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

Um Modelo de Replicação em Ambientes que Suportam Mobilidade

por

DÉBORA NICE FERRARI

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

Prof. Cláudio Fernando Resin Geyer

Orientador

Porto Alegre, janeiro de 2001

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Ferrari, Débora Nice

Um modelo de replicação em ambientes que suportam mobilidade / por Débora Nice Ferrari. – Porto Alegre: PPCG da UFRGS, 2001. 79f.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientador: Geyer, Cláudio F.R.

1. Sistemas distribuídos. 2. Replicação de Objetos. 3. Mobilidade. 4.Objetos Distribuídos. I. Geyer, Cláudio Fernando Resin. II. Título

Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Agradecimentos

À minha família, por compreender minha ausência e minha luta.

Ao Jorge, por tudo.

Ao meu orientador, prof. Cláudio Geyer, pelos seus conhecimentos,
sua compreensão e sua amizade.

Aos meus amigos, em especial a Patrícia e a Silvana, pelo companheirismo
e pela constante participação na elaboração deste trabalho.

Aos meus alunos, pela motivação de seguir sempre em frente.

À Deus, pela minha existência.

Sumário

Lista de Abreviaturas

AAPI	*Java Aglet Application Programming Interface*
DOBuider	*Distributed Object Builder*
DPC++	*Distributed Processing em C++*
GARF	*Génération Automatique d'applications Résistantes aux Fautes*
LAN	*Local Area Netware*
OMG	*Object Management Group*
ORB	*Object Request Broker*
ReMMoS	*Replication Model in Mobility Systems*
RMI	*Remote Method Invocation*
RTS	*Run Time Systems*
SBAC – PAD	*Symposium on Computer Architecture and High Performance Computing*.
UFRGS	Universidade Federal do Rio Grande do Sul
WAN	*Wide Area Netware*
WSCAD	*Workshop* de Sistemas Computacionais de Alto Desempenho

Lista de Figuras

Lista de Tabelas

Resumo

Os estudos sobre mobilidade intensificaram-se com o uso em grande escala da Internet, pois esta trouxe a possibilidade de explorar mobilidade através de redes heterogêneas, conectadas por diferentes *links* de comunicação e distantes umas das outras.

A replicação de componentes em sistemas distribuídos normalmente é utilizada para torná-los mais confiáveis e seguros ou para aumentar o desempenho da aplicação, uma vez que acessos remotos podem ser evitados através da localidade da réplica. Em qualquer um dos casos, a replicação implica a manutenção na consistência entre as múltiplas cópias, isto é, é preciso garantir que as cópias estejam com seus estados consistentes em um determinado momento.

Em sistemas que permitem mobilidade e replicação, a principal preocupação é com a consistência e o gerenciamento das réplicas do objeto móvel. Isto é, dependendo da técnica de replicação utilizada, como gerenciar um objeto e suas réplicas se estes podem mudar sua localização? Como garantir um bom desempenho do sistema? Estas perguntas esta proposta procura responder.

Este trabalho apresenta um modelo de replicação em ambientes de objetos distribuídos que permitem mobilidade chamado ReMMoS - *Replication Model in Mobility Systems*. O objetivo deste modelo é prover um ambiente de execução para suporte ao desenvolvimento de aplicações envolvendo mobilidade explicita e replicação implicita. Assim, o programador não necessita preocupar-se com o gerenciamento e a consistencia das cópias.

Palavras chaves: Sistemas Distribuídos e Paralelos, Programação Orientada a Objetos, Sistemas de Mobilidade, Replicação de Objetos

TITLE: "REPLICATION MODEL IN MOBILITY SYSTEMS"

Abstract

This work is about the studies about mobile and replication in distributed systems. The cientific studies about mobility intensified with the use of the Internet, because of this it is possible to explore mobility on heterogeneous network, connected by different communication links and distant from each other.

The replication of the components in distributed systems are used to make them more reliable and safe, so the system can survive the crash of one or more replicated component. It is necessary to keep maintenance of the consistence between backups.

The main concern regarding systems with mobility and replication is the consistence and management of the backup mobile object. The questions are: Depending on the technical replication used, how to manage an object and its backups if they can change places? How to guarantee a good speed of the systems? The proposal of this work is to answer these questions.

This work proposes an replication model in distributed object systems with mobility. This model is called ReMMoS - Replication Model in Mobility Systems. The objective is to achieve performance in applications using mobility. The mobility is explicit and the replication is implicit. So the programmer doesn´t need to worry about management and consistency of the copies.

Keywords: Distributed and Parallel Systems, Object Oriented Programming, Mobility Systems, Object Replication

1 Introdução

1.1 Tema do Trabalho

Este trabalho trata de mobilidade e replicação em ambientes de objetos distribuídos. O ponto principal diz respeito a observar como a replicação se comporta em ambientes que permitem o desenvolvimento de aplicações usando objetos móveis.

1.2 Motivação

A rápida disseminação de microcomputadores e estações de trabalho, o aumento nas suas capacidades de processamento e o surgimento de redes de comunicação com grande largura de banda tem levado ao uso cada vez maior de aplicações distribuídas.

Diversos modelos de arquitetura distribuída têm sido desenvolvidos com o objetivo de oferecer recursos que auxiliem o tratamento de características de distribuição tais como ausência de estado global e assincronismo. A característica de heterogeneidade, presente em grande parte destes modelos, impõe a necessidade de especificações abertas, com interfaces padronizadas e públicas, levando ao desenvolvimento de *middlewares* abertos. Esses sistemas oferecem interoperabilidade, escalabilidade e portabilidade. A interoperabilidade permite que sistemas diferentes convivam juntos. Já a portabilidade permite que um software desenvolvido em uma plataforma possa executar em outra com adaptações mínimas. A escalabilidade permite que novos recursos, tanto de hardware quanto de software, sejam adicionados ao sistema a medida que há necessidade de novos recursos ou de aumento do poder de processamento.

Nos últimos anos surgiram várias frentes de pesquisas para que as organizações pudessem explorar ao máximo os sistemas distribuídos. Dentre estas linhas de pesquisa, este trabalho destaca os estudos sobre mobilidade e replicação em sistemas distribuídos. As pesquisas em mobilidade intensificaram-se com o uso em grande escala das redes de longa distância (WAN - *Wide Area Network*), como a Internet. A mobilidade não é uma concepção nova, entretanto as WANs trouxeram a possibilidade de explorar mobilidade através de redes heterogêneas, conectadas por diferentes *links* de comunicação e distantes uma das outras.

As técnicas de replicação de objetos distribuídos se tornaram bastante úteis para aplicações onde requisitos como desempenho e disponibilidade são importantes. Através desta técnica, o tempo de resposta das requisições dos clientes diminui. Além disso, é possível assegurar a transparência de falhas no sistema distribuído. Na ausência de replicação, o funcionamento do sistema pode ficar limitado devido a falhas de componentes, sobrecarga de serviços e latência de acesso.

A replicação implica na manutenção da consistência entre as múltiplas cópias, isto é, é preciso garantir que todas as cópias possuam o mesmo estado. Sendo assim, o desempenho do sistema pode ser diminuído devido ao tráfego incrementado pela garantia de consistência. Para isto, pode-se utilizar políticas de posicionamento de réplicas que visam a reconfiguração do sistema com base na disponibilidade e desempenho requeridos pela aplicação. As duas políticas de replicação mais conhecidas são a **replicação ativa** e a **replicação passiva**. Na política de replicação ativa, todas as cópias do objeto replicado tratam uma requisição e retornam um resultado. Na política de replicação passiva, apenas uma cópia, geralmente denominada primária, executa os pedidos e atualiza as outras cópias.

Através da mobilidade, a execução pode ser transferida de uma máquina para outra. Portanto, em aplicações que usam mobilidade é necessária a preocupação com a integridade e disponibilidade. A replicação apresenta-se como forma de obter disponibilidade. Sendo assim, recentemente surgiram vários trabalhos envolvendo mobilidade, mais especificamente computação móvel, e replicação [OMG 98] [SOU 97] [KLE 96] [ION 96] [RAT 96]. Estes trabalhos foram apresentados em eventos específicos sobre o tema [ECO 00] [ECO 99].

Uma tendência em termos de desenvolvimento de *software* é a abstração de vários recursos necessários ou desejáveis para a construção de aplicações, sendo elas com mobilidade ou não. Desta forma, tecnologias cada vez mais robustas permitem que o desenvolvedor trabalhe em um nível cada vez mais abstrato, sem precisar se envolver com aspectos em baixo nível, que muitas vezes fazem com que o tempo de desenvolvimento de uma aplicação seja bastante grande.

Assim, novas ferramentas vêm sendo construídas com o objetivo de auxiliar o programador de ambientes distribuídos a desempenhar sua tarefa de forma eficaz e produtiva. Um trabalho em desenvolvimento no Instituto de Informática da UFRGS, pelo aluno Juliano Malacarne, é referente ao projeto e implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java denominada DOBuilder [MAL 00]. A programação é baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. Além disto, suporta o desenvolvimento de aplicações usando Voyager [OBJ 00]. Este é um ambiente que contém primitivas para o suporte à mobilidade de objetos distribuídos.

Este trabalho propõe um modelo de replicação em ambientes que permitem mobilidade de objetos. A mobilidade fica a cargo do desenvolvedor. A replicação é feita de forma transparente, facilitando o trabalho do desenvolvedor quando este necessita deste recurso em sua aplicação. Assim, este não precisará preocupar-se com o gerenciamento e consistência das réplicas. Além destas características, este modelo visa propor um desempenho satisfatório para as aplicações do usuário. Dependendo da técnica de replicação utilizada, como gerenciar um objeto, que pode mudar sua localização, e suas réplicas? Como garantir um bom desempenho do sistema na presença de réplicas? Este trabalho visa procurar respostas para estas perguntas.

Além disto, o modelo apresentado neste trabalho será integrado à ferramenta DOBuilder. Assim, as aplicações com ou sem suporte à mobilidade construídas na ferramenta, poderão comportar replicação implícita dos objetos, se assim o programador

especificar. Desta forma, a integração dos dois trabalhos tem o objetivo de oferecer ao programador um ambiente mais completo de desenvolvimento e de execução.

1.3 Objetivos do trabalho

O objetivo principal deste trabalho consiste em propor um modelo de replicação em ambientes de objetos distribuídos que suportam mobilidade. Embora a replicação permita tolerância a falhas, este não é o enfoque principal deste trabalho.

Os objetivos específicos são:

- usar um modelo de mobilidade explícita;
- propor um modelo de replicação transparente para o usuário;
- buscar uma forma eficiente de replicação em ambientes que permitem mobilidade, de forma a obter um desempenho satisfatório;
- projetar e implementar um protótipo que permita desenvolver aplicações usando o modelo proposto;
- integrar o protótipo à ferramenta DOBuilder;
- obter métricas para validar o modelo proposto e analisar o desempenho do protótipo.

Os estudos referentes a mobilidade e replicação em ambientes com suporte à mobilidade é uma área relativamente nova na comunidade científica e ainda inexplorada no Instituto de Informática. Assim, esse trabalho permitirá a criação de novas frentes de pesquisa dentro do Grupo de Processamento Paralelo e Distribuído.

1.4 Contribuição do Autor

Com relação às contribuições deste trabalho, pode-se destacar as seguintes:

1. Possibilidade de replicar objetos que podem ser movidos

A replicação de objetos distribuídos é usada em ambientes que suportam computação móvel com o objetivo de prover melhor disponibilidade, segurança e rapidez no acesso aos computadores remotos [BHA 97], [HÄR 97] e [SIL 97]. Já em ambientes que suportam a mobilidade de objetos, para tender ao pedido de objetos remotos, a replicação não costuma ser explorada. Como nestes ambientes a mobilidade do objeto é explorada de forma **explícita**, pode ocorrer que o programador não

identifique possíveis situações em que o acesso remoto aos objetos possa ser evitado. Além disso, a mobilidade permite aos clientes que estejam na mesma máquina do objeto acessá-lo localmente. Clientes em outras máquinas, mesmo necessitando freqüentemente do objeto, deverão acessá-lo remotamente. Com a possibilidade de replicação dos objetos, réplicas destes objetos poderão existir em várias máquinas do sistema.

2. Gerência implícita dos objetos replicados

A decisão quanto à replicação e gerência dos objetos replicados fica a cargo do sistema. O programador não precisa indicar ao sistema nenhuma informação para que os objetos sejam replicados e nem realizar qualquer tipo de gerência das réplicas. Conforme a característica da aplicação, o sistema decide quanto à criação e destruição das réplicas. Além disso, mantém a consistência das réplicas do objeto.

3. Não restringe a característica da aplicação

O modelo de replicação é adaptativo ao tipo de aplicação, visto que não restringe a característica da aplicação. Para isto, conforme o comportamento da aplicação, um objeto replicado pode ter suas réplicas criadas ou descartado. Desta forma, quando a aplicação se comporta de modo a predominar consultas, as réplicas são criadas. Se predominar atualizações, as réplicas ociosas do objeto vão sendo descartadas, para diminuir o custo de atualização do objeto replicado, uma vez que este custo cresce à medida que aumentam as réplicas. Entende-se por réplica ociosa a que não está processando, por um determinado período de tempo, consultas locais. No pior caso, se voltar a predominar no sistema atualizações sobre os objetos replicados, estes tornam-se objetos não-replicados, pois não existirão mais réplicas associadas a ele.

1.5 Estrutura do Texto

Além do presente capítulo, no capítulo 2 são apresentas considerações quanto à mobilidade de objetos em sistemas distribuídos. São abordados conceitos básicos sobre sistemas distribuídos, objetos distribuídos, migração de objetos e a ferramenta DOBuilder, necessários para contextualização do trabalho. O estado da arte sobre mobilidade em sistemas distribuídos também é apresentado, além de trabalhos relacionados ao assunto.

O capítulo 3 aborda replicação de objetos em sistemas distribuídos. São apresentadas algumas considerações sobre replicação, estratégias para manter a consistência das réplicas e técnicas de replicação. Além disso, o estado da arte sobre replicação em sistemas distribuídos que suportam mobilidade e trabalhos relacionados à replicação também são apresentados. As conclusões do capítulo são apresentadas ao final.

O Modelo ReMMoS é apresentado no capítulo 4. Questões com relação à concepção do modelo estão expostas neste capítulo, tais como: objetivos, princípios básicos, decisões de projeto e o modelo de replicação em ambientes que permitem

mobilidade. Finalmente, as conclusões do capítulo são apresentadas, destacando as principais contribuições do modelo.

A implementação do ReMMoS é apresentada no capítulo 5. Este apresenta a organização básica do protótipo construído, o detalhamento de seus módulos, a implantação do protótipo no Instituto de Informática da UFRGS e os resultados alcançados. Além disso, são discutidas as conclusões com relação à implementação.

Finalmente, no capítulo 6 as conclusões do trabalho são apresentadas. São destacadas as principais contribuições do trabalho e a continuidade do mesmo, através de diversas atividades que ainda podem ser realizadas. Este capítulo apresenta também as contribuições do mesmo para o Grupo de Processamento Paralelo e Distribuído do Instituto de Informática da UFRGS.

2 Mobilidade de Objetos em Sistemas Distribuídos

Este capítulo descreve os fundamentos que envolvem os estudos sobre mobilidade. Além disso, alguns conceitos importantes com relação a sistemas distribuídos e migração de objetos distribuídos são apresentados. As seções contêm os conceitos necessários para compreender o universo que envolve as aplicações que utilizam mobilidade, portanto o universo em que está inserido este trabalho. Alguns trabalhos relacionados, tais como Voyager [OBJ 00] Aglets [IBM 98] e Concordia [WON 97], também são apresentados. Finalmente, serão apresentadas as conclusões do capítulo.

2.1 Sistemas distribuídos

Sistemas distribuídos são compostos por um grande número de processadores ligados através de redes de alta velocidade. Tanenbaum [TAN 92] [TAN 95] considera sistema distribuído um sistema no qual vários processadores trabalhem juntos sem possuírem memória compartilhada. O ideal é que estas máquinas se comportem como um único computador para seus usuários (transparência).

Um sistema distribuído apresenta várias vantagens com relação aos sistemas centralizados, tais como possibilidade de acrescentar mais processadores ao sistema quando for necessário maior poder de processamento; a distribuição do processamento e possibilidade de compartilhamento de recursos, permitindo que mais de um usuário acesse uma base de dados comum e compartilhe periférico; facilidade para implementar tolerância a falhas, pois se uma das máquinas da rede não funcionar as restantes podem manter o sistema funcionando, dentre outras vantagens.

Vários aspectos devem ser observados no desenvolvimento de um software distribuído, tais como:

- Transparência: consiste em esconder do usuário o fato de estar trabalhando em um sistema distribuído. Este conceito pode ser aplicado a vários aspectos do sistema. Um sistema pode ser transparente quanto à **localização**, isto é, os usuários não devem saber onde estão os recursos; à **migração** onde os recursos podem trocar de lugar sem ter que trocar de nome; à **replicação** onde os usuários não devem saber quantas cópias existem; **concorrência**, isto é, os usuários podem compartilhar automaticamente os recursos e quanto ao **paralelismo**, onde podem ocorrer atividades paralelas;

- Confiabilidade: em geral os sistemas distribuídos são menos confiáveis que os sistemas centralizados pois possuem mais pontos possíveis de falhas. Mas técnicas de tolerância à falhas podem ser usadas para tornar o sistema mais confiável. Por exemplo, se uma das máquinas não funcionar o sistema não pára, apenas perde em desempenho;

- Desempenho: é uma característica de suma importância. Vários fatores devem ser trabalhados para melhorar o desempenho, tais como os protocolos de comunicação entre os processos, à qualidade das redes de comunicação e a granulosidade do processamento, dentre outros;

- Escalabilidade: neste caso, um sistema distribuído permite aumentar o número de recursos no sistema sem precisar trocar o sistema para isto. Por exemplo, ao necessitar de mais poder de processamento, pode-se aumentar o número de processadores no sistema.

Além destes aspectos, o desenvolvimento de um software para um sistema distribuído deve atentar para as características inerentes ao sistema. Destas características, as mais relevantes para este trabalho dizem respeito a comunicação e sincronização.

Em um sistema distribuído a comunicação é feita através de troca de mensagens, pois as máquinas estão distribuídas, cada uma com sua memória local. Além da comunicação, outro aspecto importante diz respeito ao fato de como os processos cooperam e sincronizam entre si.

A sincronização em sistemas distribuídos é um conceito mais complicado pois não existe memória compartilhada, por isto o uso de semáforos e monitores não funciona. Em geral, os algoritmos distribuídos para sincronização devem considerar que as informações relevantes estão espalhadas em diversas máquinas, os processos tomam decisões baseados nas informações disponíveis no local em que estão executando e não existe nenhuma fonte precisa de tempo global.

Uma abordagem mais aprofundada sobre este assunto pode ser encontrada em [TAN 95] e [TAN 92].

2.2 Objetos distribuídos

Objetos são representações de entidades físicas ou abstratas do mundo real, que pertencem ao contexto do sistema a ser projetado. O objeto possui como características fundamentais: os atributos, os métodos e as mensagens. Conceitos importantes relacionados com o paradigma orientado a objetos, tais como: **mensagem, encapsulamento**, **classe**, **instância**, **herança**, **associação dinâmica**, **polimorfismo** e **reusabilidade** devem ser observados em tecnologias que exploram mobilidade em objetos [AMA 97], [RUM 96].

Outra característica importante é à concorrência. Quanto a concorrência em objetos, existem dois tipos naturais:

- **Concorrência intra-objetos** - a concorrência surge da existência de vários acessos concorrentes a um mesmo objeto, que pode implicar a execução concorrente de métodos de um mesmo objeto;

- **Concorrência inter-objetos** - a forma de concorrência mais natural, que está associada ao fato de vários objetos poderem executar simultaneamente.

Havendo concorrência, deve haver sincronização. No caso de concorrência inter-objetos, é necessário sincronizar a execução de vários objetos. Já na concorrência intra-objetos, é preciso sincronizar a execução das várias *threads* dentro de um objeto.

A comunicação entre objetos pode ser feita de duas formas: **síncrona** e **assíncrona**. Na síncrona, após o envio de uma mensagem a um objeto, a execução do emissor fica suspensa enquanto não houver retorno por parte do receptor. Num contexto distribuído, deve-se tomar precauções para evitar que o emissor de uma mensagem fique indefinidamente bloqueado caso não receba resposta. Este tipo de problema levou a utilização de mecanismos de mensagens assíncronas. Neste caso, o emissor ao enviar uma mensagem ao um objeto não fica bloqueado, podendo seguir sua execução.

Com referência a objetos distribuídos, pode-se dizer que um objeto distribuído é uma espécie de código que pode "viver" em qualquer lugar de uma rede [ORF 96]. Eles são uma espécie de pacotes com código independente que podem ser acessados por clientes remotos via invocação de métodos. O cliente não precisa saber em que lugar, na rede, o objeto distribuído reside. Os objetos podem ser executados em sistemas operacionais diferentes e podem trocar mensagens com outros objetos em qualquer parte da rede [ORF 96].

Existe recentemente uma tendência de se construir sistemas computacionais abertos utilizando objetos distribuídos, tais como CORBA [OMG 98]. CORBA é um *middleware* aberto composto por objetos distribuídos: um 'barramento de *software'* é usado para conectar outros componentes de *software*. Objetos Distribuídos possuem as mesmas características principais dos objetos das linguagens de programação: encapsulamento, polimorfismo e herança, tendo, dessa forma, as mesmas principais vantagens: fácil reusabilidade, manutenção e depuração, só para citar algumas.

2.3 Migração de objetos em sistemas distribuídos

Na migração, um objeto é transferido de uma estação origem para uma destino, permitindo balanceamento de carga e aumento da eficiência em sistemas distribuídos. O maior ganho com a migração de objetos pode ser alcançado em redes de longa distância (WANs - *Wide Area Networks*), onde o custo de transferência de um objeto entre estações na rede muitas vezes é compensado pela facilidade de acesso a um objeto bem localizado [RUB 96].

A partir do pedido de migração vindo do usuário ou do sistema, um objeto que será migrado deve esperar pelo término de todos os seus métodos, isto é, não deve executar mais nenhuma tarefa além das necessárias para terminar sua tarefa. O objeto empacota seu estado em uma mensagem que pode ser enviada e interpretada em outra estação. Para isto, o objeto deve ser serealizável. Um objeto é serializável quando é possível empacotá-lo e enviá-lo via rede. Além disso, o objeto precisa informar ao serviço de nomes do sistema sua nova localização física.

Todas as mensagens recebidas pelo objeto durante sua transferência são dispostas em uma fila FIFO (*First In First Out*). Quando a transferência terminar, as mensagens enfileiradas precisam ser redirecionadas e executadas para o novo objeto. Uma mensagem é enviada da estação destino para a estação origem, indicando o

resultado da migração. Quando a estação origem receber um resultado de êxito, pode eliminar a sua cópia do objeto, finalizando assim o processo de migração do mesmo. Falhas durante a migração devem ser tratadas para que o sistema não fique inconsistente.

A migração de objetos servidores (que fornecem serviços) permite o balanceamento de carga entre as estações e eficiência ao sistema distribuído. Assim, o objeto servidor pode migrar para uma estação menos ocupada que a atual, evitando gargalos e equilibrando a carga operacional das estações no sistema distribuído [RUB 96].

Para que a migração torne o sistema distribuído eficiente, um objeto em uma rede WAN pode migrar de uma estação distante para uma mais próxima do cliente. Neste caso, o acesso ao objeto é otimizado, diminuindo o tráfego na rede e o tempo de latência da comunicação. Mesmo assim, nem sempre a localização física indica menor tráfego, pois o objeto pode ter sido migrado para uma área com tráfego intenso de informações. Isto deve ser considerado ao decidir a estação destino do objeto. Geralmente estas informações, de modo a obter otimizações, são tratadas por heurísticas.

O mecanismo de migração de objetos em sistemas distribuídos precisa manter as propriedades de tolerância a falhas, integridade referencial, transparência e eficiência. A tolerância à falhas garante disponibilidade do objeto migrado. A integridade referencial controla todas as requisições de acesso a objetos, mantendo o serviço para os clientes, apesar da migração. Para a transparência é necessário que o sistema adote um mecanismo de migração automático, implementado através de métodos de um objeto que controla a migração de outros objetos. Migração eficiente requer outros algoritmos para cooperar com o algoritmo de migração.

Maiores informações sobre migração de objetos podem ser encontradas em [RUB 96] e [BHA 97].

2.4 Uma ferramenta para programação com objetos distribuídos

Está sendo desenvolvido no Instituto de Informática da UFRGS, pelo aluno Juliano Malacarne, um trabalho referente ao projeto e implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java denominada **DOBuilder** [MAL 00]. A programação é baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. Além disto, suporta o desenvolvimento de aplicações usando Voyager [OBJ 00].

Conforme [MAL 00], a DOBuilder busca priorizar o atendimento a requisitos mais importantes para a programação distribuída. A este conjunto pertencem aplicações do tipo cliente-servidor e a interoperabilidade entre sistemas criados por diferentes desenvolvedores. Além disso, assim como todo ambiente de programação, procura buscar a facilidade de uso e oferecer recursos ao usuário que o auxiliem no desenvolvimento do *software*, tais como: reutilização, modularização, extensibilidade e geração automática de código.

Muitas das características desejáveis do ambiente estão sob a responsabilidade da sua linguagem visual. A DOBuilder adota um modelo onde a aplicação do usuário é representada através de um grafo dirigido e os objetos distribuídos são representados por nodos e os relacionamentos entre eles são visualizados explicitamente por meio de arcos que conectam tais nodos. Há vários tipos de relacionamentos possíveis entre diferentes nodos, sendo que os principais são a invocação remota de métodos e a comunicação por meio de mensagens explícitas.

A especificação da aplicação contém diferentes tipos de objetos. Os objetos distribuídos propriamente ditos são os nodos do grafo no programa visual e executam fisicamente em diferentes espaços de memória. A comunicação, entretanto, é implementada por outros objetos acoplados a esses objetos distribuídos. Tais objetos especiais são os componentes de programação distribuída, representada principalmente pelas portas de comunicação e referências remotas para chamadas de métodos. Essa abordagem de componentes permite que a comunicação seja implementada de diferentes maneiras, possibilitando ainda que seja alterada e personalizada de acordo com as necessidades de cada aplicação. Para tanto, basta que um novo tipo de componente seja implementado.

A comunicação entre o componente e o objeto que o contém é feita através de eventos. Todo o comportamento de um objeto é controlado pelos eventos que ele gera e recebe. Por exemplo, quando um objeto possuir dados prontos para receber em uma determinada porta de comunicação (um componente declarado no objeto), ele será notificado do evento que dá conta dessa condição. Durante a fase de desenvolvimento, o programador definirá a ação a ser tomada em função da ocorrência do evento em questão. O emprego do modelo de eventos é útil também como forma de aumentar a concorrência, pois um objeto poderá realizar outras tarefas enquanto espera a mensagem, não havendo a necessidade de executar sempre uma chamada de recebimento de mensagem que bloqueie o processamento antes do tempo necessário. A notificação de erros e falhas na comunicação também é facilitada através deste esquema.

O modelo de programação apresenta outros elementos além de objetos distribuídos e portas de comunicação. Tais como *locks*, dados globais, portas de chamadas de serviço, serviços e servidores virtuais. Todos eles estão organizados numa hierarquia de objetos e definidos em termos de propriedades, métodos e eventos. Esse tipo de definição faz com que os objetos da aplicação (chamados de componentes) sejam totalmente independentes entre si, tornando modular o desenvolvimento da aplicação. A possibilidade de extensão do ambiente é uma conseqüência natural da modularização, permitindo que novos componentes podem ser adicionados à ferramenta conforme a necessidade.

O projeto da ferramenta DOBuilder busca como principal contribuição a possibilidade de identificar os relacionamentos entre os diferentes objetos de uma aplicação distribuída de forma clara e imediata por parte do usuário. Isso é feito através da programação visual. Além disso, com o conceito de componente esta ferramenta objetivará aplicar noções importantes de engenharia de *software*, oferecendo a possibilidade de reutilização e encapsulamento do código dos programas.

O modelo apresentado neste trabalho (capítulo 4) poderá ser integrado a ferramenta DOBuilder. Assim, as aplicações com suporte à mobilidade construídas na

ferramenta (neste caso, o Voyager [OBJ 00]), comportarão replicação implícita dos objetos. Desta forma, a integração dos dois trabalhos tem o objetivo de oferecer ao programador um ambiente mais completo de desenvolvimento e de execução.

Maiores informações sobre a ferramenta DOBuilder podem ser encontradas em [MAL 00].

2.5 Mobilidade em sistemas distribuídos

Esta seção apresenta alguns conceitos relacionados à mobilidade em sistemas distribuídos. Um estudo mais abrangente pode ser encontrado no texto [FER 99b].

Por ser uma área de pesquisa relativamente nova, as pesquisas em mobilidade conceituam este termo de várias formas. A maioria refere-se à mobilidade através do termos código móvel (*mobile code*) [FUG 98] [CAR 97] [THO 97] [CUG 96] e agentes móveis (*mobile agent*) [DÖM 97] [CHE 96] [KNA 96]. Desta forma, não existe ainda um consenso com relação a definição de mobilidade.

Entretanto, pode-se notar um elemento presente em todos os conceitos: o **deslocamento** das entidades computacionais envolvidas (objetos ou agentes). Sendo assim, pode-se dizer que **mobilidade** é a capacidade das entidades envolvidas na computação deslocarem-se através de um sistema distribuído [FER 99b].

Mobilidade não é uma concepção nova, mas sim uma forma de organização de conceitos. As pesquisas em mobilidade vem crescendo atualmente devido principalmente à possibilidade de explorar mobilidade através de redes WANs, como a Internet por exemplo. Estas redes são interligadas de forma heterogênea e gerenciadas por organizações diferentes. Questões como migração e balanceamento de carga, por exemplo, devem ser repensadas, pois o custo de comunicação em uma rede local tende a ser menor do que o custo de comunicação através de WANs.

Tecnologias que exploram mobilidade devem prover mecanismos que suportem o deslocamento de suas entidades. Estes mecanismos necessitam de uma estrutura computacional que suporte a mobilidade. Sendo assim, a concepção de um **ambiente computacional** (*Computational Environment*) deve ser introduzida para suportar os diversos mecanismos necessários à mobilidade. Um ambiente computacional fornece suporte para que as aplicações tenham a capacidade de recolocar dinamicamente seus componentes em diferentes ambientes computacionais. Um exemplo de ambiente computacional é a máquina virtual Java (*Java Virtual Machine*) [SUN 00].

O texto [FUG 98] destaca que a mobilidade é determinada pela característica do sistema. Por exemplo, em Java é possível mover somente o código de um Applet [SUN 00]. Já em Java Aglet [LAN 97], tanto o código quanto o estado são movidos. Sendo assim, podem ser identificadas duas formas principais de mobilidade, isto é, **mobilidade forte** e **mobilidade fraca**.

Mobilidade forte permite que tanto o código quanto o estado da computação possam ser movidos para diferentes ambientes computacionais. Migração de objetos é um exemplo de mobilidade forte. Mobilidade fraca permite transferir código de um ambiente computacional para outro. O código pode conter alguns dados inicializados, mas o estado da computação não é movido. Um Applet Java é um exemplo de mobilidade fraca.

Mobilidade forte pode ser explorada através de dois mecanismos: **migração** (*migration*) e **clonagem remota** (*remote cloning*). No mecanismo de migração, a execução de uma unidade de execução é suspensa no ambiente origem e transmitida para o ambiente computacional destino, onde sua execução é retomada, existindo aogra no destino. Já no mecanismo de clone remoto, é criada uma cópia da unidade de execução no ambiente computacional destino. A unidade de execução não é isolada do ambiente computacional origem. Observa-se que replicação pode ser considerada como um tipo de clonagem remota.

2.6 Trabalhos relacionados

Existem vários trabalhos relacionados à ambientes que suportam mobilidade em sistemas distribuídos [CHE 96], [CAS 98], [FUG 98], [GLA 98], [OBJ 00], [OSH 98] [ROY 97]. Serão destacados aqui os mais significativos para este trabalho.

2.6.1 Voyager

Conforme [GLA 98] [OBJ 00], Voyager consiste de uma plataforma ORB (*Object Request Broker*) implementada em Java, suportando distribuição e mobilidade de objetos e agentes autônomos. Um ORB consiste em um componente que transfere uma requisição de um objeto cliente para um outro objeto, permitindo a utilização de um objeto independente da localização da sua implementação. Voyager suporta mobilidade de objetos e agentes, permitindo que estes possam enviar mensagens em padrão Java mesmo sem saber sua localização atual. Além disso, também inclui um conjunto de serviços que implementa, de forma transparente, persistência, comunicação em grupo e serviço básico de diretório [OBJ 00].

Voyager possui sintaxe própria para construir, enviar mensagens, mover objetos e agentes e implementar persistência distribuída. Desta forma, provê suporte para mobilidade de objetos e agentes, utilizando o modelo de objetos Java. Quando um programa Voyager inicia, ele automaticamente cria *threads* que provêem serviços como *garbage collection* e verificação do tráfego da rede. Cada programa possui um único endereço na rede formado pelo nome da máquina e pelo número da porta de comunicação.

Um objeto comunica-se com um objeto remoto através de referências virtuais (*proxy*). Mensagens enviadas para referências virtuais são automaticamente repassadas para o objeto remoto. Se uma mensagem retorna um valor, o objeto receptor envia o valor de retorno para a referência virtual, que retorna este valor para o emissor. Quando um objeto remoto é criado, é automaticamente atribuído a ele um identificador global único, que identifica o objeto em todos os programas da rede. O identificador pode ser usado para localizar ou conectar o objeto em qualquer ponto da rede.

Quanto à mobilidade, pode-se mover qualquer objeto serializável de uma máquina para outra enviando uma mensagem específica para o objeto via refêrencia remota, fornecendo o endereço do programa destino como parâmetro. A mobilidade é explícita, isto é, o programador deve especificar o endereço destino para que o objeto possa ser movido. Mobilidade para Voyager é a habilidade de mover-se independentemente de um dispositivo para outro através da rede [OBJ 00].

Uma mensagem pode ser enviada para um objeto mesmo que o objeto tenha sido movido de um programa para outro, simplesmente enviando a mensagem para o objeto no último endereço conhecido. Quando a mensagem não localiza o objeto destino, a mensagem procura por uma referência virtual do objeto. Se esta for localizada, a referência envia a mensagem para o objeto na nova localização. O valor retornado é marcado como a nova localização do objeto remoto. Após isto, mensagens são enviadas diretamente para a nova localização do objeto. Quanto às mensagens, Voyager suporta mensagens do tipo síncronas, assíncronas sem retorno, assíncrona com retorno e *multicast*. O tipo de mensagem *default* de Voyager é a mensagem síncrona.

Maiores informações a respeito de Voyager podem ser encontradas em [OBJ 00].

2.6.2 Aglets

Aglets [IBM 98] é uma plataforma para desenvolvimento de sistemas usando agentes móveis. Os agentes em Aglets são objetos Java com a capacidade de se mover de uma máquina para outra em uma rede. O Aglet leva consigo o código de programa e o estado dos objetos que o compõe, suportando a idéia de execução autônoma e rota dinâmica de seu itinerário.

A biblioteca Aglets/API (J-AAPI - *Java Aglet Application Programming Interface*) é uma proposta padrão para criar e gerenciar agentes. Contém métodos para inicialização, mensagens, movimento, ativação/desativação, etc. Plataformas escritas com estas bibliotecas podem ser executadas em qualquer computador que suporte a interface para Aglet. As principais abstrações do modelo são o Aglet, o contexto, o procurador, a mensagem, o itinerário e o identificador, como é visto a seguir:

- **Aglet** - é o elemento básico da biblioteca e consiste em um objeto Java móvel autônomo (pois possui sua própria *thread* de execução) e reativo (pois pode responder as mensagens externas);

- **Contexto** (*context*) - é um objeto estacionário que provê um ambiente para manutenção e gerenciamento de Aglets em execução;

- **Procurador** (*proxy*) - é um objeto que representa um Aglet e tem por objetivo fornecer transparência e independência de localidade e proteção a métodos públicos;

- **Mensagem** (*message*) - é um objeto transmitido entre Aglets, permitindo passagem de mensagens síncronas e assíncronas, usados para colaboração e troca de informações;

- **Itinerário** (*itinerary*) - é o plano de viagem do Aglet com o objetivo de prover uma abstração de padrões de viagens e rota;

- **Identificador** (identifier) - cada aglet possui um único e global identificador que é imutável durante todo seu ciclo de vida.

O modelo Aglet provê comportamentos essenciais com relação a um agente tais como: criação, clonagem, envio, retirada, desativação, ativação, liberação, mensagem e nomeação [LAN 97]. A criação de um Aglet implica na sua inclusão no contexto e da atribuição de um identificador único a ele. A execução começa assim que o Aglet seja inicializado com sucesso. A clonagem produz uma cópia quase idêntica do Aglet original no mesmo contexto. Somente há diferença na atribuição do identificador e o fato da execução reiniciar em um novo Aglet. Com isto, a *thread* em execução não é clonada. O envio de um Aglet de um contexto para outro remove ele do contexto corrente e insere dentro do contexto destino, onde a execução da *thread* é reinicializada. A desativação de um Aglet remove temporariamente do contexto corrente e o coloca em um armazenamento secundário. A ativação o restaura para o contexto corrente. A liberação o remove do contexto corrente.

Informações sobre Aglets podem ser encontradas em [IBM 98], [OSH 98] e [LAN 97].

2.6.3 Concordia

O Concordia [WON 97] [CAS 98], concebido pela *Mitsubishi*, constitui-se de um *framework* para o desenvolvimento e gerenciamento de aplicações de agentes móveis. Compreendem múltiplos componentes, todos escritos em Java, na qual são combinados para prover um ambiente para aplicações distribuídas. Este sistema requer uma implementação padrão do servidor Concordia, feito em cima da máquina virtual Java.

Seu ambiente é composto de um servidor e um conjunto de agentes. Um agente é gerenciado pelo servidor Concordia. Cada agente está associado a um usuário particular e carrega uma especificação que contém o caminho de migração e o método a ser executado em cada máquina. Quanto a arquitetura do sistema, cada nodo num sistema Concordia consiste em um servidor Concordia e serviços básicos para criação, migração, execução, comunicação e administração de agentes móveis.

O Gerenciador de Agentes provê a infra-estrutura de comunicação que permite aos agentes serem transmitidos e recebidos pelos nodos da rede. Além disso, gerencia o ciclo de vida do agente, provê suporte à mobilidade e interface para administração (inclusive remota) dos serviços do servidor Concordia.

A mobilidade do agente é transparente para os usuários do sistema. A viagem do agente é descrita através de um itinerário. Este é composto de múltiplos destinos. Cada destino descreve a localização do agente na viagem e o trabalho que o agente realiza nesta localização. A localização pode ser definida pela identificação da máquina na rede e o trabalho através dos métodos manipulados pelo agente.

O Concordia usa uma infra-estrutura similar a Java para mobilidade dos agentes. Durante a viagem de um agente através da rede, seu *byte code* e o *byte code* de alguns objetos agregados são carregados via uma classe especial. Durante a desserialização do agente, os bytes code para o agente e suas classes podem ser recuperados e usados para inicializar uma nova cópia do agente. A mobilidade dos agentes pode estender-se tanto em redes locais como na Internet.

O modelo de segurança Concórdia provê suporte para dois tipos de proteção: (1) proteção contra agentes com origem falsificada e (2) proteção dos recursos do servidor contra acessos não autorizados. Agentes são protegidos contra falsidade quando são armazenados no sistema cliente e quando são armazenados na base de dados persistente. Esta proteção é feita através de criptografia. Sendo assim, garante a proteção do agente no armazenamento e na migração, além de autenticar a origem do agente. O Gerenciador de Segurança gerencia os recursos de proteção. A cada agente é atribuída uma identificação, permitindo ao agente acessar os recursos.

A administração do sistema Concordia é feita pelo Gerenciador de Administração. Este inicializa e termina os servidores do sistema de agentes. Ele também gerencia a troca de política de segurança entre os agentes e os servidores fazendo uma requisição do comportamento do agente ou do servidor para o Gerenciador de Segurança. Além disso, monitora o comportamento dos agentes através da rede, provendo estatísticas sobre isto.

O Gerenciador de Diretório permite aos agentes móveis localizar os servidores de aplicação para interação. O sistema pode ser configurado para incluir um ou mais gerenciadores de diretórios. Além disso, registra os serviços de aplicações disponíveis na rede e disponibiliza uma operação de *lookup* para os agentes que estão a procura de serviços para interagir. Maiores Informações sobre Concordia podem ser encontradas nos textos [WON 97] e [CAS 98].

2.7 Considerações sobre os ambientes apresentados

A tabela 1 abaixo apresenta algumas comparações entre os ambientes Voyager, Aglets e Concordia [FER 99b].

TABELA 1 - Ambientes para explorar mobilidade

Ambientes			
Características	**Voyager**	**Aglets**	**Concordia**
Entidade computacional base objeto	X		
Entidade computacional base agente	X	X	X
Possui ambiente computacional	X	X	X
Suporta desenvolvimento em Java	X	X	X
Permite implementar mobilidade	X	X	X
Mobilidade de objetos	X		
Mobilidade de agentes	X	X	X
Explora mobilidade forte	X	X	X
Explora mobilidade fraca	X		
Mecanismos de serialização de objetos	X	X	X

Mecanismos de transporte e interação de agentes	X	X	X
Mecanismos para controle de prioridade			X
Permite implementar persistência	X	X	X
Mecanismos de segurança	X	X	X

Quanto à entidade computacional base, somente Voyager suporta objetos e agentes móveis, pois neste ambiente, um agente é considerado um objeto, mas com autonomia. Aglet e Concordia suportam somente agentes móveis. O suporte somente a agentes móveis limita um pouco o universo de aplicações que podem ser construídas.

Todos os ambientes possuem um ambiente computacional. Este fato é natural, pois para dar suporte à execução das entidades computacionais distribuídas, cada máquina envolvida na distribuição deve suportar o ambiente computacional necessário para que as entidades possam ser executadas.

Uma questão interessante é que todos os ambientes suportam desenvolvimento em Java, até porque são desenvolvidos como uma camada acima da máquina virtual Java. Embora existam outras linguagens que suportam mobilidade, Java destaca-se como linguagem principal para implementar mobilidade, uma vez que oferece todos os mecanismos necessários para implementar mobilidade, tais como serialização, acesso remoto, portabilidade, persistência, etc.

Voyager suporta mobilidade fraca e forte. A primeira é caracterizada pela capacidade de carregar classes para dentro de uma máquina virtual em tempo de execução. Sendo assim, o código para uma determinada tarefa pode ser carregado à medida que for necessário para a computação. A mobilidade forte é caracterizada pelo suporte a objetos e agentes móveis, pois eles devem levar consigo seu estado, isto é, ocorre uma espécie de migração. Aglet suporta mobilidade forte através de migração, pois o envio de um Aglet o tira de um contexto e o coloca em outro. Já Concordia permite uma mobilidade forte.

Destes ambientes, Voyager é o mais completo, pois possui suporte (primitivas especiais) para implementação de objetos e agentes móveis, utiliza padrão Java, mobilidade forte e fraca, mecanismos de serialização de objetos, segurança, transporte e interação das entidades base e persistência. Todas estas características são importantes para exploração da mobilidade.

Nenhuns dos ambientes estudados implementam replicação. Em Voyager, por exemplo, objetos são replicados em uma base de dados para garantir a persistência. Mas a replicação em nível de ambiente de execução não é oferecida.

2.8 Conclusões

Este capítulo apresentou os fundamentos que envolvem os estudos sobre mobilidade. Foram apresentados alguns conceitos importantes com relação a sistemas distribuídos e migração além de objetos distribuídos. Além disso, foram abordadas características de alguns ambientes que permitem explorar mobilidade de objetos e/ou agentes. Merece destaque a seção 2.7, onde foram apresentadas algumas considerações

quanto aos ambientes Voyager, Aglet e Concordia. O capítulo a seguir apresenta estudos relacionados à replicação em sistemas distribuídos.

3 Replicação de Objetos em Sistemas Distribuídos

Este capítulo apresenta algumas considerações com relação à replicação em sistemas de objetos distribuídos. Nas seções seguintes, serão apresentadas algumas estratégias e técnicas de replicação mais relevantes para este trabalho. Após, serão apresentados alguns trabalhos envolvendo replicação de entidades computacionais além de considerações quanto à replicação em ambientes que suportam mobilidade. Finalmente, as conclusões do capítulo são apresentadas.

3.1 Considerações quanto à replicação

A replicação é uma estratégia utilizada para distribuição de dados compartilhados, permitindo que várias cópias do mesmo item de dados residam em diferentes memórias locais. É utilizada principalmente para habilitar acessos simultâneos de diferentes nodos ao mesmo dado. A replicação é um assunto bastante estudado pela comunidade científica já há algum tempo. Alguns a estudam com o objetivo de permitir disponibilidade, outros para buscar um melhor desempenho do sistema.

Uns exemplos são os sistemas tolerantes a falhas que utilizam replicação em nível de *software*, onde múltiplas cópias do mesmo dado estão distribuídas através da rede. Como há replicação, os dados ainda podem ser acessados mesmo se um dos nodos não está respondendo [BAL 92b]. Desta forma, buscam obter disponibilidade dos dados na presença de falha de um dos nodos do sistema ou no caso de particionamento da rede. Outro exemplo é os sistemas DSM (*Distributed Shared Memory*), que normalmente optam por replicar os dados para alcançar melhor desempenho em algumas aplicações, através do acesso concorrente aos dados [STU 98]. A linguagem ORCA [BAL92a], por exemplo, usa replicação para aumentar o acesso aos dados e diminuir o *overhead* de comunicação, ocasionado pelo acesso remoto aos mesmos (operações de leitura podem ocorrer nas cópias locais). Outros exemplos de sistemas DSM podem ser encontrados em [PRO 98], [AMZ 96], [TAN 92a] e [LI 89].

3.2 Estratégias para manter a consistência das réplicas

Um problema fundamental quando se utiliza replicação é a necessidade de manter as réplicas com a mesma informação, isto é, manter a consistência da informação impedindo que um nodo acesse dados inconsistentes.

Para solucionar este problema, é necessário que as técnicas preocupem-se com **serializabilidade** e **persistência**. A serializabilidade garante que as invocações sigam uma ordem seqüencial de execução, garantindo a consistência dos dados. Isto pode ser obtido através da atomicidade e do controle da ordem das invocações. Na atomicidade, ou todos os dados tratam uma invocação ou nenhum trata. Quanto à ordem das invocações, se duas réplicas de um dado tratam duas invocações, as réplicas devem

tratar as invocações na mesma ordem. Já a persistência é uma propriedade que tem relação com o espaço/tempo do dado ou objeto, isto é, a existência do objeto independe da existência do seu criador (tempo) e o objeto pode trocar de localização (espaço).

Duas estratégias básicas são usadas para manutenção da coerência dos dados replicados: **invalidação** e **atualização**. No protocolo de invalidação, quando ocorre uma operação de escrita em um dado replicado, todas as réplicas deste dado dão invalidadas, tornando-se inacessíveis. Na próxima operação a este dado invalidado, ocorre uma falha de acesso, fazendo com que uma cópia válida deva ser buscada para a memória local. Este protocolo é mais recomendado quando há grande localidade de acesso aos dados por processador, pois menos mensagens de notificação serão enviadas e menos cópias deverão ser atualizadas.

No protocolo de atualização, uma escrita em um dado compartilhado causa a atualização de todas as suas cópias. Ao invés de uma mensagem de notificação ser enviada, é enviado um novo valor para o dado, o que, dependendo da aplicação, pode gerar muito tráfego na rede.

Segundo [BAL 92b], a escolha de qual estratégia de consistência de réplicas usar depende de vários fatores. Um fator importante é a relação de operações de leitura e escrita no objeto, o que é determinada pela aplicação. No protocolo de invalidação, se há muitas operações de escrita em um dado replicado seguida de uma operação de leitura, são necessárias muitas mensagens para buscar um dado válido. O protocolo de atualização resolve este problema, mas a operação de escrita pode tornar-se onerosa se o tamanho da mensagem é muito grande, pois todas as réplicas são atualizadas.

Outro fator está relacionado com a granulosidade do dado compartilhado. Para grão muito pequeno, uma mensagem de atualização custa aproximadamente o mesmo que uma mensagem de invalidação. Por outro lado, sistemas com grãos grandes utilizam invalidação, sendo esta proposta eficiente quando as seqüências de acessos de leituras e escritas ao mesmo item de dado por vários processadores são esparsas.

Tanto invalidação como atualização tem suas vantagens. A escolha de qual estratégia de coerência usar depende dos custos do protocolo de coerência, do tamanho do objeto ou dado, e do tamanho dos parâmetros para realizar uma operação de escrita. O melhor desempenho é alcançado quando a política de coerência adapta-se dinamicamente à situação. Sistemas que utilizam replicação em geral proporcionam um *overhead* pela necessidade de manter todas as réplicas consistentes. A configuração das réplicas deve ter como base o número e a localização destas. Os ganhos alcançados pela replicação dependem do número das réplicas, da localização destas, do protocolo de replicação, da natureza das transações executadas nas réplicas, além das máquinas e da rede que compõe o sistema [PAS 00]. Este assunto é destacado nas seções seguintes.

3.3 Estratégias de replicação

A replicação de um objeto somente é útil se este é freqüentemente consultado por um nodo remoto. Nos casos em que ocorrem mais leituras do que escritas, é vantagem replicar o objeto sempre que este for acessado [BAL 92b]. Em geral, pode-se distinguir entre três estratégias para replicação [BAL 92b]:

- Não replicar - cada objeto é armazenado em um processador específico;
- Replicação total - cada objeto compartilhado é replicado em todos os processadores no momento da carga da aplicação;
- Replicação parcial - cada objeto compartilhado é replicado em alguns dos processadores, baseados em informações em tempo de compilação, em tempo de execução ou em tempo de compilação-execução.

A primeira estratégia é usada em aplicações onde todas as operações são executadas em um único processador, isto é, operações seqüenciais. A segunda estratégia indiscriminadamente replica todos os objetos replicáveis em todos os processadores. Esta abordagem seria mais eficiente se aplicada em arquiteturas que possuem suporte de hardware para comunicação *multicast*.

A terceira estratégia seleciona os objetos replicáveis através de informações do sistema, podendo decidir dinamicamente sobre quais objetos serão replicados. Por exemplo, pode-se replicar objetos através de uma razão de *read/write* ou o compilador pode desabilitar replicação de objetos que não realizam nenhuma consulta.

3.4 Técnicas de replicação

Dentre as várias técnicas de replicação em sistemas distribuídos [BUD 93] [JAL 94], [STR 85] somente algumas, relevantes para este trabalho, serão abordadas. Muitas técnicas visam resolver problemas relacionados a tolerância a falhas. Como não é o enfoque principal deste trabalho, elas não serão abordadas.

3.4.1 Técnica de Replicação Primário-backup

Na técnica Primário-*backup* [BUD 93] existe uma réplica definida como primária e as outras como *backups*. A função da primária é receber as invocações dos clientes e enviar uma resposta. Os *backups* só interagem com à primária, não interagindo diretamente com o cliente.

O processamento da invocação é diferenciado de acordo com o tipo de operação a ser executada no dado. Caso o cliente requisite uma operação de leitura (*read*), a primária processa a requisição e retorna o valor ao cliente, sendo desnecessário informar aos *backups*, pois não há nenhuma alteração no dado.

Quando um processo cliente invoca o método de um objeto, acontecem as seguintes ações (considerando que não houve falha da réplica primária) [GUE 97]:

1. o processo envia uma invocação para a réplica primária;
2. a primária recebe e processa a invocação. No final da operação, a resposta está disponível e o estado da primária é atualizado, se for uma operação de atualização. Neste momento, a primária envia uma mensagem de atualização para os *backups* contendo o método invocado, a resposta e seu estado. Os

backups recebem a mensagem, atualizam seus estados e mandam um ACK (reconhecimento positivo) à primária;

3. após ter recebido os ACKS das cópias, a resposta da invocação é enviada para o processo cliente;

4. caso a inovação seja uma operação de leitura, assim que a resposta estiver disponível, ela é passada ao cliente.

Como todas as requisições chegam primeiramente à primária e esta envia as mensagens de atualização aos *backups*, todos eles vão receber as invocações na mesma ordem, garantindo assim a serializabilidade. Mas, para isto ser garantido, é necessário um canal de comunicação confiável onde a ordem das mensagens seja preservada. Em caso de falha, a técnica recebe um outro tipo de tratamento [GUE 97].

3.4.2 Replicação Ativa

Esta técnica atribui a todas as réplicas a mesma funcionalidade, ou seja, todas as réplicas recebem e processam pedidos dos clientes [SCH 93]. Depois que cada réplica processa a informação recebida, atualiza o seu próprio estado e envia a resposta ao cliente.

O cliente espera até receber a primeira resposta ou a maioria das respostas idênticas. Se alguma réplica não responder por motivo de falha, o processamento não está perdido, bastando existir ao menos uma réplica ativa funcionando corretamente.

Portanto, nesta abordagem não há um controle centralizado. Esta técnica requer que as invocações dos clientes sejam recebidas pelas réplicas na mesma ordem (ordem e atomicidade). Isto pode ser implementado através de *multicast* de ordem total ou *multicast* atômico. Esta técnica usa mais recursos que a primário-*backup*, pois cada réplica tem que processar a invocação de um método.

3.4.3 Técnica Combinada

Este esquema emprega uma combinação da técnica primário-*backup* e replicação ativas. Nesta técnica existem várias cópias do dado residindo em diversos nodos. Cada cópia é chamada de encarnação. Estas são organizadas em uma corrente linear. Para a encarnação i, a i+1 forma o seu *backup*. O chamador primário propaga a chamada para o seu *backup*, o qual enviará para o seu *backup*, e assim por diante, até o final da corrente. Desta maneira, todas as encarnações são chamadas. O resultado da chamada é retornado ao cliente pelo chamador primário.

A encarnação secundária executa uma função ativa. Todas as encarnações restantes executam uma chamada e cada uma destas encarnações recebe uma cópia do resultado. Entretanto, o cliente fará uma chamada única e receberá apenas uma cópia do resultado. Além disso, não é necessário um *multicast* de ordem total, por exemplo. Se o primário não responder por motivo de falha, a sua encarnação assume seu lugar e repassa a informação ao cliente [OLI 98].

Outras técnicas também são citadas na literatura, tais como [ION 96] e [REI 96], que tratam da replicação para suporte à computação móvel. Já [STR 85] e [JAL 94] são exemplos de técnicas voltadas para prover tolerância a falhas. O trabalho de Ionitoiu [ION 96] apresenta uma técnica de replicação de objetos com consistência relaxada (uma espécie de invalidação), onde os objetos são atualizados à medida que necessário. Já o trabalho de Reiher [REI 96] apresenta uma técnica baseada em reconciliação *Peer-to-peer*. A técnica da Recuperação Otimista é apresentada em [STR 85]. Esta técnica se baseia na preservação da dependência entre os processos, permitindo assim a determinação de *checkpoints* e armazenamento de mensagens para comportar-se assincronamente. Já a técnica Distribuída [JAL 94] trabalha com o armazenamento de mensagens através de um processo emissor, para garantir ordenamento total, e um receptor, garantindo ordenamento parcial. Por ser distribuída, pode atender a falhas simultâneas de múltiplos processos.

Estas técnicas não serão apresentadas detalhadamente, visto que suas aplicações diferem dos objetivos buscados neste trabalho.

3.5 Replicação e mobilidade

Replicação de objetos é muito usada em ambientes que suportam computação móvel. Na maioria dos sistemas, como em Voyager, os objetos são replicados em uma base de dados persistente para serem buscados quando o computador for conectado à rede. Em ambientes para computação móvel, a replicação é usada para obter melhor disponibilidade, segurança e desempenho.

Já em ambientes que suportam mobilidade de objetos, não é comum a replicação em nível de entidades computacionais. Já para prover desempenho e disponibilidade em sistemas de arquivos distribuídos, é comum o uso de replicação.

Sendo a migração e a replicação um tipo de mobilidade forte, então se pode destacar vários trabalhos que envolvem estes dois assuntos, geralmente relacionados a gerenciamento de arquivos, tais como [HUR 96], [HAC 89], [KUR 88], [SHE 86]. A maioria dos trabalhos trata de algoritmos, tratamento de arquivos e tolerância a falhas. Nos trabalhos que tratam de replicação, esta é utilizam para prover tolerância a falhas. Neste caso, o desempenho não deve ser fator determinante. Outros trabalhos abordam apenas mobilidade.

Alguns trabalhos, como o de [KUR 88] e [SHE 86], desenvolveram algoritmos para migração e replicação de arquivos. Mas estes algoritmos consomem um alto custo computacional para tomada de decisão, pois ambos usam algumas verificações do comportamento do sistema para decidir sobre o processo de migração/replicação. Já o trabalho de [HAC 89] apresenta um estudo em migração e replicação de arquivos e migração de processos usando um algoritmo distribuído que leva em conta a freqüência de acessos ao arquivo e seu tamanho, entre outros parâmetros obtidos do sistema.

Em [HUR 96] é apresentado um trabalho com migração e replicação de arquivos. Hurley considera a replicação de arquivos como sendo uma extensão natural da migração de arquivos. Desta forma, o trabalho trata de uma política de replicação dinâmica baseada em uma heurística estabelecida para migração: esta acontece sempre

que uma redução no tempo de resposta à operação em um arquivo pode ser alcançada. Neste caso, através de análise de acesso aos arquivos, é que decide-se quanto à replicação/migração. Além disso, como a replicação é dinâmica, sempre que uma das cópias do arquivo não esta mais sendo necessária, ela é descartada.

3.6 Trabalhos relacionados

Esta seção apresenta alguns trabalhos que utilizam replicação tanto à nível de tolerância a falhas (como DPC++, GARF, Piranha e Electra) quanto em nível de desempenho e compartilhamento de recursos (como Orca e TreadMarks). Embora TreadMarks não trabalhe em nível de objetos, sua concepção possui aspectos interessantes quando pensados com relação a um sistema de objetos.

3.6.1 Orca

Consiste em um sistema de memória compartilhada distribuída **baseado** em objetos, cujo sistema de execução gerencia o suporte da distribuição dos "objetos", fornecendo uma memória compartilhada distribuída estruturada. Caracteriza-se por permitir que processos em diferentes máquinas compartilhem dados, sendo estes dados encapsulados em **objetos de dados**, que são instâncias dos tipos abstratos de dados definidos pelo programador [BAL 92a] [BAL 92b]. Assim, os programadores definem operações para manipular as estruturas de dados compartilhadas. Cada operação em um objeto é atômica, o que dispensa o uso de *locks*. O modelo de objetos compartilhados é suportado pela linguagem Orca [BAL 98].

O sistema Orca é composto de três partes: um compilador, um sistema de execução e uma máquina virtual chamada Panda. O compilador é o responsável, dentre outras coisas, por gerar informações sobre quais operações têm atributo somente de leitura e como os processos acessam os objetos compartilhados. Estas informações serão utilizadas para auxílio na tomada de decisões quanto à replicação dos objetos de dados. O sistema de execução (RTS – *RunTime System*) é responsável por gerenciar os processos e os objetos de dados do Orca.

São mantidas estatísticas em tempo de execução sobre sua utilização. Baseado nestas informações, o sistema decide em quais máquinas colocar cópias dos objetos compartilhados.

Os objetos de dados compartilhados que são lidos freqüentemente são replicados. A vantagem da replicação é que operações de leitura (as quais são reconhecidas pelo compilador, pois a linguagem provê sintaxe própria para leitura e escrita) podem ser executadas localmente, sem a necessidade de comunicação. O protocolo de atualização é utilizado para implementar operações de escrita. Isto é feito através de expedição de função (*function shipping)*, que consiste em enviar a operação e seus parâmetros para todas as máquinas que contenham uma cópia do objeto de dado, não sendo necessário transmitir o objeto de dados inteiro.

Para realizar este processo de atualização de forma coerente, a operação é enviada utilizando comunicação de grupo totalmente ordenada. Quando uma atualização é solicitada em um objeto de dado replicado, o solicitante envia um *broadcast* da operação e bloqueia-se até que o sistema de execução tenha processado a mensagem de *broadcast* (e todas as outras mensagens que ocorreram antes dela). Quando este tipo de mensagem é recebido por um processador, o sistema de execução verifica se existe uma cópia deste objeto de dado na memória local. Se existe, o sistema chama um procedimento que executa a atualização. Senão, desconsidera a mensagem.

O protocolo utiliza um seqüenciador centralizado para ordenar todas as mensagens. Cada mensagem de *broadcast* contém um número de seqüência, o qual os receptores utilizam para ordenar as mensagens e para verificar se falta alguma mensagem. Uma desvantagem deste protocolo é que a máquina seqüenciadora precisa ser praticamente dedicada, pois ela pode ter que manipular várias requisições por segundo. Além disso, todos os processadores que participam da execução recebem mensagens, mesmo que não possuam cópias do objeto de dado que está sendo atualizado. Portanto, uma operação de escrita em um objeto replicado irá interromper todas as máquinas envolvidas na execução.

Maiores informações sobre o sistema Orca podem ser encontradas em [BAL 92a], [BAL 92b] e [BAL 98].

3.6.2 TreadMarks

TreadMarks é um software que permite programação concorrente com memória compartilhada em arquiteturas de memória distribuída. Tem como objetivo reduzir a quantidade de comunicação necessária para manter a consistência de memória. Para isto utiliza um protocolo de múltiplos escritores e o modelo de consistência de liberação preguiçosa [AMZ 96].

Quando a memória compartilhada é acessada por um processador, TreadMarks determina onde o dado está fisicamente presente, e se necessário transmite o dado para o processador sem a intervenção do programador. Quando a memória compartilhada é modificada por um processador, é assegurado que outros processadores serão notificados da mudança, portanto eles não terão dados obsoletos. Esta notificação não é imediata nem global. Notificação imediata seria realizada freqüentemente e notificação global normalmente iria enviar para processadores informações que não seriam pertinentes a maioria deles. Ao invés disto, esta notificação entre processos é realizada quando eles são sincronizados.

Os protocolos de vários escritores tratam do problema de falso compartilhamento. Caracterizam-se por consentir que vários processadores tenham permissão para escrever na cópia da página ao mesmo tempo. As páginas compartilhadas são inicialmente protegidas para escrita. Quando uma escrita ocorre por um processo P1, por exemplo, é criada uma cópia da página, chamada de *twin*, e salvada como parte das estruturas de dados do TreadMarks neste processador. O sistema então desprotege a outra cópia da página, a qual está presente no espaço de endereçamento do processo. Isto possibilita que as próximas escritas à página possam ocorrer sem a intervenção do software DSM.

No momento em que o processador P_1 for realizar uma operação de liberação, existem duas cópias da página. A cópia modificada que está no espaço de endereçamento do processo e a cópia *twin*, com a versão original (sem modificações) da página. A partir de uma comparação palavra-por-palavra da cópia do processo com a *twin*, é criada uma estrutura chamada *diff*, com as modificações realizadas na página. Como os *diffs* contêm apenas as modificações realizadas na página, seu tamanho varia de acordo com a quantidade de dados modificados. Uma vez que a *diff* tenha sido criada, a *twin* é descartada. Atualizações nas outras cópias da página presentes no sistema podem ser realizadas aplicando a *diff* sobre estas páginas.

Com exceção da primeira vez que um processador acessar uma página, sua cópia é atualizada exclusivamente aplicando os *diffs*. Portanto, não é necessário buscar novamente a página inteira. Esta estratégia reduz os efeitos de falso compartilhamento. Além disso, reduz significativamente a exigência de banda passante, pois os *diffs* normalmente são bem menores do que as páginas.

TreadMarks usa consistência de liberação preguiçosa - LRC (*Lazy Release Consistency)* como modelo de consistência de memória, permitindo que alterações a dados compartilhados se tornem visíveis a outros processadores apenas em determinados pontos de sincronização. Nos pontos de aquisição de *locks* ocorrem as invalidações das páginas alteradas por outros processadores. Entretanto, as modificações dessas páginas só são requeridas na ocorrência de uma falha de acesso.

A notificação sobre as modificações ocorridas nos dados compartilhados é realizada através de **avisos de escrita** (*write-notices*). Um aviso de escrita indica que uma determinada página foi modificada, mas não possui o conteúdo das modificações. Cada intervalo contém um aviso de escrita para cada página modificada no segmento de tempo correspondente. Ao receber um aviso de escrita, o processador invalida a página correspondente. Os *diffs* relativos às modificações em questão só são recebidos na próxima falha de acesso à página.

Maiores informações sobre TreadMarks podem ser encontradas em [AMZ 96] e [ARA 99].

3.6.3 Replicação de Objetos Distribuídos no DPC++

O DPC++ (*Distributed Processing em C++*) é uma linguagem orientada a objetos para programação de sistemas distribuídos em redes locais de estações de trabalho. A linguagem foi implementada utilizando como base a linguagem C++. O modelo de distribuição permite que objetos instanciados executem concorrentemente com outros objetos. O fluxo de execução interna ao objeto é seqüencial. Além disso, o modelo prevê que a comunicação entre dois objetos distribuídos deva ser iniciada pelo objeto Diretório. Cabe a este esperar por algum pedido vindo por parte de algum objeto distribuído. Normalmente estes pedidos são *status* de outros objetos distribuídos do sistema ou a criação de novos objetos distribuídos [CAV 94b].

O trabalho de Oliveira Júnior [OLI 98] apresenta um modelo de replicação para garantir tolerância a falhas no DPC++. Esta é obtida através da replicação do objeto Diretório. A técnica utlizada para gerenciar a replicação é a técnica primário-*backup* [BUD 93]. Este modelo garante que o ambiente de execução do DPC++ retorne sempre

a um estado global confiável e seguro, quando ocorrer uma falha de *crash* no nodo que está executando o objeto Diretório.

Na primeira vez que o objeto Diretório é criado, o DPC++ deverá também criar o mesmo objeto Diretório em outro nodo da rede distribuída. Esta situação é controlada pelo objeto Diretório Primário, no momento da criação do seu *backup*. Assim que este for criado, é armazenado no Diretório Primário a identificação dos diretórios primário e *backup*. Isto é necessário para que, ao criar um objeto distribuído, o objeto Diretório além de enviar a sua identificação, envie também a identificação do seu *backup*. Desta forma, em caso de falha do Diretório Primário, os objetos distribuídos podem reportar-se ao *backup*.

Como o modelo utiliza a técnica Primário-backup, toda vez que um objeto distribuído qualquer faz um pedido para o diretório primário, este atualiza seu backup. Somente após receber a resposta de atualização do backup, o objeto Diretório envia a resposta da solicitação ao objeto distribuído que realizou o pedido. O modelo ainda define várias situações de detecção de falhas e como o sistema comporta-se nestas situações. Maiores informações podem ser encontradas em [OLI 98].

3.6.4 Piranha e Electra

O Piranha [MAF 96] é uma interface gráfica interativa usada para gerenciar e monitorar objetos em uma rede de computadores. Essa interface é implementada em CORBA [OMG 95], usando conceitos da programação orientada a grupos. O Piranha permite migrar, replicar, inicializar e eliminar objetos.

Para prover estas facilidades, o modelo CORBA foi estendido para suportar replicação dinâmica. Neste caso, o Piranha usa um ORB para:

- replicar dinamicamente os objetos mais importantes;
- manter referências válidas até que um objeto falho se recupere;
- permitir que objetos reportem situações excepcionais através de um serviço de *notificação distribuído*. Este é implementado como um grupo de objetos;
- permitir detecção automática de objetos falhos através de um *serviço de detecção de falhas distribuído*;
- permitir reinicialização automática de objetos falhos;
- possibilitar instalar novas versões de serviços em tempo de execução, sem reinicializar aplicações de clientes, desde que a interface do serviço não seja alterada.

Para implementar estes serviços é usado um ORB especial chamado Electra [MAF 95]. O Electra realiza o gerenciamento entre os membros do grupo e é encarregado de enviar mensagens indicando a mudança da configuração do grupo. O ORB implementa o serviço de replicação dinâmica usando réplicas ativas ou réplica primária. Esses objetos são tratados como um grupo, permitindo manter um único estado. Uma invocação para o grupo terá sucesso se sobreviver pelo menos um dos

membros do grupo. O grau de replicação pode aumentar em tempo de execução através de adição de novos objetos. Para garantir a consistência do estado do grupo, requisições enviadas ao grupo devem ser sincronizadas com a transferência de estado e as mensagens para gerenciar o grupo. Todas as mensagens deverão ser ordenadas seguindo o conceito de sincronismo virtual.

Os objetos do Piranha podem migrar de um nodo para outro quando o usuário desejar. A migração é usada para manutenção preventiva: se o nodo for desativado, o objeto que está no nodo pode migrar para outra localização. A migração do objeto é realizada através da criação de uma réplica, que é sincronizada com o objeto do nodo fonte, e em seguida é eliminado. Sincronização virtual é usada para controlar as mensagens perdidas durante a migração.

Maiores informações sobre os sistemas Piranha e Electra podem ser encontradas em [MAF 96], [MAF 95] e [PAS 00].

3.6.5 GARF

O GARF (*Génération Automatique d'Applications Résistantes aux Fautes*) é uma ferramenta orientada a objetos para a construção de aplicações distribuídas confiáveis [GUB 97]. O programador desenvolve uma aplicação centralizada e esta aplicação é incrementada com uma biblioteca que permite distribuir os componentes da aplicação em diferentes nodos, aumentando a confiabilidade.

A comunicação de grupo é usada para gerenciar e manter objetos replicados em diferentes nodos do sistema. O GARF é implementado como uma camada intermediária entre um sistema de comunicação de grupo e a aplicação do usuário. O sistema permite modularidade de *software* e separa os procedimentos que permitem concorrência, distribuição e replicação dos procedimentos funcionais dos sistemas centralizados tradicionais.

A camada intermediária é uma biblioteca de abstrações para programação distribuída. Essas abstrações possibilitam implementar diferentes protocolos de consistência de réplicas. O programador pode implementar novas abstrações combinando as já existentes ou usando as primitivas de comunicação de grupo embaixo do GARF.

O GARF possui três tipos de objetos: dados (cliente e servidor), procedimentos e sistema. Os dados são entidades passivas que se comunicam por mensagens síncronas ponto-a-ponto usando clientes e servidores. Os procedimentos dos objetos implementam concorrência, distribuição ou replicação de dados. GARF suporta dois tipos de procedimentos: *encapsulators* e *mailers*. Os *encapsulators* encapsulam dados e controlam o envio e o recebimento de mensagens (objetos). Os *mailers* são usados para executar a comunicação remota entre os *encapsulators*. O GARF implementa mecanismos de baixo nível nos quais as classes *encapsulators* e *mailers* são baseadas.

Os três tipos de objetos do GARF correspondem a três diferentes níveis de programação que o programador deve seguir [GUB 97]: dados, amarração e procedimentos. No nível de dados, a aplicação é desenvolvida de forma centralizada através da modelagem de classes. No nível de amarração, a aplicação é modificada para

tratar concorrência, distribuição e replicação conforme suas necessidades. Neste nível, os dados são amarrados aos seus procedimentos. Procedimentos armazenados em classes são selecionados de acordo com as semânticas da aplicação. No terceiro nível, a biblioteca de classes é finalizada para comportar as demais necessidades da aplicação. O programador combina as classes existentes para formar novas classes para a aplicação.

O GARF foi construído para ser executado sobre o Isis [BIR 87], [BIR 93]. É codificado em Smalltalk através um *run-time* e de três bibliotecas: *encapsulator*, *mailer* e *system*. O *run-time* gerencia a criação de objetos e a comunicação, e associa dados a comportamentos especificados pelo programador. O GARF *run-time* faz essas associações interceptando a criação de dados e a comunicação entre objetos. Cada dado possui um *encapsulator* e um *mailer* associados. Um dado e o seu *encapsulator* correspondente sempre estão localizados no mesmo nodo. O *run-time* redireciona toda a comunicação de objetos através do *encapsulator*, que trata do controle de concorrência e da replicação de objetos. Uma réplica do *mailer* é criada em outro nodo cada vez que o objeto correspondente é invocado por um objeto deste nodo. A réplica do *mailer* funciona como *proxy*, redirecionando mensagens para o *encapsulator*.

Para realizar a sincronização das réplicas, inicialmente o GARF usava o Isis, que mais tarde foi substituído pelo Bast [GAR 97], [GAR 98]. O Bast é um *framework* orientado a objetos para a construção de aplicações tolerantes a falhas, sendo mais flexível que o Isis, que usa o modelo de processos. Maiores detalhes com relação ao GARF podem ser encontrados em [GUB 97] e [PAS 00].

3.7 Considerações sobre os ambientes apresentados

O sistema Orca é um sistema baseado em objetos para sistemas com memória compartilhada distribuída. É dito baseado em objetos pois manipula tipos abstratos de dados que são acessados a partir de operações de alto nível. Desta forma, a unidade de compartilhamento do Orca é o objeto de dados. Os sistemas DPC++, Piranha e GARF são voltados para construção de aplicações tolerantes a falhas. O sistema de replicação em DPC++ e o Piranha possuem como unidade de compartilhamento o objeto, sendo que o segundo é uma interface gráfica interativa usada para gerenciar e monitorar objetos em uma rede de computadores. O GARF é uma ferramenta que permite desenvolver aplicações distribuídas confiáveis, encapsulando dados em objetos. Já o TreadMarks compartilha páginas de memória.

Portabilidade é uma característica importantíssima tanto para mobilidade quanto para replicação. Tanto o Orca quanto o TreadMarks caracterizam-se por serem portáveis no sentido que os programas não precisam ser recompilados. O DPC++ também é portável, desde que o ambiente possua o compilador DPC++ e os programas sejam compilados novamente na mudança do sistema operacional. Já o Piranha apresenta uma portabilidade maior com relação ao GARF, pois o primeiro foi desenvolvido usando CORBA. O segundo necessita do sistema Isis para dar suporte à comunicação em grupo.

O Orca utiliza protocolo de atualização enquanto o TreadMarks o de invalidação com consistência relaxada. O primeiro utiliza expedição de funções para implementar o

protocolo de atualização, enquanto o segundo utiliza o sistema de *diffs* para páginas. Já no DPC++ a consistência é feita através da técnica de Primário-*backup*, onde o objeto Diretório possui um objeto *backup*. O GARF possibilita a implementação de diferentes protocolos de consistência de réplicas. O programador pode implementar novas abstrações combinando as já existentes ou usando as primitivas de comunicação de grupo embaixo do GARF. O Piranha usa um ORB para replicar dinamicamente os objetos mais importantes, usando réplicas ativas ou réplica primária. Este é responsável pelo gerenciamento entre os membros do grupo e é encarregado de enviar mensagens indicando a mudança da configuração do grupo.

No TreadMarks devem ser declarados através de comandos quais áreas de memória serão compartilhadas. No Orca o sistema é que determina quais objetos serão replicados, dependendo do comportamento da aplicação. No DPC++ o objeto Diretório sempre é replicado, embora não comporte replicação dos objetos distribuídos. No GARF e no Piranha o programador deve determinar quais objetos serão replicados, ficando o sistema responsável pelo gerenciamento do mesmo.

Maiores informações sobre comparações do sistema Orca e do TreadMarks podem ser encontradas em [ARA 99]. O texto de Márcia Pasin [PAS 00] traz considerações a respeito dos sistemas GARF e Piranha.

3.8 Conclusões

Este capítulo apresentou considerações com relação à replicação em sistemas de objetos distribuídos. Tanto sistemas que permitem replicação buscando desempenho, como sistemas que se preocupam em prover tolerância a falhas foram estudados. Foram apresentadas algumas estratégias e técnicas de replicação mais relevantes para este trabalho. Alguns trabalhos envolvendo replicação de entidades computacionais além de considerações quanto à replicação em ambientes que suportam mobilidade também foram apresentados. Vale ressaltar que os trabalhos envolvendo mobilidade e replicação está mais presente em ambientes para suporte à computação móvel. Nestes ambientes, quando o cliente se conecta na rede, é feita automaticamente uma réplica da aplicação que o cliente utilizará. Após o trabalho *off-line*, o cliente conecta-se novamente na rede para atualizações dos objetos movidos.

Trabalhos envolvendo replicação em ambientes que suportam mobilidade de objetos em nível de aplicação não são conhecidos até o momento da escrita deste texto. A autora deste trabalho considera importante a replicação nestes tipos de ambientes, uma vez que os custos de acesso remotos aos dados podem ser amenizados. Além disto, o ambiente torna-se mais completo.

O capítulo a seguir apresenta o modelo de replicação em ambientes que suportam mobilidade.

4 Um Modelo de Replicação em Ambientes que Permitem Mobilidade de Objetos

Este capítulo apresenta o modelo de replicação em ambientes que permitem o desenvolvimento de aplicações envolvendo mobilidade. O modelo é denominado REMMOS- **Re***plication* **M***odel in* **Mo***bility* **S***ystems*. Este possui como principais características a gerência implícita da replicação, a independência do modelo de mobilidade, o controle dinâmico do número de réplicas, desempenho em sistemas distribuídos que permitem mobilidade e o uso de um ambiente de programação visual para programação distribuída, onde o ambiente de execução poderá suportar replicação.

O modelo ReMMoS foi apresentado à comunidade científica brasileira através do artigo publicado no WSCAD'00 – *Workshop* de Sistemas Computacionais de Alto Desempenho em outubro deste ano [FER 00]. Este evento ocorreu dentro da programação do SBAC – PAD'00 - *12th Symposium on Computer Architecture and High Performance Computing*.

4.1 Princípios Básicos

Os princípios básicos para definição do modelo são os seguintes:

- **Desempenho em sistema de objetos distribuídos que permitem mobilidade**. A mobilidade neste tipo de ambiente permite que um objeto, que está sendo constantemente acessado, seja movido para o nodo solicitante. Mas, outros nodos que estejam acessando este objeto continuarão acessando o mesmo remotamente. Com a replicação, há a possibilidade de manter uma réplica do objeto em cada nodo solicitante. Desta forma, o número de acessos remotos ao objeto diminui;

- **Independência do modelo de mobilidade**. A interface entre o modelo de mobilidade e o modelo de replicação tende a ser o mais flexível e independente possível. Desta forma, o modelo de replicação pode ser usado em outros ambientes que utilizam modelos de mobilidade diferentes;

- **Gerência implícita de Replicação**. O desenvolvedor, embora tenha conhecimento da existência de uma camada de replicação no ambiente, não necessita realizar nenhuma gerência dos objetos replicados. Desta forma, o ambiente é responsável por todo o gerenciamento da replicação, facilitando ao desenvolvedor o uso deste recurso em seu ambiente;

- **Modelo adaptativo ao comportamento da aplicação**. O modelo de replicação não restringe o tipo de aplicação a ser usada. Assim sendo, as réplicas são criadas quando predominam consultas aos objetos servidores (que fornecem serviços) do ambiente. Quando há predominância de atualizações no ambiente, o modelo descarta as réplicas que estão ociosas no sistema. No pior caso, quando há somente atualizações em um objeto replicado, o mesmo tem suas réplicas totalmente descartadas. Assim,

procura-se diminuir o custo de atualização das réplicas e tornar o modelo adaptativo a qualquer tipo de aplicação;

- **Integração com a ferramenta DOBuilder**. Esta consiste em um ambiente de programação visual para o desenvolvimento de aplicações com objetos distribuídos [MAL 00]. Desta forma, a integração do modelo ReMMoS a DOBuilder comportará replicação implícita dos objetos nas aplicações distribuídas. A integração dos dois trabalhos tem o objetivo de oferecer ao programador um ambiente mais completo de desenvolvimento e de execução.

4.2 Visão Geral do ReMMoS

A entidade computacional base do modelo é o objeto, isto é, a mobilidade e a replicação trabalham com a noção de objetos. Sendo assim, este modelo pode ser aplicado em ambientes que permitem mobilidade de objetos. O modelo está estruturado em duas partes principais: sistema de mobilidade e sistema de replicação. A figura 4.1 mostra a organização do modelo. Cada nodo do sistema (máquinas envolvidas na computação) possui esta organização.

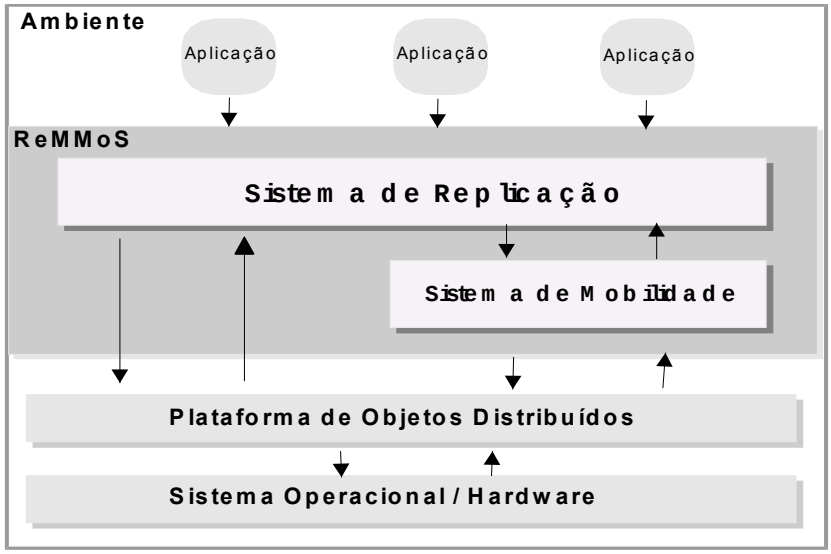

FIGURA 4.1 - Organização do Modelo

A replicação no ReMMoS é **implícita**. Desta forma, o sistema de replicação tem duas tarefas principais: **monitoração** e **gerência**. A monitoração diz respeito ao fato de que o sistema deve controlar as chamadas aos objetos **potencialmente replicáveis** e **objetos replicados**. A gerência é acionada no momento que o sistema de replicação detecta a necessidade de criação de réplica de um objeto.

A monitoração corresponde a objetos de uma classe serializada (somente objetos deste tipo de classe podem ser enviados via rede). Para obter esta informação, será necessário identificar, estaticamente, quais objetos pertencem a classes serializadas.

Desta forma, é acrescentado no código da aplicação uma chamada ao sistema de replicação sempre que for acessado um objeto deste tipo de classe. Somente acesso a objetos de classes serializadas são monitorados pelo sistema de replicação, pois somente estes podem ser replicados ou movidos. Mesmo um objeto que já possui réplicas associadas a ele, devem ser monitorado pois pode haver a necessidade de criação de novas réplicas.

A gerência é acionada no momento que o sistema de replicação detecta a necessidade de criação de réplica de um objeto, portanto torna-se um **objeto replicado**. Cada objeto deste tipo possui um módulo de gerência, que é responsável por manter a consistência do objeto e controlar dinamicamente a necessidade de descartar réplicas, conforme o comportamento da aplicação. Uma vez que um objeto replicado deixa de possuir réplicas, ele passa a ser considerado um objeto **não-replicado**, deixando de fazer parte do módulo de gerência.

O sistema de mobilidade é responsável por efetuar a mobilidade de objetos **replicados** e **não-replicados**. Como a mobilidade é **explícita**, objetos replicados, quando movidos, precisam ter alguns aspectos considerados, tais como a existência ou não de uma réplica no nodo destino. Para isto, é necessário que o sistema de replicação forneça algumas informações ao sistema de mobilidade. Já a mobilidade de objetos não-replicados não precisa ser monitorada pelo sistema de replicação, uma vez que não possui nenhuma réplica associada ao objeto.

4.2.1 Modelo de Replicação

O **modelo de replicação** é **implícito** e tem como objetivo diminuir o fluxo de mensagens no sistema através das réplicas. Além disso, permite paralelismo no acesso aos objetos. Desta forma, visa-se obter desempenho em aplicações distribuídas que usam mobilidade.

4.2.1.1 Considerações quanto à replicação na definição do modelo ReMMoS

Quando se trabalha com objetos, devido às suas características [RUM 96], vários aspectos devem ser pensados ao adaptar-se técnicas que já estão consolidadas através do uso em sistemas não orientados a objetos. Este é um fator importante uma vez que este trabalho abrange mobilidade e replicação em ambientes distribuídos usando objetos. Estudos sobre replicação foram apresentados no capítulo 3 deste trabalho.

Quanto à técnica de replicação a ser utilizada, observa-se ser favorável para o desenvolvimento deste trabalho à escolha da técnica Primário-*backup* [BUD 93], devido a cópia primária naturalmente garantir serializabilidade. Desta forma, não é preciso adotar medidas que garantam esta propriedade. Devido a isto, o gerenciamento da consistência das réplicas torna-se facilitado, uma vez que estas processam a informação na mesma ordem que o primário. Além disso, como a replicação é implícita, a aplicação conhece somente o endereço do primário, sendo o sistema de replicação responsável por prover a abstração necessária em caso de acesso às réplicas. Outro fator importante é que, devido à simplicidade da técnica, os recursos do sistema (como memória, por exemplo) são mais bem aproveitados, acarretando, de certa forma, em um menor *overhead* na execução. Como este trabalho preocupa-se com desempenho, esta questão torna-se importante.

Já na técnica de réplicas ativas [SCH 93], todas as réplicas processam uma requisição de escrita enviada para uma unidade replicada, o que ocasionaria um *overhead* de processamento, devido ao número de mensagens necessárias, ao processamento da requisição propriamente dita e ao gerenciamento da serializabilidade. A possibilidade de usar a técnica combinada foi eliminada devido a sua característica linear. Desta forma, uma operação de atualização poderia levar mais tempo para ser processada, pois as cópias não recebem a ordem de atualização ao mesmo tempo.

Quanto à consistência dos objetos replicados, optou-se por atualizar todas as cópias do objeto replicado quando incidir sobre este uma requisição de escrita. A escolha de atualizar as cópias em vez de invalidá-las ocorreu por várias razões. Uma delas é que, para atualizar um objeto, é necessário enviar a um método os parâmetros de atualização para que a operação seja realizada. Assim, possivelmente a mensagem de invalidação, que chamaria um método que invalidaria um objeto, tenha quase o mesmo custo que uma mensagem de atualização. Além disso, tornar válido um objeto não é tão trivial quanto tornar válido um dado (variável). Em um dado, geralmente é necessário somente alterar o seu valor. Para um objeto tornar-se consistente com as suas cópias, é necessário processar todas as operações realizadas enquanto o objeto estava invalidado. Isto porque a mudança de estado de um objeto pode acarretar a mudança de estado de outros objetos (objetos com objetos agregados, por exemplo).

Sendo assim, as transações ocorridas desde o momento que o objeto foi invalidado até a sua validação devem ser guardadas, o que acarretaria um *overhead*, além de gasto de memória. Uma opção seria descartar a cópia inválida e criar uma cópia atualizada do objeto. Mas se várias operações de escrita seguida de operações de leitura acontecerem, o *overhead* tende a ser grande, devido ao tráfego na rede. Como a replicação já acarreta em um gasto de recursos computacionais, optou-se por não onerar ainda mais os recursos computacionais necessários para a execução das aplicações. Somado aos fatores acima, tem-se o fato de que as réplicas do objeto replicado podem ser consultadas localmente. Isto poderia ocasionar vários pedidos de validação se as cópias são constantemente consultadas.

Para evitar que a replicação torne-se um gargalo em aplicações onde há momentos em que predomina operações de escrita, o ideal é que o gerenciamento da replicação seja feito de forma dinâmica, conforme o comportamento da aplicação [HUR 96]. No pior caso, o sistema ficaria somente com mobilidade, sem nenhum objeto replicado. Portanto, o modelo de replicação tende a ser adaptativo ao comportamento da aplicação, isto é, réplicas ociosas tendem a ser descartadas, ficando no sistema réplicas que estão sendo consultadas localmente e que possivelmente não ficarão invalidadas por muito tempo. Isto se assemelha a uma das vantagens da invalidação, onde a réplica que não é mais utilizada não necessita ser validada, permanecendo validadas somente as réplicas que estão sendo requisitadas [STU 98].

Devido às considerações acima, torna-se atraente o uso do protocolo de atualização [STU 98] neste trabalho. Além disso, a escolha da técnica de replicação a ser utilizada também favorece a escolha deste protocolo. Concluindo, acredita-se ser favorável tornar todas as réplicas do objeto replicado consistente quando incidir sobre o mesmo uma operação de atualização, evitando assim constantes pedidos de validação.

Em sistemas distribuídos como a Internet, o custo de comunicação pode ser alto devido aos tipos das redes e ao tráfego de comunicação entre os diversos computadores

espalhados pelo mundo. Desta forma, este trabalho utiliza a estratégia da replicação parcial. Os objetos são replicados dinamicamente conforme informações de acesso de leitura e escrita (*read*/*write*). Para obter esta razão, o sistema de replicação controla o tipo de acesso aos objetos. A verificação busca não ocasionar um custo adicional significativo ao sistema. Assim, os objetos são replicados conforme o comportamento da aplicação. Também por esta razão, um objeto replicado pode ter suas cópias descartadas.

4.2.1.2 Características principais do modelo de replicação

Devido às considerações acima colocadas, o modelo de replicação possui as seguintes características:

- **Utiliza uma estratégia de replicação parcial para criar as réplicas do objeto**. O objeto é replicado conforme seu padrão de acesso de leitura e escrita (*read/write*). Este padrão é verificado em tempo de compilação, onde são analisados os tipos dos métodos, e em tempo de execução, onde o tipo da operação é identificado;

- **Utiliza a técnica baseada em Primário-*backup* para gerenciar a consistência do objeto replicado**. Com esta técnica, a réplica primária do objeto replicado garante o estado consistente das réplicas, pois é ela quem trata as requisições de atualização (escrita) que incidem sobre o objeto replicado. A técnica é baseada em Primário-*backup* pois, consultas (leituras) em um objeto replicado podem ser processadas independentemente pelas réplicas deste objeto. Neste trabalho, esta escolha foi feita para aumentar o paralelismo no acesso aos objetos e diminuir do fluxo de mensagens no sistema;

- **Utiliza o protocolo de atualização para manter a consistência do objeto replicado quando este é atualizado**. A escolha do protocolo de atualização, em vez do protocolo de invalidação, ocorreu porque este modelo trabalha com replicação em objetos, permite incidência de leituras nas réplicas do objeto replicado e possui um modelo adaptativo ao perfil da aplicação. Maiores informações sobre estes protocolos podem ser obtidas no capítulo 3.

- **Atualização dinâmica do número de réplicas**. Quando operações de escrita incidem sobre os objetos replicado, suas cópias ociosas (que não estão processando leitura local) são descartadas. Desta forma, o modelo é adaptativo ao perfil da aplicação.

Muitos objetos possuem objetos agregados, pois as variáveis de um objeto podem ser também objetos, fazendo com que o estado de um objeto envolva outros objetos. No modelo de objetos, isto corresponde à agregação, o que ocasiona uma dependência inter-objetos. Neste caso, uma chamada ao método de um objeto que possui objetos agregados, pode ocasionar a chamada de outro objeto que está agregado a ele. Na maioria das vezes, objetos que fornecem serviços (objetos servidores) tem agregados a si outros objetos, que dão suporte aos serviços oferecidos pelo objeto.

Neste caso, o objeto e seus agregados são replicados, pois entende-se que os objetos agregados serão requisitados constantemente pelo objeto, para realização dos seus serviços. A plataforma de replicação deve garantir a replicação do objeto e de seus

agregados. Por isto, os objetos a serem replicados devem estar serializados. Desta forma, é possível transmitir via rede tanto o objeto como seus agregados.

4.2.2 Modelo de Mobilidade

Embora a mobilidade permita que a computação migre para onde há maior solicitação por parte de um cliente, não possibilita o acesso local de vários clientes (em diferentes máquinas) ao mesmo objeto. Em sistemas distribuídos, mesmo que um objeto migre para um determinado cliente, outros clientes, que também realizam solicitações ao objeto, só podem acessá-lo remotamente.

Muitos trabalhos consideram migração e replicação como sendo um relacionamento simbiótico [HUR 96]. A replicação pode ser vista como uma migração onde a unidade de execução não é descartada no nodo origem. Assim sendo, os dois mecanismos são muito parecidos mas precisam de gerenciamentos diferentes. Por exemplo, na migração, é necessário gerenciar as ligações da unidade de execução com o sistema origem e destino, isto é, a referência entre a unidade de execução e o sistema nunca devem ser perdidas. Em replicação, além disto, é necessário gerenciar a consistência das cópias de uma unidade de execução através do sistema.

Este trabalho utiliza o tipo de **mobilidade forte** [FUG 98] para mover os objetos através do sistema. Como foi visto no capítulo 2, a mobilidade forte assemelha-se bastante da migração e da replicação de objetos em sistemas distribuídos.

O **modelo de mobilidade** tem como objetivo permitir acesso local a objetos servidores que trocam muitas mensagens com um determinado cliente. Neste trabalho, estes objetos são chamados de **objetos não-replicados**. Desta forma, o modelo diferencia objetos não-replicados e objetos replicados. Como a mobilidade é **explícita**, objetos não-replicados são movidos conforme decisão do programador, isto é, o programador decide qual objeto mover e qual o destino deste objeto. Neste caso, o modelo de replicação não interfere na mobilidade do objeto se este não é replicado. A seção 4.6.1 mostra como ocorre a mobilidade de objetos não-replicados.

Objetos replicados recebem um tratamento especial quanto à mobilidade, pois sendo a replicação implícita, o programador pode mover um objeto que já está replicado. Neste caso, se uma réplica do objeto replicado não existir no destino, a mobilidade ocorre de forma semelhante aos objetos não-replicados. Se existir uma réplica do objeto, a mobilidade é considerada pronta. O Capítulo 4 explica como ocorre a mobilidade em objetos replicados.

A plataforma de mobilidade deve garantir a mobilidade de objetos que possuem objetos agregados. Para isto, como na replicação, os objetos a serem movidos devem estar serializados. Desta forma, tanto o objeto como os objetos agregados a ele podem ser transmitidos via rede.

4.3 Modelo de Replicação

Cada objeto replicado é responsável por gerenciar suas réplicas. Assim, a aplicação conhece explicitamente somente o endereço do objeto replicado. Este deve

direcionar as mensagens para as suas réplicas, tornando a replicação dos objetos transparente para as aplicações. Em caso de consulta ao objeto replicado, este deve determinar se a requisição vai ser remota ou local. Em caso de atualização, o objeto replicado deve controlar a escrita nas réplicas. Os objetos encontram-se na plataforma de objetos distribuídos. A figura 4.2 exemplifica a organização de um objeto replicado.

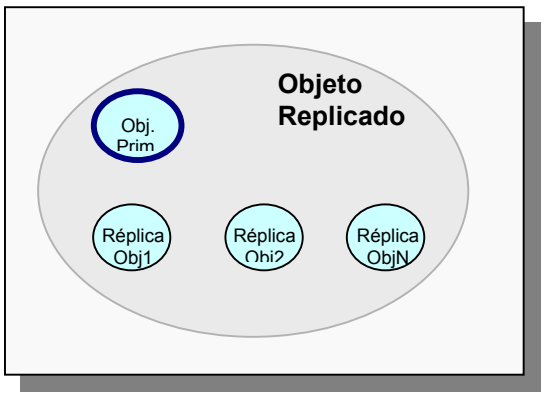

FIGURA 4.2 - Organização de um Objeto Replicado

4.3.1 Criação das Réplicas

A estratégia utilizada para a criação de um objeto replicado é a estratégia de replicação parcial, onde as réplicas são criadas somente quando necessário. Neste caso, um objeto pode ser replicado seguindo a heurística abaixo:

- quando o número de operações de leitura/escrita que incidem sobre um objeto potencialmente replicável alcançar a proporção de 3 para 1, isto é: **3 leituras** locais / **1 escrita** total (ou somente 3 leituraslocais. Esta medida está baseada em estudos feitos em [BAL92b]. Uma **leitura** local corresponde a consultas consecutivas realizadas no objeto por um determinado cliente. Uma **escrita** corresponde à atualização global de algum atributo de classe/objeto.

O sistema de replicação controla o acesso a todos os objetos **potencialmente replicáveis**. Estes correspondem a objetos de uma classe serializada, pois somente os objetos pertencentes a estas classes podem ser replicados ou movidos.

A verificação para determinar a criação de um objeto replicado é realizada cada vez que houver um acesso a um objeto potencialmente replicável por um cliente remoto. Desta forma, a cada acesso remoto a este tipo de objeto, o sistema de replicação controla o tipo de acesso ao objeto, qual cliente o está acessando e qual a origem deste acesso. Quando os acessos alcançarem a heurística acima, o sistema de replicação cria uma réplica do objeto na máquina cliente. Após isto, o objeto torna-se um **objeto replicado**. Assim, todo o objeto é replicado. O seguinte algoritmo exemplifica estas questões:

Nome: monitoraCriaObj;

Entrada: objeto, msg;
Saída:
Método:

```
início
   Verifica NºAcessoRemoto para leitura e para escrita;
   se NºAcessoRemoto >= condição de criação então
                Chama criaRéplica(objeto, destino);
   fim-se

fim
```

Para criar um objeto replicado é usado um método semelhante ao usado quando um objeto é movido, visto serem a mobilidade e a replicação simbióticas [HAC 89]. Desta forma, a criação de um **objeto replicado** consiste na seguinte seqüência de passos:

1. O objeto recebe do sistema de replicação uma mensagem indicando que uma réplica do mesmo deve ser criada;

2. O objeto é bloqueado e todas as mensagens que o objeto está processando devem ser completadas. Novas mensagens que chegam devem ser suspensas e esperar até que o objeto seja liberado;

3. O objeto e seus objetos agregados são copiados para o nodo destino;

4. O método responsável pela replicação retorna se a mesma ocorreu com sucesso ou não, isto é, se foi ou não criada no nodo destino.

Se a replicação ocorreu com sucesso:

- o estado do objeto é atualizado para **objeto replicado** se é a primeira réplica a ser criada;

- um grupo formado pela nova réplica do objeto é criado (técnica primário-*backup*). O objeto replicado é responsável pela gerência do grupo;

Se a replicação não ocorreu com sucesso:

- uma exceção deve ser gerada. Neste caso, é necessário desbloquear o objeto no nodo origem, permitindo que ele continue executando e recebendo solicitações.

5. Mensagens para o objeto, que foram suspensas, são liberadas.

A partir deste momento, o acesso à réplica deve ser feita localmente pelo cliente. Além disto, no momento que a fila de mensagens é liberada na origem, o sistema de replicação é responsável por encaminhar aos objetos **potencialmente replicáveis**, que encontram-se na máquina destino, o acesso local ao objeto replicado. A seguir é

apresentado o algoritmo para controlar a criação de um objeto replicado e de suas réplicas.

Nome: criaRéplica;

Entrada: objeto, destino;
Saída: objeto replicado, exceção, sucesso;
Método:

```
início
      Objeto recebe mensagem: Cria_réplica_OK;
      Bloquear objeto;
      Criar fila de mensagens;
      se objeto não existe no destino então
            Inicia criaRéplicaObj(objeto,destino);
            se criaRéplicaObj = Sucesso então
                  se NºRéplica = 0 então
                        EstadoObjeto = ObjetoReplicado;
                  fim-se
                  Chama gerência PrimárioBackup;
            senão
                  Gera exceção;
            fim-se
      senão
            encaminha acesso local
      fim-se
      Desbloqueia objeto na origem;
      Libera fila de mensagens;
fim
```

4.3.2 Gerenciamento e Atualização das réplicas

Na técnica de Primário-*backup*, todas as requisições, até mesmo leitura, devem ser processadas através da réplica primária do objeto replicado. Pasin [PAS 00] comenta que alguns autores, propõe uma alteração na técnica de Primário-*backup*. Neste caso, somente requisições para escrita no objeto são processadas pela cópia primária do objeto. Requisições para leitura no objeto podem ser processadas pelas réplicas locais. A técnica com esta característica é então chamada de **baseada** em Primário-*backup*. Permitir acesso local para consulta nas réplicas, acarreta em paralelismo no acesso aos objetos, um dos objetivos deste trabalho. Assim, esta técnica é usada neste trabalho para gerenciar o objeto replicado.

As réplicas de um objeto são atualizadas quando uma operação de escrita incide sobre o objeto replicado. A atualização é processada **somente** pela réplica primária do objeto replicado. Desta forma, é garantida a consistência das réplicas, pois somente uma delas é responsável por gerenciar a atualização.

4.3.2.1 Operações de consulta e atualização

Como colocado anteriormente, somente objetos serializados podem ser replicados. Desta forma, a identificação de operações de consulta ou atualização ocorre somente em objetos deste tipo. Para isto, o código da aplicação é modificado, de forma que seja identificado se um método tem função de consulta ou de atualização.

Se o método tem alguma função de atualização ele é considerado um **método de escrita**. Neste caso, deve ser incluído no código uma chamada ao sistema de replicação que controla requisições de atualização. Se o método tem somente função de consulta, ele é considerado um **método de leitura**.

A seção 4.6 especifica como deve ser a análise das características da aplicação para suporte à gerência de replicação e mobilidade.

4.3.2.2 Consulta em um Objeto Replicado

Uma requisição de leitura a um objeto replicado pode ser tratada de duas formas:

Uma requisição de leitura a um objeto replicado pode ser tratada de duas formas:

a)O cliente **não possui** uma cópia local do objeto replicado - o acesso para leitura deve ser feito remotamente, diretamente no objeto replicado. Neste caso, o sistema de replicação é encarregado de monitorar as requisições ao objeto replicado, pois pode haver necessidade de criar uma cópia do objeto na máquina cliente;

b)O cliente **possui** uma cópia local do objeto - a leitura ocorre localmente. Neste caso, o sistema de replicação é encarregado de encaminhar o processamento da consulta para a réplica local. Novas consultas serão realizadas diretamente na réplica local até que uma exceção (falha) ou solicitação de escrita ocorra e o cliente tenha que reportar-se remotamente ao objeto replicado. Além disso, o número de leituras na réplica deve ser controlado, para auxiliar o processo de descarte da réplica.

Nome: controlReadObjReplica;
Entrada: parâmetros de consulta (PR);
Saída: parâmetros consultados, excessão;
Método:

```
início
      Verifica RéplicaLocal;
      se RéplicaLocal = Não_Existe então
            acesso remoto;
            chama monitoramento para o objeto;
      senão
            acesso local;
            chama gerência de réplicas;
      fim-se
fim
```

Uma exceção pode ser gerada por falha de acesso ao objeto (o mesmo pode ter sido descartado, por exemplo). Neste caso, o cliente só saberá que o objeto não encontra-se localmente no seu próximo acesso.

Quando uma réplica do objeto replicado recebe uma requisição de escrita, esta requisição deve ser tratada de forma especial, de modo a manter a consistência do objeto replicado. A seção seguinte aborda este assunto.

4.3.2.3 Atualização em um Objeto Replicado

A figura 4.3 apresenta a técnica de Primário-*backup* quando uma requisição de atualização incide sobre um objeto replicado.

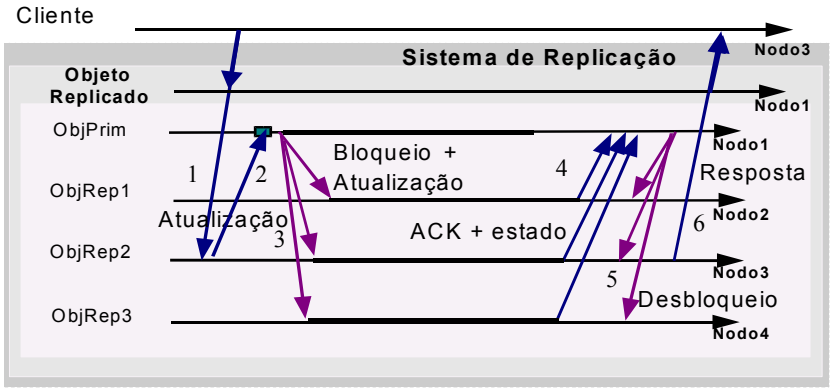

FIGURA 4.3 - Réplica do objeto replicado recebendo pedido de atualização

Quando uma réplica do objeto replicado recebe um pedido de atualização, a mesma deve ser bloqueada e a requisição de atualização encaminhada à réplica primária. A partir deste momento o protocolo de atualização é controlado pelo objeto replicado, como destacado a seguir:

1. ao receber a mensagem, a réplica primária do objeto replicado envia para as outras réplicas uma mensagem contendo uma ordem de bloqueio das mesmas e os parâmetros de atualização;

2. terminada a operação de atualização, as réplicas mandam uma mensagem de retorno à réplica primária, indicando que seus estados já encontram-se consistentes;

3. à réplica primária manda uma mensagem de desbloqueio para as demais réplicas assim que receber a mensagem de retorno de todas elas, indicando assim que o objeto replicado já está atualizado;

4. ao ser desbloqueada, a réplica do objeto que recebeu a solicitação de atualização por parte do cliente, responde ao mesmo o seu pedido.

Quando uma requisição de atualização parte de um cliente que está no mesmo endereço do objeto replicado, portanto no mesmo endereço da réplica primária o procedimento de atualização é semelhante. Neste caso, não é necessário informar o pedido de atualização à cópia primária (mensagem número 2), pois o mesmo foi recebido localmente por ela.

É importante destacar que não é permitida concorrência entre criação e atualização de um objeto replicado, de forma a manter a consistência do objeto replicado.

4.3.3 Controle dinâmico do número de réplicas

Muitas aplicações são caracterizadas por predominância de operações de consultas, atualizações, ou consultas/atualizações. A replicação pode ser explorada nestes três tipos de aplicações. A replicação em aplicações onde predominam operações de atualização não é indicada para obter desempenho, devido ao custo de atualização das réplicas, mas pode ser usado para prover tolerância à falhas, sem a preocupação de perda desempenho. Já em aplicações onde predominam operações de consulta ou consulta/atualização, a replicação pode ser usada para obter desempenho.

Com o objetivo de não restringir a característica da aplicação usada, o modelo de replicação apresentado neste trabalho é adaptativo ao tipo de aplicação. Para isto, conforme o comportamento da aplicação, um objeto replicado pode ter suas réplicas criadas ou descartadas.

Desta forma, quando a aplicação se comporta de modo a predominar consultas, as réplicas são criadas. Se predominar atualizações, as réplicas ociosas do objeto vão sendo descartadas, para diminuir o custo de atualização do objeto replicado, uma vez que este custo cresce à medida que aumentam as réplicas [PAS 00]. Entende-se por réplica ociosa a que não está processando, por um determinado período de tempo, consultas locais. No pior caso, se predominar no sistema atualizações sobre os objetos replicados, estes tornam-se objetos não-replicados, pois não existirão mais réplicas associadas a eles.

O método utilizado neste trabalho para determinar se uma réplica será descartada é o modelo LRU (*Least Recently Used*). Este método é usado principalmente na área de Sistemas Operacionais, como parte da gerência de blocos de memória virtual e cache [TAN 95]. O método seleciona para substituição um bloco de memória menos usado pelo processador, isto é, o bloco para o qual transcorreu o maior intervalo de tempo desde o último acesso. Desta forma, baseia-se na heurística de que um bloco que não foi referenciado durante um determinado período de tempo, não o será em um futuro próximo.

Conforme [WEB 00], a dificuldade deste método é manter uma heurística dos blocos menos recentemente usados. Uma solução possível é associar contadores a cada bloco da memória principal. Toda vez que um bloco é referenciado, seu contador é atualizado para o maior valor positivo. De tempos em tempos, todos os contadores são decrementados. O bloco menos recentemente usado é aquele cujo contador apresentar o menor número.

4.3.3.1 Considerações quanto à escolha do método

Não existem muitos trabalhos onde as réplicas de um objeto são criadas e descartadas dinamicamente. O trabalho de [HAC 89] apresenta um estudo em migração e replicação dinâmica de arquivos, usando um algoritmo distribuído que leva em conta a freqüência de acesso ao arquivo e seu tamanho, entre outros parâmetros obtidos do

sistema. Alguns parâmetros do sistema devem ser fornecidos pelo programador. Além disso, a decisão de manter ou não uma réplica não é muito trivial, exigindo do sistema o processamento de fórmulas que combinam as informações obtidas pelo sistema. O trabalho não preocupa-se com desempenho.

Desta forma, como este trabalho preocupa-se com o desempenho da aplicação, optou-se por um algoritmo simplificado, de modo que a decisão de manter ou não uma réplica do objeto seja obtida aproveitando-se outras mensagens necessárias na manutenção da consistência das réplicas. Assim, os métodos utilizados neste trabalho para determinar quais réplicas devem ser descartadas é baseado no algoritmo LRU. Assim, a réplica que está ociosa por mais tempo é descartada à medida que sucessivas operações de atualização incidem sobre o objeto replicado.

4.3.3.2 Funcionamento

Cada réplica do objeto replicado possui associada a si um **contador de acesso local**. Este contador é incrementado a cada solicitação de **consulta local** à réplica do objeto replicado. Esta consulta considera todas as chamadas de objetos replicados ou potencialmente replicáveis, em um determinado nodo, para aquele objeto. Isto é, se uma réplica do objeto replicado já foi criada no nodo cliente, uma nova réplica, obviamente, não necessita ser criada. Basta suas requisições de leitura serem tratadas localmente.

Como é vantagem uma réplica existir somente se está sendo acessada localmente, o uso do contador visa controlar se a réplica está ociosa ou não. Assim, a cada operação de atualização, o contador de acesso local é verificado. Após um determinado período de tempo (heurística a ser definida por experimentação), a réplica que apresenta o contador com o menor número de acessos é descartada.

Desta forma, permanecem no sistema somente réplicas que estejam localmente ativas, evitando a atualização desnecessária de réplicas que não estão sendo localmente usadas. Caso todas as réplicas do objeto replicado sejam descartadas, este objeto será considerado **não-replicado**. Desta forma, requisições para este objeto serão remotas, e não mais locais. O algoritmo abaixo exemplifica com ocorre o controle da atualização do objeto replicado, além do controle dinâmico do seu número de réplicas.

Nome: atualizaObjtReplicado
Entrada: parâmetros de gerência, parâmetros de atualização (PA);
Saída: atualização, exceção, sucesso;
Método:

```
início
   chama protocolo de atualização de réplicas;
fim
```

Nome: descartaRéplica
Entrada: parâmetros de gerência

Saída: exceção, sucesso;
Método:

```
   início
      chama ControlaNºconsultaLocal;
      se Nºconsultalocal = constante então
            descartar réplica;
            estado réplica = descatado;
      fim-se
fim
```

4.4 Modelo de Mobilidade

O modelo de mobilidade diferencia os objetos em dois tipos: **objetos não-replicados** e **objetos replicados**. Esta diferenciação é necessária pois a mobilidade e a replicação ocorrem de forma diferente conforme o tipo de objeto que está sendo tratado.

A mobilidade é explícita, isto é, o desenvolvedor deve especificar durante o desenvolvimento da aplicação **qual objeto** servidor será movido e qual o seu **destino**. Para isto, é desejável que seja usada uma abstração (método especial) de forma a facilitar para o programador o uso da mobilidade. Desta forma, o programador especifica somente qual objeto será movido e qual o seu destino, e a mobilidade é tratada pelo sistema de mobilidade. Destaca-se que um pedido de mobilidade ocasiona na mobilidade de todo o servidor e não apenas de um objeto que o compõe.

Todo objeto movido deve ter um *proxy* associado a ele. Este é criado para representar o objeto. Assim, o objeto pode mover-se através do sistema mas isto fica transparente para a aplicação, pois esta comunica-se com o *proxy* representando o objeto. Além disso, assim como os objetos replicáveis, os objetos movidos devem estar aptos a serem transmitidos e recebidos através da rede. Em linguagens de programação, isto quer dizer que a classe que do objeto deve ser serealizável. Este trabalho considera que os objetos que serão movidos estão serealizados. Neste caso, o ambiente de programação deve garantir esta propriedade.

4.4.1 Mobilidade de Objetos Não-replicados

Devido às características do modelo de mobilidade utilizado no sistema Voyager [OBJ 00], este foi escolhido como plataforma para a mobilidade de objetos não-replicados descrita neste trabalho. O modelo utilizado em Voyager é explícito, simples e eficiente, integrando vários conceitos consolidados quando se trata de mobilidade em sistemas distribuídos, tais como o uso de *proxies*; possibilidade de desenvolvimento usando mobilidade forte e fraca; uso de um método especial para mover um objeto que encapsula a mobilidade, tratando aspectos como objetos agregados e sucesso ou não da mobilidade do objeto; além vários outros serviços para suporte à mobilidade. Desta forma, o modelo de mobilidade de Voyager retrata muitas das características desejáveis em uma plataforma de mobilidade para dar suporte ao modelo ReMMoS.

Assim, a cada requisição de mobilidade, se o objeto é não-replicado, isto é, não possui réplicas, o sistema de mobilidade usado (no caso do ReMMoS o Voyager) é

responsável pela sua mobilidade. Ressalta-se que qualquer sistema de mobilidade de objetos pode ser usado, desde que este sistema tenha um método especial para tratamento da mobilidade. Basta fornecer ao ReMMoS o método que controla a mobilidade, para que o mesmo possa efetuar o controle da mobilidade dos objetos replicados.

4.4.2 Mobilidade de Objetos Replicados

A mobilidade de um objeto replicado requer uma atenção especial. Como a mobilidade é explícita e a replicação implícita, o desenvolvedor pode mover um objeto que está replicado.

Quando um objeto replicado é movido, sua referência deve ser alterada para a nova localização. Desta forma, procura-se manter a transparência quanto à replicação. Além disso, como o modelo de replicação é adaptativo, quando ocorrem muitas atualizações, somente o "objeto original" sobrevive no sistema.

Ao receber uma requisição de mobilidade, o objeto replicado é responsável por controlar se já existe uma réplica no nodo destino. Em caso de mobilidade de objetos replicados, duas possibilidades podem ocorrer:

A) Existe uma réplica do objeto no nodo destino: neste caso, não há mobilidade, há somente troca de referência para o objeto replicado, da seguinte forma:

1. O objeto é bloqueado e todas as mensagens que o objeto está processando devem ser completadas. Novas mensagens que chegam devem ser suspensas;

2. O objeto replicado verifica que existe uma réplica no endereço destino;

3. A réplica no endereço destino torna-se a nova referência para o objeto replicado (as demais réplicas são informadas na próxima requisição de atualização no objeto replicado);

4. o objeto é destruído no endereço antigo e um *proxy* para o objeto é criado neste endereço. Este *proxy* é necessário para encaminhar as mensagens que o objeto recebe no endereço antigo para o novo endereço, até que sua referência seja atualizada;

5. O objeto é desbloqueado e mensagens para o objeto, que foram suspensas, são liberadas.

B) Não existe uma réplica do objeto no nodo destino: neste caso, o objeto replicado move-se para o nodo destino utilizando um protocolo semelhante ao usado em objetos não-replicados. O protocolo é descrito a seguir:

1. O objeto é bloqueado e todas as mensagens que o objeto está processando devem ser completadas. Novas mensagens que chegam devem ser suspensas;

2. O objeto replicado verifica que não há uma réplica no endereço destino;

3. O objeto replicado é movido para o nodo destino;

4. É controlado se a mobilidade ocorreu com sucesso ou não.

   Se ocorreu com sucesso:

   - a réplica no endereço destino torna-se a nova referência para o objeto replicado (as demais réplicas são informadas na próxima requisição de atualização no objeto replicado);

   - o objeto é destruído no endereço antigo e um *proxy* para o objeto é criado neste endereço;

   Se a mobilidade não ocorreu com sucesso:

   - uma exceção deve ser gerada e o objeto deve ser restaurado no endereço antigo. Neste caso, como o objeto ainda continua existindo no endereço antigo, é necessário desbloqueá-lo, permitindo que ele continue processando as mensagens que estavam bloqueadas.

6. Mensagens para o objeto, que foram suspensas, são liberadas. O *proxy* representando o objeto é responsável por enviar a mensagem para o objeto na nova localização, até que as referências para o objeto sejam atualizadas.

Ao receber uma requisição de mobilidade, o objeto replicado tem sua referência atualizada para o endereço destino da mobilidade. Em um nível mais baixo de abstração, isto quer dizer que a referência para o objeto primário é alterada. A razão principal para alterar a referência do objeto replicado diz respeito à transparência para o sistema. Como a mobilidade é explícita, o conhece programador a localização do objeto antes e depois da mobilidade.

Para evitar que o sistema de replicação necessite monitorar a localização do objeto para o programador e a localização real do objeto, optou-se por "mover" o primário. Além disso, se todas as réplicas do objeto replicado forem descartadas, este objeto será considerado não-replicado. Desta forma, é necessário que permaneça no sistema a cópia original, isto é, permaneça no sistema o objeto criado pela aplicação do usuário. A seguir, será apresentado o algoritmo para implementar a mobilidade de um objeto replicado não-replicado e de um objeto replicado, existindo ou não uma réplica do objeto no nodo destino.

Nome: ControlaMobiObj;

Entrada: objeto, destino;
Saída: objeto replicado/movido, exceção, sucesso;
Método:

```
início
   Bloquear objeto;
   Criar fila de mensagens;
   Verifica EstadoObjeto;
   se EstadoObjeto <> ObjetoReplicado então
      MoveObjeto(objeto, destino);
   senão
```

```
            Verifica ExisteRéplicaDestino;
            se ExisteRéplicaDestino = verdadeiro então
               tipoRéplica = primária;
            senão
               MoveObjeto(objeto, destino);
               se MoveObjeto = sucesso então
                     tipoRéplica = primária;
                     DestróiObjeto(objeto, origem);
               Senão
                     Desbloqueia objeto na origem;
               fim-se
            fim-se
            Desbloqueia objeto no destino;
            Libera fila de mensagens;
         fim-se
      fim
```

4.4.2.1 Atualização do endereço do objeto replicado em caso de mobilidade

Como o objeto replicado pode alterar seu endereço por motivos de mobilidade, é necessário que as aplicações e as réplicas do objeto replicado sejam informadas da nova localização. No caso das aplicações, o sistema de replicação é responsável por encaminhar mensagens do endereço antigo para o novo, através de um *proxy*, até que a referência para a nova localização do objeto seja atualizada. No caso das réplicas do objeto replicado, o protocolo de atualização é responsável por informar também a nova localização do objeto replicado (primário). Sempre que o objeto replicado mudar seu endereço por motivo de mobilidade, a próxima alteração neste objeto atualiza seu endereço nas suas réplicas.

Se o pedido de atualização incidir sobre uma réplica do objeto replicado antes da mesma ter sido informada sobre a nova localização do objeto, o sistema de replicação, através do *proxy* do objeto, é responsável por encaminhar a requisição até a nova localização do objeto.

4.5 Análise das características da aplicação para suporte à gerencia de replicação e mobilidade

Para controlar o acesso dos objetos clientes, é necessário inferir no código destes chamadas ao sistema de replicação. Desta forma, o código do cliente deve ser pré-processado, antes de ser executado. O pré-processamento gera um código cliente alterado, com as chamadas ao ambiente de execução do ReMMoS. A figura 4.4 representa esta abordagem.

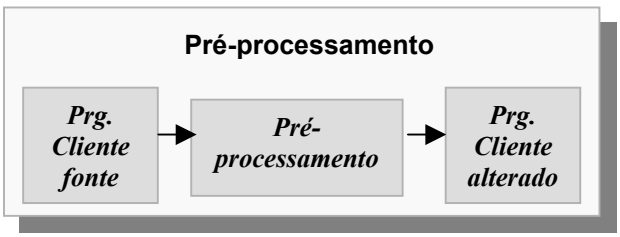

FIGURA 4.4 – Pré-processamento do código cliente

Para dar suporte à replicação implícita, é necessário um **analisador de código** para a linguagem suportada pelo ReMMoS (capítulo 5) que infira informações que serão usadas durante a execução. Para a realização deste trabalho, é importante um analisador de código Java (linguagem suportada pelo ReMMoS) para que o mesmo analise e modifique o código da aplicação. Vale ressaltar que o **código original do usuário deve ser mantido**, visto que a replicação é transparente para o mesmo. Desta forma, é necessário que esteja sempre disponível para o usuário o código original.

Para criação das réplicas de um objeto replicado, é necessário que o sistema de replicação identifique os objetos potencialmente replicáveis. Estes correspondem a objetos de uma classe serializada. Para obter esta informação, é necessário identificar, estaticamente, quais objetos pertencem a classes serializadas.

Além disto, é necessário que o sistema de replicação identifique as operações de acesso de objetos serializáveis ao objeto servidor. Desta forma, a cada acesso remoto a este tipo de objeto, o sistema de replicação controla se o acesso é de leitura ou de escrita. Este tipo de controle é feito tanto em objetos potencialmente replicáveis quanto em objetos replicados. Esta informação é usada para criar, atualizar e descartar as réplicas, e é obtida pelo analisador através da analise dos métodos.

Existem trabalhos, como [BAL 92], onde o controle de acesso a métodos de escrita e leitura é realizado. Mas neste trabalho, implementado utilizando a linguagem Orca, não é necessário um analisador de código, visto que a própria linguagem, através de métodos explicitamente de leitura e de escrita identificados estaticamente, dá suporte ao controle de acesso em tempo de execução.

Desta forma, como a linguagem suportada pelo ReMMoS não possui este suporte, é necessário que o analisador de código analise cada método para identificá-lo. Isto é feito da seguinte forma:

- Se o código do método contém o operador de atribuição (=) e esta atribuição altera variáveis além do escopo do método, então este método é considerado um método de escrita. Assim, é inserido, antes da chamada do referido método, uma chamada ao sistema de replicação;

- Se o método não contém o operador acima ou altera alguma variável dentro do escopo do método, é considerado um método de leitura. Assim, também deve ser inserida uma chamada ao sistema de replicação que indique que o método é de leitura.

Observa-se uma visão simplista para considerar o tipo do método. Trabalhos de análise estática, como o que está em desenvolvimento pela aluna Silvana Azevedo [AZE 00], podem aperfeiçoar a determinação do tipo do método. O referido trabalho, em desenvolvimento no Grupo de Processamento Paralelo e Distribuído do Instituto de Informática da UFRGS, está sendo realizado com o objetivo, dentre outros, de identificar métodos de leitura e/ou escrita, além de inferir informações sobre as características dos objetos, tais tipo do objeto, seu tamanho, comportamento do objeto com relação a outros objetos, etc. Desta forma, os resultados deste trabalho, assim que possível, serão incorporados ao modelo ReMMoS, podendo trazer maior precisão no

monitoramento dos acessos. Destaca-se aqui, a integração de trabalhos realizados pelo Grupo de Processamento Paralelo e Distribuído do Instituto de Informática.

Com relação à mobilidade, toda vez que o analisador encontrar no código uma chamada a um método que move um objeto, deve ser inserida uma chamada ao método de controle de mobilidade do ReMMos (que possui controle de mobilidade de objetos replicados). Desta forma, o método do cliente é substituído pelo método do ReMMoS.

4.6 Integração com a ferramenta DOBuilder

É cada vez mais freqüente, o uso de ferramentas de programação visual para o desenvolvimento de aplicações. Segundo Malacarne [MAL 00], apesar de a maioria dos ambientes visuais afirmarem o apoio ao desenvolvimento de aplicações distribuídas, pouco existe nessa área e a ênfase maior da programação visual se dá mesmo na programação paralela. Desta forma, encontra-se em desenvolvimento no Instituto de Informática da UFRGS, o projeto e implementação de uma ferramenta de programação visual para o desenvolvimento de aplicações com objetos distribuídos em Java denominada **DOBuilder** [MAL 00]. Esta ferramenta está sendo modelada e implementada pelo aluno Juliano Malacarne, sendo a programação baseada na manipulação de componentes, com geração de código em Java e execução em ambiente distribuído. Além disto, suporta o desenvolvimento de aplicações usando o suporte à distribuição de objetos de Voyager [OBJ 00].

4.6.1 Características principais da ferramenta

Além das características descritas acima, [MAL 00] destaca que a ferramenta busca a facilidade de uso e a oferta de recursos ao usuário que o auxiliem no desenvolvimento do *software*: reutilização, modularização, extensibilidade e geração automática de código.

Como linguagem de programação visual, a ferramenta adota um modelo onde a aplicação do usuário é representada através de um grafo dirigido, no qual os objetos distribuídos são representados por nodos e os relacionamentos entre eles são visualizados explicitamente por meio de arcos que conectam tais nodos. Há vários tipos de relacionamentos possíveis entre diferentes nodos, sendo que os principais são a invocação remota de métodos e a comunicação por meio de mensagens explícitas, estas conduzidas através das portas de comunicações declaradas nos objetos distribuídas.

A especificação da aplicação contém diferentes tipos de objetos. Os objetos distribuídos propriamente ditos são os nodos do grafo no programa visual e executam fisicamente em diferentes espaços de memória. A comunicação, entretanto, é implementada por outros objetos acoplados a esses objetos distribuídos. Tais objetos especiais são os componentes de programação distribuída, representados principalmente pelas portas de comunicação e referências remotas para chamadas de métodos. Essa abordagem de componentes permite que a comunicação seja implementada de diferentes maneiras, possibilitando ainda que seja alterada e personalizada de acordo com as necessidades de cada aplicação. Para tanto, basta que um novo tipo de componente seja implementado.

A comunicação entre o componente e o objeto que o contém é feita através de eventos. Todo o comportamento de um objeto é controlado pelos eventos que ele gera e recebe. O emprego do modelo de eventos é útil também como forma de aumentar a concorrência, pois um objeto poderá realizar outras tarefas enquanto espera a mensagem, não havendo a necessidade de executar sempre uma chamada de recebimento de mensagem que bloqueie o processamento antes do tempo necessário. A notificação de erros e falhas na comunicação também é facilitada através deste esquema.

O modelo de programação apresenta outros elementos além de objetos distribuídos e portas de comunicação. Entre outros, estão disponíveis também *locks*, dados globais, portas de chamadas de serviço, serviços e servidores virtuais. Todos eles estão organizados numa hierarquia de objetos e definidos em termos de propriedades, métodos e eventos. A possibilidade de extensão do ambiente é uma conseqüência natural da modularização, permitindo que novos componentes podem ser adicionados à ferramenta conforme a necessidade.

Maiores informações sobre a ferramenta DOBuilder pode ser encontrada na dissertação de Juliano Malacarne [MAL 00].

4.6.2 DOBuilder e ReMMoS

Conforme destacado, a ferramenta DOBuilder preocupa-se com a programação de aplicações em objetos distribuídos. O ambiente de execução fica a cargo de Java [SUN 00] e Voyager [OBJ 00]. Desta forma, o código gerado pela programação visual executa em uma ou em ambas as plataformas, sem problemas para o uso da replicação implícita, uma vez que os dois ambientes acima são a base do ambiente de execução do ReMMoS, conforme será destacado no capítulo 5.

Como a DOBuilder não possui suporte direto à replicação de objetos, a integração do ReMMoS com a ferramenta permitirá que o ambiente de execução tradicional para objetos distribuídos seja substituído por outro mais completo, que ofereça replicação de objetos transparente para o usuário com a finalidade de aumentar o desempenho. Neste caso, o ambiente de execução tradicional será substituído pelo ambiente de execução ReMMoS.

Como a replicação no ReMMoS é implícita, o ambiente de execução suportado pela DOBuilder deve ser avisado para suportar replicação de objetos de forma transparente para o usuário. Assim, a DOBuilder deve prover, na sua estrutura de menus, na janela principal, a opção do programador especificar se o ambiente de execução suportará replicação. Para manter o modelo atual da ferramenta, as aplicações utilizam, por *default*, o ambiente de execução sem suporte à replicação.

No momento que o programador do ambiente DOBuilder + ReMMoS especificar que o ambiente de execução suportará replicação, a biblioteca ReMMoS será automaticamente disponível para ser usada pelo ambiente de execução. Desta forma, no momento da execução, o código fonte do cliente sofre alterações (conforme destacado no item 4.6). Estas alterações garantem o uso do ambiente de execução do ReMMoS.

O ReMMoS garante que o que código original do usuário seja mantido, uma vez que a representação visual dos programas permanece a mesma e os objetos não sofrem alterações. Ressalta-se que, embora o programador especifique o uso da replicação implícita na execução da sua aplicação, a gerência da replicação não é preocupação do programador, conforme destacado nos objetivos do ReMMoS.

A princípio, como a replicação é implícita, não há dificuldades em integrar os dois trabalhos, pois não exige a criação de uma representação visual, isto é, um novo tipo de nodo básico, que implemente por si só a replicação. Assim, a replicação passa a ser visível somente pelo ambiente de execução e não diretamente no grafo da aplicação.

Além disto, o trabalho do mestrando Edvar Araújo [ARA 00] envolve visualização na execução das aplicações desenvolvidas com objetos distribuídos, provavelmente através da DOBuilder (trabalho em fase inicial). A princípio, com a integração, o comportamento das réplicas no ambiente também poderão ser visualizadas. Assim sendo, a integração dos dois trabalhos procura oferecer ao programador um ambiente mais completo de desenvolvimento e de execução.

4.7 Conclusão

Este capítulo apresentou o modelo ReMMoS, cuja sua principal característica é permitir replicação implícita em ambientes de objetos distribuídos que permitem mobilidade. Além disto, a integração do modelo com outros trabalhos também foi proposta.

É importante observar que o modelo deixa claro suas necessidades em termos de informações que fogem ao âmbito deste trabalho, tais como análise estática de código. Assim, o modelo ReMMoS abre fontes de trabalho e pesquisa para outras áreas da Ciência da Computação, tais como interpretação abstrata, linguagens, aplicações distribuídas e tolerância a falhas. Desta forma, este trabalho poderá ser fonte de outros trabalhos e os resultados destes podem sempre aperfeiçoar o modelo ReMMoS, assim como os projetos relacionados ao ReMMoS citados nas seções anteriores.

Além disto, como destacado do capítulo 2, os trabalhos relacionados à mobilidade em sistemas distribuídos não trabalham com replicação em nível de aplicação. A única replicação existente corresponde à persistência dos objetos movidos. Trabalhos que envolvem replicação estão mais voltados para computação móvel, sem preocupação com desempenho das aplicações. Assim sendo, o modelo apresentado neste capítulo possui vários aspectos inovadores no sentido de disponibilizar replicação em ambientes que permitem mobilidade de objetos.

Destaca-se a preocupação do modelo, e do trabalho em geral, de integração com outros trabalhos em desenvolvimento pelo Grupo de Processamento Paralelo e Distribuído do Instituto de Informática, tais como o DOBuilder [MAL 00] e o trabalho de interpretação abstrata de código Java em desenvolvimento por [AZE 00]. Além disto, o trabalho do mestrando Edvar Araújo [ARA 00] envolve visualização no DOBuilder. Com a integração, o comportamento das réplicas no ambiente poderão ser visualizadas. Esta integração é de suma importância para o desenvolvimento do grupo, uma vez que os trabalhos são direcionados para um objeto em comum, mesmo cada um deles

possuindo suas características próprias e bastante criativas, além das publicações que podem ser geradas.

O capítulo a seguir apresenta a implementação e os resultados do modelo.

5 Implementação do Protótipo ReMMoS

Este capítulo apresenta a implementação do modelo REMMOS- *Replication Model in Mobility Systems*. O sistema (protótipo) foi implementado em Java [SUN 00] utilizando Voyager [OBJ 00] para suporte a mobilidade dos objetos.

5.1 Organização Básica do Protótipo

O protótipo está organizado em dois módulos principais: módulo de gerenciamento geral e módulo de gerenciamento local. Cada módulo possui funções específicas, como demonstra a figura 5.1 abaixo.

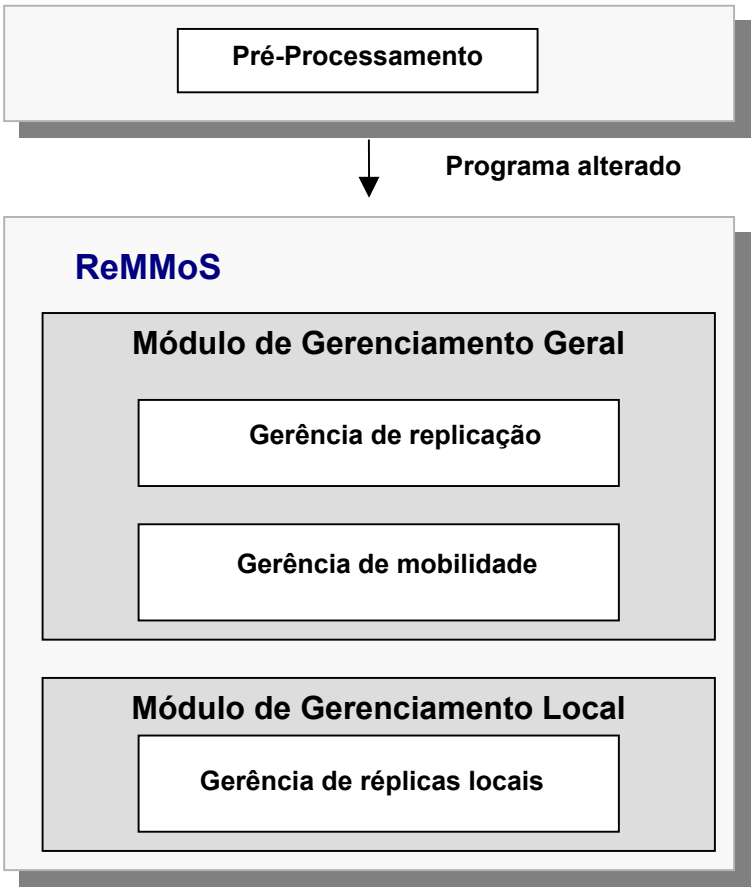

FIGURA 5.1 - Organização do protótipo ReMMoS

A função de cada módulo, bem como dos seus sub-módulos, e a representação dos principais objetos e métodos que implementam estes módulos, serão destacadas nas seções seguintes.

5.2 Pré-Processamento

Como destacado no item 4.6 do capítulo 4, para dar suporte à replicação implícita, é necessário um **analisador de código** que infira informações que serão usadas pelo ReMMoS. Este analisador deve pré-processar o código cliente original, inferindo chamadas ao sistema de execução ReMMoS, mantendo o código original do cliente intacto, uma vez que a replicação é implícita.

O primeiro passo é criar uma cópia do código original chamada `ReMMoS<nome_do_prog>.java`. Vale ressaltar que a cópia é criada toda vez que é verificada a alteração na cópia original do código.

Para criação das réplicas de um objeto, o analisador deve inserir uma chamada ao método `reMMoSControlaSerializado()` toda vez encontrar uma classe que implementa serialização (*implements Serializable*, em Java). O referente método atribuirá **verdadeiro** ao atributo `serializado` do objeto `reMMoSGerenciaGeral`. Além disso, se o objeto é serializado ele é também potencialmente replicável. Assim, o analisador também deve inserir uma chamada ao método `reMMoSControlaReplicado()`. Este método atribuirá **falso** ao atributo `replicado` do objeto `reMMoSGerenciaGeral`. Estes atributos são consultados e manipulados pelos métodos responsáveis pelo controle de chamadas à objetos potencialmente replicados, objetos replicados e de gerência das réplicas.

Como descrito no item 4.6, é necessário que o analisador de código analise cada método verificando seu tipo (escrita/leitura). Neste caso, as seguintes chamadas devem ser inseridas:

- se o método é considerado de escrita, são inseridas, antes da chamada do método de escrita, chamadas aos métodos `reMMoSControlaChamadaEscritaRemota()` e `reMMoSControlaChamadaEscritaLocal()`. O primeiro é um método do objeto `reMMoSGerenciaGeral` e controla chamadas de escritas remotas de um objeto **potencialmente replicável**. O segundo é um método do objeto `reMMoSGerenciaLocal` e controla chamadas de escritas locais. As escritas locais correspondem a solicitação de escrita, de um objeto cliente a uma réplica local, sendo que esta escrita acarreta em uma modificação global. Conforme o capítulo 4, seção 4.6, escrita dentro do escopo do método é considerada uma operação de leitura;

- se o método é considerado de leitura, são inseridas, antes da chamada do método de leitura, chamadas aos métodos `reMMoSControlaChamadaLeituraRemota()` e `reMMoSControlaChamadaLeituraLocal()`. O primeiro é um método do objeto `reMMoSGerenciaGeral` e controla chamadas de leituras remotas de um objeto **potencialmente replicável**. O segundo é um método do objeto `reMMoSGerenciaLocal` e controla chamadas de leituras locais. As leituras locais correspondem a solicitação de leitura, de objetos clientes a uma réplica local

Para controle da mobilidade de um objeto replicado, toda vez que o analisador encontrar no código uma chamada a um método *moveTo ( )* (método de mobilidade do Voyager), o método do cliente deve ser encapsulado pelo seguinte método do ReMMoS: `reMMoSmoveTo()` do ReMMoS. Este método herda o método do cliente mais o controle de mobilidade de objetos replicados.

Observa-se que, durante a execução, o método `reMMoSmoveTo()` verifica o atributo `Replicado`. Se este atributo está com valor **verdadeiro**, então a mobilidade será controlada pelo sistema de replicação. Se este atributo está com valor **falso**, então a mobilidade será controlada pelo sistema de mobilidade usado (neste caso, Voyager).

Através das informações acima, o ReMMoS possuirá todas as informações necessárias para a gerência das réplicas e da mobilidade dos objetos replicados.

5.3 Módulo de Gerenciamento Geral

Este módulo é responsável pelas seguintes ações:

- monitoramento do acesso aos objetos potencialmente replicáveis e objetos replicados;
- gerência da consistência do objeto replicado;
- gerência da mobilidade do objeto replicado.

Como a replicação é implícita, é necessário que os acessos de clientes a objetos servidores sejam monitorados pelo sistema. Este monitoramento é feito com base em informações inseridas no código do cliente. Este controle de acesso é necessário pois um objeto **potencialmente replicável** (que pertence a uma classe serializável) pode comportar-se de modo a ser necessário criar réplicas. Já, uma vez sendo um objeto **replicado** (que já possui réplicas), pode ter associado a si novas réplicas ou pode ser movido.

O objeto que implementa este módulo é denominado `reMMoSGerenciaGeral`. Este é o responsável pela maior parte do controle do ReMMoS. Possui vários métodos, que representam serviços deste objeto, conforme será colocado a seguir. Para um melhor entendimento, estes métodos serão apresentados dentro de seções que representam as principais ações do objeto `reMMoSGerenciaGeral`.

5.3.1 Monitoramento Geral

O método do objeto `reMMoSGerenciaGeral` que implementa o monitoramento geral chama-se `reMMoSMonitorGeral()`. Este método é o responsável por obter informações dos objetos potencialmente replicáveis e objetos replicados, de forma que as operações de criação das réplicas e acesso aos objetos replicados possam ser efetuadas.

Assim, a cada vez que os métodos de leitura/escrita remotos forem chamados, o objeto de monitoramento geral é responsável por controlar o tipo da chamada (escrita/leitura) e a proporção das chamadas, verificando a possibilidade de criação das réplicas. Os métodos `reMMoSControlaChamadaEscritaRemota()` e `reMMoSControlaChamadaLeituraRemota()` fornecem as informações sobre o tipo do acesso para que o método possa decidir quanto a criação das réplicas.

Uma réplica é criada quando o número de operações de leitura/escrita que incidem sobre um objeto potencialmente replicável alcançar a proporção de 3 para 1, isto é: **3 leituras** locais / **1 escrita** total (ou somente 3 leituras locais). O algoritmo que implementa a criação de réplicas de um objeto é apresentado no capítulo 4.

Se uma réplica do objeto replicado é criada pela primeira vez, o método `reMMoSMonitorGeral()` deve atualizar o atributo `Replicado` do objeto `reMMoSGerenciaGeral` para **verdadeiro** e chamar o método `reMMoSCriaReplica()`.

Caso o objeto já tenha réplicas, basta informar o local onde a réplica foi criada ao método `reMMoSControlaLocalReplicas()`. Este atualiza o vetor que controla a localização das réplicas de cada objeto replicado (atributo `vetReplicas`). O próprio método `reMMoSCriaReplica()` faz esta verificação.

Além disto, o método `reMMoSMonitorGeral()` é responsável por verificar se acessos remotos estão sendo solicitados para objetos locais. Isto é, um cliente em uma máquina pode estar solicitando serviços de um objeto do servidor e este já está replicado na máquina do cliente. Neste caso, as próximas chamadas do cliente ao objeto são locais. Para isto, o método monitor faz uma chamada ao objeto `reMMoSGerenciaLocal` , que é responsável, entre outras coisas, por monitorar os acessos dos clientes locais a uma réplica do objeto.

5.3.2 Gerência de atualização do objeto replicado

O método `reMMoSAtualizaObjReplicado()` é o responsável por atualizações no objeto replicado (portanto implementa o protocolo de Primário-*backup*). Assim, toda vez que um objeto torna-se objeto replicado, uma operação de escrita neste objeto, causa a chamada ao método `reMMoSAtualizaObjReplicado()`. Esta solicitação de escrita pode ser identificada por dois métodos diferentes. Isto é, ou pelo método `reMMoSMonitorGeral()`, do objeto `reMMoSGerenciaGeral` ou pelo método `reMMoSControlaChamadaEscritaLocal()` do objeto `reMMoSGerenciaLocal`. Assim, o protocolo de atualização é acionado toda vez que a primária receber, de um cliente local ou de suas réplicas, uma solicitação de escrita. As mensagens necessárias para controlar a atualização de um objeto estão representadas na figura 4.3, capítulo 4. Através da serialização das mensagens, garantida pela cópia primária, os objetos são mantidos no mesmo estado consistente.

A operação de atualização não ocorre se uma operação de criação está sendo realizada. Da mesma forma que um objeto não é criado se está incidindo uma operação de atualização sobre este objeto. Assim, estas operações não podem ocorrer de forma concorrente. Esta decisão procura evitar inconsistência no estado das réplicas.

A figura 5.2 indica a situação de um objeto com duas réplicas sendo atualizado por iniciativa de uma das réplicas (esta recebeu uma solicitação de atualização local). A figura mostra apenas as invocações remotas que foram implementadas com o uso do ambiente Voyager. Caso a atualização iniciasse pelo primário, a primeira mensagem não ocorreria. É importante observar que a mensagem `ACKMaisDescarte()` corresponde ao retorno da invocação remota do método `ReMMoSAtualizaBloqueia()`.

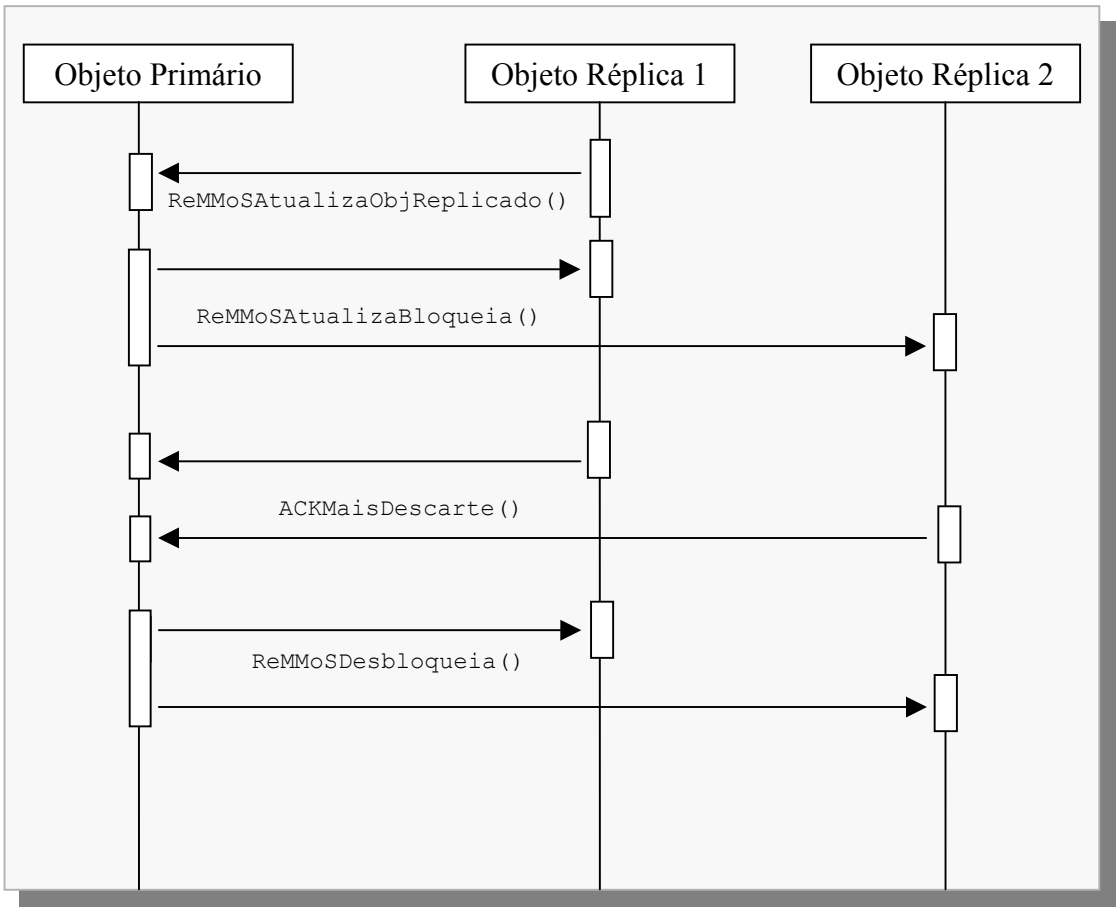

FIGURA 5.2 – Protocolo de atualização: réplica solicita atualização

5.3.3 Gerência de mobilidade do objeto replicado

Conforme já colocado, o método de mobilidade do cliente foi encapsulado, estaticamente, por um método de mobilidade do ReMMoS (`ReMMoSmoveTo`). Este método incorpora o método do cliente mais o controle de mobilidade de objetos replicados. Durante a execução, o atributo `Replicado` é verificado. Se este for **falso**, então o controle de mobilidade do ReMMos não é chamado e a mobilidade é controlada pelo sistema de mobilidade usado (no caso deste trabalho, o Voyager). Caso o atributo seja **verdadeiro**, então o ReMMoS controla a mobilidade, conforme capítulo 4.

A mobilidade de objetos não-replicados fica a cargo do sistema Voyager. A seção 5.5 destaca algumas razões para escolha do mesmo. Em Voyager, a mobilidade ocorre da seguinte forma:

1. o objeto é bloqueado e todas as mensagens que o objeto está processando devem ser completadas. Novas mensagens que chegam são suspensas e esperam em uma fila até que o objeto seja liberado;

2. o objeto e seus objetos agregados são movidos para o nodo destino;

3. o método responsável pela mobilidade retorna se a mobilidade ocorreu com sucesso ou não.

   Se a mobilidade ocorreu com sucesso:

   - objeto é destruído no endereço antigo e um *proxy* para o objeto é criado neste endereço. Este *proxy* é necessário para encaminhar as mensagens que o objeto recebe no endereço antigo para o novo endereço;

   Se a mobilidade não ocorreu com sucesso:

   - uma exceção é gerada e o objeto deve ser restaurado no endereço antigo. Como o objeto ainda continua existindo no endereço antigo, este é desbloqueado para que continue processando as mensagens que estavam bloqueadas.

4. Mensagens para o objeto, que foram suspensas, são liberadas. Quando uma mensagem não encontra o objeto no endereço, procura por um *proxy* representando o objeto. Se este é localizado, ele mesmo envia a mensagem para o objeto na nova localização.

5.4 Módulo de Gerenciamento Local

Este módulo é responsável pelas seguintes ações:

- monitoramento dos acessos locais a uma réplica de um objeto;
- descarte da réplica.

O objeto que implementa este módulo é denominado `reMMoSGerenciaLocal`. Este é o responsável pelo controle local das ações acima de cada réplica de um objeto. Possui vários métodos, que representam serviços deste objeto, conforme será colocado a seguir. Para um melhor entendimento, estes métodos serão apresentados dentro de seções que representam as principais ações do objeto `reMMoSGerenciaLocal`.

5.4.1 Monitoramento de acessos locais

Os métodos `reMMoSControlaChamadaEscritaLocal()` e `reMMoSControlaChamadaLeituraLocal()` são responsáveis pelo monitoramento local de escrita ou leitura. Assim, quando um cliente local solicita uma escrita e esta incide sobre a réplica do objeto, o método `reMMoSControlaChamadaEscritaLocal()` é responsável por encaminhar este pedido de atualização ao método `reMMoSAtualizaObjReplicado()`do objeto `reMMoSGerenciaGeral`.

Já o método `reMMoSControlaChamadaLeituraLocal()` é responsável por manter o controle do número de acessos de leitura que a réplica do objeto replicado processa localmente. O número de acessos é mantido em uma lista de acessos (`vetorAcesso`)Vale lembrar que estes acessos podem ser de qualquer cliente, desde de

que seja um cliente local. Este método é acionado, pelo método `reMMoSDescartaRéplicaLocal()`, toda vez que as réplicas são atualizadas.

5.4.2 Descarte das réplicas

O método `reMMoSDescartaRéplicaLocal()` verifica, durante os processos de atualizações, o comportamento dos números de acessos da réplica. O número de acessos de leitura é mantido em uma lista. Se durante 3 (três) chamadas de atualização do objeto replicado, o número de acessos permanecer igual, o método `reMMoSDescartaRéplicaLocal()` é responsável por descartar a réplica antes que ela seja atualizada. Quando as réplicas sinalizam que a atualização ocorreu com sucesso (ACK), o método aproveita para enviar uma chamada ao método `reMMoSControlaLocalReplicas()`, do objeto `reMMoSGerenciaGeral`, indicando qual réplica foi descartada. Automaticamente, o método de atualização considera esta resposta como um ACK da réplica.

Aproveita-se assim o próprio protocolo de atualização para realizar a gerência do descarte das réplicas. Na próxima iteração com as réplicas, a primária terá sua visão de réplicas atualizada.

5.5 Desenvolvimento do Protótipo

Como já foi colocado anteriormente, o ReMMoS suporta mobilidade e replicação em ambientes de objetos distribuídos implementados em Java [SUN 00], usando Voyager [OBJ 00] como plataforma de suporte à distribuição e mobilidade dos objetos.

A escolha de Java deu-se pelo fato da linguagem ser portável, robusta e permitir implementar os principais aspectos necessários para o controle dos objetos distribuídos. Além disto, conforme o capítulo 2, Java suporta as principais plataformas de suporte à distribuição e mobilidade de objetos [OBJ 00] [IBM 98] [CAS 98]. Assim sendo, para garantir portabilidade, o ReMMoS foi implementado usando Java (JDK versão 1.2.2).

Como plataforma para suporte a mobilidade dos objetos servidores usou-se Voyager, versão 3.0. A escolha de Voyager como sistema base para a mobilidade deu-se por várias razões:

- é um sistema projetado para utilizar a linguagem Java;
- várias aplicações distribuídas que suportam mobilidade estão sendo desenvolvidas com Voyager, pois este oferece várias facilidades para estes tipos de aplicações;
- possibilidade de obter uma versão *freeware* do sistema;
- suporte técnico facilitado;
- **não possui replicação de objetos** e **suporta mobilidade de objetos**.

O protótipo do ReMMoS foi implementado numa rede de estações de trabalho Sun gerenciadas pelo sistema operacional Solaris. Além disso, o padrão de interconexão da rede é *Ethernet* com uma taxa de transferência de 10 Mbits por segundo. A determinação de quais estações serão utilizadas na execução de uma aplicação é feita pela aplicação cliente. Para validação do protótipo, os clientes executam em diferentes máquinas da rede.

5.6 Resultados alcançados

Esta seção apresenta e discute os resultados obtidos nos principais testes, realizados com o intuito de avaliar o comportamento das aplicações clientes usando o ambiente de execução com as informações do ReMMoS. As informações no código do cliente, necessárias para o suporte a execução no ReMMoS, foram incluídas de forma manual, visto ainda não estar disponível um analisador de código Java que gere as informações necessárias. Como colocado no capítulo 4, o trabalho de [AZE 00] visa automatizar esta tarefa.

Os testes apresentados foram realizados com o intuito de avaliar o comportamento das aplicações clientes usando o ambiente de execução com as informações do ReMMoS. As informações no código do cliente, necessárias para o suporte a execução, foram incluídas de forma manual, visto ainda não estar disponível um analisador de código Java que gere as informações necessárias.

As aplicações clientes possuem características relevantes de serem avaliadas durante a execução no ReMMoS. É importante avaliar o comportamento de diferentes tipos das aplicações clientes, uma vez que a criação das réplicas adaptam-se ao comportamento da aplicação. Assim, três tipos de clientes foram implementados, representando aplicações sintéticas. Suas características são as seguintes:

1. **Caso A** - cliente com escrita e leitura proporcional, isto é, possui a mesma taxa de requisições de escrita e requisições de leitura ao objeto servidor. As solicitações são feitas de forma alternada;

2. **Caso B** - cliente com leitura predominante, possuindo assim maior taxa de requisições de leitura ao objeto servidor do que requisições de escrita;

3. **Caso C** - cliente com escrita predominante, onde ocorre maior taxa de requisições de escrita ao objeto servidor do que requisições de leitura.

Além disto, para cada tipo de cliente, 4 situações com relação à execução foram analisadas:

(A) Sem mobilidade e sem replicação (S/M S/R): o cliente executa sem o uso das chamadas ao ReMMoS;

(B) Com mobilidade e sem replicação (C/M S/R): Somente durante os processos de escrita ocorre à mobilidade. Durante os processos de leitura, não há replicação;

(C) Sem mobilidade e com replicação (S/M C/R): mobilidade e replicação. Somente ocorre a replicação do objeto para o cliente durante os processos de leitura. Antes de entrar nos processos de escrita, o cliente não solicita mobilidade do objeto;

(D) Com mobilidade e com replicação (C/M C/R): o cliente executa usando mobilidade e replicação. O objeto é replicado para o cliente durante os processos de leitura e o cliente solicita mobilidade do objeto para sua máquina (localização do cliente) antes de entrar nos processos de escrita.

Desta forma, procurou-se para cada tipo de cliente, analisar diferentes formas de execução e como os clientes comportam-se utilizando o ReMMoS nestas situações. Mas vale lembrar que o ReMMoS só detecta o comportamento da aplicação à medida que estas estão sendo executadas. A tabela 2 apresenta os resultados alcançados nos testes. A figura 5.3 apresenta o gráfico da execução.

TABELA 2 – Especificação dos Nodos

Nodo	**Especificação**
1	Sun SPARCstation 20 – 128 Mbytes RAM
2	Sun Ultra 10 – 128 Mbytes RAM
3	Sun Ultra 5 – 192 Mbytes RAM

TABELA 3 –Softwares Usados para implementação

Software	**Versão**
Sistema Operacional	SunOS Release 5.7
Voyager	Versão 3.3
Java	Versão 1.2

TABELA 4– Custo de Replicação e Descarte

Operação	**Média (ms)**
Criação da réplica	740
Descarte da réplica	6

TABELA 5 - Benchmarks (ms)

Comportamento	**Cliente A**		**Cliente B**		**Cliente C**	
	Méd.	**DesvP**	**Méd.**	**DesvP**	**Méd.**	**DesvP**
S/M S/ R	4063,5	61,3	1995,5	34,3	2134,9	30,0
C/M S/R	1335,4	42,8	1085,2	09,6	1126,8	40,4
S/M C/ R	5069,3	53,0	995,4	13,7	2150,7	48,3
C/M C/ R	1074	18,8	846,4	09,1	1115,2	11,8

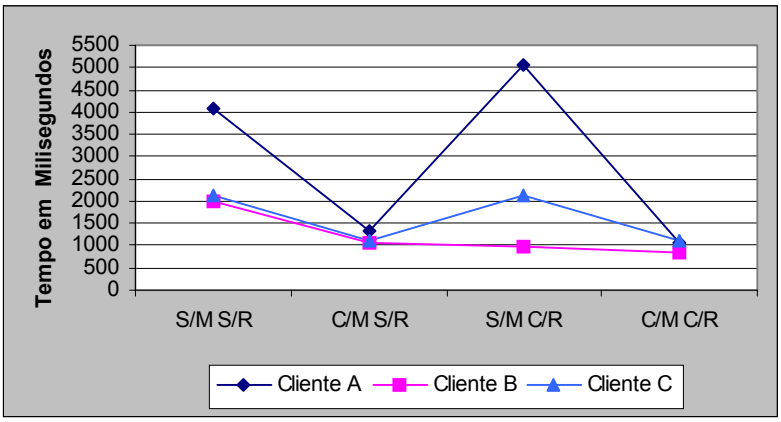

FIGURA 5.3 – Resultados dos testes

5.6.1 Análise do Comportamento das Aplicações

Considerando os resultados das execuções, alguns pontos que validam o uso de replicação e mobilidade são importantes de serem destacados. Como na implementação atual do sistema o custo de replicação é muito elevado, as aplicações que possuem mobilidade obtiveram desempenho superior uma vez que com a mobilidade não foi necessário instanciar réplicas para garantir a localidade de acesso. Assim, de um modo geral, a execução dos clientes que além de mobilidade possuem replicação foi favorecida, visto que os acessos remotos ao objeto servidor são evitados na leitura com a criação das réplicas e na escrita com a mobilidade. O caso mais favorável diz respeito ao do Cliente B que, por possuir predominantemente operações de leitura, já se esperava um resultado favorável na presença de mobilidade e replicação.

Observa-se também que os clientes comportaram-se de forma favorável na presença de mobilidade, uma vez que nestes casos as operações de escrita são feitas localmente. Este caso é especialmente favorável para os Clientes A e C. Já no caso do Cliente B, onde predominam leituras, executá-lo somente com mobilidade do objeto servidor também é favorável pois, embora as leituras sejam remotas, o que não ocasiona alteração no objeto servidor, as escritas são locais. Além disto, como as aplicações são sintéticas, a operação de leitura e escrita incide sobre uma pequena *string*. Já em aplicações reais, onde mensagens maiores trafegam em uma rede WAN (*Wide Area Netware*), provavelmente o custo de execução deste cliente seja maior.

Já na presença somente de replicação, o caso mais interessante à replicação (Cliente B) comportou-se de forma esperada, obtendo um desempenho satisfatório na execução, mesmo sem a presença de mobilidade. O Cliente C apresentou um pequeno *overhead* visto que as operações de escrita foram feitas remotamente e não houve replicação. O caso mais significativo diz respeito ao Cliente A. Uma vez que este realiza seqüências de operações de leitura e escrita intercaladas, as réplicas são criadas durante os processos de leitura, incidindo então o custo de criação de réplicas e, nos processos de escrita, as réplicas são destruídas. Na próxima incidência de leituras, as réplicas são criadas novamente. Isto poderia ser evitado se o comportamento da aplicação fosse conhecido previamente. Assim, o ReMMoS poderia executar a criação e o descarte das réplicas levando em consideração também o comportamento da

aplicação. Trabalhos relacionados à análise estática para aplicações orientadas a objetos [AZE 00] podem favorecer a execução dos clientes com leitura e escrita proporcionais.

Assim, conforme os resultados experimentais comprovaram, o custo de criação é bastante elevado. Tal informação é significativa e indica duas linhas principais de trabalhos futuros. Por um lado, seria necessário fazer uma estimativa de custos para decidir se vale ou não a pena criar uma réplica. O ideal seria ter suporte para a análise de quantidade de leitura a ser realizada para levando em conta o custo de criação da réplica em relação ao número de escritas acrescido de custo de leitura remota, verificar se compensa ou não a criação. Em outras palavras, a heurística de criação seria enriquecida pela estimativa de custos. Outra linha de trabalho relaciona-se a formas de diminuir o custo de criação de réplicas. Uma opção a ser avaliada consiste na implementação de uma rotina própria de clonagem e controle de acesso visando minimizar o tempo de disponibilização da cópia.

Pode-se concluir que a replicação em ambientes de objetos distribuídos que suportam mobilidade é viável e pode melhorar o desempenho da aplicação dependendo do seu comportamento. Além disto, verifica-se que também é interessante, dependendo da aplicação, o uso tanto da mobilidade quanto da replicação separadamente. Sendo assim, a execução de aplicações usando o ambiente de execução do ReMMoS é viável, mostrando ser um ambiente interessante de ser explorado no desenvolvimento de aplicações distribuídas.

5.7 Conclusões

Este capítulo apresentou a implementação do modelo ReMMoS. Seus módulos principais e as classes e objetos que implementam estes módulos foram apresentados. O ambiente de execução usado e a implantação do protótipo no Instituto de Informática foram discutidos.

Um dos pontos principais deste capítulo diz respeito aos resultados alcançados com a execução de aplicações no ambiente ReMMoS (seção 5.6). Observou-se que a gerência de replicação não trouxe um custo significativo na execução. Este fato é de suma importância para a validação da proposta deste trabalho. Mesmo assim, a implementação do ReMMoS deve ser otimizada de forma a buscar melhores resultados.

Além disto, testes com aplicações utilizando o ReMMoS, sendo executadas em ambientes como Linux e redes de interconexão mais rápida (como *Myrinet*, por exemplo), estão previstos para realização futura. Desta forma, espera-se que melhores resultados sejam alcançados.

6 Conclusão

Em ambientes distribuídos, a preocupação com o desempenho das aplicações é importante, uma vez que vários fatores, tais como topologia das redes e os *links* de comunicação, podem afetar o tempo de resposta das aplicações. Embora a mobilidade permita que o programador mantenha a localidade de objetos que trocam muitas mensagens entre si, não permite que vários objetos distribuídos acessem localmente a mesma informação. Clientes em outras máquinas, mesmo necessitando freqüentemente de um objeto, deverão acessá-lo remotamente. O acesso local de vários objetos à mesma informação é permitido através da replicação de um objeto para os locais onde as requisições estão ocorrendo.

Trabalhos relacionados à mobilidade de objetos realizam a mobilidade **explícita**. Neste caso, pode ocorrer que o programador não identifique possíveis situações em que replicar o objeto, para evitar acessos remotos de outros objetos, seja possível. Com a possibilidade de replicação dos objetos, estes poderão existir em várias máquinas do sistema. Além disto, se a replicação é **implícita**, o programador não precisa preocupar-se com as questões relacionadas à gerência da replicação e às decisões referentes à localização e descarte das réplicas.

Assim, este trabalho apresentou o modelo **ReMMoS** – *Replication Model in Mobility Systems*. O modelo permite explorar replicação implícita, com o objetivo de prover desempenho, em ambientes onde a mobilidade explícita dos objetos distribuídos pode ocorrer. No decorrer do texto, foram apresentados conceitos relacionados à replicação e à mobilidade em objetos distribuídos, de forma a obter o embasamento teórico necessário para a definição do modelo.

O capítulo 2 apresentou conceitos relacionados à mobilidade de objetos em sistemas distribuídos. Foram abordados também conceitos básicos sobre sistemas distribuídos, objetos distribuídos e migração de objetos. A ferramenta DOBuilder e trabalhos relacionados à mobilidade também foram apresentados. O ambiente de execução do ReMMoS será integrado à DOBuilder, com o objetivo de prover um ambiente mais completo, onde o programador somente especifique que a replicação esteja presente na execução das aplicações, e o ambiente gerencie todos os aspectos necessários para dar suporte a execução.

No capítulo 3 foram apresentados aspectos referentes à replicação de objetos em sistemas distribuídos, desde os conceitos básicos até considerações sobre replicação, estratégias para manter a consistência das réplicas e técnicas de replicação. Além disso, o estado da arte sobre replicação em sistemas distribuídos que suportam mobilidade e trabalhos relacionados à replicação também foram apresentados.

O Modelo ReMMoS foi apresentado no capítulo 4. Os objetivos, princípios básicos e decisões de projeto foram colocados. O modelo apresentou aspectos inovadores no sentido de disponibilizar replicação em ambientes que permitem mobilidade de objetos. Conforme abordado no capítulo 3, os trabalhos relacionados à mobilidade em sistemas distribuídos não trabalham com replicação em nível de

aplicação. Trabalhos que envolvem replicação estão mais voltados para computação móvel, sem preocupação com desempenho das aplicações. O modelo ReMMoS foi apresentado à comunidade científica brasileira através **do artigo** publicado no WSCAD'00 – *Workshop* de Sistemas Computacionais de Alto Desempenho em outubro deste ano. Este evento ocorreu dentro da programação do SBAC – PAD'00 - *12th Symposium on Computer Architecture and High Performance Computing*.

Além disto, a replicação implícita é favorecida, uma vez que o programador da aplicação não necessita ter conhecimentos à gerência da replicação. O uso da mobilidade implícita também seria favorecido pelo mesmo motivo. O ReMMoS facilita estes aspectos, aproximando os conhecimentos necessários para a criação de um ambiente onde tanto a replicação como a mobilidade sejam implícitas.

A implementação do modelo foi apresentada no capítulo 5. O desenvolvimento do protótipo no Instituto de Informática da UFRGS, os testes e os resultados alcançados foram abordados. Concluiu-se que a gerência de replicação não trouxe uma perda de desempenho significativa na execução das aplicações testada, o que foi importante para a validação do modelo proposto neste trabalho. Concluiu-se também que a implementação do ReMMoS pode ser otimizada de forma a buscar melhores resultados.

O modelo traz como **principais contribuições** o uso de replicação em ambientes que permitem mobilidade; a independência do modelo de mobilidade de objetos, uma vez que está é explícita, bastando informar apenas a primitiva responsável por mover o objeto; não restringe o comportamento da aplicação, embora melhores resultados sejam obtidos quando predominam leituras; e a integração deste trabalho com outros trabalhos do grupo de Processamento Paralelo e Distribuído do Instituto de Informática, como a DOBuilder [MAL 00] e a análise estática [AZE 00].

Considera-se de suma **importância** a **integração** dos vários trabalhos em desenvolvimento pelo grupo, pois cada trabalho pode enriquecer os demais. Além disto, favorece tanto a criação de trabalhos acabados e mais completos (como DOBuilder + ReMMoS + análise estática), como as publicações dos referidos trabalhos em congressos científicos de várias áreas da Ciência da Computação.

O modelo proposto neste trabalho abre vários frentes de pesquisa, uma vez que vários aspectos podem ser incluídos ou melhorados. Assim, destacam-se **como trabalhos futuros**:

- o estudo e a implementação de outras estratégias de replicação e de consistência dos objetos replicados;

- a integração do ReMMoS com a ferramenta DOBuilder, de forma a obter um ambiente mais completo;

- a integração no ReMMoS com trabalho em desenvolvimento pela aluna Silvana Azevedo [AZE 00], de forma que as informações necessárias ao ReMMoS sejam inseridas automaticamente no código. Além disto, outras informações, tais como, comportamento da aplicação, tamanho dos argumentos das mensagens e tamanho dos objetos, podem ser obtidas e usadas pelo ReMMoS;

- a implementação do ReMMoS usando outros sistemas que permitem mobilidade em sistemas distribuídos;

- o estudo, projeto e a implementação de mobilidade implícita;

- melhorias do protótipo construído, de forma que melhores resultados sejam alcançados;

- testes com outras aplicações; rede, sistema operacional e máquinas diferentes podem ser feitos, de forma a buscar novas medidas de desempenho e observar o comportamento do ReMMoS nestes ambientes;

Bibliografia

[AMA 97] AMANDI, Anília. **Programação de Agentes orientada a Objetos**. Porto Alegre: CPGCC da UFRGS, 1997. 208p. Tese de Doutorado.

[AMZ 96] AMZA, Cristiana et al. TreadMarks: Shared Memory Computing on Network of Workstation. **Computer**, New York, v. 29, n. 2, p.18-28, Feb.1996.

[ARA 99] ARAÚJO, E. **Um Estudo sobre Sistemas de Memória Compartilhada Distribuída**. Porto Alegre: CPGCC da UFRGS, 1999. Trabalho Individual.

[ARA 2000] ARAÚJO, E. **Uma Ferramenta de Visualização para Aplicações Distribuídas em Java**. V Semana Acadêmica do PPCG da UFRGS. Porto alegre, PPGC da UFRGS, 2000.

[AZE 2000] AZEVEDO, S. **Análise Estática de Programas Orientados a Objetos**. V Semana Acadêmica do PPCG da UFRGS. Porto alegre, PPGC da UFRGS, 2000.

[BAL 92a] BAL, Henri E. et al. Orca: A Language for Parallel Programming of Distributed Systems. **IEEE Transactions on Software Engineering**, New York, v. 18, n. 3, p. 190-205, Mar. 1992.

[BAL 92b] BAL, Henri E. et al. Replication Techniques for Speeding Up Parallel Applications on Distributed Systems. **Concurrency Practice & Experience**. v. 4, n. 5, p. 337-355, Aug. 1992.

[BAL 98] BAL, Henri E. et al. Performance Evaluation of the Orca Shared Object System. **ACM Transactions on Computer Systems**, New York, v. 16, n. 1, Feb. 1998.

[BAR 98] BARBOSA, Jorge L. V. **Paradigmas de desenvolvimento de Sistemas Computacionais**. Porto Alegre: PPGC da UFRGS, 1998. 49p. Trabalho Individual II.

[BAU 97] BAUMANN, J.; HOHL, F. et al. Communication concepts for mobile agent systems. In: INTERNATIONAL WORKSHOP ON MOBILE AGENTS, 1., 1997. **Proceedings...** Berlin:[s.n], 1997.

[BHA 97] BHARAT, K. CARDELLI, L. Migratory Applications. In: ANNUAL THE PROGRAMMABLE INTERNET, 8., 1997 **Proceedings…**, Berlin: Springer-Verlag, 1997. p. 131-149.

[BIR 87] BIRMAN, Kenneth; JOSEPH, T. A. Reliable communication in the presence of failures. **ACM Transactions on Computer Systems**, New York, v.5, n.1, Feb. 1987

[BIR 93] BIRMAN, Kenneth; VAN RENESSE, R. Reliable distributed computing with the Isis Toolkit. **IEEE Computer Society Press**. 1993.

[BIR 96] BIRMAN, Kenneth. **Building Reliable and Secure Network Applications**. [s.e.]: Prentice Hall, 1996. 550p.

[BUD 93] BUDHIRAJA, N. et al. The primary-backup approach. In: MULLENDER, Sape (Ed.). **Distributed Systems**. 2nd ed., New York: ACM Press, 1993. p.199-216.

[CAS 98] CASTILLO, A.; KAWAGUCHI, M. et al. **Concordia as Enabling technology for cooperative Information Gathering**. USA: Mitsubishi Eletric ITA, USA, 1998.

[CAR 97] CARZANIGA, A.; PICCO, G.; VIGNA, G. Designing distributed applications with a mobile code paradigm. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 12., 1997 **Proceedings...** Boston:[s.n],1997.

[CAR 95] CARDELLI, L. A Language with Distributed Scope. **Proceedings...** pág. 286-297. ACM Press, 1997. Disponível em: <http://www.research.digital.com/SRC/personal/luca_cardelli/obliq/>. Acesso em: ago. 1999.

[CAV 94b] CAVALHEIRO, G. **Um Modelo para Linguagens Orientadas a Objetos Distribuídos**. Porto Alegre: PPGC da UFRGS, 1994. 145p. Dissertação de Mestrado.

[CRI 91] CRISTIAN, F. Understanding Fault-Tolerant Distributed Systems, **Communications of the ACM**, New York, v.34, n.2, p.56-78, February 1991.

[CUG 96] CUGOLA, G. et al. Analyzing Mobile Code Languagens. Mobile Object Systems. Second Internationa Workshop, MOS'96. **Proceedings...** Linz: Springer Verlag,1996.

[CHE 96] CHESS, D.; HARRISON, C.; KERSHENBAUM, A. Mobile agents: are they a good idea? Mobile Object Systems. Second Internationa Workshop, MOS'96. **Proceedings...** Linz: Springer Verlag,1996.

[DÖM 97] DÖMEL, P.; LINGNAU, A.; DROBNIK, O. Mobile Agent Interaction in Heterogeneous Environments. Mobile Object Systems. Second International Workshop, MOS'96. **Proceedings...** Linz: Springer Verlag,1996.

[FER 99b] FERRARI, Débora. **Um Estudo sobre Mobilidade em Sistemas Distribuídos**. Porto Alegre: PPGC da UFRGS, 1999. Trabalho Individual II.

[FER 2000] FERRARI, D.; KAYSER, P.; GEYER, C. ReMMoS - um modelo de replicação em ambientes que permitem mobilidade de objetos. WSCAD' 00 - Workshop de Sistemas Computacionais de Alto Desempenho - SBAC-PAD'00. **Anais.....** São Pedro: [s.n], 2000.

[FLE 98] FLEISCHHAUER, L. **Agentes**. 1998. Disponível em: <http://www.eps.ufsc.br/disserta97/amaral/cap3.htm.> Acesso em: set.1999.

[FUG 98] FUGGETTA, A.; PICCO, G. P.; VIGNA, G. Understanding Code Mobily. **IEEE Transactions on Software Engineering**. New York, v.24, n. 5, May 1998.

[GAR 97] GARBINATO, Benoît; GUERRAOUI, Rachid. **Bast, A Framework for Reliable Distributed Computing**. Operating Systems Laboratory of the Swiss Federal Institute of Technology,1997. Technical Report. Disponível em:< http://lsewww.epfl.ch/bast/ >. Acesso em: abr, 2000.

[GAR 98] GARBINATO, Benoît; GUERRAOUI, Rachid. An Open Framework for

Reliable Distributed Computing. In **ACM Computing Survey**, 1998. Disponível em:< http://lsewww.epfl.ch/bast/ >. Acessado em Abr. 2000.

[GLA 98] GLASS, Graham. ObjectSpace Voyager – The Agente ORB for Java. Worldwide Computing and its Applications (WWCA'98). Second International Conference. **Proceedings...** Tsukuba:[s.n], 1998.

[GUB 97] GUERRAOUI, R.; GARBINATO, B.; MAZOUNI, K. GARF: a tool for programming reliable distributed applications. **IEEE Concurrency**, v.5, n.4, p.32-39, Nov. 1997. Disponível em: < http://lsewww.epfl.ch/~rachid/ papers/obj.html>. Acesso em: nov. 1999.

[GUE 97] GUERRAOUI, R.; SCHIPER, A. Software-based replication for fault tolerance. **IEEE Computer**, v.30, n.4. pp. 68-74. Apr. 1997.

[GUE 98] GUERRAOUI, R. *et al*. System support for object groups. OOPSLA'98. ACM Conference on Object Oriented Programming Systems, Languages and Applications. **Proceedings...** Vancouver: [s.n], 1998.

[HAC 89] HAC, A. A Distributed Algorithm for Performance Improvement Through File Replication, File Migration and process Migration. **IEEE Trans. Software Engineering**, v. 15, n. 11, p. 1459-1470. Nov. 1989

[HÄR 97] HÄRTIG, H. LARS, Reuther. Encapsulanting mobile Objects. **In Proceedings of the 17th International Conference on Distributed Computing Systems**, Baltimore:[s.n], 1997.

[HUR 96] HURLEY, R. T. File Migration and File Replication: A Symbiotic Relationship. **IEEE Transactions on Parallel and Distributed Systems**,v. 7, n. 6, p. 578-586, June 1996.

[IBM 98] Industrial and Business Machine corp. **The Aglet Homepage**. Disponível em< http://www.trl.ibm.co.jp/aglets/>. Acessado em Dez.1999.

[ION 96] IONITOIU, Cristian et al. Replicated Objects with Lazy Consistency Second International Workshop, ECOOP'96. **Proceedings...** Politechnica Univeristy of Timisoara.[S.L.:s.n], 1996.

[JAL 94] JALOTE, P. **Fault Tolerance in Distributed Systems**. Englewood Cliffs. Prentice Hall, 1994. 432p.

[KLE 96] KLEINÖDER, J.; GOLM, M. **Transparent and Adaptable Object Replication Using a Reflexive Java**. University of Erlangen-Nürnberg. Erlangen: Computer Science Department, 1996. Technical Report.

[KNA 96] KNABE, F.; Na Overview of mobile agente programming. Fifth LOMAPS workshop on Analysis and Verification of Multiple Agent Languages. **Proceedings...** Stockholm, Springer Verlag, 1996.

[KUR 88] KURE, Q. **Optimization of File Migration in Distributed Systems**. Berkeley, University of California, 1988. PhD Thesis.

[LAN 97] LANGE, Danny; OSHIMA, Mitsuru. et al. Aglets: programming Mobile Agents in Java. Worldwide Computing and its Applications (WWCA'97). International Conference. **Proceedings...** Tsukuba:[s.n], 1997.

[LI 89] LI, Kai ; HUDAK, Paul. Memory Coherence in Shared Virtual Memory Systems**. ACM Transactions on Computer Systems**, New York, v. 7,

n. 4, p. 321-359, Nov. 1989.

[MAF 95] MAFFEIS, Silvano. Adding group communication and fault tolerance to CORBA. USENIX CONFERENCE ON OBJECT ORIENTED, **Proceedings...** Monterey: [s.n],1995

[MAF 96] MAFFEIS, S. PIRANHA: A Hunter Crashed CORBA Objects.Disponível em <http://www.cs.cornell.edu/Info/People/maffeis/electra.html>. Acessado em Out, 1999.

[MAL 00] MALACARNE, J. **Ambiente Visual para Programação Distribuída em Java**. Porto Alegre: PPGC da UFRGS, 2000. Dissertação de Mestrado.

[OBJ 00] ObjectSpace Voyager Core Tecnology 3.0 Disponível em http://www.objectspace.com. Acessado em Fev, 2000.

[OLI 98] OLIVEIRA JUNIOR, E.R. **Replicação de Objetos Distribuídos no DPC++**. Porto Alegre: PPGC da UFRGS, 1998. Dissertação de Mestrado.

[OMG 95] OBJECT MANAGEMENT GROUP. **The Common Object Request Broker: Architecture and Specification**, [S.L.: s.n],1995.

[OMG 98] Object Management Group. Disponível em <http://www.omg.org/corba/> Acessado em Abr.1999

[ORF 96] ORFALI, R. et al. **The Essential Distributeds Objects Survival Guide**. John Wiley & Sons, 1996.

[OSH 98] OSHIMA, M. ; LANGE, D. **Mobile Agentes with Java**: The Aglet API. Disponível em: < http://www.aglet.com >. Acesso em: jan. 2000.

[PAS 00] PASIN, Márcia. **Tolerância a Falhas através de Réplicas de Objetos Distribuídos**. Porto Alegre: PPGC da UFRGS, 2000. Exame de Qualificação.

[PRO 98] PROTIC, Jelica et al. **Distributed Shared Memory**: Concepts and Systems. Los Alamitos, California: IEEE Computer Society Press, 1998.

[RAT 96] RATNER, D.; POPEK, G. J; REIHER, P. **The Ward Model**: A Scalable Replication Architecture for Mobility.

[REI 96] REIHER, P. et al. Peer-to-Peer Reconciliation Based Replication for Mobile Computers. **ECOOP'96 – II Workshop on Mobility and Replication**. Linz: [s.n],1996

[ROY 97] ROY, Peter et al. Mobile Objects in Distributed Oz. ACM Transactions on Programming Languages and Systems. **Proceedings...** New York, v.19, n.5, p.804-851, 1997.

[ROY 98] ROY, Peter. et al. **Three moves are not as bad as fire**. 1998. Tecnical Report.

[RUB96] RUBIN, Ryan.; KISSIMOV, Valentin. Object Migration in the Distributed Computing Environment. ECCOP 96. **II Workshop on Mobility and Replication**. Linz: [s.n], 1996. Disponível em http://www.diku.dk/distlab/workshops/ wmr96/pgm.html >Acessado em Dez.1999

[RUM 96] RUMBAUGH, James; et al. **Modelagem e Projetos baseados em Objetos**. Rio de Janeiro: Campus, 1996.

[SCH 93] SCHNEIDER, F. B. Replication management using the state machine approach. In: MULLENDER, Sape (Ed.). **Distributed Systems**. 2nd. ed., New York: ACM Press, 1993. p. 169-198.

[SHE 86] SHENG, Lui O. R. **Models for Dynamic File Migration in Distributed Computer Systems**. University of Rochester, 1986. PhD Thesis

[SIL 97] SILVA, M. M. Mobility and Persistence. **In Mobile Objects Systems: Towards the Programmable Internet**. P. 157-176. Springer-Verlag, 1997.

[SIN 97] SINHA, Pradeep K. **Distributed Operating Systems**: Concepts and Design. IEEE Computer Society Press, 1997, 743p.

[SOU 97] SOUZA, Luís; OLIVEIRA, Rui. **A CORBA Object Replication Service**. Departamento de informática. Universidade do Minho. Braga:[s.n], 1997.

[STR 85] STROM, R. E. and YEMINI, S. Opmistic Recovery in Distributed Systems. **ACM Transactions on Computer Systems**, New York, v.3, n.3, p.204-226, Aug.1985

[STU 98] STUMM, Michael; ZHOU, Songnian. Algorithms Implementing Distributed Shared Memory. In: PROTIC, Jelica et al. **Distributed Shared Memory**: Concepts and Systems. Los Alamitos, California: IEEE Computer Society Press, 1998. p. 54-64.

[SUN 2000] SUN MICROSYSTEMS. **The Source for Java tecnology**. Disponível em <http://java.sun.com/>. Acesso em: jan. 2000.

[THO 97] THORN, T. **Programming Languages for Mobile Code**. INRIA, France,1997. Tecnical Report 3134

[TAN 92] TANENBAUM, Andrew S. **Sistemas Operacionais Modernos**. Prentice Hall. 1992

[TAN 95] TANENBAUM, A. **Distributed Operating Systems**. Inc. Englewood Cliffs: Prentice-Hall, 1995, 614p.

[WEB 2000] WEBER, Raul F. **Arquitetura de Computadores Pessoais**. Porto Alegre: Instituto de Informática da UFRGS. Editora Sagra Luzzatto, 2000

[WON 97] WONG, D.; PACIOREK, N.; WALSH, T. et al. Concordia: na Infrastructure for Collaborating Mobile Agents. In: INTERNATIONAL WORKSHOP ON MOBILE AGENTS, 1., 1997. **Proceedings...** Berlin:[s.n], 1997.