicação é confiável. Além disso, uma mensagem é recebida em até δ unidades de tempo depois de enviada. O parâmetro δ inclui não só o tempo necessário para transportar a mensagem pela rede, mas também os atrasos decorridos do processamento de envio e recebimento da mesma. O modelo assume que cada processo tem um relógio local, que avança na mesma proporção que os outros, e é utilizado somente para mensurar intervalos de tempo, não precisando estar sincronizado aos demais [BAB93].

A partir do modelo recém descrito e da suposição de que não há falhas de comunicação, a máquina de replicação utiliza *timeouts* (prazos máximos) para detectar processos falhos.

5.2 Modelo de Replicação

Seja **x** um objeto que representa um conjunto de réplicas [$\mathbf{x_1,x_2,...,x_n}$], sendo cada réplica (*backup*) localizada em um processador $\mathbf{p_k}$. O objeto **x**, além de representar um conjunto de réplicas, é responsável pela disseminação de mensagens (chamadas remotas de métodos) e serviços de recuperação de réplicas e seus estados. Por conseguinte, ele deve conter um conjunto de funcionalidades que não existem no objeto original definido pelo usuário, atuando como uma "máquina de replicação" que deve oferecer vários serviços, que correspondem às exigências de implementação da técnica de replicação. Os principais serviços são:

1. **Obtenção e distribuição de cópias**: após a criação do objeto crítico **OC** na aplicação, seu estado deve ser transmitido para o conjunto de réplicas [$\mathbf{x, x_1, x_2,..., x_n}$] localizadas em outros processadores, para que formem a base de um sistema de replicação com vistas à tolerância a falhas. Nesta etapa, cada cópia do objeto **OC** é serializada e transmitida ao processador de destino. Ao chegar ao destino, a cópia serializada é novamente convertida em um objeto executável, apto a fornecer os mesmos serviços que o objeto original [AMA99].

2. **Disseminação de mensagens**: após formar a base do sistema de replicação, para cada invocação com alteração de estado ao objeto crítico **OC**, a aplicação fará um desvio para o método de atualização de estado da máquina de replicação. Ao receber a invocação de atualização de estado desviada, a máquina de replicação transmitirá a mesma invocação a todas as réplicas, a fim de que estas atualizem seus estados. Desta forma, uma solicitação de serviço **R** com alteração de estado ao objeto **OC** causará uma solicitação de atualização de estado **S** ao objeto **x** (máquina de replicação). Ao receber esta solicitação **S** (FIGURA 19), o objeto **x** realiza as seguintes ações:

   a) executa a solicitação e envia uma mensagem de atualização de estado para cada réplica $\mathbf{x_i}$;
   b) objeto **x** aguarda até receber a resposta de todas as réplicas não falhas;
   c) cada réplica $\mathbf{x_i}$ atende a solicitação, indicando se houve atualização de estado ou se ocorreu uma falha.

3. **Consistência**: a primitiva fundamental para manter a consistência é a atomicidade no recebimento das mensagens, visto que a ordenação total é garantida pela centralização feita pelo primário [GUE96]. Por isso, a transmissão de uma invocação será feita através de um protocolo de **commitment atômico** de duas fases. O protocolo commitment atômico usado neste trabalho, utiliza *timeout* e permite que o primário comunique-se com as réplicas, uma a uma, mantendo a consistência mesmo na presença de falhas.

Na interação do objeto $\boldsymbol{x}$ com suas réplicas [$\boldsymbol{x_1,x_2,...,x_n}$] podem ocorrer as seguintes situações:

1. Serviço correto de consulta: todas as réplicas fornecem o serviço solicitado, sem atualização de estado;
2. Serviço correto de escrita: todas as réplicas atendem ao serviço solicitado e atualizam seus estados;
3. Falha de atualização de estado: uma ou mais réplicas responde negativamente a mensagens de atualização de estado;

As hipóteses (1) e (2) identificam um comportamento adequado do sistema que o habilitam a prosseguir o atendimento de novas solicitações de serviço.

A hipótese (3) caracteriza um comportamento excepcional que exige a consistência de estado das réplicas. Esta hipótese somente se realizará em caso de falha da réplica no intervalo de tempo decorrido entre o envio do serviço solicitado e o envio da mensagem de atualização de estado.

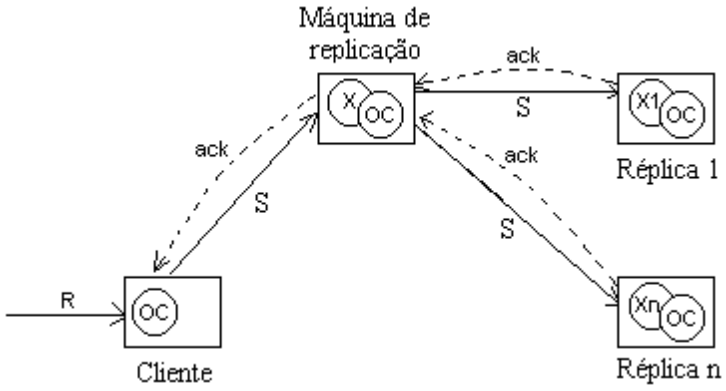

FIGURA 19 - Modelo de replicação

5.3 Um Protocolo para Garantir a Atomicidade

De acordo com a seção anterior, a atomicidade no recebimento das mensagens é crítica para a máquina de replicação baseada na técnica do primário-*backup*. Em função disso, as seções a seguir, baseadas em Babaoglu [BAB93], abordam o assunto de forma a oferecer uma solução que garanta o cumprimento da propriedade.

5.3.1 Transações Distribuídas

Uma transação distribuída é a execução de um programa, ou parte de um programa, que executa acessos a dados compartilhados em múltiplos nodos.

Para cada transação, os processos que realizam alterações em si mesmos são chamados de **participantes**. Cada participante atualiza seus dados locais e para encerrar uma transação deve coordenar suas ações com os outros participantes de tal forma que todos atualizem seus dados locais ou ninguém o faz. Entre os participantes, deve haver um elemento que coordena as ações. Além disso, cada transação deve possuir um identificador global único para permitir a identificação em sistemas com transações concorrentes. Neste exemplo, supõe-se que há somente uma transação de cada vez e elimina-se assim a identificação das mesmas.

O algoritmo, mostrado a seguir, ilustra o esquema de execução de uma transação distribuída e serve de contexto para a especificação e solução do problema do *commitment* atômico.

```
Algoritmo Transação
   % Algum participante (o invocador) executa
   Envia  (T_Start: transaction, Δc, participants)
          para todos os participantes
```

```
% Todos executam (inclusive o invocador)
Uma vez (Recebido (T_Start: transaction, Δc,
          participantes))
    Cknow:= Local_clock

    % realiza as operações da transação
    se ( pronto para tornar as alterações
         permanentes )
        Voto:= Yes
    Senão Voto:= No

    % Decide confirmar ou abortar a transação
    Atomic_commitment(transação, participantes)

Fim algoritmo
```

A prova detalhada do algoritmo pode ser encontrada em [BAB93]. Neste momento, o que interessa é que, como pode-se observar, o algoritmo não tem preocupações sobre quem é ou como foi escolhido o coordenador. Sua única necessidade é atribuir uma transação a um coordenador de tal forma a satisfazer os três axiomas seguintes:

AX1: no máximo um participante assumirá o papel de coordenador.

AX2: se não ocorrer nenhuma falha, um participante assumirá o papel de coordenador.

AX3: existe uma constante Δc, tal que nenhum participante assume o papel de coordenador além de Δc unidades de tempo físico depois de iniciada a transação.

Os axiomas AX1 e AX2 garantem que não mais que um participante assumirá o papel de coordenador de cada vez e, na ausência de falhas, com certeza um assumirá. O axioma AX3 permite limitar a duração de uma transação mesmo quando o coordenador falha antes de executar alguma ação.

É importante notar que o algoritmo não prevê a indivisibilidade de suas operações, isto é, o invocador pode falhar depois de ter enviado a mensagem para alguns mas não para todos os participantes.

5.3.2 Protocolo de *Commitment* Atômico

O objetivo do protocolo de *commitment* atômico é levar uma transação a um estado global consistente, a despeito da presença de falhas.

Para cada participante, deve selecionar entre duas possíveis decisões - *commit* (confirmar) e *abort* (cancelar). Decidir pela confirmação indica que todos os participantes farão a alteração tornar-se permanente, enquanto decidir pelo cancelamento indica que ninguém o fará. As decisões individuais são irreversíveis e uma decisão de confirmação é baseada na unanimidade de votos *YES* entre os participantes.

As quatro propriedades seguintes formalizam estas noções e definem o problema do *commitment* atômico:

AC1 : todos os participantes que decidem atingem a mesma decisão;
AC2 : se qualquer participante decide pela confirmação, então todos os participantes devem ter votado *YES*;
AC3 : se todos os participantes votam YES e não ocorrem falhas, então todos os participantes decidem confirmar;
AC4 : cada participante decide pelo menos uma vez.

Um protocolo que satisfaz todas as propriedades acima é chamado de **protocolo de *commitment* atômico**.

O algoritmo, a seguir, mostra tal protocolo que consiste de duas tarefas concorrentes, uma executada somente pelo coordenador (tarefa 1) e outra executada por todos os participantes, incluindo o coordenador (tarefa 2).

A prova detalhada do algoritmo pode ser encontrada em [BAB93].

```
Algoritmo Atomic_Commitment(transação, participantes)
    % Tarefa 1: executada pelo coordenador
    Envia [VOTE_REQUEST] para todos % inclusive para
                                      o coordenador
    Set-timeout-to local_clock + 2δ
    Espera-por ( votos de todos os participantes )
        se ( todos os votos são positivos )
            então broadcast(Commit,participantes)
            senão broadcast(Abort,participantes)
    on-timeout
        broadcast(Abort,participantes)

    Tarefa 2: executada por todos (incluindo o
                                   coordenador)
    Set-timeout-to Cknow + Δc + δ
    Espera-por ( VOTE_REQUEST do coordenador )
        Envia Voto para o coordenador
        se ( Voto = No )
            então decide Abort
            senão
               Set-timeout-to Cknow + Δc + 2δ + Δb
               Espera-por ( mensagem de decisão )
                   Se decisão é abortar
                      Então decide Abort
                      Senão decide Commit
               On-timeout
                   decide Abort
    on-timeout
        decide Abort

fim Algoritmo
```

Inicialmente o coordenador coleta os votos dos participantes através do envio da mensagem VOTE_REQUEST. Se ocorrer um *timeout* na espera da resposta dos participantes ou algum deles responder com um voto negativo, então o coordenador enviará uma mensagem de cancelamento da transação. Caso contrário, enviará uma mensagem de confirmação.

Cada participante espera do coordenador uma mensagem de VOTE_REQUEST e outra de confirmação. Um *timeout* na espera de qualquer uma delas causa um cancelamento da operação por parte do participante. Além disso, se não ocorrer um *timeout* mas o participante tiver votado negativamente, também haverá um cancelamento. Em uma situação sem *timeout*s e com voto positivo, o participante confirma a transação.

Finalmente, um ponto fundamental para o algoritmo `Atomic_Commitment` é a disseminação das decisões pelo coordenador, para todos os participantes, na tarefa 1. A primitiva para obter esta disseminação é chamada de *broadcast* (difusão) e tem uma ação correspondente no destino chamada *deliver* (entrega). É claro que *broadcast* e *deliver* são implementadas utilizando múltiplas operações de envio e recebimento através da rede.

A maneira mais simples de um processo **p** transmitir uma mensagem **m** para os membros de um conjunto **G** é **p** seqüencialmente enviar **m** para cada processo em **G**. Quando um processo em **G** recebe tal mensagem, ele a transmite para todos os participantes e então executa a entrega local.

Um algoritmo deste tipo, chamado de *Uniform Timed Reliable Broadcast (UTBR)*, satisfaz as propriedades abaixo ($\Delta b = \delta$):

B1 (Validade): se um processo correto transmite uma mensagem **m**, então todos os processos corretos em **G** entregam **m** em um tempo finito.

B2 (Integridade): para qualquer mensagem **m**, cada processo em **G** entrega **m** pelo menos uma vez, e somente se algum processo, de fato, transmitiu ***m***.

B3 (Δb-*Timeliness*): existe uma constante conhecida Δb tal que, se uma transmissão de **m** é iniciada no tempo físico **t**, nenhum processo em **G** entrega **m** depois do tempo $t + \Delta b$.

B4 (Acordo uniforme): se qualquer processo (correto ou não) em **G** entrega uma mensagem **m**, então todos os processos corretos em **G** entregam **m** em um tempo finito.

A seguir, esta reproduzido o algoritmo de broadcast UTBR:

```
Algoritmo broadcast(m,G)
   % o transmissor executa
   Envia [deliver m] para todos os processo em G
   deliver m

   % todos em G executam (menos o transmissor)
   Uma vez recebido (primeiro deliver m que chegar)
       Envia [deliver m] para todos os processo em G
       deliver m
Fim Algoritmo
```

A máquina de replicação utiliza uma adaptação dos algoritmos **Transação**, ***Atomic_Commitment*** e **broadcast**, recém descritos, para garantir que o primário obtenha atomicidade em relação aos *backups*, ou seja, uma mensagem de atualização de estado do primário para os *backups* deve ser recebida por todos os *backups* ou por nenhum deles. A principal adaptação feita deve-se ao fato de que os algoritmos foram elaborados para suportar qualquer tipo de transação e admitem transações que podem dar certo ou não, daí o esquema de unanimidade de votos. Porém, na máquina de replicação, um *backup* que não responde é eliminado do grupo dinamicamente de tal forma que o grupo é sempre formado somente por *backups* corretos. Desta forma, nunca há votos *NO* e, sendo assim, o controle da unanimidade dos votos foi suprimido da implementação.

5.4 Diagrama de Classes

A máquina de replicação possui as classes MaquinaReplicacao, InterfaceMaquinaReplicacao, FrameMaquina e Conexao. A FIGURA 20 representa o diagrama de classes da máquina de replicação.

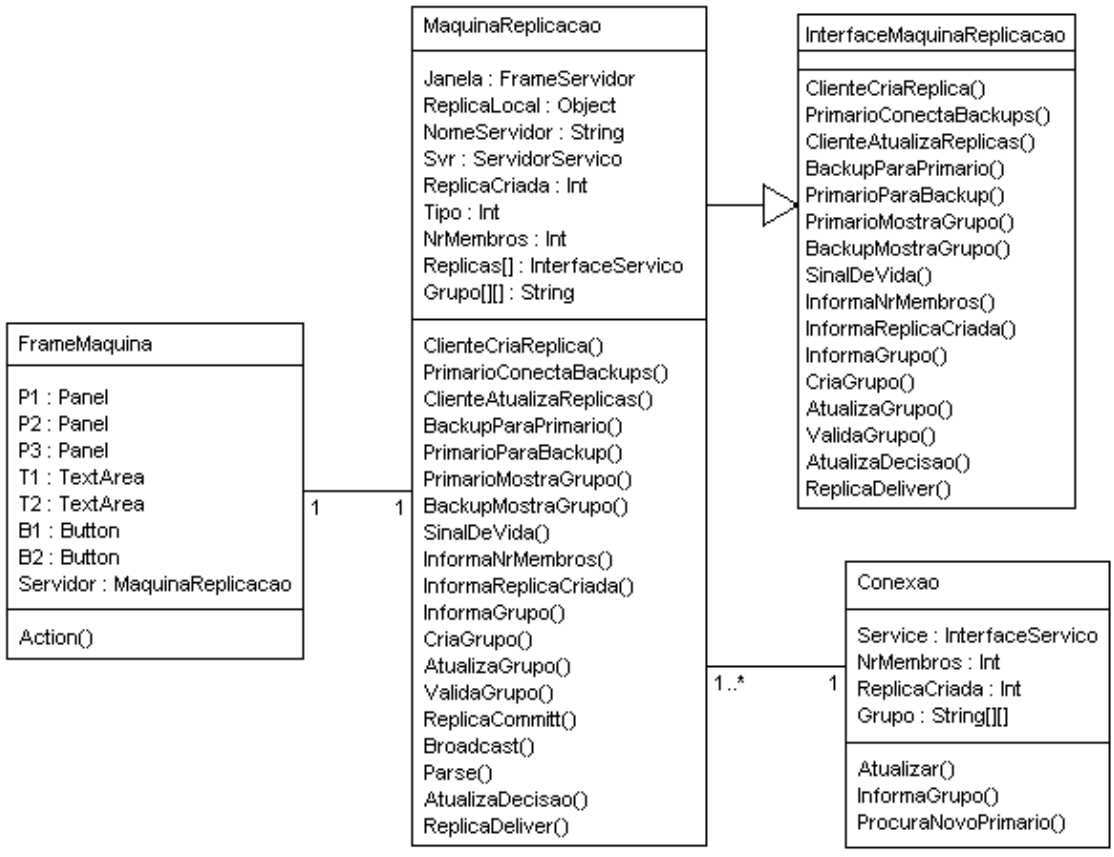

FIGURA 20 - Diagrama de classes da máquina de replicação

A classe InterfaceMaquinaReplicacao é uma interface que define objetos remotos. A classe MaquinaReplicacao implementa os métodos definidos na classe InterfaceMaquinaReplicacao. A classe FrameMaquina não faz parte do mecanismo de replicação e só é utilizada para implementar um *frame* que permite uma

interação amigável com a máquina de replicação. A classe Conexao faz a ligação entre a aplicação do usuário e a máquina de replicação. Seu objetivo é tornar transparente o mecanismo de conexão da aplicação com a maquina de replicação.

5.5 Algoritmos

Os principais serviços da máquina de replicação são:

- Criação das cópias
- Disseminação de mensagens
- Consistência das réplicas

É importante perceber que a máquina de replicação é servidora para a aplicação do usuário e é cliente para os *backups* que formam a base de replicação. Em relação à técnica primário-*backup*, a máquina de replicação é o primário. No entanto, os *backups* também são máquinas de replicação com um comportamento diferenciado. Quando há a necessidade de escolher um novo primário, um dos backups é designado para isso e passa a comportar-se como tal. Por isso, é preciso olhar os algoritmos e lembrar que eles se comportam ora de acordo com as necessidades de um primário e ora de acordo com as necessidades de um *backup*.

A criação das cópias é realizada pelo algoritmo `ClienteCriaReplicas` mostrado abaixo. Este algoritmo recebe o objeto crítico como parâmetro e utiliza variáveis de estado da classe da máquina de replicação para determinar se deve ser criada uma réplica ou se o cliente deve ser atualizado, a partir de um backup, com uma réplica já existente, isto é, deve ser recuperado após uma falha. O algoritmo devolve a réplica recém criada ou uma réplica que já estava criada.

```
Algoritmo ClienteCriaReplicas(Objeto t)

   { a variável de estado ReplicaCriada indica se a
     máquina de replicação já tem uma réplica criada
     ou não }
   se ReplicaCriada == 0
         { a variável de estado ReplicaLocal é na
           realidade uma instância da classe Objeto
           que é utilizada para armazenar uma cópia
           do objeto crítico }

         ReplicaLocal  = t
         ReplicaCriada = 1

   Fim se

FimAlgoritmo(ReplicaLocal)
```

O algoritmo `ClienteCriaReplicas` tem por objetivo configurar o estado da máquina de replicação (inicializada ou não inicializada). Na realidade, a

criação propriamente dita do objeto que receberá uma cópia do objeto crítico é feita quando ocorre a inicialização da própria máquina de replicação.

A disseminação das mensagens e consistência das réplicas é feita inicialmente pelo algoritmo `ClienteAtualizaReplica` mostrado abaixo. Este algoritmo apoia-se no conceito de transação, já discutido na seção 5.3, e utiliza a variável de estado `Tipo`, que indica o tipo da réplica (primário ou *backup*), para determinar a ação a ser executada em função do tipo da réplica.

```
Algoritmo ClienteAtualizaReplica(Objeto t)

   se ( Tipo == 1 )
      ValidaGrupo()

      Para i = 0 até NrMembros

        Bloco de Exceção
           { A matriz Grupo é composta pelo nome das
             referências remotas onde estão as
             réplicas, o tipo da réplica (primário ou
             backup) e o status da réplica (ativo ou
             inativo).
             O vetor Replica é composto pelas
             referências remotas das réplicas.
           }

           se Grupo[i][1]="Backup" e
              Grupo[i][2]="Ativo"
              Replica[i].ClienteAtualizaReplica(t)
           Fim se

        Ao ocorrer Exceção
           Grupo[i][2] = "Inativo"

        Fim exceção

      Fim para
   Fim se

   { A variável de estado ReplicaCriada indica se
     está réplica já foi inicializada com uma cópia
     do objeto crítico.
     A variável ReplicaTemp é utiliza para armazenar
     a atualização enquanto a réplica espera a
     mensagem de Commit.
     A variável Decisao indica o recebimento da
     mensagem de Commit. }
```

```
   ReplicaCriada = 1
   ReplicaTemp   = t
   Decisao       = 0

   ReplicaCommit()

FimAlgoritmo
```

Os algoritmos da máquina de replicação manipulam os índices da matriz Grupos e do vetor Replica sempre em ao mesmo tempo. Assim, uma linha **i** da matriz tem sua correspondência na posição **i** do vetor.

O algoritmo `ReplicaCommit`, mostrado abaixo, dá continuidade ao esquema de consistência e tem o objetivo de iniciar o processo de decisão das réplicas. Novamente a variável `Tipo` é utilizada para determinar a ação. O primário simplesmente envia uma mensagem de confirmação através do algoritmo *`Broadcast`* para todos os membros do grupo. Cada *backup* inicia uma contagem de *timeout* e espera a chegada da mensagem de confirmação neste período. Se a mensagem chegar, o *backup* executa a sua confirmação através do algoritmo `ReplicaDeliver` e então repassa a mensagem para os outros membros do grupo. Se a mensagem não chegar e ocorrer um *timeout*, o *backup* cancela a operação. O cancelamento significa simplesmente não executar a confirmação feita por `ReplicaDeliver`.

```
Algoritmo ReplicaCommit

   se ( Tipo = 1 )
      então Broadcast
      senão Inicia contagem de timeout
                ReplicaDeliver()
                Envia ReplicaDeliver() para todos do
                      grupo
            Ao ocorrer timeout
                Cancela a operação

   Fim se

Fim algoritmo
```

O algoritmo `ReplicaCommit` é uma simplificação do algoritmo de *commitment* atômico não bloqueante, proposto por Babaoglu [BAB93], que se deve ao fato de não ser necessário o controle de votos. Isto acontece porque as réplicas nunca votam NO. Isto não significa que elas não falham. Significa que ao falhar, a réplica é excluída do grupo e não mais se espera resposta dela. Desta forma, o grupo é sempre composto por réplicas operantes e que respondem YES.

Finalizando o processo de consistência, o algoritmo `Broadcast` é responsável por enviar a mensagem de confirmação para todos os membros do grupo:

```
Algoritmo Broadcast
   Para i = 0 até NrMembros
      Bloco de exceção
        Se Grupo[i][1]="Backup" e Grupo[i][2]="Ativo"
           Replica[i].AtualizaGrupo(nrMembros,Grupo,R
                                        eplicas)
           Replica[i].AtualizaDecisao()
        Fim se

      Ao ocorrer Exceção
        Grupo[i][2] = "Inativo"

      Fim Exceção
   Fim para

   ReplicaDeliver()

Fim Algoritmo
```

O algoritmo `ValidaGrupo`, utilizado por `ClienteAtualizaReplica`, é mostrado abaixo. Este algoritmo percorre a lista de membros e tenta recuperar os membros que estão inativos.

```
Algoritmo ValidaGrupo

   para i = 0 até NrMembros

      se Grupo[i][1] = "BACKUP"
         se Grupo[i][3] = "INATIVO"

            Replica[i] = obtém referência a máquina
                         Grupo[i][0]+Grupo[i][2]

            se Replica[i].SinalDeVida
               então Grupo[i][3] = "ATIVO"
               senão Grupo[i][3] = "INATIVO"
            fim se

         fim se
      fim se

   fim para

FimAlgoritmo
```

Os algoritmos acima compõem a estratégia de consistência baseada na atomicidade na entrega das mensagens.

Por outro lado, há situações onde uma réplica pode falhar em um momento em que não há troca de mensagens. Quando isto acontece, os algoritmos da máquina de replicação utilizam a estratégia de perguntar às réplicas se elas estão ativas. Assim, quando um *backup* cai, o primário não é avisado e irá perceber a falha quando não obtiver resposta para uma atualização de estado. Uma vez detectado que o *backup* caiu, a informação é repassada aos outros *backups*. Os *backups* mantêm a matriz de grupo porque, quando um deles é promovido a primário, ele necessitará conhecer os membros do grupo.

A recuperação de um backup é detectada porque o algoritmo de atualização de estado sempre pede um sinal de vida dos inativos. Quando um dos inativos responde ele é imediatamente identificado como "ativo" e tem seu estado atualizado.

A idéia central para manter o grupo sempre com a informação correta sobre quem está ativo ou não é atualizar o estado das réplicas uma a uma e identificar como inativas aquelas não responderam à mensagem de atualização. Uma réplica que está identificada como inativa não é considerada quando houver a necessidade de escolher um novo primário. No entanto, ela fica no grupo e, a cada operação de atualização realizada pelo primário, é requisitada a responder com um sinal de vida. Quando o fizer, ela voltará a ser atualizada e identificada como ativa pelo primário.

O algoritmo `ProcuraNovoPrimario` tem um papel importante na obtenção da consistência da réplicas. O algoritmo é mostrado abaixo:

```
Algoritmo ProcuraNovoPrimario(Object t)

   para i = NrMembros-1 até 0

      se Grupo[i][1] = "PRIMARIO"
         então
            Grupo[i][1] = "BACKUP";
            Grupo[i][3] = "INATIVO";
         senão
            se !Ok
               replica = obtém referência a máquina
                         Grupo[i][0]

               se replica.SinalDeVida()
                  então
                     Grupo[i][1]   = "PRIMARIO"
                     Grupo[i][2]   = "ATIVO"
                     Ok            = true
                  senão
                     Grupo[i][2]   = "INATIVO"
               FimSe

            Fim Se
```

```
      Fim se

   FimPara

   { a atualizacao é feita aqui porque o grupo estará
     atualizado somente após a execução completa do
     laço acima }

   se Ok
      então

         Bloco de exceção
            replica.BackupParaPrimario();
            replica.AtualizaGrupo(NrMembros, Grupo);
            replica.PrimarioConectaBackups();
            replica.ClienteAtualizaReplica(t);

         Ao ocorrer exceção
             Ok = false
             Escreva "a replica " + Grupo[i][0] +
                     "não pode assumir o primário"

         Fim exceção

      senão
         Ok = true
         Escreva "nenhuma replica pôde assumir o
                  primário"

   Fim se


Fim Algoritmo
```

Quando o primário cai, o cliente e os backups não são avisados. O cliente irá perceber a falha quando não receber resposta a uma invocação de atualização. Neste momento, o algoritmo `ProcuraNovoPrimario` é executado pelo cliente e percorre a matriz de membros para identificar o antigo primário como um backup inativo. O algoritmo seleciona o novo primário e pede uma sinal de vida. Se receber resposta, a matriz é atualizada para representar a nova situação. Finalmente, após escolher o novo primário, o algoritmo comunica a ele sua promoção, atualiza sua matriz de grupo, solicita sua conexão aos backups restantes e atualiza seu estado. Quando o novo primário conecta-se aos backups ele os informa das alterações ocorridas no grupo.

Quando o antigo primário se recuperar, ele será reintegrado ao grupo porém, agora como uma réplica backup.

Se o cliente não conseguir obter nenhum primário, o algoritmo continuará tentando a cada vez que houver uma alteração no estado do objeto crítico até que alguma das réplicas se recupere e possa assumir o lugar do primário.

Quando o primário cai e recupera-se antes que o cliente perceba, ele volta a seu estado inicial e perde os valores do objeto crítico. Nesta situação, se o primário receber uma mensagem de atualização de estado não haverá problemas porque ele atualizará seu estado, de acordo com a mensagem recém recebida, e repassará a mensagem aos backups normalmente. No entanto, se o primário receber um pedido de atualização do estado do cliente (no caso do cliente ter caído e também ter se recuperado) ele deve ser capaz de perceber que seu estado está desatualizado e que o estado atual do objeto crítico encontra-se em qualquer um dos backups ativos. Para isso, quando o primário conecta-se aos *backups* ele compara o valor da sua variável local `ReplicaCriada` ao valor dessa mesma variável em cada um dos backups. Esta variável indica se uma réplica, seja primário ou *backup*, já recebeu alguma atualização ou não. Assim, se o primário encontra alguma réplica onde a variável `ReplicaCriada` indica a existência de uma atualização de estado, ele identifica a si próprio, através de variáveis locais, como desatualizado. Quando o cliente invoca o primário e percebe o estado de desatualizado ele procede à seleção de um novo primário entre as *backups* que estão atualizados.

A algoritmo de conexão do primário aos backups `PrimarioConectaBackups` é mostrado abaixo:

```
Algoritmo PrimarioConectaBackups
   para i = 0 até NrMembros
      Bloco de exceção
         se Grupo[i][1] = "BACKUP" e
            Grupo[i][2].= "ATIVO"
            então
               Replica[i]   =   obtém   referência   a
                                máquina Grupo[i][0]

               Replica[i].AtualizaGrupo(NrMembros,
                                        Grupo,
                                        Replica)

               { se o primario não tem replica criada
                 ele verifica se algum dos backups
                 tem }

               se ReplicaCriada == 0
                 se Replica[i].InformaReplicaCriada()
                                                  = 1
                    então ReplicaCriada    = 2
                 FimSe
               FimSe
         FimSe

      Ao ocorrer exceção
         Escreva "erro na réplica " + Grupo[i][0]
      Fim exceção
```

```
    FimPara

Fim Algoritmo ReplicaCriada
```

O algoritmo `PrimarioConectaBackups` percorre a matriz de grupo e, para os elementos que são backups e estão ativos, obtém o acesso remoto. A `ReplicaCriada`, como já foi descrito, é utilizada para identificar o estado de atualizado ou desatualizado de uma réplica.

5.6 Esquema Hierárquico dos Algoritmos

A FIGURA 21 representa de forma hierárquica a utilização dos algoritmos. No topo da hierarquia está a aplicação do usuário que utiliza os algoritmos através de sua implementação na classe de replicação.

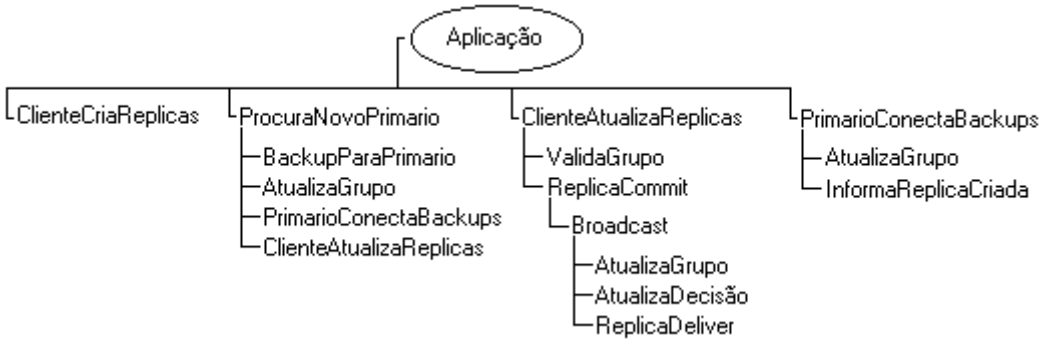

FIGURA 21 – Esquema hierárquico dos algoritmos

6 Aspectos de Implementação

Antes da descrição da implementação deste trabalho, é importante visualizar a máquina de replicação do ponto de vista da arquitetura RMI.

Primeiramente é preciso entender que o conjunto de réplicas é formado pela máquina de replicação e pelas réplicas espalhadas em outros servidores.

Como baseiam-se em Java RMI, estas réplicas comportam-se segundo a arquitetura cliente/servidor: há um cliente (a máquina de replicação) e vários servidores (as réplicas). Assim, a máquina de replicação é um servidor para a aplicação do usuário e um cliente das réplicas. Por isso, a denominação de cliente ou servidor depende da operação e do momento em análise. Esta dualidade de comportamento é mostrada através da FIGURA 22: na situação 1, a máquina de replicação comporta-se como um servidor RMI para a aplicação do usuário; ao mesmo tempo, na situação 2, ela comporta-se como um cliente RMI dos servidores que representam as réplicas.

É importante perceber também que a máquina de replicação e as réplicas são execuções diferentes da mesma aplicação. Por isso, todas as réplicas são máquinas de replicação e esta, por sua vez, é uma réplica.

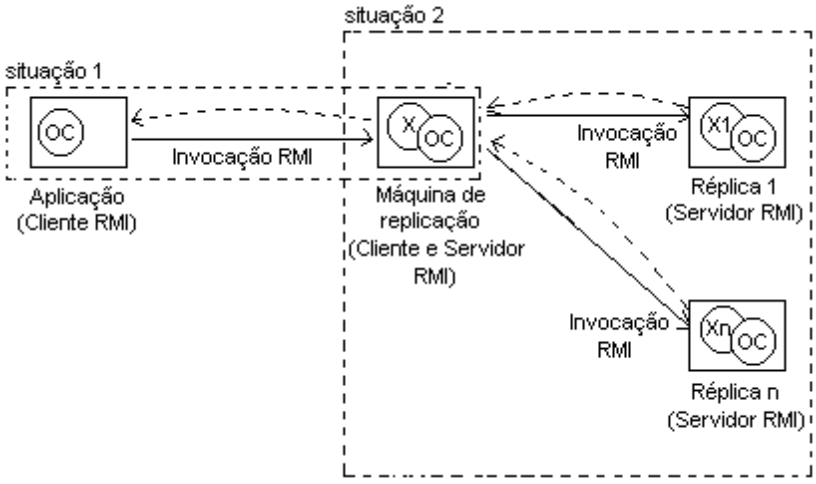

FIGURA 22 - Máquina de replicação do ponto de vista do RMI

6.1 Implementação de Aplicações RMI

As aplicações RMI são muitas vezes divididas em dois programas [JDK99]. Uma aplicação servidora típica cria alguns objetos remotos, torna as referências aos objetos acessíveis aos clientes e aguarda que clientes realizem invocações sobre esses objetos remotos. Uma aplicação cliente típica obtém uma referência remota para um ou mais objetos remotos e então invoca métodos desses objetos. RMI oferece o mecanismo pelo qual um servidor e um cliente comunicam-se e passam informações de um para o outro. Tal aplicação é algumas vezes denominada de *aplicação objeto distribuída*.

Este tipo de aplicação precisa:

- localizar objetos remotos: aplicações podem usar um ou mais mecanismos para obter referências aos objetos remotos. Uma aplicação pode registrar

seus objetos remotos através do serviço de nomeação do Java RMI, *rmiregistry*, ou a aplicação pode passar e retornar referências a objetos remotos como parte de sua operação normal. O *rmiregistry*, recém citado, é um serviço simples de nomeação de objetos remotos que permite a clientes remotos obterem referências para objetos remotos através de seu nome;

- comunicar-se com objetos remotos: detalhes da comunicação entre objetos remotos são gerenciados pelo RMI. Para o programador, a comunicação remota parece com uma invocação local padrão;
- ler *bytecodes* de classes que são transmitidas pela aplicação: a passagem de objetos para objetos remotos é permitida através de mecanismos de leitura do código do objeto, bem como de mecanismos para a transmissão de seus dados.

A máquina de replicação utiliza todas as características descritas acima. O serviço de registro é utilizado para fazer referência às diferentes réplicas onde ficam instalados os servidores. A comunicação entre os objetos remotos é transparente devido ao gerenciamento feito pelo RMI e, finalmente, os objetos considerados críticos serão transmitidos para os objetos remotos localizados nos servidores.

6.2 Arquitetura do Sistema

Por utilizar a arquitetura RMI, a máquina de replicação é dividida em duas partes: cliente e servidor. Na realidade, há vários servidores distribuídos onde cada um deles é a base de funcionamento de uma réplica. A máquina de replicação faz a mesma invocação para todos os objetos remotos localizados em cada um dos servidores

6.2.1 Implementação dos Servidores RMI

O papel dos servidores é receber a tarefa de atualização de estado da máquina de replicação, executá-la e retornar algum resultado. Os servidores são compostos por uma interface e uma classe. A interface oferece as definições para os métodos que podem ser chamados pela máquina de replicação. A classe é a implementação da interface.

6.2.2 Definição da Interface Remota

A interface de servidor `InterfaceMaquinaReplicacao` permite que objetos sejam submetidos aos servidores e define as operações que implementam os algoritmos de replicação:

```
package InterfaceMaquina;

import java.rmi.*;

public interface InterfaceMaquinaReplicacao
                              extends java.rmi.Remote
{
   Object ClienteCriaReplicas(Object t)
```

```
        throws RemoteException;

int     PrimarioConectaBackups()
        throws RemoteException;

void    ClienteAtualizaReplica(Object t)
        throws RemoteException;

void    BackupParaPrimario()
        throws RemoteException;

void    PrimarioParaBackup()
        throws RemoteException;

void    PrimarioMostraGrupos()
        throws RemoteException;

void    BackupMostraGrupo()
        throws RemoteException;

void    SinalDeVida()
        throws RemoteException;

int     InformaNrMembros()
        throws RemoteException;

int     InformaReplicaCriada()
        throws RemoteException;

String[][] InformaGrupo()
        throws RemoteException;

void    CriaGrupo(int TotalDeMembros, String[]
        Argumentos) throws RemoteException;

void    AtualizaGrupo(int NumeroDeMembros,
                      String[][] GrupoAtualizado,
                      InterfaceMaquinaReplicacao[]
                      Referencias)
                      throws RemoteException;

void    ValidaGrupo()
        throws RemoteException;

void    AtualizaDecisao()
        throws RemoteException;

void    ReplicaDeliver(String OrigemDaMensagem)
        throws RemoteException
```

```
}
```

Através da extensão da interface `java.rmi.Remote`, a interface `InterfaceMaquinareplicacao` faz com que seus próprios métodos possam ser chamados a partir de qualquer máquina virtual; além disso, qualquer classe que implemente esta interface torna-se um objeto remoto.

6.2.3 Implementação da Interface Remota

Uma vez definidas as interfaces necessárias aos servidores de réplicas da máquina de replicação, pode-se detalhar a classe `MaquinaReplicacao`, que implementa a interface `InterfaceMaquinaReplicacao.`

Em geral, a implementação de uma classe de uma interface remota deve pelo menos:

- declarar a interface remota que está sendo implementada;
- definir um método construtor para o objeto remoto;
- oferecer uma implementação para cada método remoto da interface remota.

Um servidor RMI tem um procedimento padrão: criar e instalar objetos remotos. Isto pode ser feito dentro do método `main` na implementação do objeto remoto da própria classe do servidor ou pode ser feito em outra classe especifica para este propósito. Este procedimento deve:

- criar e instalar um gerenciador de segurança;
- criar uma ou mais instâncias de um objeto remoto;
- registrar pelo menos um dos objetos remotos com um serviço de nomeação, tal como o *rmiregistry*.

A implementação completa da classe `MaquinaReplicacao` pode ser encontrada no ANEXO 1, mas o detalhamento de seus componentes será discutido nos parágrafos a seguir.

A implementação da classe `MaquinaReplicacao` é declarada como:

```
public class MaquinaReplicacao
             extends UnicastRemoteObject
             implements  InterfaceMaquinaReplicacao,
                         Runnable
```

Esta declaração afirma que a classe `MaquinaReplicacao` implementa a interface `InterfaceMaquinaReplicacao`, portanto define um objeto remoto, e também a interface *`Runnable`*, que permite a execução de *threads*.

Além disso, a classe `MaquinaReplicacao` é derivada da classe `java.rmi.server.UnicastRemoteObject`, a qual é definida pela API do RMI, que pode ser usada como uma superclasse para a implementação de objetos

remotos. Assim, a classe `MaquinaReplicacao` pode ser usada para criar um único objeto remoto que suporta comunicação remota *unicast* (ponto-a-ponto) e que usa os mecanismos da arquitetura RMI para comunicação.

O método construtor da classe `MaquinaReplicacao` simplesmente chama o construtor da superclasse `UnicastRemoteObject.`

O método `main` é utilizado para iniciar a máquina de replicação e por isso deve ser capaz de prepará-la para aceitar chamadas dos clientes. A primeira ação a ser feita é criar e instalar um gerente de segurança, que protege o sistema de acessos não confiáveis:

```
if (System.getSecurityManager() == null)
    System.setSecurityManager (new
                                RMISecurityManager());
```

A seguir, ele cria uma instância da classe `Servidor`:

```
svr = new MaquinaReplicacao();
```

Neste momento, como já mencionado, o método construtor chama o construtor da classe `UnicastRemoteObject` que, ao ser executado, exporta o objeto recém criado para a *runtime* do RMI. Uma vez que a exportação é completada, o objeto remoto servidor está pronto para receber chamadas de clientes em uma porta anônima escolhida pelo RMI ou pelo sistema operacional.

Embora esteja pronto para receber chamadas de clientes, antes que um cliente possa invocar um objeto remoto, deve obter uma referência ao objeto remoto. Neste ponto, utiliza-se o *rmiregistry* para encontrar referências a objetos remotos. Este serviço é tipicamente utilizado somente para localizar o primeiro objeto remoto que um cliente RMI pode utilizar.

Para determinar o nome do servidor, a classe `MaquinaReplicacao` utiliza o método `Parse`:

```
NomeDoServidor = rmi://" + svr.Parse(args[1])
```

Este método recebe um argumento da linha de comando e extrai dele o nome e o identificador do servidor. A descrição da forma da linha de comando para a inicialização da máquina de replicação é descrita da seção 6.3.

Após definir o nome, a interface `java.rmi.Naming` é utilizada como *API front-end* para ligar, registrar ou procurar objetos remotos dentro do *rmiregistry*. Uma vez que um objeto remoto é registrado com o *rmiregistry* em um hospedeiro local, clientes em quaisquer outros hospedeiros podem procurar este objeto pelo nome, obter sua referência e então invocar os métodos remotos do objeto.

A classe `MaquinaReplicacao` inclui o nome de objeto no **Rmiregistry** com a sentença:

```
Naming.rebind (NomeDoServidor, svr);
```

O *frame* onde está inserida a interface de entrada/saída é criado partir da classe `FrameMaquina`:

```
Janela = new FrameMaquina(svr);
```

Finalmente, após a nomeação do servidor e a criação do *frame* da interface, a condição mostrada abaixo determina se a máquina de replicação deve comportar-se como primário ou como *backup*. Esta condição leva em consideração o argumento de linha de comando `args`[0] que representa o tipo do servidor (primário ou *backup*).:

```
if (args[0].toUpperCase().equals("PRIMARIO"))
{
   Janela.setTitle("Primario " + NomeDoServidor);

   svr.Tipo             = 1;
   svr.NrMembros        = args.length - 1;
   svr.CriaGrupo(svr.NrMembros, args);
   svr.ReplicaCriada = svr.PrimarioConectaBackups();
   Janela.t1.append("\n");
}
else
{
   Janela.setTitle("Backup " + NomeDoServidor);
}
```

A classe `MaquinaReplicacao` implementa também o método `CriaGrupo` que é responsável pela configuração da matriz `Grupo` que representa o conjunto das réplicas. Este método recebe o número de membros, calculado pelo método `main`, e os argumentos da linha de comando e os utiliza em um laço `for` para preencher a matriz Grupo:

```
public void CriaGrupo(int TotalDeMembros, String[]
                      Argumentos)
{  int i = 2 , j = 0;

   // cria lista de membros
```

```
   for ( j=0 ; j < (TotalDeMembros - 1); j++ )
   {
      Grupo[j][0] = svr.Parse(Argumentos[i]);
      Grupo[j][1] = "BACKUP";
      Grupo[j][2] = "ATIVO";
      i++;
   }

   Grupo[j][0] = svr.Parse(Argumentos[1]);
   Grupo[j][1] = "PRIMARIO";
   Grupo[j][2] = "ATIVO";
}
```

Cada linha da matriz representa uma réplica e as três colunas representam o nome da referência remota, o tipo e o estado da réplica respectivamente. O nome da referência remota é obtido pelo método `Parse` através da concatenação do nome do hospedeiro e do identificador. A matriz começa a ser preenchida pelos *backups* e em sua última posição é inserido o primário.

O método `PrimarioConectaBackups` utiliza a matriz `Grupo` preenchida pelo método `CriaGrupo` para conectar-se aos *backups*. Um laço `for` é utilizado para percorrê-la e para as linhas que indicam um backup ativo é obtida uma referência àquele objeto remoto. A referência é armazenada no vetor `Replica`.

```
public int PrimarioConectaBackups()
{ int i;

 for ( i=0 ; i < NrMembros ; i++ )
 {
   try
   { // só conecta aos backups ativos
     if(Grupo[i][1].toUpperCase().equals("BACKUP") &&
        Grupo[i][2].toUpperCase().equals("ATIVO") )
     {
        Janela.t1.append("\nConectando backup rmi://+
                         Grupo[i][0]);

        Replica[i] = (InterfaceMaquinaReplicacao)
                     Naming.lookup("rmi://"+
                     Grupo[i][0]);

        Replica[i].AtualizaGrupo(NrMembros, Grupo,
                                 Replicas);

        // se o primario não tem replica criada ele
           verifica se algum dos backups tem

        if ( ReplicaCriada == 0 )
        {
```

```
                if(Replica[i].InformaReplicaCriada() ==1)
                {
                    ReplicaCriada    = 2;
                }
            }
          }
        }
        catch (Exception e)
        {
          Janela.t1.append("\n Erro na conexão da replica
                           ... " + Grupo[i][0]);
        }
      }

      return ReplicaCriada;
    }
```

Uma característica importante do método `PrimarioConectaBackups` é que ele utiliza a variável de estado `ReplicaCriada` para determinar se a máquina de replicação está desatualizada. Quando uma réplica é inicializada, a variável `ReplicaCriada` recebe o valor 0. Ao ser atualizada, a variável `ReplicaCriada` recebe o valor 1. Assim, se no momento em que é inicializada, a máquina de replicação detecta alguma réplica já atualizada, ela atribui à sua variável `ReplicaCriada` o valor 2 para indicar à aplicação cliente que há alguma réplica mais atualizada.

Os métodos `ClienteAtualizaReplica`, `ReplicaCommit`, *`Broadcast`*, *`run`*, `ReplicaDeliver` e `AtualizaDecisao` fazem parte do mecanismo de *commitment* atômico que garante a consistência das réplicas através da atomicidade na entrega das mensagens. Este métodos são detalhados a seguir.

O método `ClienteAtualizaReplica` é acessado remotamente pelo cliente e é responsável pelo início da transação que atualiza o estado do primário e dos *backups*. Quando este método é executado pelo primário, ele inicialmente testa as informações da matriz `Grupo` para detectar possíveis falhas e então a utiliza para iniciar a transação de atualização em todos os *backups*. Após ter acessado a todos os *backups*, ele inicializa as variáveis `ReplicaCriada`, `ReplicaTemp` e `Decisao`. Essas variáveis são utilizadas pelo mecanismo de *commitment* atômico.

Por outro lado, quando este método é executado por um *backup*, ele simplesmente atribui os mesmos valores às variáveis `ReplicaCriada`, `ReplicaTemp` e `Decisao` e também executa o método `ReplicaCommit`.

```
public void ClienteAtualizaReplica(Object t)
{ int i


  if (svr.Tipo == 1)
  {
```

```
    ValidaGrupo()

    for ( i = 0 ; i < NrMembros ; i++ )
    {
      try
      {
       if (Grupo[i][1].toUpperCase().equals("BACKUP")
           &&
          (Grupo[i][2].toUpperCase().equals("ATIVO")
       {
          Replicas[i].ClienteAtualizaReplica(t);
       }
      }
      catch (Exception e)
      {
       Grupo[i][2] = "INATIVO"
       Janela.t1.append("\n Replica " + Grupo[i][0] +
                        "esta inoperante e não
                        iniciou a transação ");
      }
    }
  }

  // Todos os participantes executam.
  // A variável ReplicaCriada indica se a réplica já
     recebeu alguma atualização de estado.
  // A variável ReplicaTemp armazena temporariamente
     o novo estado do objeto replicado.
  //  A variável Decisao indica se a réplica já
     recebeu a mensagem de Commit para a atualização.
  ReplicaCriada = 1;
  ReplicaTemp   = 1

  Decisao       = 0;

  ReplicaCommit();
}
```

O método `ReplicaCommit` faz parte do mecanismo de *commitment* atômico. A sua execução também é diferenciada para primários e *backups*. Quando executado pelo primário, ele executa o método `Broadcast` que é responsável por enviar a mensagem de confirmação de atualização para todos os *backups*. Quando é executado por um *backup*, o método `ReplicaCommit` inicia uma contagem de tempo para determinar a ocorrência ou não de um *timeout*. Esta contagem de tempo é feita através da *thread* `myThread`.

```
public void ReplicaCommit()
{
```

```
    if (svr.Tipo == 1)
    {
       // executado somente pelo primário
       BroadCast();
    }
    else
    {
       myThread = new Thread(this);
       myThread.start();
    }
}
```

O método `Broadcast` consiste de um laço *for* que percorre a matriz `Grupo` e, para as entradas que indicam *backups* ativos, executa remotamente os métodos `AtualizaGrupo` e `AtualizaDecisao` das réplicas. O método `AtualizaGrupo` atualiza, em cada réplica, a situação presente da matriz `Grupo`. O método `AtualizaDecisao` atualiza, em cada réplica, o valor da variável `Decisao` que representa a mensagem de confirmação de atualização. Ao final, é executado o método local `ReplicaDeliver` que é responsável pela atualização de fato do objeto local.

```
public void BroadCast()
{
  int i;

  for ( i=0 ; i < NrMembros ; i++ )
  {
    try
    {
      if(Grupo[i][1].toUpperCase().equals("BACKUP") )
      {
        if(Grupo[i][2].toUpperCase().equals("ATIVO")
                                                    )
        {
          Replicas[i].AtualizaGrupo(NrMembros, Grupo,
                                    Replicas);
          Replicas[i].AtualizaDecisao();
        }
        else
        {
          Janela.t1.append("\n Replica "+Grupo[i][0]+
                           " esta inoperante ...");
        }

      }
    }
    catch (Exception e)
    {
```

```
            Grupo[i][2] = "INATIVO";
            Janela.t1.append("\n Replica "+Grupo[i][0]+
                             " esta inoperante ...");
          }
        }

        ReplicaDeliver(NomeDoServidor);
      }
```

O método `AtualizaDecisao` simplesmente atualiza o valor da variável `Decisao`. Esta variável é utilizada pelo mecanismo de *timeout* para verificar se a réplica já recebeu a mensagem de confirmação de atualização.

```
public void AtualizaDecisao()
{
   Decisao = 1;
}
```

O método `ReplicaDeliver` é executado localmente por cada réplica e simplesmente atualiza a variável `ReplicaLocal` a partir da variável `ReplicaTemp`. Esta atribuição representa a operação ***commit*** do mecanismo de *commitment* atômico.

```
public void ReplicaDeliver(String OrigemDaMensagem)
{  ReplicaLocal = ReplicaTemp;

   Janela.t1.append("\n A réplica"+NomeDoServidor+
                    "foi atualizada por " +
                    OrigemDaMensagem);
}
```

A operação *abort* do *commitment* atômico é representada pela não execução do método `ReplicaDeliver`.

O mecanismo de *timeout* é implementado pelo método `run`. Como já foi descrito, a contagem de tempo é feita pela *thread* `myThread` que utiliza, como todas as *threads* em Java, este método para a sua implementação.

Inicialmente, é utilizado um laço `while` para esperar por 30 segundos a chegada da mensagem de confirmação (*commit*). Ao chegar dentro do prazo, esta mensagem altera o valor da variável `Decisao` e causa a interrupção do laço. Se a mensagem de confirmação chegar no prazo e não ocorrer um *timeout*, a réplica executa o seu método local `ReplicaDeliver` e envia também uma mensagem de confirmação para as outras réplicas. Este envio garante que, se uma réplica receber a mensagem de confirmação, todas a receberão, em um tempo finito, mesmo que o primário falhe durante a operação.

```
public void run()
{  int i, TempoDecorrido = 0;

   // Espera 30 segundos
   while ( Decisao == 0 && TempoDecorrido < 30 )
   {
     try
     {
        Thread.sleep(1000);
     }
     catch (InterruptedException e)
     {

     }

     TempoDecorrido++;
   }

   // Se a variável Decisao tiver valor 1 é porque a
      mensagem de confirmação chegou
   if ( Decisao == 1 )
   {
     // Atualiza o seu estado
     ReplicaDeliver(NomeDoServidor);

     // Envia a mensagem de confirmação para as
        outras réplicas
     for ( i=0 ; i < NrMembros ; i++ )
     {
       try
       {
         if(Grupo[i][1].toUpperCase().equals("BACKUP"
                                              ))
         {
           if(Grupo[i][2].toUpperCase().equals("ATIVO
                                              "))
           {
              if ( !NomeDoServidor.equals("rmi://"+
                   Grupo[i][0].toLowerCase() ) )
              {
                 Janela.t1.append("\n Enviando
                                  Delivery para " +
                                  Grupo[i][0] + " ...
                                  ");

                 Replicas[i].AtualizaGrupo(NrMembros,
                                           Grupo,
                                           Replicas);
```

```
                    Replicas[i].AtualizaDecisao();

                    Janela.t1.append(" Ok ");
                  }
                }
              }
            }
            catch (Exception e)
            {
              Grupo[i][2] = "INATIVO";
              Janela.t1.append("\n Replica "+Grupo[i][0]+
                               " esta inoperante ...");
            }
          }
        }
        else
        {
          Janela.t1.append("\n Ocorreu um timeout e a
                           replica "+NomeDoServidor + "
                           abortou a operacao ...");
        }
    }
```

Não faz parte dos objetivos deste trabalho determinar exatamente qual o melhor valor de intervalo de tempo para ocorrência de *timeout*. Por isso, o valor de 30 segundos foi escolhido de forma despretensiosa. Uma discussão a este respeito, no contexto do problema do *commitment* atômico, pode ser encontrada em [BAB93].

A seguir, são detalhados métodos da máquina de replicação que não fazem parte diretamente do mecanismo de *commitment* atômico mas também ajudam na manutenção da consistência das réplicas através de operações de criação e atualização da matriz `Grupo`, recuperação de réplicas falhas, transformação de um *backup* em primário e vice-versa e troca de informações entre as réplicas.

Para que o primário faça a atualização dos *backups* ele precisa saber quem são e qual a referência remota a cada *backup*. Além disso, o cliente também precisa saber porque eventualmente será necessário escolher um novo primário entre os *backups*.

Para testar as informações armazenadas na matriz `Grupo`, o método `ValidaGrupo` utiliza um laço *for* para percorrê-la e para cada *backup* pede um sinal de vida através da execução remoto do método `SinalDeVida`. Como resultado da execução, este método obtém a matriz `Grupo` atualizada com a situação atual das réplicas.

```
public void ValidaGrupo()
{  int i;

   for ( i=0 ; i < NrMembros ; i++ )
   {
```

```
        if (Grupo[i][1].toUpperCase().equals("BACKUP"))
        {
          try
          {
             if(Grupo[i][2].toUpperCase().equals("INATI
                                                  VO"))
             { Replica[i] =(InterfaceMaquinareplicacao)
                           Naming.lookup
                           ("rmi://"+Grupo[i][0]);
             }

             Replica[i].SinalDeVida();
             Grupo[i][2] = "ATIVO";
          }
          catch (Exception e)
          {
             Grupo[i][2] = "INATIVO";
          }
        }
    }
}
```

O método `SinalDeVida` é obtido remotamente e simplesmente emite uma mensagem local. Na verdade, o sinal de vida esperado é a não ocorrência de uma exceção na invocação do método. Por isso, quem invoca o método `SinalDeVida` deve tratar a ocorrência de uma exceção como uma sinalização de que a réplica esta inativa.

```
public void SinalDeVida()
{
   Janela.t1.append("\n Replica "+NomeDoServidor+"
                        operante ...");
}
```

O cliente também tem acesso remoto ao método `ClienteCriaReplicas`, que é responsável pela criação da réplica. Na realidade, a criação da réplica não significa criar um objeto local para armazenar o estado do objeto crítico do usuário. O objeto local é criado automaticamente, no momento da inicialização da réplica, seja ele um primário ou um *backup*, e a variável de estado `ReplicaCriada` é inicializada com o valor 0 indicando que o objeto local ainda não recebeu nenhuma atualização, ou seja, ainda não é uma réplica do objeto crítico.

Para criar a réplica, o método `ClienteCriaReplicas` utiliza a variável `ReplicaCriada` para determinar o estado atual do objeto local `ReplicaLocal`. Se há a indicação de que ainda não ocorreu atualização do objeto local, a variável `ReplicaCriada` recebe o valor 1, para indicar que a partir deste momento o objeto local já recebeu uma atualização, e a variável `ReplicaLocal`

recebe a atualização de estado representada pelo parâmetro `t`. Ao final, o método retorna ao cliente o estado do objeto local.

Como resultado da execução do método `ClienteCriaReplicas`, o cliente recebe um objeto com estado inicialmente zerado ou um objeto com o estado igual ao estado da réplica local. Esta operação permite que o cliente recupere-se de uma falha.

```
public Object ClienteCriaReplicas(Object t)
{
   if ( ReplicaCriada == 0)
   {
      ReplicaLocal  = t;
      ReplicaCriada = 1;

      Janela.t1.append("\n Foi criada uma réplica ");
   }
   else
   {
      Janela.t1.append("\n Foi encontrada uma replica
                        criada e o cliente foi
                        atualizado... ");
   }

   return ReplicaLocal;
}
```

A aplicação do usuário tem acesso remoto aos métodos `BackupParaPrimario` e `PrimarioParaBackup`, os quais responsáveis por transformar um *backup* em primário e um primário em *backup* respectivamente. O primeiro é utilizado quando ocorre a queda de um primário e é necessário que algum *backup* assuma seu lugar. O segundo é utilizado quando um primário cai e recupera-se sem que o cliente perceba. Nesta situação, ao recuperar-se, o primário não tem mais o estado atual do objeto crítico e por isso deve ser atualizado. A atualização é feita transformando-o em *backup* e escolhendo um dos *backups* para primário. O processo de atualização depende também do método `SelecionaNovoPrimario` responsável pela escolha do novo primário. Este método será detalhado na seção referente a aplicação do usuário.

```
public void BackupParaPrimario()
{
   ReplicaCriada     = 1;
   Tipo              = 1;

   Janela.setTitle("Primario " + NomeDoServidor);
}

public void PrimarioParaBackup()
```

```
{
   Tipo                = 0;

   Janela.setTitle("Backup " + NomeDoServidor);
}
```

O método `AtualizaGrupo` é responsável pela atualização da variável `Grupo`, o número de membros do grupo e o vetor `Replica`. Este método é utilizado em conjunto com o método `ClienteAtualizaReplica` para manter a réplica atualizada em relação ao objeto crítico e a matriz `Grupo`. Desta forma, qualquer réplica *backup* ativa tem, a qualquer momento, a situação atual do grupo e pode, se for necessário, tornar-se um primário. Além disso, cada réplica mantém o vetor de referências `Replicas` para realizar a disseminação da mensagem de confirmação de atualização de estado (*commit*) no método *`run`*.

```
public void AtualizaGrupo(int NumeroDeMembros,
                          String[][] GrupoAtualizado,
                          InterfaceMaquinaReplicacao[]
                          ReplicaDoGrupo) throws
                          RemoteException
{
   Grupo          = GrupoAtualizado;
   NrMembros      = NumeroDeMembros;
   Replicas       = ReplicasDoGrupo
}
```

Os métodos `InformaNrMembros`, `InformaGrupo` e `InformaReplicaCriada` são utilizados remotamente pela aplicação do usuário para obter, do primário, os valores das variáveis de estado `NrMembros`, `Grupo` e `ReplicaCriada` respectivamente. Estas variáveis são utilizadas pela aplicação nas operações de conexão ao primário e escolha de um novo primário.

```
public int InformaNrMembros() throws RemoteException
{
   return NrMembros;
}

public String[][] InformaGrupo() throws
                                 RemoteException
{
   return Grupo;
}

public int InformaReplicaCriada() throws
                                 RemoteException
{
```

```
   return ReplicaCriada;
}
```

Finalmente, os métodos `PrimarioMostraGrupos` e `BackupMostraGrupo` são utilizados para listar o conteúdo da matriz Grupo e sua finalidade básica é facilitar as operações de depuração feitas durante o desenvolvimento da máquina de replicação.

```
public void PrimarioMostraGrupos()
{  int i;

   for ( i = 0 ; i < NrMembros ; i++ )
   {
      if(Grupo[i][1].toUpperCase().equals("PRIMARIO")
                                                    )
      {
         svr.BackupMostraGrupo();
      }
      else
      {
         try
         {
            if(Grupo[i][3].toUpperCase().equals("ATIV
                                                  O"))
            {
               Replica[i].BackupMostraGrupo();
            }
         }
         catch (Exception e)
         {
            Janela.t1.append("\n Erro ao mostra o
                              grupo da replica "
                              + Grupo[i][2]);
         }
      }
   }
}

public void BackupMostraGrupo()
{  int i, j;

   Janela.t2.setText("\n");

   for (i = 0; i < NrMembros; i++)
   {
      Janela.t2.append("Host/Tipo/Id/Status: ");

      for (j = 0; j <= 3; j++)
```

```
            Janela.t2.append(Grupo[i][j] + " / ");

         Janela.t2.append("\n");
      }

      Janela.t2.append("\n Tipo: " + Tipo);
   }
```

É importante salientar que a máquina de replicação utiliza a classe `FrameDoServidor` para definir e implementar sua interface de entrada/saída. Esta interface tem a finalidade acadêmica de permitir uma interação com a máquina de replicação e com isso a sua operação. No entanto, o funcionamento da replicação não depende da interface podendo ela ser, até mesmo, suprimida.

A implementação completa da classe `FrameDoServidor` pode ser encontrada no ANEXO 2.

6.2.4 Implementação da Classe de Conexão

A classe `Conexao` compõe o pacote `Conexao` que deve ser importado pela aplicação do usuário. O objetivo desta classe é fazer a ligação entre a aplicação do usuário e a máquina de replicação.

Desta forma, o usuário não precisa programar na sua aplicação nenhum comando referente ao RMI, isso é feito na classe `Conexao`, sendo necessário somente que ele crie uma objeto desta classe.

Os parágrafos a seguir detalham o funcionamento dos principais componentes desta classe; sua implementação completa pode ser encontrada no ANEXO 3.

O método construtor da classe tem a tarefa fundamental de conectar a aplicação do usuário à máquina de replicação que atua como primário. Para isso, ele recebe como entrada um argumento da linha de comando de onde será obtido o nome da referência remota do primário. Um componente do tipo TextArea é utilizado meramente para efeitos de interface entrada/saída:

```
public Conexao(String Argumento, TextArea Resultado)
                throws Exception
```

Como em qualquer conexão RMI, o primeiro passo é criar um gerente de segurança:

```
if (System.getSecurityManager() == null)
    System.setSecurityManager ( new
    RMISecurityManager() );
```

Em seguida, o parâmetro `Argumento` é utilizado para formar um *string* que identifica a referência remota desejada. Para isso, um laço *`while`* percorre o *string* `Argumento` separando o nome do hospedeiro do primário e seu identificador, que é opcional, nas variáveis *`host`* e `id` respectivamente. Os detalhes da inicialização da aplicação cliente são apresentados na seção 6.3.

```
// define o nome da referência remota do primario a
   partir da linha de comando

   while ( i < Argumento.length() )
   {

      if("=".equals(String.valueOf(Argumento.charAt(i
                                                  ))))
      {
         if ( i + 4 <= Argumento.length() )
         {
           if("i".toLowerCase().equals(String.valueOf
                  (Argumento.charAt(i+1))) &&
              "d".toLowerCase().equals(String.valueOf
                  (Argumento.charAt(i+2))) )
           {
              for (j=i+4 ;j<Argumento.length() ;j++ )
                 id = id + Argumento.charAt(j);

              break;
           }
         }
      }
      else
      {
         host = host + Argumento.charAt(i);
      }

      i++;
}
```

Após a obtenção do valor das variáveis *`host`* e `id`, a variável `service` é inicializada com esta referência:

```
service = (InterfaceMaquinaReplicacao) Naming.lookup
          ("rmi://" + host.toLowerCase()+"/"+
           host.toLowerCase()+id);
```

A partir deste momento, já existe uma conexão estabelecida com o primário e sua referência é a variável `service`. Para permitir que a aplicação cliente

também conheça todo o grupo de réplicas, são executados dois métodos remotos no primário:

```
NrMembros     = service.InformaNrMembros();
Grupo         = service.InformaGrupo();
```

O primeiro obtém do primário o número de membros do grupo e o segundo obtém o grupo propriamente dito. É importante lembrar que o primário tem estas informações deste o momento em que é inicializado.

Neste ponto, a conexão entre a aplicação cliente e o primário está estabelecida e a aplicação conhece todo o grupo de réplicas. A replicação está pronta para funcionar.

No entanto, para garantir a consistência do primário, o método construtor faz um teste que começa com a execução do método remoto `InformaReplicaCriada` no primário:

```
ReplicaCriada = service.InformaReplicaCriada();
```

Este método retorna o valor 0, se o primário ainda não recebeu atualizações de estado, ou o valor 1, se o primário já recebeu atualizações de estado. Nestes dois casos a consistência esta correta. Porém, se o valor retornado for 2, significa que o primário está desatualizado, ou seja, está inconsistente. Como já foi descrito na seção de descrição da classe `MaquinaReplicacao`, quando o primário é inicializado, ele compara o seu estado com o estado das outras réplicas para determinar a sua consistência.

Se for detectada a inconsistência, o construtor transforma o primário inconsistente em *backup* e procura um novo primário entre as réplicas consistentes.

```
if ( ReplicaCriada == 2 )
{
   for ( i = NrMembros-1 ; i >= 0 ; i-- )
   {
     if(Grupo[i][1].toUpperCase().equals("PRIMARIO"))
     {
        Grupo[i][1] = "BACKUP";
        Grupo[i][2] = "ATIVO";

        service.PrimarioParaBackup();
     }
     else
     {
        if(Grupo[i][2].toUpperCase().equals("ATIVO")
             && !Achou)
        {
```

```
                try
                {
                  service = (InterfaceMaquinaReplicacao)
                            Naming.lookup
                            ("rmi://"+Grupo[i][0]);

                  if (service.InformaReplicaCriada()==1)
                  {

                     Grupo[i][1]   = "PRIMARIO";
                     Achou         = true;

                     Resultado.append("\nO primario
                                       especificado esta
                                       desatualizado. Novo
                                       primario: " +
                                       Grupo[i][2]);
                  }
                }
                catch (Exception e)
                {
                   Grupo[i][2]   = "INATIVO";
                }
            }
          }
        }

        // a atualizacao é feita aqui porque o grupo
           estará atualizado somente após a execução
           completa do for acima

        try
        {
          service.BackupParaPrimario();
          service.AtualizaGrupo(NrMembros,Grupo,Replicas);
          service.PrimarioConectaBackups();
        }
        catch (Exception e)
        {
          Resultado.append("\nErro na busca de novo
                            primario ...");
        }
    }
```

É importante observar que o laço `for`, descrito acima, somente configura a matriz `Grupo`. O novo primário é representado na matriz mas não é avisado para comportar-se como tal. Para isso, são executados, no novo primário, os métodos remotos: `BackupParaPrimário`, `AtualizaGrupo` e `PrimarioConectaBackups`. Estes métodos são responsáveis, respectivamente, por

mudar o comportamento do antigo *backup* para o comportamento de primário, atualizar a visão que o novo primário tem das réplicas e conectar o novo primário às réplicas. Como já foi descrito na seção de descrição da classe `MaquinaReplicacao`, quando o primário conecta-se aos *backups* ele atualiza a visão de todos eles.

Um outro método da classe `Conexao`, o método `Atualizar`, é responsável pelo acesso remoto ao método `ClienteAtualizaReplica` no primário. O método `ClienteAtualizaReplica`, por sua vez, dá início ao mecanismo de *commitment* atômico. No momento da atualização dos *backups*, é possível que o primário detecte alguma falha e reconfigure a matriz `Grupo`. Por isso, o método `Atualizar` também executa remotamente o métodos `InformaNrMembros` e `InformaGrupo` para atualizar a visão que a aplicação cliente tem do grupo.

```
public void Atualizar(Object t) throws Exception
{
   Ok = false;

   While ( ! Ok )
   {
     try
     {
        service.ClienteAtualizaReplica(t);

        NrMembros   = service.InformaNrMembros();
        Grupo       = service.InformaGrupo();

        Ok = true;
     }
     catch (Exception e)
     {
        ProcuraNovoPrimario(t,t1);
     }
   }
};
```

Se o primário falhar, o método `Atualizar` executa o método `ProcuraNovoPrimario` que é responsável por determinar e conectar um novo primário. Se houver uma falha também do novo primário, o laço `while` utiliza a variável `Ok` para buscar outro até que não haja mais candidatos.

O método `ProcuraNovoPrimario` é utilizado pela aplicação cliente para escolher um novo primário quando o atual falha. Seu funcionamento é análogo ao descrito no método construtor quando o primário está desatualizado, com a diferença de que ele manipula a variável `Ok` para determinar a ocorrência de falhas no novo primário.

Por fim, o método `InformaGrupo` é utilizado somente para facilitar a tarefa de depuração da aplicação. Seu objetivo é informar o conteúdo da matriz `Grupo` na aplicação cliente.

6.2.5 Implementação da Aplicação Cliente

A aplicação cliente utilizada neste trabalho é bastante simples. Trata-se de uma aplicação que acumula o valor do saldo de uma conta bancária através de operações de saque e depósito. A implementação completa desta classe pode ser encontrada no ANEXO 4.

A aplicação importa a classe `Conexao` através da qual realiza a conexão ao primário e à classe `InterfaceMaquinaReplicacao` por ser a interface do primário. Além disso, ela é derivada da classe `Frame` do Java e seu método construtor simplesmente define e cria a interface entrada/saída utilizada.

Para o funcionamento da replicação, a aplicação precisa de um objeto da classe `Conexao` para fazer referência ao primário e um objeto da classe `Operacoes` onde é definida a classe do objeto crítico:

```
public class ContaBancaria extends Frame
{
   static Conexao Primario;
   static Operacoes ObjetoCritico;

   public Panel p1, p2, p3;
   public TextField tf1;
   public Button b1,b2,b3,b4;
   public MenuBar MenuPrincipal;
   public Menu Opcoes;

   static TextArea  t1;

   .
   .
   .
}
```

As demais variáveis de estado são utilizadas pela interface entrada/saída. A classe `Operacoes` será detalhada mais tarde, por enquanto basta entender que ela é a classe crítica da qual um objeto será replicado.

O método `action` da aplicação é responsável pelo tratamento dos eventos associados aos componentes da interface entrada/saída, tais como: clique em botões ou em itens da barra de menu. O tratamento de um evento que implica em alteração no estado do objeto crítico significa executar a operação associada ao evento e repassar o objeto atualizado ao primário para que este tome as devidas providências.

Por exemplo, o evento associado ao clique no botão Depósito executa inicialmente o método `Deposito` do objeto crítico. Este método recebe o valor digitado no campo `tf1` e o soma ao valor do saldo:

```
ObjetoCritico.Deposito(Integer.parseInt(tf1.getText()
                                                    ));
```

Em seguida, como o método `Deposito` produz uma alteração no estado do objeto crítico, o primário é atualizado:

```
try
{
  Primario.Atualizar(ObjetoCritico,t1);
}
catch (Exception e)
{
  t1.append("\nErro na atualização do primário ");
}
```

O parâmetro `t1` é um objeto da classe TextArea e tem a função de ajudar a visualização do resultado da operação. Ele não tem nenhuma interferência no funcionamento da máquina de replicação. Assim, o saldo atualizado da conta é informado através do objeto `t1`.

```
t1.append("\nSaldo: " + MeuObjeto.Local);
```

O tratamento de um evento que não implica na alteração do estado do objeto crítico significa simplesmente agir sobre os métodos e variáveis do próprio objeto, não sendo necessário interagir com o primário.

Por exemplo, o evento associado ao clique no botão Saldo simplesmente permite o acesso ao valor do saldo do objeto crítico, armazenado na própria aplicação, e o informa através do objeto `t1`:

```
t1.append("\nSaldo: " + MeuObjeto.Local);
```

O método `main` da aplicação cliente é bastante simples, sendo composto basicamente de comandos de criação de objetos:

```
public static void main(String args[]) throws
                                        Exception
{
   new ContaBancaria();

   Primario      = new Conexao(args[0],t1);
   ObjetoCritico = new Operacoes();
```

```
   ObjetoCritico = (Operacoes)
                   Primario.service.ClienteCriaRepli
                   cas(ObjetoCritico);
}
```

Inicialmente, é criada uma instância da própria classe `ContaBancaria` que define a interface entrada/saída e seu comportamento. Na seqüência, é criada uma instância da classe `Conexao` que estabelece a ligação ao primário. Por fim, é criada uma instância da classe crítica `Operacoes`. Após criado o objeto crítico e o objeto de conexão, a aplicação executa no primário o método remoto `ClienteCriaReplicas`.

6.2.6 Implementação da Classe Crítica

A definição da classe crítica do usuário é feita de tal forma a implementar um mecanismo que permita que um objeto qualquer seja submetido ao servidor. Este mecanismo é composto por uma interface suportada pelo servidor e pelos objetos que são submetidos ao servidor.

A interface `Tarefa` define as operações da classe crítica do usuário. Como já foi descrito nas seções anteriores, neste exemplo, a classe crítica tem duas operações simples: uma operação de depósito de valores e uma operação de saque de valores. O valor da operação é passado como parâmetro:

```
package InterfaceMaquina;

import java.io.Serializable;

public interface Tarefa extends Serializable
{
   public Integer Deposito (int numero);
   public Integer Saque (int numero);
}
```

O ponto fundamental desta interface é que ele é derivada da interface `java.io.Serializable`. Esta interface é utilizada pelo RMI para transportar objetos por valor entre máquinas virtuais Java. Isto permite que os objetos de uma classe possam ser convertidos em um *array* de *bytes* que podem ser usados para reconstruir uma cópia exata do objeto serializado quando ele é lido a partir do *array*. Esta característica permite que a máquina de replicação faça a atualização do estado das réplicas através do envio de cópias do objetos da classe crítica.

A implementação da interface `Tarefa` é feita pela classe `Operacoes` e é detalhada a seguir:

```
package Conta;
```

```
import InterfaceMaquina.*;

public class Operacoes implements Tarefa
{
   int Saldo = 0;

   public Operacoes()
   {
      super();
   }

   public Integer Deposito (int numero)
   {
      Saldo = Saldo + numero;
      String literal = String.valueOf(Saldo);
      Integer resultado = new Integer(Saldo);

      return (resultado);
   }

   public Integer Saque (int numero)
   {
      Saldo = Saldo - numero;
      String literal = String.valueOf(Saldo);
      Integer resultado = new Integer(Saldo);

      return (resultado);
   }
}
```

Pode-se perceber que a classe `Operacoes` é extremamente simples e o único fator de complexidade implícita é o fato dela implementar a interface `Tarefa` que é serializável.

Utilizando-se o mecanismo descrito acima, pode-se definir qualquer classe crítica e os objetos instanciados dela podem ser replicados pela máquina de replicação.

6.3 Execução da Máquina de Replicação

Inicialmente, é preciso considerar a versão de Java onde será executada a máquina de replicação. A partir do JDK 1.2, é necessário especificar um arquivo de configuração para o gerente de segurança da máquina de replicação e da aplicação cliente. A listagem a seguir representa uma configuração genérica:

```
grant
{
```

```
    permission java.net.SocketPermission "*:1024-
                65535","connect,accept,resolve";

    permission java.net.SocketPermission "*:1-
                1023","connect,resolve";
};
```

Este exemplo assume que o arquivo de configuração chama-se `java.policy` e encontra-se na raiz do drive `C:` . Em versões anteriores, este arquivo não era necessário porque o `RMISecurityManager` continha internamente todas as configurações necessárias.

6.3.1 Inicialização da Máquina de Replicação

Para formar uma base de replicação, deve-se executar os *backups*, o primário e por último a aplicação cliente. Este exemplo assume que *backups*, primário e aplicação cliente estão localizados em hospedeiros diferentes que compartilham a mesma rede Windows98.

A forma geral de inicialização da máquina de replicação do tipo *backup* é dada por uma linha de comando que deve ter o seguinte formato:

```
java -Djava.security.policy = c:\java.policy
      Maquina.MaquinaReplicacao
      [tipo da réplica]
      [host da réplica]<=id:id da réplica>
```

Os sinais [ ] representam argumentos obrigatórios e os sinais <> argumentos opcionais. A especificação de um **id** para a réplica só é necessária quando há mais de uma réplica no mesmo hospedeiro. Neste caso, o identificador compõe o nome e diferencia uma da outra para o *rmiregistry*.

Assim, para inicializar uma máquina de replicação do tipo *backup* em um hospedeiro chamado `lab0` temos:

```
java -Djava.security.policy = c:\java.policy
      Maquina.MaquinaReplicacao
      backup
      lab0=id:0
```

O resultado da execução da linha de comando acima é mostrado na FIGURA 23.

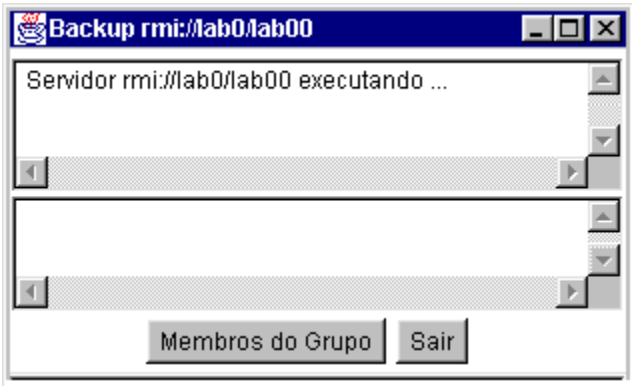

FIGURA 23 - *Backup* no *host* `lab0`

Da mesma forma, para inicializar outra máquina de replicação do tipo *backup* em um hospedeiro chamado `lab1` tem-se:

```
java -Djava.security.policy = c:\java.policy
      Maquina.Replicacao
      backup
      lab1=id:1
```

O resultado da execução da linha de comando acima é mostrado na FIGURA 24.

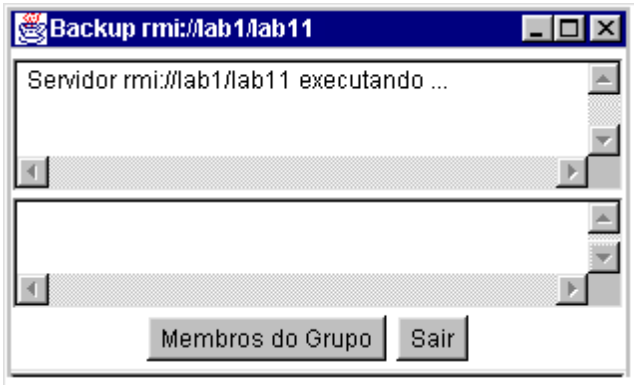

FIGURA 24 - *Backup* no *host* `lab1`

A forma geral de inicialização da máquina de replicação do tipo primário é dada por uma linha de comando que deve ter o seguinte formato:

```
java -Djava.security.policy = c:\java.policy
      Maquina.MaquinaReplicacao
      [tipo da réplica]
      [host da réplica] <=id:id da réplica>
      [host do backup 1]<=id:id do backup 1>
      ...
      [host do backup n]<=id:id do backup n>
```

Assim, para inicializar uma máquina de replicação do tipo primário em um hospedeiro chamado `lab2` a linha de comando deve ser como mostrado abaixo:

```
java -Djava.security.policy = c:\java.policy
      Maquina.MaquinaReplicacao
      primario
      lab2=id:2 lab0=id:0 lab1=id:1
```

O resultado da execução da linha de comando acima é mostrado na FIGURA 25.

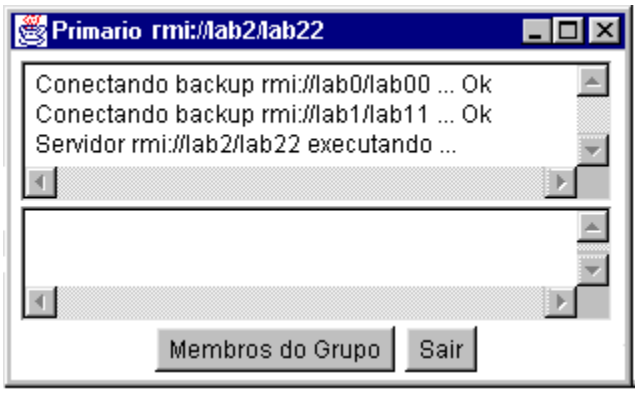

FIGURA 25 - Primário no *host* `lab2`

Em relação à aplicação cliente, a forma geral de inicialização é dada por uma linha de comando que deve ter o seguinte formato:

```
java -Djava.security.policy = c:\java.policy
      Conta.ContaBancaria
      [host do primário] <=id:id do primário>
```

Assim, no exemplo deste trabalho, para iniciar a aplicação cliente tem-se:

```
java -Djava.security.policy = c:\java.policy
      Cliente.ClienteServidor
      lab2=id:2
```

A interface da aplicação cliente é mostrada na FIGURA 26.

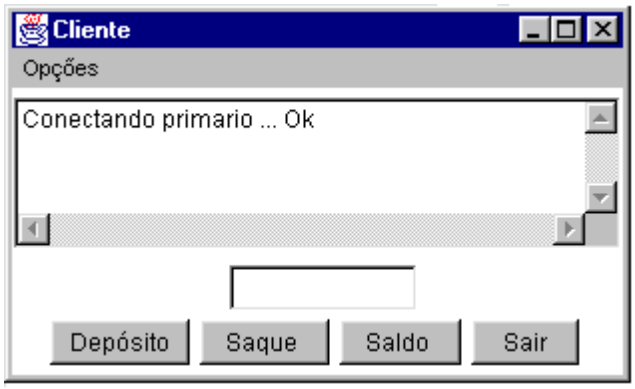

FIGURA 26 - Cliente

A FIGURA 27 mostra a alteração no primário devido a conexão do cliente.

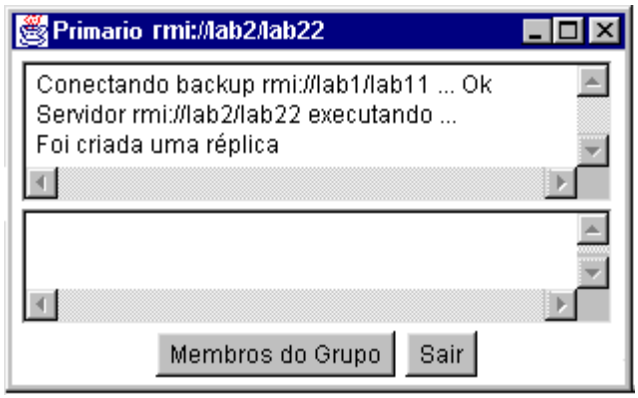

FIGURA 27 - Primário após conexão do cliente

Uma vez conectada ao primário, a aplicação cliente pode executar suas operações normalmente como se não houvesse a replicação. A FIGURA 28 mostra o resultado da execução de uma operação de depósito.

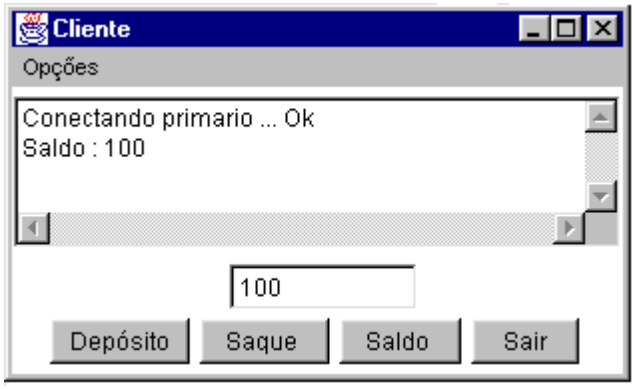

FIGURA 28 - Cliente após operação de depósito

A operação de depósito produz alterações também no primário, como é mostrado na FIGURA 29.

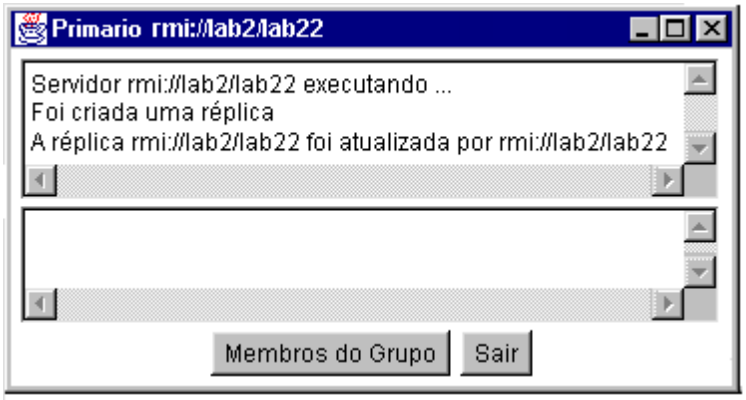

FIGURA 29 - Primário após operação de depósito

As FIGURA 30 e 31 mostram a alteração produzida na interface dos *backups*.

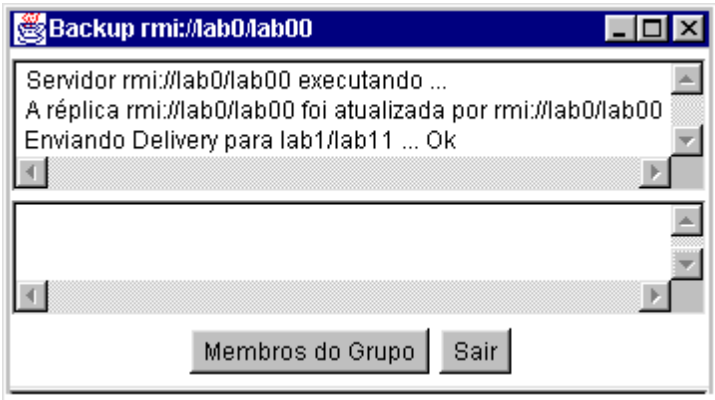

FIGURA 30 - *Backup* do hospedeiro `lab0` após operação de depósito

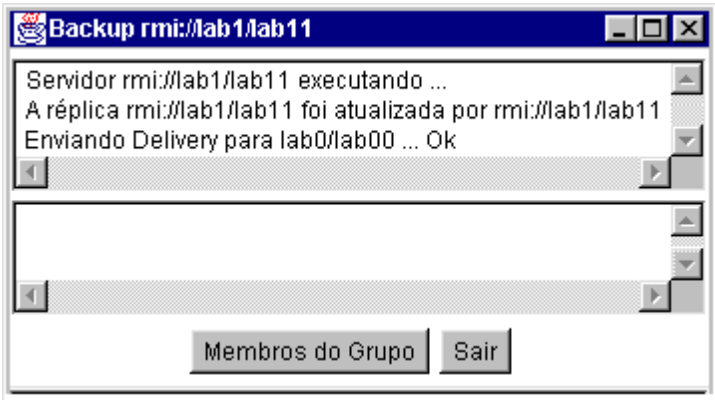

FIGURA 31 - *Backup* do hospedeiro `lab1` após operação de depósito

7 Trabalhos Relacionados

Muitas pesquisas têm desenvolvido plataformas para a construção de aplicações tolerantes a falhas e de alta disponibilidade. Pode-se destacar dois enfoques para estas pesquisas: RMI e CORBA. Independentemente da escolha, é desejável esconder a replicação tanto quanto possível do programador da aplicação.

Em relação ao RMI, pode-se citar o Jgroup que é uma extensão do modelo de objetos distribuídos do Java baseados em comunicação de grupo [MON99]. O Jgroup é particularmente apropriado para o desenvolvimento de aplicações dependentes de rede em ambientes sujeitos a partições voluntárias ou involuntárias. Ele adapta a semântica da visão síncrona, tipicamente definida para sistemas de comunicação com mensagens de grupo, para a invocação remota de métodos de objetos.

O Jgroup apresenta uma interface de programação uniforme baseada inteiramente na invocação remota de métodos. Além disso, ele estende a semântica da visão síncrona para controlar interações entre o grupo e clientes externos ao grupo sem precisar que estes clientes tornem-se membros do grupo. Desta maneira, o alto custo associado com as fortes garantias sobre as invocações remotas são limitados aos membros do grupo enquanto as garantias aos clientes podem ser mais "leves" e assim diminuir o custo.

É importante ressaltar que o Jgroup não é um serviço de replicação, mas sim, um serviço de comunicação de grupo particionável. Seu objetivo é oferecer suporte ao desenvolvimento de aplicações confiáveis em sistemas distribuídos particionáveis.

Quanto ao CORBA, pode-se citar o Electra e o Orbix+Isis, desenvolvidos sobre as plataformas Horus e Isis respectivamente, e que oferecem suporte a objetos CORBA replicados [MON99]. Ainda em relação ao CORBA porém, apoiado na plataforma Totem há o sistema Eternal. Diferentemente do Eternal, os sistemas Electra e Orbix+Isis são não-hierárquicos, suportam somente a replicação ativa e usam o enfoque de integração onde os mecanismos de replicação e comunicação de grupo são integrados ao ORB e requerem modificações no mesmo [MOS98].

O sistema Eternal utiliza uma combinação das técnicas de replicação ativa, primário-*backup* e votação. Além disso, usa o enfoque da interceptação onde não há necessidade de modificações no ORB mas há dependência de mecanismos de baixo nível do sistema operacional.

O enfoque da integração é bem aceito devido à sua transparência, uma vez que os clientes não precisam saber se o serviço requerido por eles é oferecido por um único objeto ou por um grupo. Porém, os ORBs resultantes não são compatíveis com o CORBA [MON99]. Outra alternativa é o enfoque de serviço onde a comunicação de grupo é oferecida como um serviço sobre o ORB. Um exemplo de ferramenta baseada neste enfoque é o Object Group Service (Felber, Guerraoui e Schiper, apud [MON99]). Neste caso não há incompatibilidade entre os ORBs e o CORBA porém, os clientes não podem acessar transparentemente um grupo de objetos como se eles fossem uma única entidade, eles devem utilizar as primitivas oferecidas pelo OGS.

Neste ponto é importante salientar que o resultado deste trabalho, assim como o Jgroup, é baseado totalmente no Java RMI e por isso não sofrem de nenhum dos problemas dos enfoques descritos acima.

Outros dois trabalhos que utilizam o RMI são o iBus (Maffeis, apud [MON99]) e o Filterfresh (Baratloo, apud [MON99]). O iBus é um produto comercial escrito em Java que tem como objetivo oferecer suporte a aplicações de intranet, *groupware* e sistemas cliente/servidor tolerantes a falhas. O Filterfresh compartilha os mesmos objetivos do Jgroup, isto é, integração do paradigma de comunicação de grupo com o modelo de objetos distribuídos de Java. Uma característica importante e comum a estes dois sistemas é a implementação de um serviço de registro distribuído. A grande diferença entre os dois é que o Jgroup é baseado em um serviço de comunicação de grupo particionável enquanto o Filterfresh é baseado em um sistema de partição primária.

Há ainda outros sistemas de objetos distribuídos que oferecem suporte a tolerância a falhas ou alta disponibilidade e não são baseados em CORBA ou RMI [MOS98].

O sistema Arjuna (Parrington e Shrivastava, apud [MOS98]) usa a replicação de objetos com uma estratégia de transações atômicas para oferecer tolerância a falhas. Os tipos de replicação suportados incluem a replicação ativa, primário-*backup* com *coordinator-cohort* e primário-*backup* com cópia única. Uma estratégia similar a *checkpointing* é utilizada para a atualização de estados na replicação por primário-*backup*.

Os sistemas PHOENIX e GARF (Felber e Garbinato, apud [MOS98]) oferecem suporte à replicação ativa e permitem operações aninhadas com replicação para o objeto cliente e o objeto servidor.

8 Conclusões

Este trabalho obteve como resultado principal, a implementação de uma biblioteca de classes de replicação na linguagem Java. Estas classes permitem que um programador utilize a replicação de forma transparente em suas aplicações.

A utilização da arquitetura Java RMI ajudou muito porque simplificou o trabalho de programação através de sua abstração. Além disso, a utilização das bibliotecas de serialização permitiu, de forma totalmente transparente, a transmissão de objetos pela rede.

Como resultado, temos uma classe de replicação simples e funcional que pode replicar qualquer tipo de objeto, visto que as réplicas locais foram implementadas a partir de um objeto genérico da classe `Object` do Java. Este trabalho não utilizou nenhuma ferramenta de comunicação de grupo, optando por, ele mesmo, fazer um controle dos membros do grupo. Apesar da implementação de um ferramenta de comunicação de grupo não ser o objetivo do trabalho, esta característica diminui muito a complexidade do sistema para o usuário final, visto que, ele não precisa interagir com nenhuma tipo de ferramenta adicional para utilizar a máquina de replicação.

A consistência das réplicas é garantida por dois mecanismos. O primeiro exclui do grupo qualquer réplica que não responde à uma invocação e tenta recuperá-la mais tarde. Desta forma, o grupo é sempre composto somente por réplicas em funcionamento. O segundo mecanismo utiliza um protocolo de *commitment* atômico para garantir a primitiva de atomicidade na transmissão das mensagens. Assim, o estado do objeto replicado é mantido a salvo mesmo na presença de falhas no primário ou nos *backups*.

Um ponto muito importante na simplicidade da máquina de replicação é a forma de sua utilização pelo usuário. O que ele deve fazer é nada mais que importar algumas classes para o seu código e implementar a sua classe crítica na forma de uma interface. Utilizar a máquina de replicação significa simplesmente acessar um de seus métodos. Se considerarmos a utilização de uma ferramenta de reflexão sobre o código do usuário, a chamada à máquina de replicação pode ser incluída no código da aplicação do usuário de forma totalmente transparente. Para ele é como se estivesse programando uma aplicação comum sem nenhum detalhe que lembre a replicação.

Um trabalho futuro, que teria grande valia para outros trabalhos de tolerância a falhas, é a implementação de uma classe especializada em comunicação de grupos. A máquina de replicação utiliza seu próprio mecanismo para fazer esta tarefa, contudo, este mecanismo poderia ser melhorado com o uso de uma biblioteca mais aprimorada. Já que a máquina de replicação pretende tirar do usuário o trabalho de implementar a replicação, então uma biblioteca especializada poderia tirar da máquina de replicação o trabalho de implementar a comunicação.

Além disso, um melhoramento importante seria a extensão da máquina de replicação para que ela replique mais de um objeto ao mesmo tempo e até mais de um objeto, de classes diferentes, ao mesmo tempo. Esta extensão não implica em modificar o esquema de replicação já implementado, mas sim, em acrescentar a máquina de replicação uma estrutura de dados capaz de armazenar vários objetos e um esquema de identificação entre os objetos armazenados.

Anexo 1 Implementação da Classe MaquinaReplicacao

```
package Maquina;
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import java.util.Date;
import InterfaceMaquina.*;

public class MaquinaReplicacao
             extends UnicastRemoteObject
             implements InterfaceMaquinaReplicacao,
                        Runnable
{
   static FrameMaquina Janela;
   Thread myThread;

   Object ReplicaLocal          = new Object();
   Object ReplicaTemp           = new Object();

   static String NomeDoServidor = new String();
   static MaquinaReplicacao svr;

   public int ReplicaCriada=0, Tipo=0, NrMembros=0
   static int Decisao=0;

   static InterfaceMaquinaReplicacao Replicas[] =
          new InterfaceMaquinaReplicacao[10];

   static String[][] Grupo = new String[10][3];

   public MaquinaReplicacao () throws
                               RemoteException
   {
      super();
      myThread = new Thread(this);
   }

   public int InformaNrMembros() throws
                                 RemoteException
   {
      return NrMembros;
   }

   public String[][] InformaGrupo() throws
                                    RemoteException
   {
      return Grupo;
   }
```

```
public int InformaReplicaCriada() throws
                                    RemoteException
{
   return ReplicaCriada;
}

public void AtualizaGrupo(
    int NumeroDeMembros,
    String[][] GrupoAtualizado,
    InterfaceMaquinaReplicacao[] ReplicasDoGrupo)
    throws RemoteException
{
   Grupo         = GrupoAtualizado;
   NrMembros     = NumeroDeMembros;
   Replicas      = ReplicasDoGrupo;
}

public void ValidaGrupo()
{ int i;
  for ( i=0 ; i < NrMembros ; i++ )
  {
   if(Grupo[i][1].toUpperCase().equals("BACKUP"))
   {
     try
     {
       if(Grupo[i][2].toUpperCase().equals("INATI
                                           VO"))
       { Replicas[i]=(InterfaceMaquinaReplicacao)
                     Naming.lookup ("rmi://" +
                     Grupo[i][0]); }

         Replicas[i].SinalDeVida();
         Grupo[i][2] = "ATIVO";
       }
     catch (Exception e)
     {
       Grupo[i][2] = "INATIVO";
     }
   }
  }
}

public void CriaGrupo(int TotalDeMembros,
                      String[] Argumentos)
{  int i = 2 , j = 0;

   // cria lista de membros

   for ( j=0 ; j < (TotalDeMembros - 1); j++ )
```

```
    {
      Grupo[j][0] = svr.Parse(Argumentos[i]);
      Grupo[j][1] = "BACKUP";
      Grupo[j][2] = "ATIVO";
      i++;
    }
    Grupo[j][0]    = svr.Parse(Argumentos[1]);
    Grupo[j][1]    = "PRIMARIO";
    Grupo[j][2]    = "ATIVO";
}

public void PrimarioMostraGrupos()
{  int i;

    for ( i = 0 ; i < NrMembros ; i++ )
    {
      if(Grupo[i][1].toUpperCase().equals("PRIMARI
                                            O"))
      {
        svr.BackupMostraGrupo();
      }
      else
      {  try
         {
            if(Grupo[i][2].toUpperCase().equals("A
                                              TIVO"))
            Replicas[i].BackupMostraGrupo();

         }
         catch (Exception e)
         {
            Janela.t1.append("\n Erro ao mostra o
                             grupo da replica " +
                             Grupo[i][0]);
         }
      }
    }
}

public void BackupMostraGrupo()
{  int i, j;

    Janela.t2.setText("\n");

    for (i = 0; i < NrMembros; i++)
    {
      Janela.t2.append("Host/Tipo/Id/Status: ");

      for (j = 0; j <= 2; j++)
        Janela.t2.append(Grupo[i][j] + " / ");
```

```
      Janela.t2.append("\n");
    }

    Janela.t2.append("\n Tipo: " + Tipo);
}


public void BackupParaPrimario()
{
    ReplicaCriada     = 1;
    Tipo              = 1;

    Janela.setTitle("Primario " + NomeDoServidor);
}

public void PrimarioParaBackup()
{
    Tipo              = 0;

    Janela.setTitle("Backup " + NomeDoServidor);
}

public Object ClienteCriaReplicas(Object t)
{
    if ( ReplicaCriada == 0)
    {
       ReplicaLocal  = t;
       ReplicaCriada = 1;

       Janela.t1.append("\n Foi criada uma réplica
                                              ");
    }
    else
    {
       Janela.t1.append("\n Foi encontrada uma
                        réplica criada e o cliente
                        foi atualizado ");
    }

    return ReplicaLocal;
}

public int PrimarioConectaBackups()
{  int i;

    for ( i=0 ; i < NrMembros ; i++ )
    {
      try
      {  // só conecta aos backups ativos
```

```
            if(Grupo[i][1].toUpperCase().equals("BACK
                                              UP") &&
               Grupo[i][2].toUpperCase().equals("ATIV
                                              O") )

            {
               Janela.t1.append("\n Conectando backup
                              rmi://"+Grupo[i][0] +
                              " ... ");

               Replicas[i]=(InterfaceMaquinaReplicaca
                           o) Naming.lookup
                           ("rmi://"+Grupo[i][0]);

               Replicas[i].AtualizaGrupo(NrMembros,
                                         Grupo,
                                         Replicas);

               Janela.t1.append(" Ok ");

               // se o primario não tem replica
                 criada ele verifica se algum dos
                 backups tem

               if ( ReplicaCriada == 0 )
               {
                 if(Replicas[i].InformaReplicaCriada(
                                           ) == 1 )
                 {
                    ReplicaCriada    = 2;
                 }
               }
            }
         }
         catch (Exception e)
         {
            Janela.t1.append("\n Erro na conexão da
                             replica " + Grupo[i][0]
                             + e.toString());
         }
      }

      return ReplicaCriada;
   }


   public void ClienteAtualizaReplica(Object t)
   {  int i;

      if (svr.Tipo == 1)
```

```
   {  // envia inicio de transação para todos os
         participantes

      ValidaGrupo();

      for ( i=0 ; i < NrMembros ; i++ )
      {
        try
        {
          if(Grupo[i][1].toUpperCase().equals("BA
                                        CKUP") )
          {

            if(Grupo[i][2].toUpperCase().equals("
                                        ATIVO") )

            {
               Replicas[i].ClienteAtualizaReplica
                                             (t)
            }
          }
        }
        catch (Exception e)
        {
          Grupo[i][2] = "INATIVO";
          Janela.t1.append("\n Replica " +
                           Grupo[i][0] + " esta
                           inoperante e não
                           iniciou a transação
                           ...");
        }
      }
   }

   // todos os participantes executam
   ReplicaCriada = 1;
   ReplicaTemp   = t;

   Decisao       = 0;

   ReplicaCommit();
}

public void ReplicaCommit()
{
   if (svr.Tipo == 1)
   {
     // executado somente pelo primário
     BroadCast();
   }
```

```
   else
   {
     myThread = new Thread(this);
     myThread.start();
   }
}

public void BroadCast()
{  int i;

   for ( i=0 ; i < NrMembros ; i++ )
   {
     try
     {
       if(Grupo[i][1].toUpperCase().equals("BACKU
                                            P") )
       {
         if(Grupo[i][2].toUpperCase().equals("ATI
                                           VO") )
         {
           Replicas[i].AtualizaGrupo(NrMembros,
                                     Grupo,
                                     Replicas);

           Replicas[i].AtualizaDecisao();
         }
         else
         {
           Janela.t1.append("\n Replica " +
                            Grupo[i][0] + " esta
                            inoperante 1...");
         }
       }
     }
     catch (Exception e)
     {
       Grupo[i][2] = "INATIVO";
       Janela.t1.append("\n A replica " +
                        Grupo[i][0] + " esta
                        inoperante ...");
     }
   }

   ReplicaDeliver(NomeDoServidor);
}

public void SinalDeVida()
{
```

```
   Janela.t1.append("\n\n A replica
                    "+NomeDoServidor+" esta
                    operante ...");
}

String Parse(String Argumento)
{  int i = 0, j = 0;
   String host="", id="";

   while ( i < Argumento.length() )
   {
     if("=".equals(String.valueOf(Argumento.charA
                                          t(i))) )
     {
       if ( i + 4 <= Argumento.length() )
       {
         if("i".toLowerCase().equals(String.value
                    Of(Argumento.charAt(i+1))) &&
            "d".toLowerCase().equals(String.value
                    Of(Argumento.charAt(i+2))) )
         {
           for(j=i+4 ;j<Argumento.length() ; j++)
           {
             id = id + Argumento.charAt(j);
           }

           break;
         }
       }
     }
     else
     {
       host = host + Argumento.charAt(i);
     }

     i++;
   }

   return host.toLowerCase() + "/" +
          host.toLowerCase()+id.toLowerCase();
}

public void run()
{  int i, TempoDecorrido = 0;

   while ( Decisao == 0 && TempoDecorrido < 30 )
   {
     try
     {
       Thread.sleep(1000);
```

```
  }
  catch (InterruptedException e)
  { }

  TempoDecorrido++;
}

if ( Decisao == 1 )
{
  ReplicaDeliver(NomeDoServidor);

  for ( i=0 ; i < NrMembros ; i++ )
  {
    try
    {
      if(Grupo[i][1].toUpperCase().equals("BAC
                                        KUP") )
      {
        if(Grupo[i][2].toUpperCase().equals("A
                                      TIVO") )
        {
          if(!NomeDoServidor.equals("rmi://"+
              Grupo[i][0].toLowerCase() ) )
          {
             Janela.t1.append("\n Enviando
                              Delivery para "
                              + Grupo[i][0] +
                              " ... ");

             Replicas[i].AtualizaGrupo(
                                   NrMembros,
                                   Grupo,
                                   Replicas);

             Replicas[i].AtualizaDecisao();

             Janela.t1.append(" Ok ");
          }
        }
      }
    }
    catch (Exception e)
    {
      Grupo[i][2] = "INATIVO";
      Janela.t1.append("\n Replica " +
                       Grupo[i][0] + " esta
                       inoperante ...");
    }
  }
}
```

```
   else
   {
     Janela.t1.append("\n Ocorreu um time out e a
                      replica " + NomeDoServidor
                      + " abortou a operacao
                      ...");
   }
}

public void ReplicaDeliver(String
                           OrigemDaMensagem)
{
  ReplicaLocal = ReplicaTemp;

  Janela.t1.append("\n A réplica " +
                   NomeDoServidor + " foi
                   atualizada por "  +
                   OrigemDaMensagem);
}

public void AtualizaDecisao()
{
  Decisao = 1;
}

public static void main ( String args[] )
                          throws Exception
{
   // cria um security manager
   if (System.getSecurityManager() == null)
       System.setSecurityManager ( new
       RMISecurityManager() );

   // cria uma instancia da classe
      MaquinaReplicacao

   svr = new MaquinaReplicacao();

   // escreve a instancia criada no servico de
      registro

      NomeDoServidor="rmi://"+svr.Parse(args[1]);
      Naming.rebind NomeDoServidor, svr);

   // cria o frame
   Janela = new FrameMaquina(svr);

   if (args[0].toUpperCase().equals("PRIMARIO"))
   {
     Janela.setTitle("Primario "+NomeDoServidor);
```

```
            svr.Tipo              = 1;
            svr.NrMembros         = args.length - 1;
            svr.CriaGrupo(svr.NrMembros, args);
            svr.ReplicaCriada=svr.PrimarioConectaBackups
                                                  ();
            Janela.t1.append("\n");
          }
          else
          {
            Janela.setTitle("Backup " + NomeDoServidor);
          }

          Janela.t1.append(" Servidor "+NomeDoServidor+"
                           executando ...");
      }
  }
```

Anexo 2 Implementação da Classe FrameMaquina

```
package Maquina;

import java.awt.*;
import java.lang.*;

public class FrameMaquina extends Frame
{
   public Panel p1, p2, p3;
   public TextArea t1, t2;
   public Button b1,b2;
   public MaquinaReplicacao Servidor;

   public FrameMaquina(MaquinaReplicacao t)
   {
      Servidor = t;
      setLayout(new BorderLayout());
      setBounds(100,10,450,450);

      p1 = new Panel();
      p2 = new Panel();
      p3 = new Panel();
      t1 = new TextArea();
      t2 = new TextArea();
      b1 = new Button("Membros do Grupo");
      b2 = new Button("Sair");

      add("North",p1);
      add("Center",p2);
      add("South",p3);
      p1.add(t1);
      p2.add(t2);
      p3.add(b1);
      p3.add(b2);

      show();
   }

   public boolean action(Event Evento, Object
                                          Objecto)
   {
      if (Objecto.equals("Sair")) System.exit(0);
      else if (Objecto.equals("Membros do Grupo"))
              Servidor.BackupMostraGrupo();
      return true;
   }
}
```

Anexo 3 Implementação da Classe Conexao

```
package Conexao;

import java.awt.*;
import java.io.*;
import java.lang.*;
import java.rmi.*;
import java.rmi.Naming;
import InterfaceMaquina.*;

public class Conexao extends Object
{
   public InterfaceMaquinaReplicacao service;

   boolean Ok;
   int NrMembros, ReplicaCriada;
   String[][] Grupo;
   InterfaceMaquinaReplicacao[] Replicas = new
                     InterfaceMaquinaReplicacao[10]

   public Conexao(String Argumento, TextArea
                  Resultado) throws Exception
   {
      int i=0, j=0;
      String host="", id=""
      boolean Achou = false;

      Resultado.append("Conectando primario ... ");

      try
      {
        if (System.getSecurityManager() == null)
            System.setSecurityManager ( new
            RMISecurityManager() );

        // define o nome da referência remota do
           primario a partir da linha de comando

        while ( i < Argumento.length() )
        {
          if("=".equals(String.valueOf(Argumento.char
                                          At(i))) )
          {
            if ( i + 4 <= Argumento.length() )
            {
              if("i".toLowerCase().equals(String.valu
                     eOf(Argumento.charAt(i+1))) &&
```

```
            "d".toLowerCase().equals(String.valu
                eOf(Argumento.charAt(i+2))) )
          {
             for (j=i+4;j<Argumento.length();j++)
                id = id + Argumento.charAt(j);

             break;
          }
        }
      }
      else
      {
        host = host + Argumento.charAt(i);
      }

      i++;
    }

    // conecta ao primario especificado na linha
       de comando

    service = (InterfaceMaquinaReplicacao)
              Naming.lookup
              ("rmi://"+host.toLowerCase()+"/"+
              host.toLowerCase()+id);

    NrMembros    = service.InformaNrMembros();
    Grupo        = service.InformaGrupo();
    ReplicaCriada=service.InformaReplicaCriada();

    // se o primario especificado estiver
       desatualizado, seleciona um dos backups
       para ser o novo primario

    if ( ReplicaCriada == 2 )
    {
      for ( i = NrMembros-1 ; i >= 0 ; i-- )
      {
         if(Grupo[i][1].toUpperCase().equals("PRI
            MARIO"))
         {
            Grupo[i][1] = "BACKUP";
            Grupo[i][2] = "ATIVO";

            service.PrimarioParaBackup();
         }
         else
         {
            if(Grupo[i][2].toUpperCase().equals("
               ATIVO") && !Achou)
```

```
            {
               try
               {
                  service =
                            (InterfaceMaquinaRep
                            licacao)
                            Naming.lookup("rmi:/
                            /"+Grupo[i][0]);

                  if(service.InformaReplicaCriada
                                          () == 1)
                  {
                    Grupo[i][1]   = "PRIMARIO";
                    Achou         = true;

                    Resultado.append("\nO
                            primario especificado
                            esta desatualizado.
                            Novo primario: " +
                            Grupo[i][0]);
                  }
               }
               catch (Exception e)
               {
                  Grupo[i][2]   = "INATIVO";
               }
            }
         }
      }

      // a atualizacao é feita aqui porque o
         grupo estará atualizado somente
         após a execução completa do for acima

      try
      {
         service.BackupParaPrimario();
         service.AtualizaGrupo(NrMembros,
                               Grupo,Replicas);
         service.PrimarioConectaBackups();
      }
      catch (Exception e)
      {
         Resultado.append("\nErro na busca de
                               novo primario ...");
      }
   }
}
catch (Exception e)
{
```

```
      Resultado.append("\nErro na conexao ao
                        primario ...");
   }

   Resultado.append(" Ok ");
};

public void Atualizar(Object t) throws Exception
{
   Ok = false;

   while ( ! Ok )
   {
      try
      {
         service.ClienteAtualizaReplica(t);

         NrMembros = service.InformaNrMembros();
         Grupo     = service.InformaGrupo();

         Ok        = true;
      }
      catch (Exception e)
      {
         ProcuraNovoPrimario(t,t1);
      }
   }

};

public void InformaGrupo(TextArea Resultado)
{  int i, j;

   Resultado.setText("\n");

   for (i = 0; i < NrMembros; i++)
   {
      Resultado.append("Host/Tipo/Status: ");

      for (j = 0; j <= 2; j++)
      {
         Resultado.append(Grupo[i][j] + " / ");
      }

      Resultado.append("\n");
   }
}

public void ProcuraNovoPrimario(Object t, TextArea
                                 Resultado)
```

```
{  int i;

   for ( i = NrMembros-1 ; i >= 0 ; i-- )
   {
       if(Grupo[i][1].toUpperCase().equals("PRIMAR
          IO") )
       {
          Grupo[i][1] = "BACKUP";
          Grupo[i][2] = "INATIVO";
       }
       else
       {
          if ( ! Ok )
          {  try
             { service =(InterfaceMaquinaReplicao)
                         Naming.lookup ("rmi://" +
                         Grupo[i][0] );

               service.SinalDeVida();

               Grupo[i][1]   = "PRIMARIO";
               Grupo[i][2]   = "ATIVO";
               Ok            = true;
               Resultado.append("\nNovo primario:
                              " + Grupo[i][2]);
             }
             catch (Exception e)
             {
               Grupo[i][2]   = "INATIVO";
               Resultado.append("\nA replica " +
                              Grupo[i][0] + "
                              não pode ser
                              assumir o
                              primário");
             }
          }
       }
   }

   // a atualizacao é feita aqui porque o grupo
      estará atualizado somente após a execução
      completa do for acima

   if ( Ok )
   {
      try
      {
         service.BackupParaPrimario();
         service.AtualizaGrupo(NrMembros, Grupo);
         service.PrimarioConectaBackups();
```

```
                service.ClienteAtualizaReplica(t);
            }
            catch (Exception e)
            {
                Ok = false;

                Resultado.append("\nA réplica " +
                                 Grupo[i][0] + "não pode
                                 assumir o pirmário");
            }
        }
        else
        {
            Ok = true;
            Resultado.append("\nNenhuma réplica pode
                                 assumir o primário ");

        }
    }
}
```

Anexo 4 Implementação da Classe ContaBancaria

```
package Conta;

import java.awt.*;
import java.rmi.*;
import java.rmi.Naming;
import java.io.*;
import InterfaceMaquina.*;
import Conexao.*;

public class ContaBancaria extends Frame
{
   static Conexao Primario;
   static Operacoes ObjetoCritico;

   public Panel p1, p2, p3;
   public TextField tf1;
   public Button b1,b2,b3,b4;
   public MenuBar MenuPrincipal;
   public Menu Opcoes;

   static TextArea  t1;

   public ContaBancaria()
   {
      setLayout(new BorderLayout());

      p1  = new Panel();
      p2  = new Panel();
      p3  = new Panel();

      t1  = new TextArea();
      tf1 = new TextField(10);

      b1  = new Button(" Depósito ");
      b2  = new Button("  Saque   ");
      b3  = new Button("  Saldo   ");
      b4  = new Button("   Sair   ");

      MenuPrincipal = new MenuBar();
      Opcoes        = new Menu("Opções");
      Opcoes.add(new MenuItem("Membros do grupo"));
      Opcoes.add(new MenuItem("Sinal de vida"));
      MenuPrincipal.add(Opcoes);
      setMenuBar(MenuPrincipal);

      add("North",p1);
```

```
   add("Center",p2);
   add("South",p3);

   p1.add(t1);
   p2.add(tf1);
   p3.add(b1);
   p3.add(b2);
   p3.add(b3);
   p3.add(b4);

   setTitle("Cliente");
   setBounds(100,10,450,300);
   show();
}

public boolean action(Event Evento, Object
                                    Objecto)
{
   if (Objecto.equals("   Sair   "))
      System.exit(0);
   else
      if (Objecto.equals("Membros do grupo"))
      {
       try
       {
         Primario.InformaGrupo(t1);
         Primario.service.PrimarioMostraGrupos();
       }
       catch (Exception e)
       {
         t1.append("Erro: Mostra Grupos");
       }
      }
   else
      if (Objecto.equals(" Depósito "))
      {
         ObjetoCritico.Deposito(Integer.parseInt(
                              tf1.getText()));

         try
         {
            Primario.Atualizar(ObjetoCritico,t1);
         }
         catch (Exception e)
         {
            t1.append("\nErro na atualização do
                                    primário");
         }
```

```
            t1.append("\nSaldo: " +
                      ObjetoCritico.Saldo);
         }
      else
         if (Objecto.equals("  Saque   "))
         {
            ObjetoCritico.Saque(Integer.parseInt(tf1
                                .getText()));

            try
            {
              Primario.Atualizar(MeuObjeto);
            }
            catch (Exception e)
            {
              t1.append("\nErro na atualização do
                                        primário")
                                        ;
            }

            t1.append("\nSaldo: " +
                      ObjetoCritico.Saldo);
         }
      else
         if (Objecto.equals("  Saldo   "))
         {
            t1.append("\nSaldo: " +
                      ObjetoCritico.Saldo);
         }

      return true;
   }

   public static void main(String args[]) throws
                                        Exception
   {
       new ContaBancaria();

       Primario      = new Conexao(args[0],t1);
       ObjetoCritico = new Operacoes();

       ObjetoCritico = (Operacoes)
                       Primario.service.ClienteCriaR
                       eplicas(ObjetoCritico);
   }
}
```

Bibliografia

[AGR90] AGRAWAL, D.; ABBADI, A. El. **The Tree Quorum Protocol: Na Efficient Approach for Managing Replicated Data**. Santa Barbara: Dept. of Computer Science, University of California, 1990.

[AMA98] AMARAL, J. F. B.; PASIN, Márcia.; LEITE, F. O.; JANSCH-PÔRTO, I. E. S. Implementando Réplicas de Arquivos Através de uma Ferramenta de Comunicação Confiável. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, 8., 1999. **Anais**..., São Paulo: Unicamp, 1999. p 84-98.

[AMA98a] AMARAL, J. B. **Protocolos para o Envio Confiável de Mensagens em Multicast**: trabalho individual. Porto Alegre: CPGCC da UFRGS, 1999.

[AMA99] AMARAL, J.F.B.; BERTAGNOLLI, S.C., LISBÔA, M.L.B**.** Componentes para apoiar o desenvolvimento de aplicações tolerantes a falhas. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, 8., 1999. **Anais**..., São Paulo: Unicamp, 1999. p 142-146.

[BAB93] BABAOGLU, O.; TOUEG, S. Non-Blocking Atomic Commitment. In: MULLENDER, Sape (Ed.). **Distributed Systems**. Workingham : Addison-Wesley, 1993.

[BAB98] BABAOGLU, O.; DAVOLI, R.; MONTRESOR, A. **Group Communication in Partitionable Systems:** Specification and Algorithms. Bologna: Dept. of Computer Science, University of Bologna, 1998. (Technical Report UBLCS-98-01.)

[BIR87] BIRMAN, K.; JOSEPH, T. Reliable Communication in the Presence of Failures. **ACM Transactions on Computer Systems**, New York, v.5, n.1, Feb. 1987.

[BIR93] BIRMAN, K. The Process Group Approach to Reliable Distributed Computing. **Communications of the ACM**, New York, v.36, n.12, p.37-53,103, Dec. 1993.

[BIR96] BIRMAN, K. **Building Secure and Reliable Network Applications**. Greenwich: Manning Publ., 1996. 591p.

[BUD93] BUDHIRAJA, N.; MARZULLO, K.; SCHNEIDER, F. B.; TOUEG, S. The Primary-Backup Approach. . In: MULLENDER, Sape (Ed.). **Distributed Systems**. Workingham : Addison-Wesley, 1993.

[BUR96] BURNS, Alan; WELLINGS, Andy**. Real-Time Systems and Programming Languages**. Harlow: Addison-Wesley, 1997. 611p.

[CAR97] CARDOSO, Afonso J. F. **Servidores de Arquivos Duplicados em Redes Locais com Ambiente UNIX**. Porto Alegre: CPGCC da UFRGS, 1997. Dissertação de Mestrado.

[CEC98] CECHIN, S. L. **Recuperação Assíncrona de Processos**. Porto Alegre : CPGCC da UFRGS, 1998. Exame de Qualificação.

[FAR98] FARLEY, Jim, **Java Distributed Computing**. Sebastopol, CA : Ed. O'Reilly, 1998.

[FIL99] FILHO, J. C. F. **Implementação de Objetos Replicados usando Java RMI** : trabalho individual. Porto Alegre: PPGC da UFRGS, 1999.

[FIL00] FILHO, J. C. F.; BERTAGNOLLI, S.C. Adicionando Replicação Utilizando Componentes de Software e um Ambiente Interativo. In: WORKSHOP DE TESTES & TOLERÂNCIA A FALHAS, 2., 2000. **Anais**... Curitiba: UFPR, 2000. p.40-45.

[GUE96] GUERRAOUI, Rachid; SCHIPER, A. Fault-Tolerance by Replication in Distributed Systems. In : ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 2., 1996. **Proceedings**... Berlim : Springer Verlag, 1996. p.38-57. (Lecture Notes in Computer Science, v.1088).

[GUE98] GUERRAOUI, Rachid et al. **System Support for Object Groups**. Lausanne, Switzerland : Department of Computer Science, Swiss Federal Insitute of Technology, 1998.

[HOF96] HOFF, Arthur Van; SHAIO, Sami; STARBUCK, Orca. **Ligado em Java**. São Paulo : Makron Books do Brasil, 1996.

[JAL94] JALOTE, P. **Fault Tolerance in Distribuited Systems**. New Jersey : Prentice-Hall, 1994. 432p.

[JDK99] JDK DOCUMENTACION. **JDK1.1 Beta Documentacion**. Disponível em : <http://deis12.cineca.it/local/manuals/java/docs/index.html>. Acesso em : 10 fev. 1999.

[LIA90] LIANG, L.; CHANSON, S.; NEUFELD, G. Process Groups and Group Communications: Classifications and Requirements. **IEEE Computer**, New York, p.56-66, Feb. 1990.

[LIS95] LISBÔA, M. L. B. **MOTF**: Metaobjetos para Tolerância a Falhas. Porto Alegre : CPGCC da UFRGS, 1995. Tese de Doutorado.

[LIS97] LISBÔA, M. L. B. A new trend on the development of fault-tolerant applications: software meta-level architectures. **Journal of the Brazilian Computer Society**, [S.l.], v4. n.3, p. 31-38, Nov. 1997.

[NEW97] NEWMAN, Alexander. **Usando Java :** O Guia de Referência mais Completo. Rio de Janeiro : Campus, 1997.

[NUN98] NUNES, Raul Cerreta**. Programação Orientada a Grupos**: Ponto de Vista das Aplicações : trabalho individual. Porto Alegre : PPGC da UFRGS, 1998.

[MON99] MONTRESOR, Alberto. **The Jgroup Reliable Distributed Object Model**. Bologna, Italy : Department of Computer Science, 1999.

[MOS98] MOSER, L. E.; MELLIAR-SMITH, P. M.; NARASIMHAN, P. Consistent Object Replication in the Eternal System. **Theory and Practice of Object Systems**, [S.l.], v.14, n. 2, p.81-92, 1998.

[RIT96] RITCHEY, Tim. **Programando Java & JavaScript**. São Paulo : Quark do Brasil, 1996.

[SCH93] SCHNEIDER, Fred. B. What Good are Models and What Models are Good ? In: MULLENDER, Sape (Ed.). **Distributed Systems**. Workingham : Addison-Wesley, 1993.

[SCH93a] SCHNEIDER, Fred. B. Replication Management using the State-Machine Approach. In: MULLENDER, Sape (Ed.). **Distributed Systems**. Workingham : Addison-Wesley, 1993.

[SIN94] SINGHAL, Mukesh; SHIVARATRI, Nirajan G**. Advanced Conceptes in Operating Systems**. [S.l.] : Mcgraw-Hill, 1994.

[SOR98] SORIA-RODRIGUEZ, Pedro. ***Multicast*-Based Interactive-Group Object-Replicaton for Fault Tolerance**. Massachussets : Worcester Polytechnic Institute, 1998. Dissertação de Mestrado. Disponível em : <http://xfactor.wpi.edu/Works/PSoria/thesis.html>. Acesso em : 22 abr. 1999.

[VER92] VERÍSSIMO, P.; RODRIGUES, L.; VOGELS, W**. Group Orientation: a Paradigma for Modern Distributed Systems**, Mont Saint-Michel, France: Proceedings of the 5th ACM SIGOPS European Workshop, Sep. 1992.

[WUT97] WUTKA, Mark. **Java Técnicas Profissionais**. São Paulo : Berkley, 1997.