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

Tolerância a Falhas e Reflexão Computacional num Ambiente Distribuído

por

WILLINGTHON PAVAN
pavan@upf.tche.br

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

Prof. Dr. Paulo Alberto de Azeredo
Orientador

Porto Alegre, março de 2000

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Pavan, Willingthon

Tolerância a Falhas e Reflexão Computacional num Ambiente Distribuído / por Willingthon Pavan. – Porto Alegre: PPGC da UFRGS, 2000.

86p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR – RS, 2000. Orientador: Azeredo, Paulo Alberto de.

1. Tolerância a falhas. 2. Orientação a objetos. 3. Reflexão computacional. I. Azeredo, Paulo Alberto de. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Maria Panizzi
Pró-Reitor de Pós-Graduação: Prof. Franz Rainier Semmelmann
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenadora do PPGC: Profa. Dra. Carla Maria Dal Sasso Freitas
Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Agradecimentos

Chegando ao fim deste trabalho, agradeço a todas as pessoas que, de uma forma ou outra, contribuíram para o seu desenvolvimento, pois, se não fossem elas, certamente ele não seria concluído. De forma especial, agradeço:

Ao professor Dr. Paulo Alberto de Azeredo, pela atenciosa orientação, pelo auxílio despendido sempre que necessário e pelo amigo que demonstrou ser nos momentos difíceis.

À minha esposa que, de forma grandiosa, me incentivou a continuar, não importando quais fossem os problemas nos momentos difíceis dessa caminhada.

À professora Drª. Maria Lúcia Blanck Lisbôa, pela atenção em momentos de dúvida.

Ao CNPq, pelo auxílio à realização deste trabalho; ao Instituto de Informática da UFRGS e ao PPGC, pelo apoio à pesquisa.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

API	Aplication Programming Interfaces
BR	Blocos de Recuperação
CCN	*Consistent Checkpoint Number*
CNPq	*Conselho Nacional de Desenvolvimento Científico e Tecnológico*
CPGCC	Curso de Pós-Graduação em Ciência da Computação
DBMS	*Database Management System*
DCOM	*Distributed Component Object Model*
HTML	*Hipertext Markup Language*
I/O	*Input/Output*
IP	*Internet Protocol*
JDBC	*Java Database Connectivity*
JDK	*Java Development Kit*
JVM	*Java Virtual Machine*
MB	Mega Bytes
MHz	Mega Hertz
ODMG	*Object Data Management Group*
OO	Orientação a Objetos
PC	*Personal Computer*
PNV	Programação N-Versões
PPGC	Programa de Pós-Graduação em Computação
PSE	*Persistent Storage Engine*
RC	Reflexão Computacional
RMI	*Remote Method Invocation*
RMIC	*Remote Method Invocation Compiler*
SOTR	Sistema Operacional de Tempo Real
SQL	*Structured Query Language*
STR	Sistemas de Tempo Real
TCP	*Transfer Control Protocol*
TF	Tolerância a Falhas
UDP	*User Datagram Protocol*
UFRGS	Universidade Federal do Rio Grande do Sul

Resumo

O modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de *software* tolerante a falhas em virtude de características inerentes ao próprio modelo de objetos, tais como abstração de dados, encapsulamento, herança e reutilização de objetos (componentes). O uso de técnicas orientadas a objetos facilita o controle da complexidade do sistema porque promove uma melhor estruturação de seus componentes e também permite que componentes já validados sejam reutilizados [LIS96].

Técnicas básicas para tolerância a falhas em *software* baseiam-se na diversidade de projeto e de implementação de componentes considerados críticos. Os componentes diversitários são gerenciados através de alguma técnica que tenha por objetivo assegurar o fornecimento do serviço solicitado, como, por exemplo, a conhecida técnica de blocos de recuperação.

Reflexão Computacional é a capacidade que um sistema tem de fazer computações para se auto analisar. Ela é obtida quando o programa pára sua execução por um período de tempo para fazer computações sobre si próprio; analisa seu estado, se o processamento está correto, se pode prosseguir com a execução e atingir o objetivo satisfatoriamente; se não precisa mudar de estratégia ou algoritmo de execução, fazendo, ainda, processamentos necessários para o sucesso da execução.

Um sistema de programação distribuída consiste basicamente em vários aplicativos executados em diferentes computadores, os quais realizam troca de mensagens para solucionar um problema comum. A comunicação entre os computadores é realizada através da rede que os interliga. As Redes que controlam sistemas críticos são normalmente de pequena escala pois redes de grandes dimensões podem apresentar atrasos e baixa confiabilidade.

Portanto, a abordagem aqui proposta consiste em utilizar, em um ambiente distribuído, uma arquitetura reflexiva aliada a técnicas do domínio da tolerância a falhas para promover a separação entre as atividades de controle, salvamento, recuperação, distribuição e validação de componentes e as funcionalidades executadas pelo próprio componente, a fim de que falhas não venham a prejudicar a disponibilidade, confiabilidade e clareza de determinadas computações. A proposta apóia-se num estudo de caso, implementado na linguagem de programação Java, com seus protocolos de reflexão computacional e de comunicação.

PALAVRAS-CHAVE: Tolerância a falhas, orientação a objetos, reflexão computacional, salvamento e recuperação de estados de objetos, objetos distribuídos, replicação, RMI e Java.

TITLE: “FAULT TOLERANCE AND COMPUTATIONAL REFLECTION IN A DISTRIBUTED ENVIRONMENT”

Abstract

The model of objects comes as a promising model for the development of tolerant software you fail it by virtue of inherent characteristics to the own model of objects, such as abstraction of data, protection of data, inheritance and reuse of objects (components). THE use of techniques guided to objects facilitates the control of the complexity of the system because it promotes a better structuring of its components and it also allows that components already validated they are reused [LIS96].

Basic techniques for tolerance fault it in software they base on the project diversity and of implementation of critical considered components. The diversity components are management through some technique that has for objective to assure the supply of the requested service, as, for example, the acquaintance technique of recovery blocks.

Reflection Computational is the capacity that a system has to do computations for if solemnity analyze. It is obtained when the program stops its execution for a period of time to do computations on itself own; it analyzes its state, if the processing is correct, if it can continue with the execution and to reach the objective satisfactorily; if doesn't need to move of strategy or execution algorithm, doing, still, necessary processings for the success of the execution.

A system of distributed programming consists basically of several applications executed in different computers, which accomplish change of messages to solve a common problem. The communication among the computers is accomplished through the net that the connect. The Nets that control critical systems are usually of small scale because nets of great dimensions can present back payments and it lowers reliability.

Therefore, it is verified the importance of new rescue techniques and recovery of states of the computation so that they don't harm the readiness, reliability and clarity. The approach here proposal consists of using, in a distributed system, an allied reflexive architecture the techniques of the domain of the tolerance fault to promote the separation among the control activities, rescue, recovery, distribution and validation of components and the functionalities executed by the own component. The proposal leans on in a case study, implemented in the programming language Java, with its protocols of reflection computational and of communication.

KEYWORDS: Tolerance fault, orientation to objects, reflection computational, rescue and recovery of states of objects, distributed objects, replication, RMI and Java.

1 Introdução

O modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de *software* tolerante a falhas em razão de características inerentes ao próprio modelo de objetos, tais como abstração de dados, encapsulamento, herança e reutilização de objetos (componentes). O uso de técnicas orientadas a objetos facilita o controle da complexidade do sistema porque promove uma melhor estruturação de seus componentes e também permite que componentes já validados sejam reutilizados [LIS96].

Técnicas básicas para tolerância a falhas em *software* baseiam-se na diversidade de projeto e de implementação de componentes considerados críticos. Os componentes diversitários são gerenciados por meio de técnicas que tenham por objetivo assegurar o fornecimento do serviço solicitado, como, por exemplo, a conhecida técnica de blocos de recuperação [PAV97]. Entretanto, um dos pontos críticos da técnica de blocos de recuperação é a questão do salvamento e posterior recuperação do estado da computação, de forma que cada variante seja executada a partir das mesmas condições iniciais.

No modelo de objetos, o estado de um objeto relaciona-se com cada uma das suas ativações. Objetos definem ambientes dinâmicos, visto que são criados e destruídos por mensagens específicas no decorrer da execução do programa [LIS95].

Além do ambiente local explícito de um objeto, que compreende as suas variáveis de instância, há que se considerar o ambiente local implícito, que abrange os dados temporários usados pelo ambiente de execução do objeto e seu ambiente de parâmetros, que se estabelece a cada mensagem parametrizada recebida. Acrescente-se, ainda, o ambiente global, que torna visível ao objeto outros componentes (p. ex., nomes de variáveis, classes e objetos) não definidos em sua “cápsula”, e, sim, no seu contexto de definição ou chamada [LIS95].

A cada ativação de um objeto, essas informações são mantidas em um registro de ativação, cuja existência se encerra com o término da ativação do objeto. Na semântica de execução de um objeto, o registro de ativação reflete o seu estado: ambiente local, parâmetros e referência ao ambiente global [LIS95].

Por essas breves considerações, verifica-se a grande complexidade e importância de se estabelecer um processo de salvamento do estado da computação para uma possível recuperação no caso de uma falha vir a ocorrer em alguma computação. No uso de algumas técnicas de tolerância a falhas, como, por exemplo, na de blocos de recuperação, esse processo é de fundamental importância para o sucesso da técnica.

1.1 Estados consistentes

Checkpoints, ou pontos de recuperação, tratam do estabelecimento e armazenamento do "estado" global do sistema. O mecanismo de *checkpoint* e retorno é realizado em três etapas [CAM96]: primeiro, ele sincroniza os processos para estabelecer o momento em que cada processador está com a memória virtual consistente; segundo, esse estado consistente é salvo em disco pelo mecanismo de *checkpoint* antes de ser modificado pelos processos, para que a imagem salva esteja num estado consistente, a fim de que possa ser recarregado mais tarde. Dessa maneira, o

sistema tem um estado intermediário a ser restaurado no caso de falhas provocarem a parada do sistema; terceiro, ele fornece uma maneira de recuperar, após a ocorrência de uma falha no sistema, a memória virtual previamente salva no disco.

Para sistemas simples, compostos por apenas um único processo, o "estado global" é geralmente representado pelo espaço de endereçamento do processo e pelo estado de seus registradores [PLA96].

Em um sistema mais complexo, formado por vários processos, executado em máquinas distintas, podem ocorrer problemas adicionais que dificultam a determinação de um estado global consistente. Como exemplo, a troca de mensagens entre os processos do sistema resulta em problemas de sincronização.

1.2 Abordagem Proposta

Neste trabalho, a proposta consiste em utilizar técnicas do domínio de tolerância a falhas juntamente com técnicas para salvamento e recuperação de estados de objetos, empregando uma arquitetura reflexiva em ambiente distribuído para a implementação de uma biblioteca de classes, que possa ser utilizada para o salvamento e recuperação desses estados.

A reflexão computacional será utilizada como técnica de programação metanível, com a finalidade de controlar e manipular a execução dos objetos em nível-base, visando garantir, no caso de falha, que o sistema retorne a um ponto de recuperação (estado consistente), instancie um novo objeto, o qual retome a execução impedindo, assim, que haja prejuízos ao sistema.

1.3 Organização da Dissertação

Em razão da grande extensão e abrangência do assunto abordado neste trabalho, fez-se um estudo genérico do problema em seu aspecto mais amplo para chegar a um estado mais detalhado do aspecto específico. Entre os tópicos abordados de forma genérica, encontram-se os sistemas de tempo-real, os sistemas distribuídos e a programação com objetos distribuídos. Este trabalho visa propor uma estrutura que se caracteriza pelos aspectos apresentados, objetivando resolver o problema da falta de confiabilidade e de disponibilidade dos sistemas em ambientes distribuídos.

No capítulo 2, faz-se uma descrição da prevenção e tolerância a falhas; no capítulo 3, descreve-se a técnica de reflexão computacional; no capítulo 4, expõe-se uma descrição sobre projetos de programas; o capítulo 5 contém as considerações sobre as técnicas de salvamento e recuperação de estados de objetos; no capítulo 6, apresentam-se a proposta, problemas e soluções para salvamento e recuperação de estados; no capítulo 7, apresenta-se um estudo de caso e, finalmente, no capítulo 8, sintetizam-se as conclusões.

2 Prevenção e Tolerância a Falhas

2.1 Introdução

Para dotar os sistemas computacionais da propriedade de segurança de funcionamento, ou seja, capacitá-los a executar um serviço de acordo com a especificação, exige-se o emprego de procedimentos e/ou métodos em nível de realização, que podem ser classificados em técnicas de prevenção de falha e técnicas de tolerância a falhas.

As técnicas de prevenção de falhas visam impedir, por construção, a ocorrência ou introdução de falhas. Objetivam a confiabilidade do sistema por meio de uma fase de especificação e estruturação cuidadosa, seguida de uma extensa bateria de testes antes da implantação do sistema.

As técnicas de prevenção utilizadas para o *hardware* empregam o uso de componentes confiáveis e tecnologias de interconexão testadas, metodologias para lidar com complexidades de projeto, bem como métodos formais para verificação dos projetos lógicos. Para o *software*, é realizada a prevenção com a inclusão de definições de necessidades, uso de especificações precisas de metodologias de projeto, de técnicas de programação estruturada, de técnicas de teste e documentação, desenvolvimento em linguagens de alto nível e gerenciamento de *software* com objetivo de prevenir enganos.

Essas técnicas, entretanto, não são suficientes, pois sempre haverá falhas residuais que poderão culminar na interrupção da execução de um ou mais módulos do sistema. Para tratar desses resíduos e aumentar a integridade, a disponibilidade de recursos e a segurança do sistema, devem-se utilizar técnicas de tolerância a falhas que permitam ao sistema continuar a executar corretamente, mesmo na presença de falhas.

Conforme Conceição [CON97], de um modo geral, tolerância a falhas pode ser idealizada em três níveis: da máquina, do sistema operacional e das aplicações. No nível de máquina, as falhas estão se tornando pouco freqüentes, pois seus componentes têm apresentado uma confiabilidade bastante satisfatória. Em nível de sistema operacional, as falhas estão sendo tratadas com bastante sucesso com o uso de métodos de tolerância a falhas, tais como replicação de sistemas de arquivos, espelhamento de disco e *checkpointing* baseado em transações. Finalmente, no nível de aplicação, as falhas devem ser detectadas e corrigidas por novas técnicas, tais como métodos de *checkpointing*, tratamento de exceções e replicação ativa de processos.

De acordo com Huang e Kintala [HUA95], citado por [CON97], a tolerância à falha no nível de aplicação pode ser implementada em cinco níveis:

- Nível 0:

  Não há tolerância à falha em nível de aplicação.

  Quando uma aplicação apresenta uma falha, ela deve ser restaurada manualmente a partir do estado inicial.

- Nível 1:

  Detecção automática e restabelecimento de execução.

  Quando uma aplicação apresenta uma falha, essa é automaticamente detectada e a aplicação é restaurada a partir do estado inicial.

- Nível 2:

  Nível 1 mais *checkpointing* periódico, rastreamento de mensagens e restabelecimento a partir do último estado local salvo.

  Neste nível, o estado local da aplicação é periodicamente salvo em disco durante o período de execução normal.

  Quando uma falha é detectada, a execução da aplicação é retomada a partir do último estado local salvo e as mensagens trocadas entre os processos são reprocessadas para que se reconstrua completamente o estado no qual a aplicação se encontrava no momento anterior à falha.

- Nível 3:

  Nível 2 mais *checkpointing* de arquivos.

  Os arquivos manipulados pela aplicação são replicados em nós (de *backup*) da rede e são mantidos consistentes em relação aos arquivos presentes no nó primário durante a execução da aplicação. No caso de uma falha, a nova instância da aplicação retoma a execução no nó de *backup* e os arquivos replicados reconstroem da melhor forma possível o estado da aplicação.

- Nível 4:

  Execução contínua sem interrupção.

  Este nível garante a melhor tolerância à falha teoricamente possível. Normalmente, pode ser atingido replicando-se processos em máquinas distintas.

2.2 Tipos de Falhas

Muitos tipos de falhas são freqüentemente encontrados na literatura de sistemas distribuídos. Conforme livro de Pankaj Jalote [JAL94] (FIGURA 1), encontram-se as seguintes falhas:

- *Falha de crash*: é uma falha que causa a parada de um componente ou a perda do seu estado interno;
- *Falha de omissão*: ocorre quando um componente não responde a determinadas entradas (engloba falhas de *crash*);
- *Falha de temporização*: verifica-se quando o componente responde ou muito cedo ou muito tarde; também é chamada falha de desempenho (engloba falha de omissão);
- *Falha de resposta*: dá-se por computação incorreta; o componente produz respostas incorretas para algumas entradas (não engloba as anteriores);
- *Falhas bizantinas* (arbitrárias ou maliciosas): é a falha arbitrária que provoca um comportamento totalmente arbitrário e imprevisível do componente durante o defeito (engloba todas as classes de falhas).

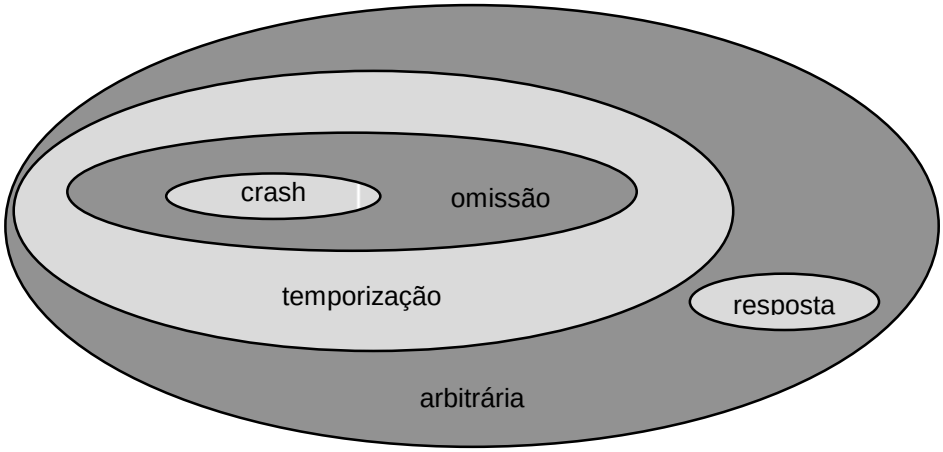

FIGURA 1 - Classes de falhas.

Podem-se encontrar também outros tipos de falhas [CON97]:

- *Falha de parada* (*fail-stop*): ocorre quando há uma interrupção permanente na execução, sendo, entretanto, possível detectar a indisponibilidade do processo;
- *Falhas crash + link*: ocorrem quando um processo sofre uma falha do tipo crash, aliada ao fato de algumas mensagens poderem ser perdidas; contudo, não há demora, duplicação nem corrupção de mensagens;
- *Falha omissão de recepção*: ocorre quando um processo deixa de receber do subsistema de comunicação algumas mensagens enviadas através de um meio de comunicação sem falhas;
- *Falhas omissão de envio*: ocorrem quando um processo deixa de enviar mensagens por um meio de comunicação sem falhas;
- *Falha omissão genérica*: é caracterizada por uma falha de omissão de recepção e/ou omissão de envio de mensagens.

Os tipos de falhas crash+link, omissão de recepção, omissão de envio e omissão genérica são relativos às mensagens trocadas entre aplicações distribuídas.

2.3 Checkpoints

O *checkpoint* local de um processo de aplicação corresponde a seu estado local salvo em disco. O ato de obter o *checkpoint* local é denominado *checkpointing*. O trecho no código da aplicação onde as rotinas de obtenção do *checkpoint* local são invocadas denomina-se *checkpoint*.

É importante notar que um processo pode sofrer uma falha durante a fase de *checkpointing*, o que justifica a existência de mais de um *checkpoin*t local por processo. Portanto, métodos de *checkpointing* devem manipular ao menos dois *checkpoints* locais: um em fase de efetuação (denominado *candidato a checkpoint*) e outro completamente salvo (denominado *checkpoint permanente*), relativo à última operação completa de *checkpointing* efetuada pelo processo.

O estado de um objeto relaciona-se com cada uma das suas ativações. Objetos definem ambientes dinâmicos, visto que são criados e destruídos por mensagens específicas no decorrer da execução do programa [LIS95].

Além de seu ambiente local explícito, que compreende as suas variáveis de instância, há que se considerar o seu ambiente local implícito, que abrange os dados temporários usados pelo ambiente de execução, e o ambiente de parâmetros, que se estabelece a cada mensagem parametrizada recebida. Acrescente-se ainda o ambiente global, que torna visível ao objeto outros componentes (p. ex. nomes de variáveis, classes e objetos) não definidos em sua "cápsula", e, sim, no seu contexto de definição ou chamada [LIS95].

A cada ativação de um objeto, essas informações são mantidas em um registro de ativação, cuja existência se encerra com o término da ativação do objeto. Na semântica de execução de um objeto, o registro de ativação reflete o seu estado: ambiente local, parâmetros e referência ao ambiente global [LIS95].

Esse estado, que se pode dizer *estado profundo*, permite ações semânticas como suspender/retornar o fluxo de execução e a sua preservação por cópia num determinado instante, técnica que é conhecida como *ponto de recuperação* [LIS95].

2.3.1 Retomada de execução

A retomada de execução de processos faltosos é uma das principais aplicações do *checkpointing*. Na fase de recuperação, o sistema deve tornar-se livre dos efeitos causados pelas falhas; recuperar, portanto, é restabelecer um estado livre de erros após uma falha. São duas as abordagens básicas para recuperação de erros em processos:

- Se a natureza dos erros causados pelas falhas pode ser completamente avaliada, é possível remover esses erros do estado do processo e habilitá-lo a prosseguir. Essa técnica é conhecida como *recuperação por avanço*.
- Se não for possível avaliar a natureza das falhas e remover todos os erros no estado do processo, esse deve ser restaurado para um estado prévio (pelo qual o sistema já passou) livre de erros. Essa técnica é conhecida como *recuperação por retorno*.

A especificação e implementação de mecanismos de recuperação por retorno são mais simples do que a recuperação por avanço, pois independem do conhecimento do tipo de falha e dos erros causados pela mesma. Entretanto, ocorrem problemas com essa técnica, dos quais os principais são:

- pode haver falhas durante a execução dos procedimentos de retorno;
- alguns componentes do sistema podem ser irrecuperáveis.

A recuperação por avanço, por outro lado, possui custo menor durante a execução visto que somente as partes do sistema que não atingiram o valor esperado são corrigidas, contudo:

- a técnica pode ser usada somente para os danos que puderem ser completamente identificados;
- a implementação desses mecanismos depende da capacidade de antecipação das possíveis falhas;
- as soluções são particulares a cada caso (aplicações e danos).

2.3.2 Custos

A principal característica de qualquer mecanismo de tolerância à falha é a existência de custos adicionais. Dentre os custos relativos às técnicas de tolerância à falha baseadas em *checkpointing*, podem ser destacados [CON97]:

- **Tempo e espaço:**

O gasto adicional de espaço advém do fato de a operação de *checkpointing* salvar em disco os dados relativos ao estado local dos processos. Obviamente, essa operação aumenta o tempo total de execução da aplicação já que, tipicamente, essa permanece temporariamente sem apresentar progressos durante a transferência dos dados para o disco, além do fato de que o tempo de acesso a disco é lento.

- **Instrumentação:**

Técnicas de *checkpointing* exigem a instrumentação do programa da aplicação com chamadas a procedimentos de bibliotecas que salvam e carregam o estado local em disco. Para tanto, é necessário fazer novas codificações e compilações do programa da aplicação, o que representa um custo adicional em termos de pré-execução.

- **Sincronização:**

Como o estado de uma aplicação distribuída é composto pelo estado de seus processos, algumas técnicas de *checkpointing* necessitam de mecanismos que permitam a sincronização durante o armazenamento ou recuperação dos mesmos. De modo geral, tal sincronização requer complexos protocolos de comunicação, que degradam o desempenho da aplicação.

2.3.3 Restabelecimento do estado

O estado do sistema consiste em todas as informações relativas ao seu funcionamento, devendo ser armazenados os valores das variáveis, o ambiente de execução e as informações de controle. Em outras palavras, devem ser armazenados os valores de todos os registradores e o conteúdo do espaço de memória utilizado pelo sistema [JAL94].

Para um sistema de um único processo, se o estado é periodicamente armazenado em um ponto de recuperação, o programa pode voltar a este último ponto ao invés de retornar ao início do programa. Em um sistema distribuído, o armazenamento do ponto de recuperação e a operação de retorno a esse ponto são mais complexos. Isso ocorre porque, em sistemas com troca de mensagem, se cada processo armazena periodicamente informações de seu estado local, pode-se registrar um estado global inconsistente. Esse estado global inconsistente pode se dar em razão do aparecimento de mensagens órfãs (mensagens recebidas e não enviadas) e perdidas (mensagens enviadas, mas não recebidas) após a recuperação de estados de processos por motivo de falhas.

A recuperação por retorno é o procedimento responsável por fazer retornar um processo faltoso para um estado consistente, ação que é realizada utilizando os pontos de recuperação estabelecidos.

No processo de retorno, devem ser consideradas, além dos estados salvos, as mensagens trocadas entre os processos envolvidos após o estabelecimento dos pontos de recuperação. Os processos que se comunicavam com o antigo processo faltoso devem

ser notificados de que esse retornou a um ponto anterior. Se existir alguma mensagem órfã ou perdida, os demais processos também deverão retornar para os seus pontos de recuperação, repetindo esse processo de verificação e retorno até que não existam mais mensagens órfãs ou perdidas. Essa ação pode ser propagada, de forma que o sistema retorne diversos pontos de recuperação, podendo atingir o seu estado inicial. Isso é chamado de *efeito dominó* (FIGURA 2).

Os algoritmos de recuperação por retorno podem sofrer sempre efeito dominó. As implementações que seguem essas características devem procurar minimizar esse efeito para não causar a degradação da performance do sistema quando da ocorrência de uma falha.

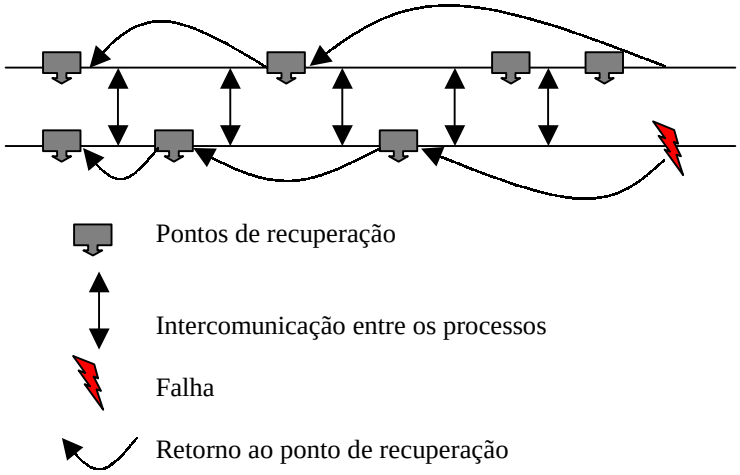

FIGURA 2 - Efeito dominó.

Em sistemas distribuídos, existem duas técnicas para estabelecimento de ponto de recuperação. A primeira é a técnica de ponto de recuperação assíncrono ou *asynchronous checkpoiting*, pela qual o estabelecimento de pontos de recuperação em diferentes nodos é realizado de maneira não coordenada, ou seja, cada nodo pode estabelecer o seu ponto de recuperação independentemente dos demais. A segunda técnica consiste no estabelecimento do ponto de recuperação de maneira coordenada em diferentes nodos, o que é conhecido como ponto de recuperação síncrono ou *synchronous checkpoiting* [JAL 94].

Analisar-se-ão duas técnicas de restabelecimento de estado, denominadas *checkpointing coordenado* ou *distribuído* (*checkpointing síncrono*) e *checkpointing assíncrono*.

2.3.3.1 Checkpointing coordenado ou distribuído

Conforme Conceição [CON97], em métodos de *checkpointing* coordenado ou distribuído, todos os processos ou um subconjunto deles armazenam os estados locais em conjunto, de forma que o conjunto de *checkpoints* locais forme um estado global consistente. A idéia principal é retomar a execução do sistema a partir de um estado global consistente quando da ocorrência de falha em algum processo.

De acordo com Jalote [JAL94], um conjunto de *checkpoints* locais *K* forma um estado global consistente se:

1. o conjunto *K* contém apenas um *checkpoint* local por processo;

2. não existe um evento de envio de mensagem em um processo *p* posterior a seu *checkpoint* local *Cp*, cujo correspondente evento de recebimento da mensagem pelo processo *q* ocorra antes do *checkpoint* local *Cp*, para *Cp* e *Cq* pertencentes ao conjunto de *checkpoints* locais *K*. Isso equivale a dizer que não existem mensagens órfãs no conjunto *K*;
3. não existe um evento de envio de mensagem em um processo *p* anterior a seu *checkpoint* local *Cp*, cujo correspondente evento de recebimento da mensagem no processo receptor *q* ocorra após o *checkpoint* local para e *Cq* pertencentes a *K*. Isso equivale a dizer que não existem mensagens perdidas em *K*.

Um conjunto de *checkpoints* que satisfaça essa condição é chamado de *linha de restabelecimento*. Métodos de *checkpointing* coordenados devem, então, retomar a execução do sistema a partir da linha de restabelecimento mais recente após a ocorrência de falha em um processo.

Veja-se um exemplo de como utilizar a definição dada para determinar uma linha de restabelecimento no seguinte sistema formado pelos processos *p*, *q* e *r*.

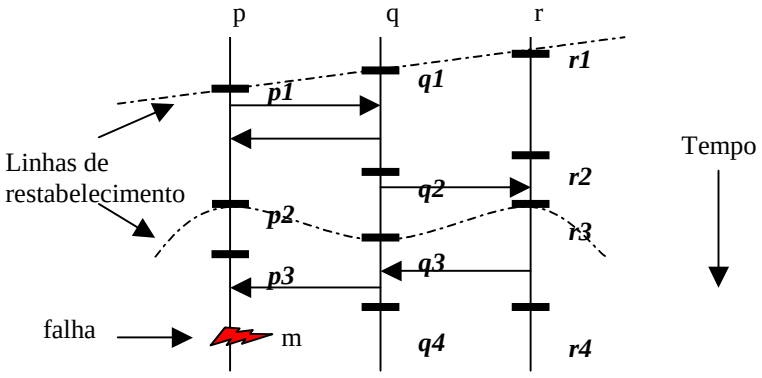

FIGURA 3 - Determinando linhas de restabelecimento.

Suponha-se que o processo *p* sofra uma falha no ponto indicado. Caso se considerem os *checkpoints* locais *p3* e *q4*, ver-se-á que o envio da mensagem *m* não satisfaz a condição 3. No entanto, o conjunto K = {*p2, q3* e *r3*} satisfaz as três condições, formando, portanto, a linha de restabelecimento (recuperação) mais recente. O conjunto K' = { *p1, q1* e *r1*} também representa um estado global consistente.

2.3.3.1.1 Protocolo de sincronização

Dentre os vários protocolos de sincronização de estados utilizados para construir linhas de restabelecimento, descrever-se-á aquele proposto e implementado por Zawaenepoel et al. [ZAW92].

O protocolo em questão parte do princípio de que um processo age como coordenador e envia mensagens aos outros processos para que sincronizem a obtenção do estado global consistente. Cada *checkpoint* local é identificado por um número denominado CCN (*Consistent Checkpoint Number*), incrementado a cada rodada do protocolo. Qualquer envio de mensagem entre os processos carrega o valor do contador CCN do remetente.

O protocolo é executado da seguinte forma:

1. o coordenador efetua um *checkpointing*, incrementa seu contador CCN e envia mensagens com marcadores que contêm CCN a cada processo do sistema;
2. ao receber a mensagem com marcador, cada processo efetua um *checkpointing* armazenando seu estado relevante em disco. Cada processo também efetua um *checkpointing* se receber alguma mensagem cujo CCN anexado seja maior que o valor do contador CCN local. Sendo tal mensagem enviada após o remetente ter iniciado a gravação do estado local, o receptor deve efetuar o *checkpointing* antes de processar a mensagem recebida;
3. após finalizar a gravação do estado local, cada processo envia uma notificação ao coordenador;
4. o coordenador coleta as notificações recebidas. Se todos conseguiram efetuar o *checkpointing* com sucesso, este envia uma mensagem informando-os de que está ciente do fato. Caso algum processo não tenha notificado o coordenador, este envia uma mensagem informando a cada um dos objetos um processo que descarte o *checkpoint* local recém-obtido.

Esse protocolo constrói linhas de restabelecimento, garantindo, portanto, que o conjunto de *checkpoints* locais individuais represente um estado global consistente.

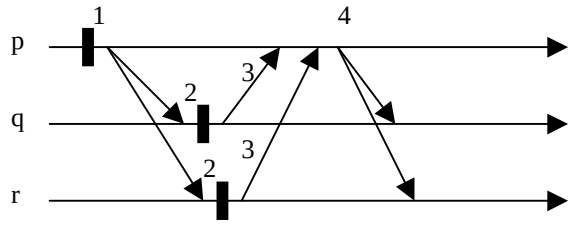

FIGURA 4 - Protocolo de sincronização.

2.3.3.1.2 Restabelecimento de estado

Para restabelecer o estado do sistema na presença de falha, deve-se reiniciar a execução a partir da linha de restabelecimento mais recente. Como existe no máximo um *checkpoint* local por processo e o conjunto de *checkpoints* forma um estado global consistente, este método de restabelecimento levará o sistema a um estado consistente.

Essa técnica, entretanto, não é a forma mais eficiente. Por exemplo, é provável que um dado processo faltoso não tenha efetuado comunicação com todos os processos do sistema, de forma que seja desnecessário restabelecer a execução de todos os processos a partir da linha de restabelecimento mais recente. Para ilustrar essa situação, considere-se o seguinte sistema formado por três processos (FIGURA 5):

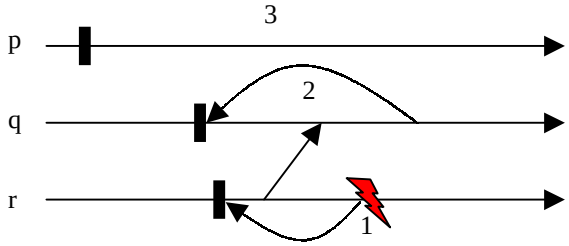

FIGURA 5 - Restabelecimento de estado.

1. o processo *r* sofre uma falha e retorna ao seu *checkpoint;*
2. como o processo *r* enviou uma mensagem ao processo *q*, detecta-se a existência de uma mensagem órfã, e o processo *q* é obrigado a voltar para o seu *checkpoint*;
3. como não houve comunicação do processo *p* com os demais, nem deles com o processo *p*, a recuperação do processo *p* a seu *checkpoint* não seria necessária, pois não existem mensagens órfãs nem mensagens perdidas, e o conjunto do estado local de *p* com os demais (*q* e *r*) forma um estado global consistente.

Pode-se dispor de uma técnica que forneça um maior nível de otimização, proposta por Koo e Toueg [KOO87].

A vantagem dos métodos de *checkpointing* coordenado é que o restabelecimento de estado do sistema torna-se bastante simples, pois basta que cada processo retome a execução a partir do respectivo *checkpoint* local determinado pela linha de restabelecimento mais recente para que um estado global do sistema seja naturalmente reconstruído. No entanto, é necessário efetuar trocas de mensagens entre os processos para garantir a sincronização no armazenamento dos estados locais, o que causa uma degradação de desempenho.

2.3.3.2 Checkpointing assíncrono

Fazendo uma comparação dos *checkpoints* síncronos com os assíncronos, pode-se dizer que, com os síncronos, tem-se facilidade na tarefa de recuperação, contudo com custo de processamento para estabelecer os *checkpoints*. Já, no assíncrono, não se têm problemas com o custo do processamento, porém a recuperação mostra-se mais complexa em razão da falta de um estado global consistente.

A idéia central é obter estados locais consistentes, e não um estado global consistente. A principal vantagem desse tipo de abordagem está em não ser necessário utilizar complexos protocolos para garantir que os estados locais reflitam um estado global consistente [CON97].

O estado local de cada processo é armazenado em disco (memória estável) no momento do *checkpointing* para possíveis futuras recuperações. Para obter os *checkpoints* locais consistentes, podem-se utilizar duas técnicas: automática e semi-automática. Na primeira, o programador não precisa se preocupar com a obtenção do estado local consistente, sendo esse efetuado pelo próprio sistema, o que implica um aumento do volume do *checkpoint* local e da complexidade dos algoritmos encarregados de armazená-los. Na técnica semi-automática, o programador da aplicação encarrega-se de invocar as rotinas de *checkpointing* em pontos específicos do programa onde esteja formado um estado local consistente.

Conforme já visto, a existência de comunicação implica que mais de um processo retome a execução a partir do último *checkpointing* efetuado. A solução para essa situação é garantir que todas as mensagens processadas pela nova instância do processo faltoso estejam disponíveis a partir de um *LOG* de mensagens em disco, por exemplo.

Dentre os algoritmos existentes para resolver esses problemas, apresenta-se o algoritmo proposto por [Xu93], cujos pontos de recuperação são efetuados em cada um

dos processos de forma assíncrona, ou seja, sem a existência de uma coordenação para criação dos mesmos.

O algoritmo utiliza os conceitos de dependência causal, caminho cruzado e ponto de recuperação útil, os quais são assim descritos:

- Dependência causal ou *causal path*: Diz-se que o processo *p* tem dependência causal de *q* quando sua execução depende diretamente da execução de *q*. Em outras palavras, o processo *p* depende dos dados de *q* para poder ser executado (ocorre uma troca de mensagem entre *p* e *q*). Em um sistema de *checkpoints*, um *checkpoint* não pode sofrer dependência causal de outro *checkpoint* em outro processo, sob pena de haver um efeito dominó. Portanto, para que um ponto de recuperação não sofra dependência causal de outro ponto de recuperação, não pode existir troca de mensagens entre eles.
- Caminho cruzado ou *zigzag path*: Um caminho cruzado existe se ocorrer uma das três condições:
  a) entre dois pontos de recuperação em processos distintos, se existir o envio de uma mensagem do primeiro para o segundo após a gravação do primeiro ponto de recuperação e antes da gravação do segundo ponto;
  b) uma mensagem recebida por um processo e a próxima mensagem é o envio pelo processo no mesmo ou posterior intervalo de recuperação;
  c) a mensagem é recebida após o estabelecimento de um ponto de recuperação.
- Ponto de recuperação útil ou *useful checkpoint*: Um ponto de recuperação é considerado útil se pertencer a um ponto de recuperação global consistente. Caso ocorra um caminho cruzado ou a uma dependência causal, este ponto de recuperação é considerado não-útil.

Conforme Rebonatto [REB98], o algoritmo proposto tem como finalidade minimizar o efeito dominó (FIGURA 6), tornando útil o maior número de pontos de recuperação; ele procura detectar dependências causais e caminhos cruzados entre os pontos de recuperação. Caso os detecte, o algoritmo cria novos pontos de recuperação.

O algoritmo segue as dependências causais entre pontos de recuperação, fazendo com que cada processo mantenha um vetor de dependência (DV). Cada processo controla os intervalos de seus próprios pontos de recuperação e o vetor com os números dos pontos de recuperação, um para cada processo que participa do sistema. O conteúdo da *i*-ésima entrada do vetor DV é o índice do último ponto de recuperação do processo *i*, que tem relação causal com o ponto atual do processo. A entrada referente ao próprio processo contém o número atual do seu ponto de recuperação.

Para manter esse vetor, cada processo adiciona seu vetor DV a cada mensagem que envia; ao receber uma mensagem, atualiza seu vetor com os valores máximos contidos no vetor anexado à mensagem recebida. Para permitir a detecção de caminhos cruzados, cada processo copia seu vetor DV para um outro vetor ZV sempre que fizer um ponto de recuperação. Assim, quando um processo *p* envia uma mensagem a um processo *q*, também acrescenta a *q*-ésima entrada do seu vetor ZV (chamada de Z*id*) à mensagem. Esse valor indica o último ponto de recuperação no processo *q* que tem um caminho causal com o processo *p*.

Para localizar mensagens que completam caminhos cruzados, o algoritmo é executado a cada recepção de mensagens. Após a mensagem ser recebida, mas antes de ser processada, o algoritmo verifica se existe um caminho causal desde o seu ponto de

recuperação atual até o ponto de recuperação que precede o envio da mensagem enviada. Essa verificação é feita comparando-se o valor Z*id* anexado à mensagem ao seu número atual de ponto de recuperação. Se esses valores forem iguais, o referido caminho causal existe e um novo ponto de recuperação é feito antes de a mensagem recebida ser processada. Em caso contrário, a mensagem pode ser processada sem a necessidade de um novo ponto de recuperação.

```
; DV_REC é o vetor DV anexado a mensagem recebida
1)  Zid = O índice acrescentado à mensagem recebida
2)  Se Zid = número do ponto de recuperação atual então
3)     Criar um novo ponto de recuperação
4)  Fim se
5)  Para j = 1 até tamanho de DV
6)      Se DVj < DV_RECj então
7)         DVj = DV_RECj
8)      Fim se
9)  Fim para
```

FIGURA 6 – Algoritmo para minimizar o efeito dominó.

A FIGURA 7 ilustra a utilização do algoritmo anteriormente descrito. Conforme essa figura, o ponto de recuperação 1 do processo *q* é um ponto de recuperação útil, formando um estado global consistente com o ponto de recuperação 2 do processo *p*.

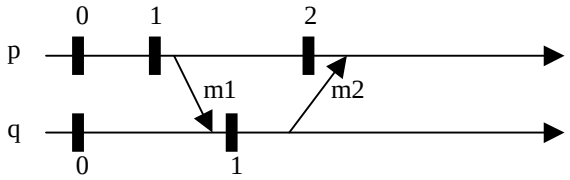

FIGURA 7 - Pontos de recuperação consistentes.

Conclui-se, com o algoritmo apresentado, que há a diminuição do efeito dominó e que sua principal virtude é transformar a maior parte dos pontos de recuperação em pontos de recuperação consistentes, detectando quando existem caminhos cruzados ou dependência causal entre os pontos. Porém, o algoritmo utiliza memória adicional e acrescenta processamento a todos os recebimentos de mensagens.

2.4 Blocos de Recuperação

A estrutura de blocos de recuperação foi criada por um grupo de pesquisadores da Universidade de Newcastel-on-Tyne [RAN75] e representa uma abordagem dinâmica de tolerância a falhas em *software*.

O ponto de partida para a utilização de blocos de recuperação é a disponibilidade de várias versões de programas aplicativos que atendam aos requisitos definidos na especificação do problema a ser solucionado.

A estrutura, basicamente, consiste de três elementos: uma rotina primária, que executa funções críticas; um teste de aceitação, que testa a saída da rotina primária após cada execução, e uma rotina alternativa, que realiza a mesma função da rotina primária

(porém, com menor capacidade ou mais devagar) e é disparada pelo teste de aceitação ao detectar uma falha.

A execução inicia-se pela primeira versão (rotina primária), sendo seu resultado é submetido ao teste de aceitação: se for aceito, as demais versões não são executadas; em caso contrário, a segunda versão (rotina alternativa 1) é executada e submetida ao teste de aceitação. Para recuperação de erros, o bloco de recuperação emprega o mecanismo de recuperação por retrocesso. Antes de executar a rotina primária, um ponto de recuperação é estabelecido, salvando o estado naquele ponto. Se um erro é detectado pelo teste de aceitação, o processo é restaurado ao estado, salvo no momento em que o ponto de recuperação foi estabelecido. Caso isso aconteça, todos os efeitos deste módulo são desfeitos, e o sistema está em um estado consistente. Após a recuperação de erros ter sido realizada, uma nova alternativa é executada, e o processo continua até que o resultado de execução de alguma versão seja aceito ou se tenham esgotado todas as alternativas possíveis. Nesse caso, uma exceção é levantada, indicando que nenhum módulo passou pelo teste de aceitação (FIGURA 8).

As rotinas alternativas (módulos) devem ser projetadas sobre a mesma especificação. Isto é, elas realizam as mesmas tarefas da rotina primária, mas são projetadas diferentemente. Caso a rotina primária venha a falhar, as rotinas alternativas são utilizadas para realizar as tarefas requeridas (tarefas nomeadas na especificação). É muito importante que as rotinas alternativas tenham independência de projeto, para que quando uma falha na rotina primária acontecer, as alternativas não falhem também [JAL94].

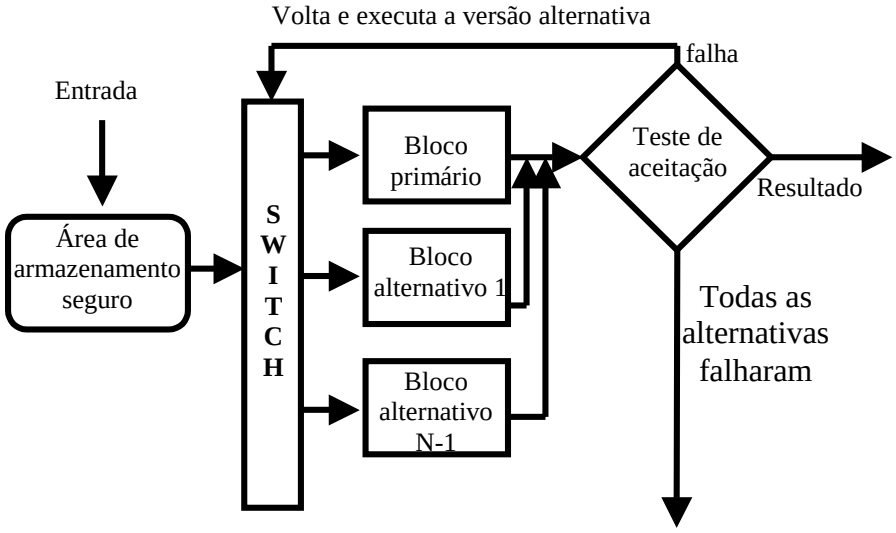

FIGURA 8 - Blocos de recuperação.

O esquema de bloco de recuperação combina diversas versões de *software* organizadas (conceitualmente, pelo menos) em ordem seqüencial, embora as versões também possam ser organizadas para executar concorrentemente.

Em seqüência, apresenta-se a estrutura básica de um bloco de recuperação, no qual *T* é a condição do teste de aceitação que se espera encontrar pela execução com sucesso da rotina primária *P* ou pela rotina alternativa *Q* (FIGURA 9). A estrutura também pode ser expandida para acomodar diversas alternativas *Q1*, *Q2*, … *Qn*. A cláusula erro sinaliza uma exceção caso todas as alternativas falhem.

Antes da execução de um bloco de recuperação, um ponto de recuperação (*checkpoint*) é estabelecido, salvando, assim, o estado atual de qualquer variável que poderia ser mudada durante a execução do bloco de recuperação em uma área de armazenamento seguro. A rotina primária (versão 1) é executada depois que o teste de aceitação é avaliado para fornecer uma decisão de resultado dessa rotina primária. Se um erro for detectado pelo teste de aceitação, ou se qualquer erro for descoberto através de outros meios durante a execução do módulo, uma exceção é levantada e a recuperação de erro por retorno é invocada, restabelecendo-se o estado do sistema ao que era na entrada. Depois, a execução continua com a primeira alternativa (versão 2) usada como um objeto de reserva temporária. Esse processo de recuperação continua até que o teste de aceitação seja aceito ou o fornecimento de alternativas seja esgotado.

```
Aceite T < Teste de aceitação>
Por P <Rotina Primária>
Senão por Q < Rotina Alternativa>
Senão erro
```

FIGURA 9 - Operação de um bloco de recuperação 1.

Se todas as alternativas falharem, o teste resultará em uma exceção (em razão de um erro interno que é descoberto), e uma exceção de falha será sinalizada ao ambiente do bloco de recuperação. Considerando que podem ser aninhados blocos de recuperação, o levantamento de uma exceção de um bloco de recuperação interno invocará a recuperação no bloco incluído. A operação do bloco de recuperação é ilustrada na Figura 10.

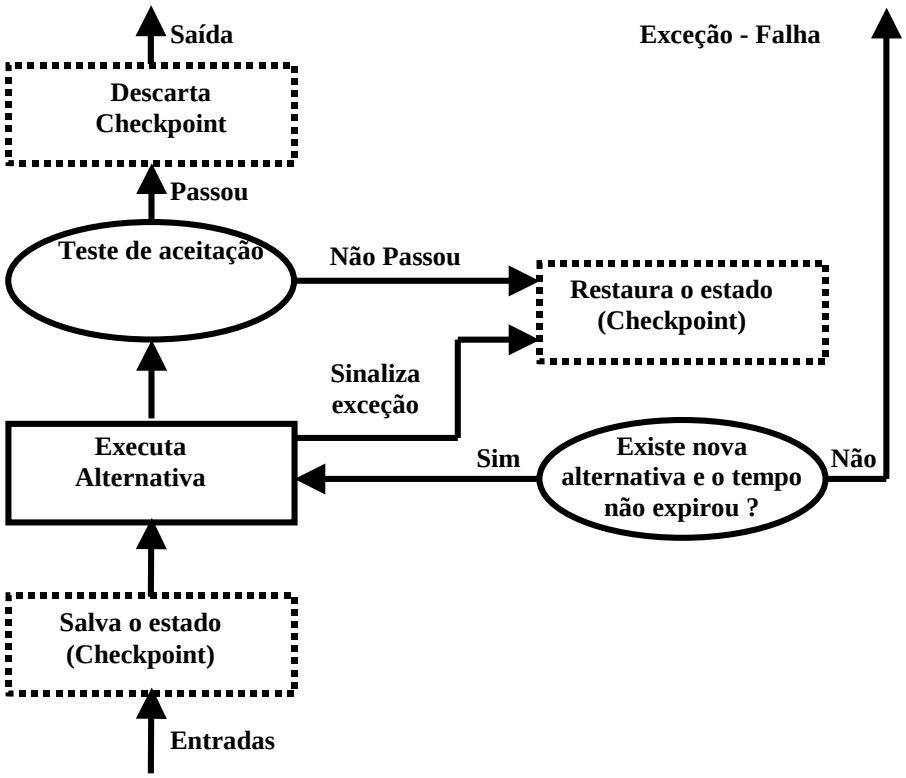

FIGURA 10 - Operação de um bloco de recuperação 2.

Obviamente, a estrutura lingüística para blocos de recuperação requer um mecanismo satisfatório para fornecer automaticamente recuperação de erro por retorno. Randell produziu o primeiro esquema de recuperação encoberta (*recovery cache*), uma descrição na qual foi incluído o primeiro artigo em blocos de recuperação [HOM74] (embora esse esquema tenha sido substituído por [Anderson e Kerr, 1976]. Esse artigo também inclui uma discussão de "procedimentos recuperáveis" - um mecanismo bastante complexo que Lauer e Randell tinham proposto como um meio de estender o esquema de recuperação encoberta para lidar com tipos de dados definidos pelo programador. Essa parte do artigo estaria, indubitavelmente, muito mais clara se as idéias tivessem sido expressas em condições orientadas a objetos.

O sucesso global do esquema de blocos de recuperação descansa em grande parte na efetividade dos mecanismos de detecção de erro usados, especialmente (mas não somente) em testes de aceitação. Um teste de aceitação altamente confiável é essencial à implementação de um módulo de bloco de recuperação. O teste de aceitação deve ser mantido razoavelmente simples, minimizando, assim, a probabilidade de que o próprio teste de aceitação contenha erros para prover uma alta cobertura de detecção de erro. Além disso, o teste de aceitação acarretará uma sobrecarga de tempo de execução, que pode ser inaceitável se ele for muito complexo [RAN94]. "Uma das maiores dificuldades na utilização de blocos de recuperação é a determinação de testes de aceitação adequados." [POR90].

De fato, teste de aceitação em um bloco de recuperação deveria ser considerado como uma última linha de detecção de erros, mais do que um único meio de detecção de erro. A expectativa é de que seja sustentado através de declaração de afirmações executáveis dentro dos módulos e verificação em tempo de execução de um módulo que conduzirá à mesma ação de recuperação adotada para o teste de aceitação falho. A falha da alternativa final, se não passar no teste de aceitação, por exemplo, constitui uma falha do módulo inteiro que contém o bloco de recuperação, invocando recuperação em nível de bloco de recuperação circunvizinho, que deveria estar lá.

Em outras palavras, cada alternativa deveria ela própria ser um componente ideal tolerante à falha. Uma exceção levantada por declarações de afirmação de tempo de execução dentro da alternativa, ou através de mecanismos de erro de *hardware*, pode ser negociada pelas próprias capacidades de tolerância a falhas da alternativa. Uma exceção de falha é levantada para notificar o sistema (o componente de controle do nosso modelo) se, apesar do uso de suas próprias capacidades de tolerância à falha, a alternativa estiver impossibilitada de fornecer o serviço requisitado. O componente de controle pode, então, invocar outra alternativa.

Embora cada uma das alternativas dentro de um bloco de recuperação compreenda satisfazer o mesmo teste de aceitação, não há nenhuma exigência de que todas elas devam produzir os mesmos resultados [LEE78]. A única reserva é que os resultados devem ser aceitáveis, como for determinado pelo teste. Assim, enquanto a alternativa primária deve tentar produzir o resultado desejado, a alternativa adicional só pode tentar fornecer um serviço degradado.

O papel do bloco de recuperação consiste simplesmente em descobrir e recuperar erros, ignorando a operação que descobriu a falha.

2.5 Programação N-Versões

A Programação N-Versões proposta por Avizienis [AVI85] consiste na geração de n-versões de uma aplicação. Com n maior ou igual a 2, são essas aplicações originárias da mesma especificação, mas programadas com ferramentas, linguagens, métodos e testes diferentes. Essa técnica utiliza um sistema de votação das n-versões para geração do resultado final, trabalhando, portanto, com um mascaramento de falhas.

As diferentes versões são executadas paralelamente ou seqüencialmente, e os resultados das versões são transmitidos ao votador responsável por determinar um resultado consensual para a computação, de acordo com algum critério. Por exemplo, com maior freqüência, o resultado será aquele dado pela maioria dos componentes; caso o votador não seja capaz de determinar o resultado, uma exceção será sinalizada. A FIGURA 11 mostra um mecanismo de n-versões composto por três componentes e um votador.

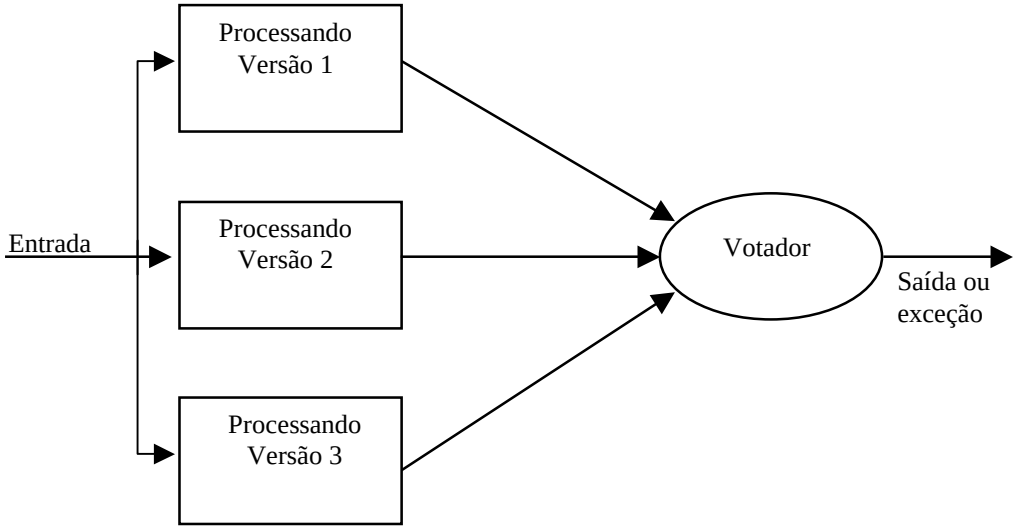

FIGURA 11 - Programação n-versões.

Um método para controlar falhas de projeto é ter n-versões de *software* com projetos diferentes, mas sobre as mesmas especificações. Nas n-versões, é importante eliminar modos comuns de fracasso da seguinte forma:

- usando linguagens diferentes para cada versão;
- usando compiladores diferentes e ambientes de apoio para cada versão;
- usando, se possível, algoritmos diferentes para cada versão.

Se apenas duas versões forem usadas, o sistema poderá detectar um único erro, comparando a saída das duas versões. Por isso, o método oferece certa dificuldade para identificar a versão errônea. Sistemas que usam essa configuração geralmente adotam uma técnica pela qual o sistema termina sua execução assim que uma diferença é detectada entre as duas versões. São exigidas três versões independentes para que se chegue a um consenso dos resultados através da maioria.

Característica de blocos de recuperação, a técnica de n-versões provê tolerância à falha de forma transparente ao usuário; assim, um projetista de um sistema pode incorporar tolerância a falhas apenas nos módulos críticos. O mecanismo de decisão (votador) em um ambiente de *software* n-versões pode ser o único ponto para falha, devendo, portanto, ser extremamente confiável.

3 Reflexão Computacional

O paradigma reflexivo permite a um sistema fazer computações sobre si mesmo com o objetivo de alterar sistemas de forma dinâmica. Define uma arquitetura em níveis, denominada *arquitetura reflexiva*, composta por um metanível, onde se encontram as estruturas de dados e as ações a serem realizadas sobre o sistema-objeto localizado no nível-base [LIS98]. A FIGURA 12 mostra o funcionamento de uma arquitetura reflexiva, na qual as ações no metanível são executadas sobre os dados que representam informações sobre o programa de nível-base, o qual executa ações sobre os seus próprios dados, com a finalidade de atender aos usuários de seus serviços.

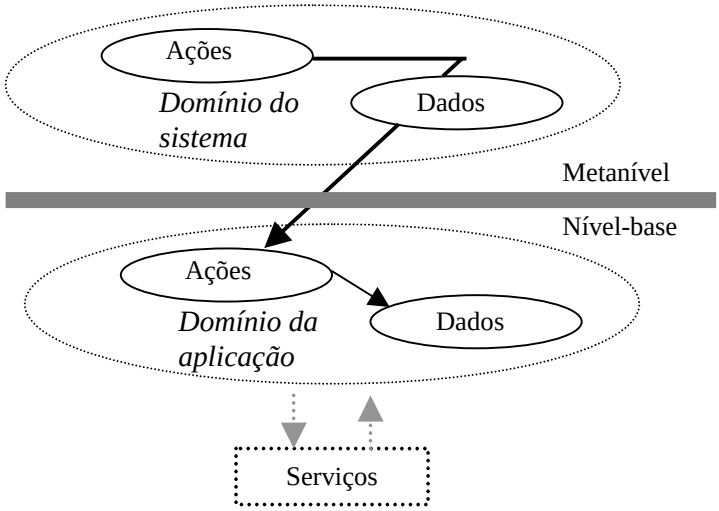

FIGURA 12 - Arquitetura reflexiva [LIS98].

3.1 Conceitos Básicos e Terminologias

Nos tópicos seguintes, abordam-se os principais conceitos de reflexão computacional e terminologias e apresentam-se algumas linguagens que têm características reflexivas.

Basicamente, reflexão computacional é obtida quando o programa pára sua execução por um período de tempo para fazer computações sobre si próprio; analisa seu estado, se o processamento está correto, se pode prosseguir com a execução e atingir o objetivo satisfatoriamente; se não precisa mudar de estratégia ou algoritmo de execução, fazendo, ainda, processamentos necessários para o sucesso da execução.

Em linguagens orientadas a objetos, a reflexão computacional é realizada através de metaclasses ou metaobjetos que se distinguem por sua abrangência: a reflexão realizada por meio de uma metaclasse altera todas as instâncias da classe, enquanto a reflexão realizada por meio de um metaobjeto refere-se a uma única instância. Em ambas as formas, o objetivo é fornecer informações sobre a representação dos métodos e das propriedades das classes ou objetos da aplicação [LIS98].

A reflexão computacional pode ser implementada em diferentes modelos, destacando-se o modelo de metaclasses e de metaobjetos.

O modelo de metaclasses é adotado em linguagens que estruturam as suas classes a partir de metaclasses. Esse modelo é conhecido como reflexão estrutural, visto que permite obter informações e realizar transformações sobre a estrutura estática de uma classe.

No modelo de metaobjetos, a cada objeto da aplicação pode ser associado um metaobjeto, que representa aspectos estruturais e comportamentais de um objeto a ele conectado. As classes do objeto e do metaobjeto são distintas, tornando esse modelo mais flexível do que o modelo de metaclasses, por ser a associação realizada através de objetos, e não de classes.

Conforme Lisboa [LIS98], um *metaobjeto* pode ser definido como um objeto que representa aspectos estruturais e comportamentais de um certo objeto a ele conectado. A estrutura do objeto é representada como atributos do metaobjeto, e os aspectos computacionais são descritos como métodos desse *metaobjeto*.

A adoção de reflexão em uma aplicação torna possível ao programador acessar as políticas de controle de execução. As mudanças de políticas em nível-meta permitem alterar o comportamento do sistema sem precisar modificar o código da aplicação de nível-base. Desse modo, podem-se implementar técnicas necessárias para fornecer segurança e credibilidade aos sistemas de *software*, uma vez que a separação em níveis apresenta vantagens, como o desenvolvimento individual de objetos e metaobjetos.

3.2 Linguagens com Características Reflexivas

Segundo Lisboa [LIS98], as linguagens orientadas a objetos oferecem condições propícias à utilização da reflexão como uma ferramenta adicional para a implementação de extensões mais flexíveis.

Algumas linguagens orientadas a objetos utilizam características reflexivas; já, para outras, foram desenvolvidas extensões que dão apoio à reflexão.

A seguir, descrevem-se rapidamente as vantagens permitidas pelas características reflexivas encontradas nas linguagens Open C++ e Java.

3.2.1 Metaobjetos de Open C++

O pré-processador Open C++ Versão 1.2, desenvolvido por Chiba, é uma extensão de C++, uma vez que essa linguagem não oferece facilidades reflexivas.

O Open C++ permite ao programador estender métodos e acesso a atributos em tempo de execução. Ele adota o modelo de metaobjetos, isto é, para cada objeto há um único metaobjeto que pode controlar a chamada de métodos e o acesso a variáveis. Metaobjetos são instâncias de metaclasses que descendem da classe MetaObj.

Em Open C++, existem dois tipos de objetos: objetos reflexivos, que são controlados pelo seu metaobjeto correspondente, e objetos não reflexivos, que são objetos C++ normais.

Quando um método reflexivo é invocado, sua execução é desviada para o metanível, ou seja, o metaobjeto passa a controlar a execução do método (FIGURA 13).

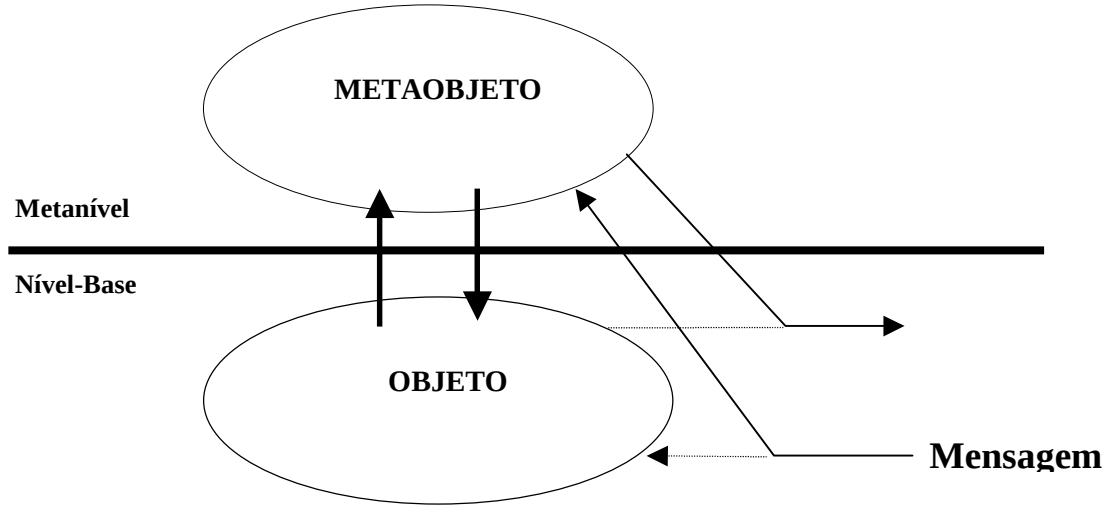

FIGURA 13 - Reflexão computacional em Open C++.

O metaobjeto precisa conhecer o método em questão e seus argumentos. Possuindo tais informações, é executado o método do nível-base, retornando, em seguida, os resultados para o usuário.

Com base em Lisboa[LIS98], metaobjetos podem ser criados como classes comuns de C++, tendo como classe ancestral a classe MetaObj, a qual define os métodos que um metaobjeto pode usar para controlar seu referente. Por ser uma classe comum de C++, um metaobjeto pode receber herança múltipla, isto é, ter uma ou mais classes ancestrais além de MetaObj; e pode sofrer especializações, isto é, pode ser usado como classe-base de outras classes descendentes, transmitindo, assim, indiretamente o protocolo herdado da classe MetaObj.

A computação no metanível é realizada por redefinição dos métodos adquiridos da classe MetaObj e pelos métodos particulares (definidos ou herdados) da classe do metaobjeto em questão.

3.2.2 Reflexão computacional em Java

O modelo abstrato da Máquina Virtual Java (JVM) determina que todas as classes e interfaces são objetos do tipo *Class*. Objetos do tipo *Class* são automaticamente criados quando novas classes são carregadas para ser executadas e contêm informações sobre o nome da classe, a superclasse, a lista de interfaces implementadas e o carregador de classes utilizado [LIS98].

A linguagem Java apresenta características reflexivas por concepção, pois informações sobre os objetos da aplicação, como o nome da classe e o nome da superclasse do objeto, são proporcionadas em tempo de execução através de métodos que se referem ao objeto. Por exemplo, this.getName() e this.getClass() retornam, respectivamente, o nome da superclasse e o nome da classe do objeto em questão.

A seguir, dá-se um exemplo da utilização de métodos que proporcionam informações de identificação dos objetos, como os nomes da classe e da superclasse desses:

```
import java.awt.*;
public class exemplo extends Button
{
	public exemplo(String nome, boolean estado)
	{
		setLabel( nome );
		setEnabled( estado );
	}
	public static void main( String args[] )
	{
		Label l = new Label();
		exemplo b1 = new exemplo( " Botao 1 ", true);
		exemplo b2 = new exemplo( " Botao 2 ", false);
		System.out.println( " Superclasse do Objeto b1: " + b1.getName() );
		System.out.println( " Classe do Objeto b1: " + b1.getClass() );
		System.out.println( " Superclasse do Objeto b2: " + b1.getName() );
		System.out.println( " Classe do Objeto b2: " + b2.getClass() );
		System.out.println( b1.getClass().getName() );
		System.out.println( l.getClass().getName() );
	}
}
/* Resulta em:
Superclasse do Objeto b1: button
Classe do Objeto b1: class exemplo
Superclasse do Objeto b2: button
Classe do Objeto b2: class exemplo
exemplo
java.awt.Label
*/
```

Recentemente, a linguagem Java vem inserindo novos mecanismos de reflexão computacional em seu ambiente de execução através de protocolos de metaobjetos.

O protocolo metaJava, desenvolvido por Michael Golm [GOL97], implementa características reflexivas em Java pelo adicionamento de extensões à máquina virtual Java. Implementa métodos de interface-meta e apresenta modificações na estrutura do objeto Java.

Na abordagem reflexiva do protocolo, os eventos ocorridos na computação do nível-base são transmitidos ao nível-meta, que reage de acordo com a avaliação do evento. Quando o metaobjeto recebe um evento, executa um método, podendo sincronizar a execução desse com outro método do objeto-base. A computação no nível-base é suspensa enquanto o metaobjeto processa o evento, concedendo ao metanível o controle sobre a atividade no nível-base.

Em metaJava, um metaobjeto pode estar associado a uma referência, a uma classe ou a um objeto. Quando associado a um objeto, os eventos desse são enviados a seu respectivo metaobjeto, bem como todos os eventos das instâncias de uma classe, quando essa é associada a um metaobjeto. Os eventos criados por operações de uma referência são enviados ao metaobjeto quando este está associado à referência em questão.

A associação entre objeto e metaobjeto é dinâmica em tempo de execução, especialmente se o metanível controla uma arquitetura distribuída. Se nenhum metaobjeto está associado a uma aplicação, a arquitetura meta não causa sobrecarga.

Múltiplos metaobjetos podem ser associados a um componente de nível-base, formando uma hierarquia de metaobjetos na qual a ordem é importante, ou seja, o metaobjeto de mais recente associação é executado em primeiro lugar [LIS98].

Para executar suas tarefas, o metaobjeto tem acesso ao conjunto de métodos com os quais pode manipular o estado interno da máquina virtual. Esses métodos são chamados de *metainterface da máquina virtual* [GOL97].

O exemplo a seguir ilustra a invocação de métodos, imprimindo o nome do método, bem como os argumentos do método invocado, e dando prosseguimento à computação no nível-base.

```
public class MetaTrace extends MetaObject
{
        public void attachObject(object o, String methodnames[]){
                createStubs(o,methodnames);
                attachObject(this,o);
        }
        public void eventMethodEnterVoid(EventMethodCall event){
                System.out.println("Method" + event.methodname + "called!");
                System.out.println("Signature" + event.signature);
                for(int i=0; i<event.n_args; i++){
                        System.out.println("Arg:" + event.arguments[i].toString());
                 }
                 continueExecution Void(event);
        }
        public Object eventMethodEnterObject( eventMethodCall event){
                System.out.println("Method"+event.methodname + "called!");
                System.out.println("Signature" + event.signature);
                for(int i=0; i<event.n_args; i++){
                        System.out.println("Arg:" + event.arguments[i].toString());
                 }
                Object ret = continueExecutionObject(event);
                System.out.println("returned:" + ret.tostring());
                return ret;
        }
}
```

Primeiramente, é criado no metanível um metaobjeto correspondente ao objeto, contendo os métodos do componente do nível-base; após, é realizada a associação entre o metaobjeto e o objeto. O metaobjeto realiza o seu processamento retornando ou não valores para o objeto. Se o metaobjeto retorna algo ao objeto, este retorna uma resposta a quem lhe enviou a mensagem. Metaobjetos também são utilizados na invocação de métodos em sistemas distribuídos.

O metaJava apresenta flexibilidade e funcionalidade na aplicação sob forma de biblioteca; concede aos metaobjetos a manipulação e o controle de métodos e variáveis, bem como a identificação desses como parte da ação realizada sobre o objeto. Permite a utilização de metaobjetos na implementação de normas de segurança, de forma que um metaobjeto pode conferir todos os acessos a um objeto especificado.

Esse protocolo permite também que mecanismos de tolerância a falhas sejam implementados no metanível, possibilitando a resolução de vários problemas como, por exemplo, a sincronização.

3.3 Conclusões

A reflexão computacional trata do comportamento da aplicação não interferindo na aplicação em si; permite ao programador acessar as políticas de controle de execução, tornando, assim, menores as chances de ocorrer algum erro. A reflexão computacional apresenta uma arquitetura chamada *arquitetura reflexiva*, composta por dois níveis: o nível-base, composto de objetos responsáveis pelos aspectos de funcionalidade da aplicação, e o nível-meta, composto de objetos responsáveis pela administração da aplicação.

O emprego do comportamento reflexivo em uma linguagem de programação possibilita o desenvolvimento de aplicações no modelo de objetos através da separação entre código funcional e não-funcional. Como o código reflexivo está separado do código base, ambos podem ser reutilizados.

Verifica-se que o número de linguagens de programação que oferecem bons mecanismos de reflexão é bem reduzido. Assim, a linguagem Java é uma boa proposta em vista de sua atualidade, simplicidade e por ser um modelo flexível de segurança para os programas.

O protocolo metaJava amplia o ambiente Java adicionando mecanismos de reflexão computacional através de uma interface-meta. Com isso, tem-se um maior controle de qualquer componente do sistema em tempo de execução, sem necessidade de modificar o código funcional da aplicação.

4 Projetos de Programas

Neste capítulo, introduzem-se os conceitos e idéias ligados ao projeto de programas no modelo orientado a objetos, programação paralela, distribuída e de tempo real.

4.1 Programação Orientada a Objetos

A busca atual por melhores linguagens de programação está concentrada nas linguagens orientadas a objetos, cujas técnicas e ferramentas incluem linguagens, sistemas, interfaces, ambientes de desenvolvimento, bases de dados, bibliotecas de classes, etc. As linguagens orientadas a objetos possuem diversas características que facilitam a sua implementação em relação às outras linguagens imperativas [HAE98].

O modelo de programação orientado a objetos utiliza conceitos já conhecidos nas áreas de linguagens de programação, como ocultamento de informações, tipo de dados abstratos, etc. Estendendo esses conceitos, criou-se um novo estilo de programação. Um programa orientado a objetos consiste num conjunto de classes que definem objetos de estrutura e comportamento idênticos. Essa característica é semelhante ao tipo de dados de uma linguagem convencional, como o integer do Pascal, que define variáveis com propriedades do tipo inteiras. A classe é um tipo que define objetos com as propriedades com que foi construída.

Uma das principais características do modelo de objetos é o suporte para abstração de dados, a habilidade de definir novos tipos de objetos cujo comportamento é descrito de forma abstrata, sem se fazer referência a detalhes de implementação, como as estruturas de dados e algoritmos usados para representar os objetos. Com a abstração de dados, os objetos podem ser manipulados somente através de requisições de uma operação pública do objeto (troca de mensagens), o que sugere a necessidade de ser precisamente definida a interface entre os objetos.

O encapsulamento apresenta muitas vantagens, como maior simplicidade para a compreensão e modificação dos programas e maior garantia de integridade dos dados. O ideal é que nenhuma parte de um sistema complexo dependa dos detalhes internos de qualquer outra parte, devendo-se minimizar a exposição de detalhes internos de implementações nas interfaces. A interface deve ser a mínima possível. É possível construir aplicações com bons níveis de segurança e confiabilidade através do encapsulamento de dados e funções dentro de objetos, que também representam a unidade básica de execução em uma linguagem OO.

Os componentes de programas orientados a objetos são basicamente classes e objetos.

4.1.1 Classes

Os programas desenvolvidos seguindo o modelo de orientação a objetos são organizados como coleções de objetos que interagem entre si, cada um representando uma instância de alguma classe, os quais são membros de uma hierarquia de classes unidas através do relacionamento de herança. Um conjunto de objetos com estrutura de dados e comportamento comum representa uma classe.

Herança é o mecanismo através do qual é possível a criação de novas classes a partir de outras já existentes, havendo uma transferência da estrutura de dados e implementação já existente. Com isso, é possível a criação de classes mais especializadas a partir de outras mais específicas, constituindo uma hierarquia de classes. A herança permite a construção de sistemas de forma incremental e evolutiva, possibilitando a reutilização de códigos já escritos [HAE98]. Uma classe herda a estrutura de dados e a implementação de outra de nível hierárquico mais elevado (a superclasse) e, da mesma forma, pode ter sua estrutura de dados e implementação herdada para outra classe (a subclasse). Uma classe pode herdar o comportamento de somente uma classe (herança simples), ou de mais de uma classe (herança múltipla). O mecanismo de herança introduz duas vantagens:

- qualquer alteração na classe será assimilada por todas as suas subclasses;
- facilita-se a reutilização de classes já existentes.
- Um modelo genérico de classe contém os seguintes itens:
- nome da classe;
- lista de superclasses que transferem propriedades a essa classe;
- declaração de variáveis que irão compor o estado interno de cada objeto dessa classe;
- declaração das operações que irão compor os métodos que definirão o comportamento dos objetos dessa classe.

O modelo de programação seguindo esse paradigma introduz características bastante interessantes, sendo a mais imediata delas a modularidade encontrada no conceito de objetos que encapsula dados e funções.

4.1.2 Objetos

A programação orientada a objetos utiliza objetos em vez de algoritmos como seus blocos fundamentais. Todos os objetos são criados ou instanciados a partir de classes, que são extensões dos tipos de dados abstratos. Os objetos que possuem a mesma estrutura de dados e operações pertencem à mesma classe. Cada objeto é constituído por uma estrutura de dados e um conjunto de métodos associados que implementam as operações realizadas por aquele objeto e manipulam os seus dados (FIGURA 14) [HAE98].

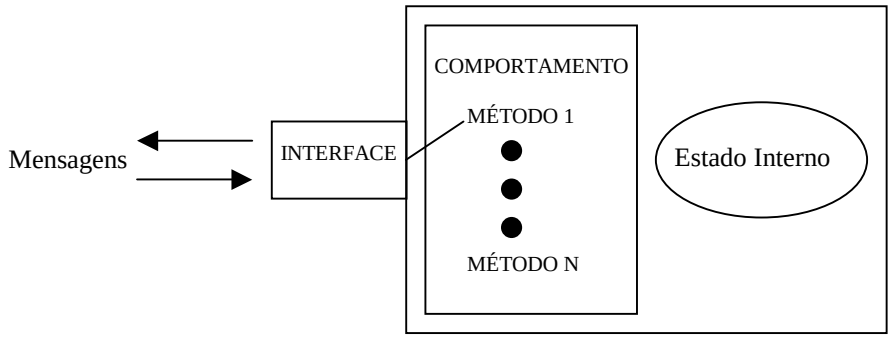

FIGURA 14 - Modelo genérico de objetos.

- Estado interno: nessa área, são mantidas as variáveis privadas do objeto acessíveis por qualquer método do objeto.

- Comportamento do objeto: contém o conjunto de métodos responsáveis pela implementação do objeto.
- Método: função ou procedimento que realiza as operações sobre os dados do objeto.
- Interface: é a parte do objeto visível externamente, compreendendo métodos e dados públicos.
- Mensagens: é formada pelo objeto destinatário, pelo método e argumentos para o método; é a principal forma de comunicação entre objetos.

O ocultamento de informações é importante por assegurar a integridade dos dados, pois não permite que o estado interno de um objeto seja modificado por outro objeto qualquer. As variáveis contidas no estado interno são privadas, e somente os métodos do objeto podem modificá-las. A modificação do estado interno do objeto é feita através da interface.

A maneira como um método implementa uma ação pode variar em função dos argumentos que ele recebe, podendo gerar respostas diferentes para argumentos diferentes, o que é chamado de *polimorfismo*. O polimorfismo exige algumas amarrações em tempo de execução; assim, o código executável de um objeto possui uma estrutura dinâmica, dificultando a verificação e validação de programas.

Objetos são criados e destruídos dinamicamente durante a execução de um programa. Cada objeto novo instanciado dá origem a uma nova estrutura com dados próprios; assim, num programa orientado a objetos, não existe uma noção de estado global, sendo mais natural falar-se do estado independente de cada um dos objetos instanciados.

4.2 Programação Paralela

Entende-se por *computação paralela* a execução de mais de uma tarefa por um ou mais processadores no mesmo intervalo de tempo. Dois processos são ditos *paralelos* quando são executados de forma concorrente e independente, exceto em pontos em que se comunicam ou sincronizam [HAE98].

Quando se fala em sistemas paralelos, dois tipos podem ser definidos:

- *Sistemas logicamente paralelos*: as várias tarefas são executadas em um único processador. Tem-se a ilusão de que todas as tarefas estão sendo executadas ao mesmo instante, porém, na verdade, o tempo de processamento é dividido entre as várias tarefas, cada uma ocupando uma fatia do tempo do processador. Esse tipo de sistema apresenta maior desempenho, pois ocupa o tempo ocioso durante uma operação de I/O para executar outra tarefa, sendo chamado de *multiprogramado* ou *multitarefa*.

- *Sistemas fisicamente paralelos*: neste tipo de sistema, as várias tarefas são executadas em mais de um processador no mesmo instante. São chamados de *sistemas multiprocessados*. O desempenho do sistema pode ser aumentado adicionando-se mais processadores para executar um maior número de tarefas simultaneamente. Os vários processadores compartilham a memória primária, o que não acontece nos sistemas distribuídos.

A programação paralela surgiu para possibilitar o desenvolvimento de programas que possam tirar o máximo proveito desses tipos de sistemas.

Os multicomputadores assemelham-se a multiprocessadores, pois ambos executam diferentes tarefas sobre um conjunto diferentes de dados. Porém, os muliprocessadores possuem uma área de memória comum e compartilham o mesmo relógio do sistema, ao passo que os multicomputadores possuem memória distribuída em mais de um sistema e não compartilham um relógio em comum, efetuando a troca de informações através de um sistema de comunicação.

Na cooperação de tarefas visando resolver um problema comum, esses detalhes são bastante significativos. Em sistemas multiprocessados, a comunicação e a sincronização são realizadas através da memória comum entre os processadores; já, em sistemas distribuídos, isso é feito por meio de troca de mensagens através da rede na qual estão interligados. Dessa forma, os sistemas distribuídos também consistem em uma forma de processamento paralelo, já que se baseiam na execução cooperativa e simultânea de processos.

Uma linguagem de programação paralela deve atender a algumas exigências básicas, tais como controle de processos, controle de acesso a dados compartilhados e mecanismos de sincronização. Para isso, deve oferecer mecanismos sintáticos que implementam unidades de paralelismo, permitindo a execução de diferentes unidades de programas em processadores distintos [HAE98].

Os processos paralelos são interativos e suas ações podem manter uma relação de cooperação ou competição. Existe competição quando um certo recurso é compartilhado por dois processos e ambos o disputam através de exclusão mútua. Em oposição, existe cooperação, quando os processos mantêm relação de dependência entre si e cooperam para atingir um objetivo comum [HAE98].

As informações ou recursos compartilhados devem ser protegidos contra acessos indevidos por meio de mecanismos de controle para proteção de código e de dados. Na proteção de código, parte de um trecho de código crítico deve ser executada somente por um processo de cada vez. A proteção e integridade de dados são particularmente importantes em sistemas concorrentes, nos quais deve ser evitado o acesso simultâneo aos dados para sua atualização. Um exemplo é o bloqueio da consulta em uma variável no momento em que ela estiver sendo alterada. Assim, a solução é que as operações de atualização e consulta sejam efetuadas em série.

4.3 Programação Distribuída

Um sistema de programação distribuída consiste basicamente em vários aplicativos executados em computadores diferentes, os quais realizam troca de mensagens para solucionar um problema comum. Como cada aplicativo é executado em um computador diferente, a memória e o relógio do sistema são privativos de cada computador, sendo inacessíveis aos outros. A comunicação entre os computadores é realizada através da rede que os interliga.

Redes de grandes dimensões, muito dispersas, possuem taxas de transferências relativamente baixas, não sendo muito bem controladas do ponto de vista tempo-real, por apresentarem atrasos e baixa confiabilidade. Redes que controlam sistemas críticos são normalmente de pequena escala, indo de confiáveis a muito confiáveis e apresentando taxas de transferência elevadas.

Sistemas distribuídos podem ser utilizados como servidores dedicados a tarefas específicas, funcionando como prestadores de serviços. A interação nesse sistema se dá pela requisição dos serviços prestados pelos diferentes processadores.

A melhora no desempenho das aplicações é obtida particionando-se uma tarefa em subtarefas menores que são executadas em diferentes processadores. Esses, por sua vez, executam as subtarefas e enviam o resultado de volta para o cliente.

Outras aplicações utilizam o armazenamento ou processamento de dados redundantes, ou seja, a mesma tarefa é enviada para os diversos computadores, visando à integridade dos dados processados. Aplicações desse tipo visam implementar uma técnica de tolerância a falhas, objetivando manter a consistência dos dados em sistemas concorrentes e não-confiáveis.

A comunicação entre os processos pode se dar de forma:

- Síncrona: o aplicativo envia a sua mensagem e fica suspenso, aguardando a chegada da resposta;
- Assíncrona: a mensagem é enviada e o aplicativo continua seu processamento; este tipo de operação é dito *não-bloqueante*.

A comunicação entre aplicativos distribuídos pode ser efetuada através de *sockets*. Outro tipo de processo de comunicação é a chamada a *métodos remotos*, pelo qual um objeto faz uma chamada pela rede a um método de outro objeto. Existem várias tecnologias em desenvolvimento para efetuar esse tipo de operação, dentre as quais se podem citar DCOM, Corba e RMI.

DCOM é uma tecnologia desenvolvida principalmente pela Microsoft e visa possibilitar a comunicação entre objetos distribuídos escritos em qualquer linguagem, os quais, entretanto, devem, obrigatoriamente, ser executados em um sistema operacional Windows.

Corba trata-se de um protocolo utilizado para a comunicação entre objetos remotos, independentemente do sistema operacional e da linguagem utilizada, criado por um conjunto de empresas. Essa tecnologia tem como objetivo permitir que objetos "conversem" uns com os outros independentemente da linguagem em que foram escritos. Ainda se encontra em fase de teste e não está totalmente implementado.

RMI é a alternativa para comunicação entre objetos da linguagem Java, independentemente de plataforma, já estando disponível para uso.

4.3.1 Sockets

Um *socket* é um ponto final de uma conexão virtual entre aplicativos que podem ou não estar no mesmo *host*. Por meio dessa conexão, os aplicativos podem enviar e receber dados através de *streams*. Os *sockets* são uma interface de programação de baixo nível para comunicação em rede sendo, geralmente, muito complicados e cheios de detalhes. A maioria das API de *sockets* pode ser usada com qualquer protocolo de rede, porém, como os protocolos podem ter características radicalmente diferentes, a API do *socket* é bem complexa.

Em teoria, qualquer família de protocolo pode ser usada para trabalhar com *sockets*, contudo a linguagem Java suporta somente a família de protocolos IP(Internet Protocol), utilizada atualmente para comunicação entre computadores conectados na internet. Para a realização de uma conexão através de uma rede IP, existem dois protocolos: TCP (*Transfer Control Protocol*) e UDP (*User Datagram Protocol*)

[MAH96]. Esses são protocolos-padrões e estão disponíveis em qualquer sistema que esteja conectado com a internet.

O TCP é orientado à conexão [MAH96]. Um protocolo orientado à conexão é equivalente a uma conversa através de um telefone. Após a conexão ter sido efetuada, as duas aplicações podem enviar dados uma para outra até que a conexão seja perdida ou fechada por algum dos lados. O protocolo assegura que nenhum dado seja perdido e que as informações sejam recebidas na mesma ordem em que foram enviadas.

O UDP não é orientada à conexão [MAH96], enviando pacotes de dados independentes chamados *datagramas*. Pode ser comparado ao sistema postal, que permite a troca de informações entre duas aplicações, mas não oferece nenhuma garantia de que as informações chegarão ao seu destinatário; caso esses cheguem, o protocolo não garante que serão recebidas na mesma ordem em que foram enviadas.

O tipo de protocolo utilizado depende do tipo de aplicação que está sendo desenvolvida. Em aplicações em que a integridade e organização dos dados é indispensável, é recomendável a utilização do TCP. Já o UDP é utilizado para aplicações que fornecem serviços de *streaming* em tempo real, como transmissões de vídeo e som pela rede. Esse protocolo é menos confiável, porém, em compensação, muito mais rápido, pois utiliza menos informações de controle para serem transmitidas.

Em uma rede IP, para se realizar uma conexão, é necessário informar o endereço IP do computador e a porta com a qual se deseja conectar. Isso porque, em um computador, podem estar sendo executadas várias aplicações diferentes, sendo a porta a forma de se especificar com qual delas se deseja realizar a comunicação [JAV99]. O computador que aguarda a conexão é chamado *servidor;* o computador que requisita a conexão é chamado *cliente* (FIGURA 15).

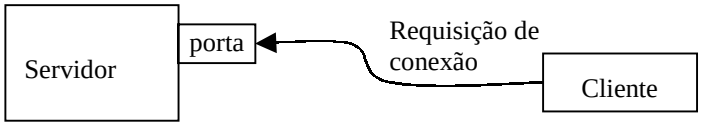

FIGURA 15 – Requisição de conexão em porta específica.

Após a requisição da conexão por parte do cliente, um novo *socket* é criado no servidor em outra porta livre, e a conexão é estabelecida entre o novo *socket* criado e o *socket* cliente (FIGURA 16).

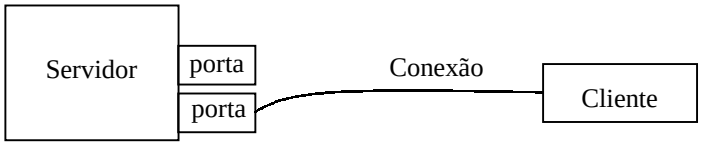

FIGURA 16 – Comunicação estabelecida em nova porta.

Estabelecida essa conexão, a troca de informações é realizada através de *streams* de entrada/saída da mesma forma do lado do cliente e do lado do servidor, ou seja, ambos podem enviar e receber informações [JAV99] (FIGURA 17).

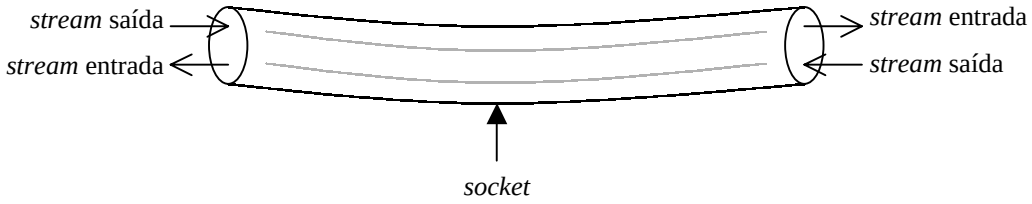

FIGURA 17 – Comunicação com uso de *streams*.

O exemplo de *socket* apresentado utiliza o protocolo TCP, pois uma conexão virtual é estabelecida entre cliente/servidor. Já, na conexão UDP, a criação dos *sockets* no cliente e servidor é praticamente igual, não havendo necessidade de uma requisição de conexão; basta, nesse caso, a criação dos *sockets*, pois as mensagens podem ser enviadas, não através de *streams*, mas de datagramas endereçados ao *socket*.

A linguagem Java suporta uma interface orientada a objetos simplificada que torna a comunicação em rede por meio de *sockets* consideravelmente fácil. A diferença de complexidade entre um programa para comunicação em rede utilizando uma linguagem estruturada como C e um programa em Java é surpreendente, visto que, em Java, todos os detalhes relacionados com a comunicação estão encapsulados nos objetos. As classes *Socket/ServerSocket* utilizam o TCP; as classes *DatagramSocket* e *MulticastSocket* utilizam UDP, sendo a *MulticastSocket* uma variação da DatagramSocket utilizada para o envio de dados para vários recipientes (*multicasting*).

4.3.2 RMI

RMI (*Remote Method Invocation*) é um conjunto de classes e interfaces desenvolvidas em Java para suportar chamadas de métodos remotos. RMI integra a tecnologia de objetos distribuídos diretamente à linguagem Java, fazendo a chamada remota de forma quase transparente para o cliente e adicionando apenas algumas novas necessidades estruturais para o servidor [HOR98]. É importante lembrar que o computador que está executando o código que faz a chamada remota é o *cliente* para aquela chamada; já o computador que está hospedando o objeto que processa a chamada é chamado de *servidor*.

Uma característica importante de RMI é que apresenta uma interface de programação para rede de mais alto nível, mais conveniente e mais natural em muitos casos do que o uso de *sockets* e *streams*. Uma exigência, entretanto, é que ambos, clientes e servidores, sejam escritos em Java, pois RMI é uma tecnologia nativa dessa linguagem [SUN99].

RMI permite que objetos sejam criados como objetos remotos e ofereçam métodos que podem ser invocados por outros objetos em outras máquinas virtuais Java, normalmente usando a rede. Pode-se chamar um método de um objeto remoto como se se estivesse chamando um método de um objeto local. RMI permite aos clientes interagirem com o objetos remotos via interfaces públicas; os clientes não interagem diretamente com as classes que implementam as interfaces.

Os programas clientes precisam manipular os objetos no servidor, mas não têm cópias deles. Os objetos realmente residem no servidor. O código cliente precisa saber o que pode fazer com aqueles objetos. As capacidades desses objetos são expressas

através de interfaces que especificam para os clientes quais são os métodos que podem ser chamados remotamente.

Quando um objeto chama um método em um objeto remoto, na verdade, está chamando um método num objeto intermediário chamado *stub*. O *stub* reside na máquina cliente e tem a função de enviar a chamada para o servidor, juntamente com os parâmetros do método, recebendo o retorno, se houver.

O *stub* utiliza um dispositivo independente de codificação para cada parâmetro. Por exemplo, números são sempre enviados no formato *big-endian*. Por sua vez, objetos locais, quando são passados como parâmetros, representam um caso particular, pois, na verdade, o que está sendo passado como parâmetro é a referência para aquele objeto. Para o servidor, não faz sentido receber a referência para um objeto no cliente, pois o objeto passado está na memória desse; assim, o *stub* usa o mecanismo de serialização para enviar uma cópia do conteúdo desse objeto através da rede. Os objetos serializáveis incluem os tipos primitivos, objetos remotos (são serializáveis por construção) e objetos não-remotos que implementem a interface *Java.io.Serializable*. Algumas classes podem, por razões de segurança, não se permitir ser passadas como parâmetro. Quando se passam objetos remotos como parâmetros, apenas o *stub* desses objetos é passado [HOR98]. Esse processo de codificação de parâmetros é chamado de *parameter marshalling*. O *stub* também produz um bloco de informação que consiste em:

- um identificador do objeto remoto que será usado;
- um número de operação que identificará o método que será chamado;
- os parâmetros codificados.

O *stub*, então, envia essas informações para o servidor, de onde elas não são enviadas diretamente para o objeto remoto que será executado, mas, sim, para o objeto *skeleton* associado a ele. O *skeleton* executa, basicamente, cinco ações para cada chamada remota [WOL99]:

- decodifica os parâmetros;
- chama o método no objeto remoto;
- recebe o valor de retorno;
- codifica o valor de retorno;
- envia o resultado para o *stub* no cliente.

O *stub* decodifica o valor de retorno ou avalia se alguma exceção foi gerada e retorna esse valor para o local de onde o método remoto foi chamado.

Pode ocorrer que se declare um método remoto com um tipo de retorno conhecido para o cliente, porém, quando o método é chamado, é retornado um objeto de uma subclasse desse tipo conhecido. O carregador de classes irá, então, carregar essa classe derivada de algum lugar, que pode ser o sistema local ou outro sistema na rede [HOR98]. Isso é muito similar ao carregamento de classes de um *Applet*, ou seja, toda classe necessária para sua execução só poderá ser carregada do computador de onde a página que contém o *Applet* está hospedada, visto que o seu gerenciador de segurança não permite que classes sejam carregadas de outros locais. O gerenciador de segurança do *stub* é um pouco menos restrito: se a classe não for encontrada no sistema local, ela pode ser carregada de vários outros locais. Essa obtenção dinâmica de classes é muito útil em programas distribuídos, que se unem para realizar uma computação problemática e precisam buscar uma classe no mesmo local central; isso também estende o comportamento de um programa distribuído dinamicamente.

No lado cliente, precisa-se do programa cliente, do *stub* gerado e da interface remota; no lado servidor, deve-se colocar o programa servidor, o *skeleton* gerado e a interface remota. Porém, antes de executar a aplicação, deve-se executar o *rmiregistry*, que é um registro de objetos remotos também fornecido pelo Java. Ao se criar um objeto remoto no servidor, precisa-se registrá-lo. Quando o cliente cria a referência local para o objeto remoto, essa referência terá de ser buscada no servidor de registro; desse modo, essa referência pode ser usada como se fosse um objeto local. Todo trabalho de comunicação, envio e recebimento de parâmetros é realizado automaticamente.

A FIGURA 18 representa o fluxo de informações desse processo.

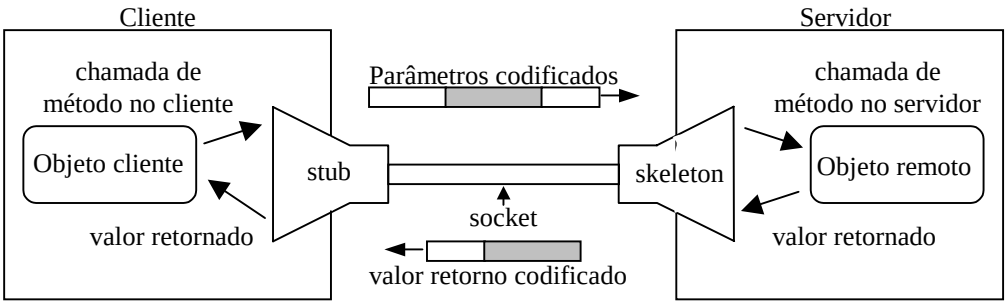

FIGURA 18 - Fluxo de informações em RMI.

Todo esse processo é muito complexo, porém tudo é realizado automaticamente e de forma transparente ao programador. O *stub* e o *skeleton* são gerados a partir da classe do objeto remoto através do RMIC, compilador que é fornecido com o Java. Se o objeto remoto se chamar *Remote.class*, o stub gerado será *Remote_stub.class* e o *skeleton*, *Remote_skeleton.class*.

Um exemplo de programa usando RMI é apresentado abaixo (FIGURA 19, FIGURA 20 e FIGURA 21):

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface HelloInterface extends Remote {
	String say(String message) throws RemoteException;
}
```

FIGURA 19 - Interface do objeto remoto, arquivo HelloInterface.java.

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
public class HelloServer extends UnicastRemoteObject implements HelloInterface {
	public HelloServer() throws RemoteException { super(); }
	public String say(String message) {
		System.out.println(message);
		return  "Obrigado por sua mensagem "+message;
	}
	public static void main(String args[]) {
		try {	HelloServer obj = new HelloServer();
			Naming.rebind("HelloServer", obj);
			System.out.println("HelloServer registrado");
		} catch (Exception e) { ... }
	}
}
```

FIGURA 20 - Código servidor, arquivo HelloServer.java.

```
import java.rmi.*;

public class HelloClient extends Object {
        public static void main(String args[]) {
                HelloInterface obj = null;
                String message;
                try {
                        obj = (HelloInterface)Naming.lookup("rmi://localhost/HelloServer");
                        message = obj.say("Hello World !!");
                        System.out.println("Retornado do servidor: "+message);
                } catch (Exception e) { ... }
        }
}
```

FIGURA 21 - Código cliente, arquivo HelloCliente.java.

O código citado (FIGURA 20) cria um objeto remoto que possui somente um método o qual pode ser chamado remotamente, o método *say*(). Esse possui um único parâmetro do tipo *string* e retorna uma *string*. O método exibe a *string* no servidor e retorna uma mensagem para o cliente.

Objetos distribuídos podem falhar pelos mais variados motivos e mais freqüentemente do que em objetos locais. Deve-se, então, ser capaz de tratar exceções que possam ocorrer durante a chamada de um método remoto. Para isso, RMI estende as características de tratamento de exceções em Java, criando novas classes de exceção, facilitando, assim, o tratamento e a captura de falhas com objetos remotos. Como pode ser visto no código citado, todos os métodos chamados remotamente devem gerar a exceção remota.

4.4 Programação Tempo-Real

Sistemas de Tempo-Real (STR) são sistemas de computação associados a eventos do mundo exterior, ou seja, são aplicações construídas pelo usuário, tendo como finalidade monitorar ou controlar algum evento do mundo exterior. Como um exemplo de um sistema de tempo-real pode-se citar um computador que controle a temperatura de uma caldeira em uma usina termoelétrica.

Um Sistema Operacional de Tempo Real (SOTR) pode, além das funções básicas de qualquer sistema operacional, fornecer uma interface amigável para o usuário e gerenciar de forma eficiente os recursos do computador; pode, ainda, fornecer ferramentas adequadas para a construção de uma aplicação de tempo-real. Quanto mais completo for esse conjunto de ferramentas, maior será a facilidade do usuário em programar a sua aplicação de tempo real. Em razão de o tempo ser um fator crítico nesses tipos de aplicações, os SOTRs devem fornecer mecanismos para que tais aplicações possam estabelecer seus prazos, suportar tolerância a falhas, bem como ter algoritmos de escalonamento que possam, sempre que possível, estabelecer um escalonamento viável para as tarefas, isto é, uma seqüência de execução que possibilite às tarefas o cumprimento de seus prazos.

Os STR, segundo [BEC98], podem ser classificados em:

- *não-críticos*: as tarefas devem ser executadas tão logo quanto possível e não há uma conseqüência grave para o sistema caso as tarefas não sejam executas em um tempo predeterminado;

- *críticos*: restrições temporais são consideradas, de tal forma que conseqüências graves podem decorrer se essas restrições não forem satisfeitas. Tais sistemas podem ser definidos como aqueles em que a correção do sistema não depende apenas dos resultados lógicos da computação, mas também do momento em que esses resultados são produzidos.

4.4.1 Previsibilidade

Previsibilidade é a capacidade de se estabelecer um procedimento para determinar se uma tarefa, quando ativada, cumprirá todos os seus prazos desde que certas hipóteses sejam obedecidas [BEC98]. Embora essa característica deva estar associada a todo sistema de tempo real, ela assume um papel preponderante em sistemas de tempo real críticos.

Como em um sistema crítico o desempenho pode comprometer instalações ou vidas humanas, é importante que tais sistemas sejam previsíveis [BEC98]. Isso quer dizer que é possível, quando da realização do projeto, mostrar que todas as restrições temporais serão atendidas sob certas hipóteses.

Um sistema de tempo real tem como principal meta ser previsível, ou seja, deve ser possível, durante a sua fase de projeto, mostrar que, para um dado conjunto de tarefas, essas podem ou não ter seus requisitos temporais satisfeitos, dependendo de parâmetros a serem analisados durante a execução do sistema. Muitos desses parâmetros podem ser eventos externos que fogem do controle do programador. Ressalta-se que não se pode prever nada que esteja fora do controle do projeto, como, por exemplo, falhas de *hardware*, contudo tudo o que se puder garantir deverá ser pesquisado e previsto [BEC98].

A velocidade de um sistema de tempo real só deve ser avaliada se esse não tiver capacidade para atender à velocidade dos eventos externos que precisa gerenciar, de forma que, quanto mais recursos computacionais forem colocados à disposição, mais complicadas serão as aplicações que serão projetadas.

4.4.2 Escalonamento

Escalonamento consiste em definir onde e quando uma tarefa deve ser executada, garantindo-se, assim, que ela seja feita dentro do intervalo de tempo previsto. Essa é, basicamente, a parte vital do sistema, pois o escalonamento é que garante a execução da tarefa no prazo determinado. Cada tarefa pode possuir um conjunto de restrições, isto é, prazos, uso de recursos, necessidade de sincronização com outras tarefas, etc. A parte de um sistema computacional responsável pelo escalonamento de tarefas é chamada de *escalonador*, que emprega para tal um algoritmo de escalonamento.

Escrever um algoritmo de escalonamento pode ser uma tarefa difícil, ou até mesmo impossível, pois o escalonamento de tarefas é um campo da computação no qual há muitos problemas em aberto [BEC98].

4.4.3 Tarefas

Tarefa é uma entidade selecionada para execução por um escalonador. Uma tarefa é completamente descrita por um contexto de *hardware* e *software* e um espaço de endereçamento virtual. Essas informações estão contidas em várias estruturas de dados localizadas em diferentes lugares do espaço de endereçamento da tarefa. Por contexto de *hardware*, podem-se entender todos os registradores do computador que são utilizados na execução de uma tarefa. Por contexto de *software*, podem-se entender todos os dados necessários para que o sistema operacional gerencie uma tarefa, tais como sua prioridade, estado, privilégios, restrições temporais, etc. [BEC98].

As tarefas de um sistema de tempo real podem ter várias características, como periodicidade, restrições, prioridades e preempção, relevantes para o estudo sobre sistemas de temo real.

Segundo a periodicidade, as tarefas podem ser:

- *periódicas*: ativadas em intervalos de tempo regulares;
- *aperiódicas*: ativadas somente quando um determinado evento ocorre;
- *esporádicas*: são tarefas aperiódicas com prazo determinado para o término da execução.

As tarefas periódicas são utilizadas no tratamento de eventos rotineiros; já as tarefas aperiódicas servem para tratar eventos fora do que se pode chamar de operação normal do sistema, tais como a detecção de um míssil, a elevação súbita da temperatura de uma caldeira ou a resposta a algum comando do usuário do sistema.

Em um sistema de tempo real, uma tarefa pode ser caracterizada pelos quatro tipos de restrições: temporais, de precedência, de exclusão e exigência de recursos.

A restrição temporal implica que uma tarefa deverá acabar no prazo determinado. Os prazos podem ser críticos ou não-críticos. Um prazo é dito *crítico* quando o seu não-cumprimento resulta em conseqüências desastrosas; assim, diz-se que uma tarefa é crítica quando possui, pelo menos, um prazo crítico. Uma tarefa possui uma restrição temporal estrita quando o seu prazo é muito pequeno, ou quando possui um prazo grande, mas tem uma folga muito pequena, devendo ser executada imediatamente. Uma tarefa com restrição temporal estrita não necessariamente é crítica, de forma que prazos grandes podem ser críticos e prazos pequenos podem não ser necessariamente críticos.

As restrições de *precedência* surgem quando há tarefas que, para prosseguir no seu processamento, necessitam da execução de outras tarefas. Por exemplo, a tarefa A só pode continuar seu processamento quando a tarefa B tiver executado até um determinado ponto.

As restrições de *exclusão* ocorrem quando tarefas utilizam recursos não compartilháveis do sistema. Para impedir que ocorram inconsistências, pode-se recorrer, por exemplo, ao uso de semáforos para sincronização das tarefas.

As restrições de *recursos* ocorrem quando há tarefas que necessitam alocar recursos para a sua computação.

No escalonamento de tarefas, a prioridade pode ser fixa ou variável. Um sistema trabalha com prioridades fixas quando a prioridade de uma tarefa não muda durante a sua execução; em caso contrário, o sistema trabalha com prioridades variáveis, ou seja, a prioridade pode variar durante o tempo de execução.

Um sistema que utiliza prioridades variáveis é mais flexível, porém ocorre perda de desempenho em virtude do gerenciamento das modificações de prioridades. Sistemas desse tipo permitem o uso de técnicas como a do envelhecimento, na qual uma tarefa vai aumentando a sua prioridade com o passar do tempo para evitar que o processamento seja adiado indefinidamente. O adiamento indefinido ocorre quando uma tarefa não consegue tomar posse do processador para iniciar a sua execução.

Uma tarefa é dita *preemptiva* quando sua execução pode ser interrompida por outras tarefas a qualquer momento e reassumida mais tarde. Uma tarefa não-preemptiva deve rodar até sua finalização depois que for iniciada. A natureza do ambiente que o sistema deve atender é que define se uma tarefa deve ou não ser preemptiva.

5 Salvamento e Recuperação de Estados de Objetos

Neste capítulo, apresentam-se algumas das opções existentes para realizar o salvamento e a recuperação do estado de objetos.

5.1 Serialização

Tipicamente, a serialização de objetos é usada para armazenar cópias dos objetos em um arquivo. A serialização permite direcionar objetos para dentro de *streams* de *byte*s, os quais, quando lidos, recriam os objetos que foram escritos para as *streams*.

A serialização define interfaces para escrita e leitura de objetos e classes que implementam essas interfaces. Quando se escreve um objeto para um objeto *stream* de saída, um fluxo de *bytes* é gerado. O fluxo de *bytes* pode ser lido por um objeto *stream* de entrada correspondente para criar um novo objeto, que é equivalente ao objeto original.

No JDK versão 1.1.5, existe um novo atributo de classe, chamado *serializable*, que pode ser usado para marcar as classes que podem ser serializadas. Não é necessário escrever um código especial para que uma classe seja serializada; basta sinalizar através do uso do atributo *serializable*. Um mecanismo-padrão é fornecido para a classe sinalizada, o qual implementa os métodos *writeObjetc* e *readObjetc*, que escreve e lê cada campo do objeto. A FIGURA 22 mostra o processo de escrita de um objeto.

```
FileOutputStream f = new FileOutputStream("Arquivo");
ObjectOutputStream s = new ObjectOutputStream(f);
s.writeObject(new Date());
s.close();
```

FIGURA 22 - Salvando objeto usando serialização.

Primeiro, uma *stream* de saída (*FileOutputStream*) é necessária para receber os *bytes*. Assim, um *ObjetOutputStream* é criado para escrever na *stream* de saída. Após, o objeto é escrito para a *stream*, o que é feito com o uso do método *writeObject*.

Para ler um objeto de uma *stream* (FIGURA 23), é necessário, primeiro, uma *stream* de entrada (*FileInputStream*) como fonte. Assim, um *ObjetInputStream* é criado para ler da *stream* de entrada; após, o objeto é lido da *stream*. Os objetos são lidos com o método *readObject*.

```
FileInputStream in = new FileInputStream("Arquivo");
ObjectInputStream s = new ObjectInputStream(in);
Date d = (Date) s.readObject();
s.close();
```

FIGURA 23 - Restaurando objeto usando serialização.

A serialização fornece um mecanismo simples extensível para armazenar objetos persistentes. O tipo do objeto Java e a segurança das propriedades são mantidos na

forma serializada; a serialização apenas requer implementação de classe para customizações especiais.

A serialização deve ser suficiente para aplicações que operam com um pequeno número de dados persistentes e nas quais o armazenamento de confiança não seja um requerimento absoluto. Porém, a serialização não é uma ótima escolha para aplicações que tenham de gerenciar mega-*bytes* de objetos persistentes por tornar lento o armazenamento dos objetos e não oferecer segurança no procedimento.

5.2 Persistent Storage Engine (PSE)

As limitações da serialização são melhoradas por um PSE (*Persistent Storage Engine*). Um PSE fornece a simplicidade e a extensibilidade da serialização, ao mesmo tempo em que supera os problemas de confiabilidade e performance quando gerencia vários mega-*bytes* de objetos [OBR97].

À semelhança da serialização, o PSE garante facilidade de uso para armazenamento e recuperação de objetos. Classes persistentes Java, seus campos e métodos são definidos da mesma maneira que classes Java transientes. O programador declara classes para serem persistentes e, após, usa construtores-padrões para criar e manipular tanto instâncias persistentes como transientes. A transparência dos objetos persistentes através de PSE habilita o desenvolvedor a fazer uso do poder de Java, havendo a facilidade de incorporar bibliotecas de classes existentes com PSE.

A funcionalidade de banco de dados é fornecida através de PSE por uma API Java que oferece, sobretudo, funções para:

- criar, abrir e fechar bancos de dados;
- iniciar e terminar transações;
- armazenar e recuperar objetos persistentes.

O PSE gera automaticamente os equivalentes *readObject* e *writeObject* para cada classe capaz de ser persistente. Do mesmo modo que na serialização, o desenvolvedor pode sobrescrever a implementação desses métodos.

A seguir mostra-se um exemplo de programa (FIGURA 24) para ilustrar como se cria um banco de dados PSE com um objeto cliente:

```
class Exemplo {
        public static void main() {

        ObjectStore.initialize(null, null);
        Database db = Database.create("clientes.odb", Database.allRead | Database.allWrite);
        Transaction t = Transaction.begin(Transaction.update);
        Cliente c = new Cliente("Willingthon Pavan", 50, 90000, null);

        db.createRoot("chefe", c);
        t.commit();
        db.close();
        }
}
```

FIGURA 24 - Salvando objeto com o uso do PSE.

No exemplo apresentado (FIGURA 24), o programa, inicialmente, inicializa o *software* PSE e, depois, cria e abre um banco de dados nomeado *clientes.odb*. Após,

inicia-se uma transação de atualização, sendo criado um novo objeto cliente. Então, um banco de dados raiz é criado com o nome *chefe*, sendo associado com o objeto cliente que foi previamente criado. Finalmente, a transação é atualizada, e o objeto cliente é armazenado no banco de dados.

À semelhança da serialização, PSE é de fácil utilização e possibilita uma razoável transparência de programação. Nele, objetos de qualquer classe podem facilmente ser armazenados e recuperados. Diferentemente da serialização, PSE fornece explicitamente um fino controle sobre o acesso e busca de objetos, resultando em alta performance e armazenamento gerenciado para um grande número de objetos. PSE introduz uma semântica de transações que é um conceito adicional para que os programadores aprendam. Os benefícios adicionados de transações e recuperação, em particular, integridade de dados e prevenção de falhas em aplicações e sistemas de banco de dados corrompidos, são certamente o preço do esforço para muitas aplicações.

Quando um banco de dados excede cem MBs, a performance do PSE começa a degradar. Quando isso ocorre e é necessário um grande número de usuários concorrentes, a escolha de um sistema gerenciador de banco de dados deve ser considerada.

5.3 Sistemas Gerenciadores de Banco de Dados

Sistemas Gerenciadores de Banco de Dados (DBMS) são projetados para suportar um grande número de usuários concorrentes, alto volume de atualizações e pesquisas sobre uma grande coleção de objetos. Um DBMS assegura a integridade e a confiabilidade do banco de dados e fornece total suporte para a administração do banco de dados, com altíssima disponibilidade.

Consideram-se duas classes de DBMS: os bancos de dados relacionais suportados pela API JDBC do Java e os bancos de dados orientados a objetos, suportados através da ligação do Java com ODMG.

5.3.1 Banco de Dados Relacionais

A JavaSoft [JAV99] tem introduzido uma interface-padrão de acesso a banco de dados SQL, API JDBC. JDBC permite que programas Java façam solicitações SQL e processem os resultados. Essa API oferece uma interface uniforme para um grande número de banco de dados relacionais, além de uma base comum na qual ferramentas de mais alto nível e interfaces podem ser construídas.

JDBC é, deliberadamente, uma API de “baixo nível” projetada para construir ferramentas de aplicação e servir como base para APIs de mais alto nível. Ela é baseada no padrão Xopen SQL CLI e no ODBC padrão da Microsoft®. ODBC foi uma escolha pragmática, pois é amplamente aceita e implementa um padrão para acesso a banco de dados SQL. Virtualmente, todos os bancos de dados suportam ODBC, que tem sido estendida além das plataformas Microsoft, para ser suportada pela maioria das plataformas Unix.

A API JDBC define classes Java para representar conecções, declarações SQL, conjunto de resultados e banco de dados metadata. Nos termos das classes Java, a API JDBC consiste em:

- java.sql.Connection;

- java.sql.Statement;
- java.sql.PreparedStatement;
- java.sql.Callablessatement;
- java.sql.ResultSet.

Uma conexão representa uma seção com um banco de dados específico. Dentro do contexto de uma conexão, declarações SQL são executadas e os resultados são retornados.

Um objeto é usado para executar uma declaração SQL estática e obter os resultados produzidos. O método *executeQuery* é usado para declarações *Select* que retornam um conjunto simples de resultados (*ResultSet*). O *executeUpdate* é usado para declarações *Insert, Update, Delete* e outros tipos simples que não retornam resultados. O *execute* é usado para tratar as mais variadas declarações, por exemplo, solicitações que retornem múltiplos resultados.

Uma simples consulta que restaura todos os clientes em um banco de dados é mostrada na FIGURA 25.

```
// Abre uma conecção a um bando de dados
        Connection com = DriverManager.getConnection("jdbc:odbc:wombat");
// Cria e executa uma declaração
        Statement stmt = com.createssatement();
        ResultSet rs = stmt.executeQuery("SELECT cli_nome, cli_idade, cli_salario FROM clientes");
// Passando pelas linhas resultantes
        While (rs.next()) {
                // pega os valores de cada linha
                String nome = rs.getString("cli_nome");
                int idade = rs.getInt(2);
                int salario = rs.getInt(3);
                System.out.println("Nome: " + nome +
                                " , idade: " + idade +
                                " , salário: " + salario);
        }
```

FIGURA 25 - Restaurando dados em um banco de dados relacional.

Java é uma linguagem orientada a objetos, portanto os desenvolvedores trabalham com classes, instâncias de objetos e invocam métodos. Já os desenvolvedores que utilizam bancos de dados relacionais trabalham com tabelas, inserem registros e chamam procedimentos armazenados. A diferença entre os dois modelos, objetos e SQL, pode resultar em dificuldade na concretização dos objetivos, tais como performance e aumento do custo de desenvolvimento. Portanto, o uso de um banco de dados relacional em uma aplicação orientada a objeto, por exemplo na linguagem Java, pode ser problemático por causa do mapeamento necessário entre os objetos e o modelo de registros.

Por outro lado, os bancos de dados são produtos já consolidados, os quais, por suas características, suportam um grande número de usuários trabalhando de forma concorrente em ambientes de alta disponibilidade. Existem também várias ferramentas de desenvolvimento baseadas em SQL, utilitários para administração e produtos que permitem ao usuário final o acesso ao banco de dados.

Para muitas aplicações Java, o acesso a um banco de dados relacional é necessário, e o JDBC fornece uma maneira uniforme de acessá-los.

5.3.2 Banco de Dados Orientado a Objetos

A maioria dos bancos de dados orientados a objetos atuais oferece um nível de funcionalidade equivalente ao dos bancos de dados relacionais quanto ao número de usuários concorrentes, alta disponibilidade, confiabilidade, segurança, linguagem de pesquisa, ferramentas de desenvolvimento e utilitários para administração [OBR97].

Em termos de arquitetura e modelo de dados, os bancos de dados orientados a objetos e os relacionais apresentam muitas diferenças entre si. Uma das principais é a performance procurada pelos desenvolvedores. Como exemplo de banco de dados, utiliza-se o *Object Design's ObjectStore DBMS*.

O *ObjectStore DBMS* oferece aos programas uma alta performance no armazenamento de objetos, além de alta disponibilidade e confiabilidade; permite o acesso concorrente a múltiplos bancos de dados em um ambiente distribuído multi-camada [OBR98].

Como no PSE, o esquema é gerado de classes Java que são marcadas como persistentes, existindo um pequeno conjunto de classes que fornecem a interface para as extensões *ObjectStore*, tais como banco de dados e transações.

As extensões fornecidas pelo *ObjectStore* incluem uma robusta coleção de bibliotecas que suportam o armazenamento e a recuperação indexada de um grande grupo de objetos e uma biblioteca de gerenciadores de objetos, os quais fornecem suporte ao gerenciamento de multimídia (imagens, áudio, vídeo, textos e HTML).

A FIGURA 26 mostra um programa *ObjectStore* que consulta um banco de dados para encontrar um conjunto de empregados (*employees*) que ganham menos que R$ 45.000,00, imprimindo os seus respectivos nomes (*name*), salários (*salary*) e gerentes (*manager*).

```
    Dtabase db = Database.open("employee.db",Database.ReadWrite);

Transaction t = Transaction.begin(Transaction.readOnly);

SetOfEmployee employees = (SetOfEmployee) db.getRoot("AllEmployees");
SetOfEmployee PoorEmployees = employees.query("salary < 15000");
EmployeeEnumeration ei = PoorEmployees.elements();

While (ei.hasMoreElements()) {
    Employee e = ei.nextelement();
    System.out.println(e.name + " tem um salario de " + e.salary);
    System.out.println(e.manager.name + " é gerente de " + e.name);
}
t.commit();
db.close();
```

FIGURA 26 - Restaurando objetos de banco de dados orientado a objetos.

O *ObjectStore* fornece um ambiente de gerenciamento de dados para aplicações Java; oferece uma ligação entre ele e a linguagem Java, que resulta em uma facilidade

de uso, reduzindo drasticamente a quantidade de código requerido para gerenciar objetos persistentes.

Desenvolvedores podem facilmente estender os tipos fornecidos ou definir novos tipos. A extensão de tipos de dados fornece uma grande flexibilidade e liberdade para alavancar a extensibilidade que um modelo de objeto oferece.

Os bancos de dados orientados a objetos oferecem uma enorme economia no custo da aplicação, pois reduzem o tempo gasto com desenvolvimento, embora ainda não tenham alcançado o nível de difusão de uso que os bancos de dados relacionais desfrutam.

5.4 Conclusões

O tempo de vida de um objeto persistente excede a vida do processamento da aplicação que o cria. O gerenciamento de objetos persistentes é um mecanismo que serve para armazenar o estado de objetos em um local não volátil a fim de que, no caso de uma aplicação parar, os objetos continuem a existir, podendo ser recuperados quando necessário [OBR97].

Pode-se escolher entre várias opções, quando se determinam as necessidades para gerenciar objetos persistentes satisfatoriamente. A técnica mais simples consiste em usar diretamente o sistema de arquivos. Felizmente, Java fornece serialização de objetos que ajuda a tratar o direcionamento de objetos para *streams*, as quais podem ser escritas para arquivos e, subseqüentemente, lidas para reconstruir os objetos do programa.

Apesar de a serialização ser de fácil utilização, podem-se encontrar problemas na performance com um grande número de objetos. Existe a classe de *softwares* PSE (*Persistent Storage Engine*) que aperfeiçoa algumas das limitações da serialização, oferecendo uma melhor performance, com um grande número de objetos, um armazenamento seguro, além da gravação de objetos em disco de forma transparente e do mesmo modo que na serialização.

Quando as aplicações tiverem suporte a um grande número de usuários concorrentes, provavelmente irá se considerar um Sistema de Gerenciamento de Banco de Dados (DBMS) para manipular os objetos. Em Java, podem-se encontrar duas categorias diferentes de bancos de dados: Banco de Dados Relacional (JDBC) ou Orientado a Objetos (ODMG).

Como já foi visto, existem várias opções para se efetuar o salvamento e a recuperação de objetos. Em cada situação, deve-se verificar de que aplicação realmente se necessita para que se possa fazer a escolha/opção. Para exemplificar, mostra-se na TABELA 1 cada uma das opções mencionadas com uma descrição de sua utilização.

TABELA 1 - Opções para armazenar o estado de objetos

Opções	**Descrição**
Serialização	Desenvolvimento de uma lista pessoal de amigos, à qual se possam adicionar alguns campos, como telefone, nome e e-mail. Deseja-se que suporte uma indexação simples para fazer pesquisas, procurando tanto por nome como por e-mail. Essa lista não será grande (mais de 200 entradas) e será acessada por apenas um usuário.
PSE	Construção de uma lista telefônica de uma grande companhia que armazena informações sobre clientes, seus nomes, telefones, endereços, renda mensal e contatos. A busca de informações pode ser considerada simples: localizar por nome e número do telefone. Diferentemente da lista pessoal, essa terá milhares de entradas além de muitos usuários acessando-a concorrentemente. Essa lista é apenas para leitura.
JDBC	A extensão da lista telefônica descrita acima permite que os clientes atualizem suas próprias informações e possibilita consultas mais sofisticadas; por exemplo, todos os clientes que ganham mais de R$ 1.000,00. Não se pode usar PSE, pois não permite atualizações concorrentes e não fornece todos os tipos de consulta, sendo necessário programar as consultas à mão, o que encarece o custo do desenvolvimento.
ODMG	Construção de sistema baseado na Web para uma grande companhia, com muitas facilidades de consultas e informações de produtos. Uma das funcionalidades consideradas é a manutenção da foto dos empregados com seus registros. Deverão ser geradas automaticamente plantas que permitam localizar o escritório de um empregado ou, até mesmo, a sala de conferências. Informações sobre linhas telefônicas, configuração de rede e endereços IP também deverão ser fornecidas.

6 Proposta, Problemas e Soluções

Apresenta-se neste capítulo a descrição de uma proposta de sistema utilizando técnicas de tolerância a falhas com reflexão computacional.

O modelo é estruturado em ambiente distribuído e de tempo real, sendo composto por um programa controlador, programas servidores e serviços.

O programa controlador faz solicitações aos servidores (chamadas remotas a métodos) e recebe informações após o término do processamento desses. Os programas servidores são responsáveis pela obtenção e/ou atualização de informações externas a eles (serviços), as quais serão repassadas ao controlador. Podem-se visualizar o controlador e os servidores como objetos que executam a quase-totalidade das funcionalidades da aplicação. A FIGURA 27 esquematiza a arquitetura básica do sistema adotado como modelo.

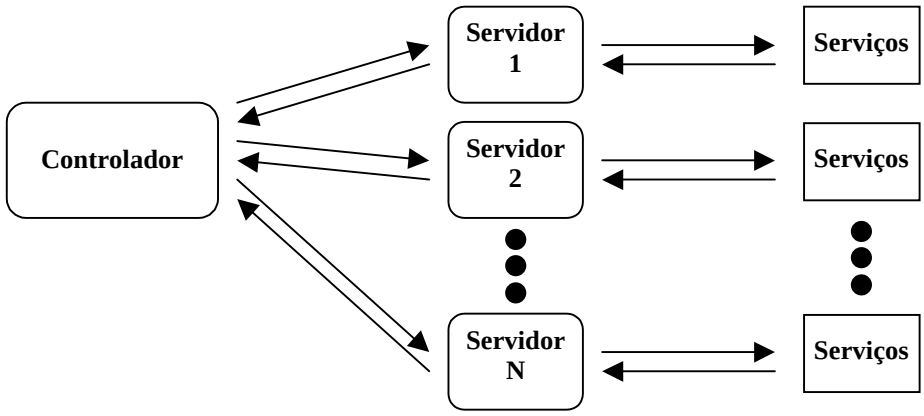

FIGURA 27 - Componentes de um STR no modelo de objetos.

Cada objeto é um elemento ou componente integrante do programa coordenador, mas externo a ele. O programa controlador não consegue operar ou funcionar de forma plena com a ausência de um dos objetos servidores. Por esse motivo, há a necessidade de inclusão de técnicas de tolerância a falhas nos servidores a fim de que a execução do controlador não fique prejudicada por causa de alguma falha dos servidores.

Neste modelo, podem existir vários objetos servidores acoplados a tantos serviços quantos forem necessários. O controle desses servidores fica sob a responsabilidade do controlador e, conseqüentemente, do *software* da aplicação, tornando essa tarefa muito trabalhosa.

As técnicas de tolerância a falhas podem tornar o objeto muito grande e complexo, além de sua programação muito difícil. Visando simplificar a programação e isolar as estruturas e técnicas de TF das funcionalidades da aplicação, tem sido adotada a técnica de reflexão computacional [LIS98]. Os objetos da aplicação são controlados por metaobjetos que atuam em um metanível entre o objeto e o usuário externo desse objeto [HAE98].

O modelo proposto tem por base um conjunto de objetos servidores que realizam operações e tarefas básicas da aplicação. Os resultados desses objetos são enviados ao objeto controlador, que realizará suas operações gerando novas tarefas, as quais serão encaminhadas aos objetos servidores. Esses executam suas tarefas, retornando novamente os resultados ao controlador, realizando, assim, o ciclo de execução do sistema. O presente modelo é considerado crítico, não podendo, portanto, sofrer interrupções em seu funcionamento.

Como o modelo aqui proposto é projetado para um ambiente distribuído, os objetos ficam instalados em nodos diferentes (remotos), onde executam suas funcionalidades e prestam serviços através da troca de mensagens pela rede de comunicação.

No modelo, o papel do controlador é o de solicitar a um conjunto de servidores informações sobre os sensores. De posse dessas informações, ele será capaz de analisar, atualizar a base de dados e tomar decisões, sinalizando o que deve ser feito ao servidor responsável pelo atuador. O servidor responsável pelo atuador executará as “ordens” do controlador, alterando o estado do atuador (FIGURA 28).

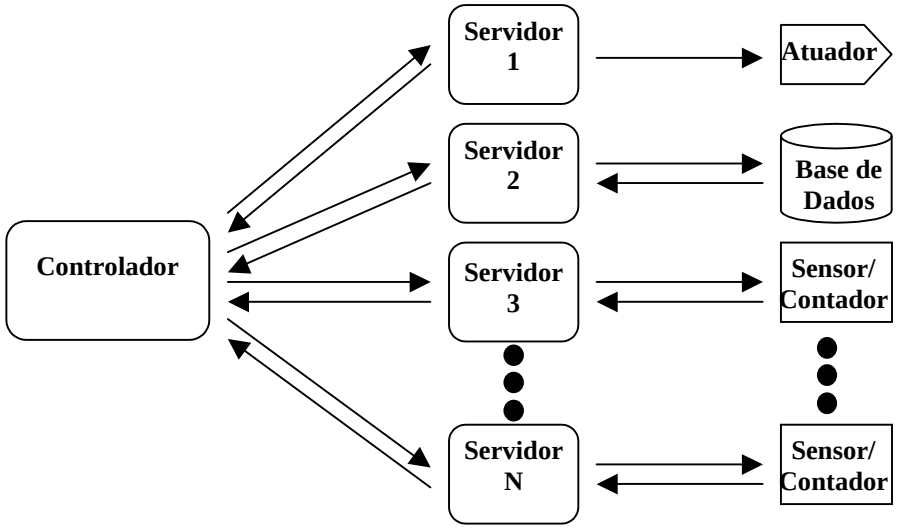

FIGURA 28 - Comunicação entre controlador, servidores e serviços.

Falhas podem ocorrer no modelo, comprometendo a confiabilidade e disponibilidade. Por isso, podem-se aplicar técnicas de TF ao modelo a fim de que os problemas sejam solucionados sem necessidade de parada total do sistema.

Visando não dificultar o desenvolvimento, em virtude da inclusão de técnicas de tolerância a falhas, adota-se o modelo reflexivo, dividindo o sistema em dois níveis: nível-meta e nível-base. O nível-base é responsável pela aplicação em si e o nível-meta, pela supervisão e aplicação da técnica de TF (detecção de falhas, confinamento de danos, registro de erros, etc.).

Podem-se utilizar as técnicas de replicação ativa e replicação passiva para visualizar qual dos servidores se encontra com problemas na sua computação, e também para encontrar a resposta correta que possibilite dar continuidade ao sistema. A técnica de replicação ativa trabalha basicamente através de votação, ao passo que a de replicação passiva utiliza-se do salvamento do estado da computação, de um teste de

aceitação e recuperação do estado em caso de falha para a execução do bloco alternativo.

A aplicação da técnica de replicação ativa é feita de forma transparente ao sistema, de forma que o controlador apenas enxergue um servidor, o qual, por sua vez, não deve ter conhecimento dos outros. Para ilustrar essa situação, imagina-se um conjunto de servidores que estão conectados a sensores, os quais são responsáveis por obter informações junto aos sensores e repassá-las ao controlador, quando isso for solicitado. Cada servidor não tem conhecimento dos demais. Quando uma mensagem é enviada pelo controlador (C) a um objeto servidor (O1) para que retorne as informações solicitadas, a mensagem é desviada para um metaobjeto (M1) ligado ao objeto servidor. O metaobjeto, por sua vez, solicita a execução da mensagem ao objeto local (O1) e envia a mensagem desviada aos outros metaobjetos remotos (M2,...,MK), que estão ligados a seus respectivos objetos (O2,...,OK), para que solicitem a execução da mensagem e retornem o resultado. O metaobjeto solicitante aguarda até que o objeto local e todos os metaobjetos remotos retornem seus resultados para realizar a comparação das respostas através do votador (V) que integra o metaobjeto solicitante. A resposta que obtiver a maioria será, então, enviada como retorno, realizando-se, dessa forma, o ciclo da computação (FIGURA 29).

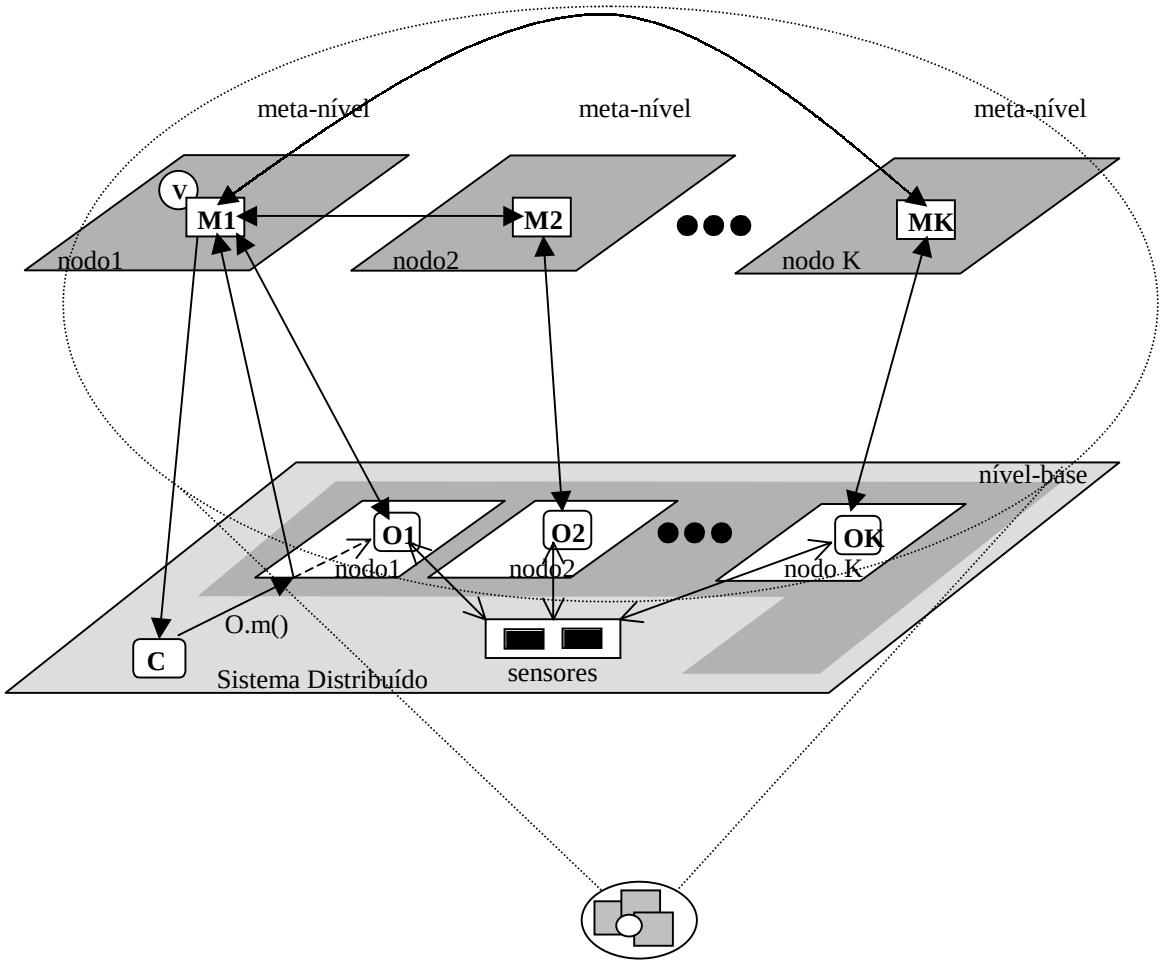

FIGURA 29 - Modelo reflexivo com técnica de replicação ativa.

Durante a execução, cada metaobjeto que está relacionado com um objeto servidor executa esporadicamente um método responsável pelo salvamento do estado da computação para que, se alguma falha no objeto for observada, um novo objeto servidor seja instanciado e o estado da computação salvo seja restaurado com o intuito de dar continuidade ao serviço.

A aplicação da técnica de replicação passiva também é feita de forma transparente ao sistema, de forma que o controlador apenas enxergue um servidor. Para ilustrar essa situação, imagine-se um conjunto de nodos servidores que estão conectados a sensores. Esses nodos servidores são, respectivamente, o nodo servidor primário, o nodo servidor alternativo 1, o nodo servidor alternativo 2, ..., nodo servidor alternativo K. A computação é realizada apenas no objeto servidor primário que está no nodo servidor primário; caso esse venha a falhar, o próximo objeto servidor será executado, e assim por diante, até que se encontre uma resposta correta ou que o último objeto servidor seja executado.

Assim como na técnica de replicação ativa, os objetos servidores são responsáveis por obter informações junto aos sensores e repassá-las ao controlador, quando solicitado. Quando uma mensagem é enviada pelo objeto controlador (**C**) ao objeto servidor primário (O1) para que retorne as informações solicitadas, a mensagem é desviada para um metaobjeto (M1) ligado ao objeto servidor. O metaobjeto, por sua vez, executa o salvamento do estado da computação do objeto local (O1) e solicita a execução da mensagem ao objeto, aguardando o término da operação.

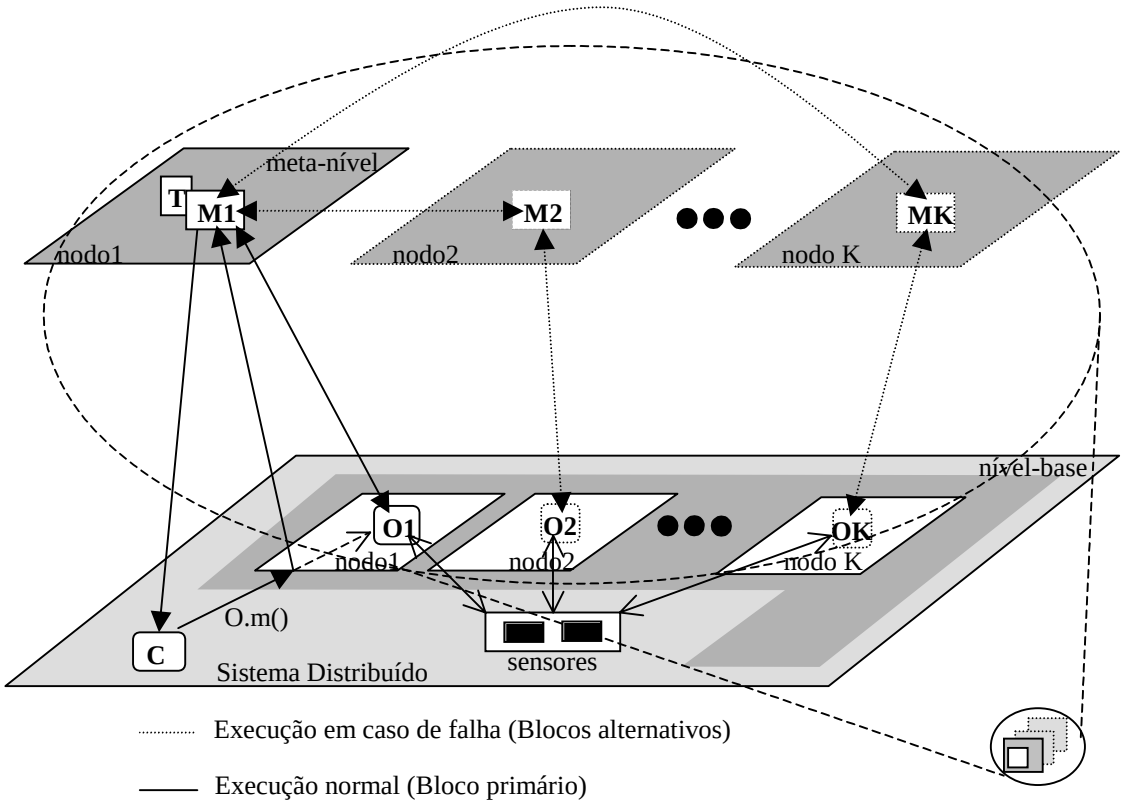

FIGURA 30 - Modelo reflexivo com técnica de replicação passiva.

Terminada a execução, o metaobjeto realiza o teste de aceitação (T) que integra o metaobjeto. Caso passe pelo teste, o resultado da execução é enviado como resposta ao objeto controlador, e o estado da computação salvo é descartado; caso não passe pelo teste de aceitação ou não haja retorno, o metaobjeto envia a mensagem interceptada ao próximo bloco alternativo, objeto (O2) e metaobjeto (M2). O metaobjeto do bloco alternativo responsável pelo controle do objeto restaura o estado interno do objeto com o estado salvo anteriormente, instanciando um novo objeto, e solicita a execução da mensagem interceptada. O metaobjeto aguarda o retorno do resultado para submetê-lo ao teste de aceitação. Se o resultado passar pelo teste, esse será enviado ao objeto controlador, assim como enviará uma notificação avisando que o metaobjeto corrente é, a partir de agora, o primário. Caso o resultado não passe pelo teste, o ciclo é repetido para os outros blocos alternativos até que uma resposta seja aceita pelo teste ou uma exceção seja sinalizada por não haver mais blocos alternativos.

6.1 Salvamento e Recuperação do Estado

Durante o tempo de execução de um objeto, esse deve fazer salvamentos periódicos de seu estado interno a fim de ser recuperado caso venha a ocorrer alguma falha. Esse procedimento é tomado para que o objeto que venha a ter seu estado recuperado mantenha o funcionamento normal como se nenhuma falha tivesse ocorrido.

O salvamento do estado da computação pode ser programado nos métodos construtor e destrutor ou em um método independente que possa ser chamado esporadicamente. O metaobjeto responsável pela supervisão do objeto pode fazer a invocação desse método de tempos em tempos ou em momentos predefinidos.

Caso um objeto possua estrutura de dados interna diferente da dos demais e em um certo momento venha a utilizar um estado salvo de outro objeto, uma reestruturação de dados deverá ser feita. Essa necessidade de reestruturação deverá ser observada no momento da criação da versão pelo projetista. Um exemplo de reestruturação pode ser: em uma versão, os dados são armazenados em uma estrutura na forma de pilha e, em outra, na forma de lista (FIGURA 31).

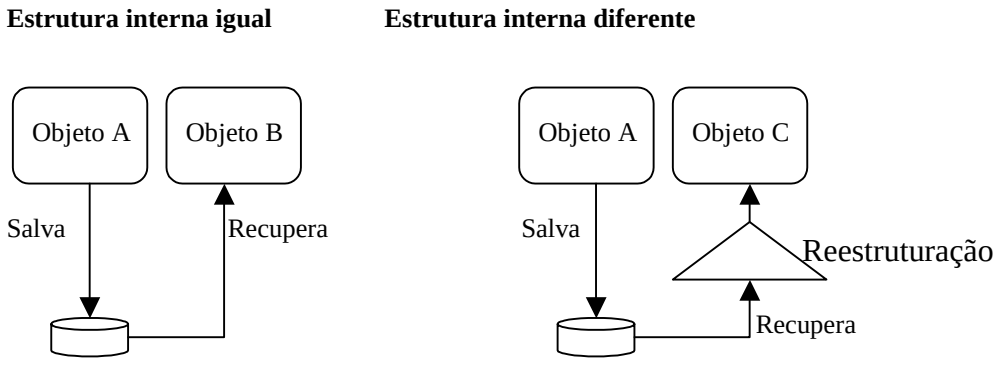

FIGURA 31 - Salvamento e recuperação do estado da computação.

No modelo anteriormente proposto usando a técnica de replicação ativa, o salvamento e a recuperação do estado da computação de um objeto se dão da seguinte forma: todos os objetos têm seus estados salvos periodicamente em um meio estável e, caso um metaobjeto detecte a falha do objeto-base que está sendo observado, instancia

um novo objeto-base restaurando o estado do objeto, usando o último estado salvo de um objeto-base que não falhou para poder continuar a fazer computações na próxima chamada a algum método.

Já, no modelo usando a técnica de replicação passiva, o salvamento e a recuperação do estado do objeto se dão da seguinte forma: a cada chamada a métodos do objeto-base, o metaobjeto efetua o salvamento do estado interno do objeto; se o objeto-base vier a falhar, uma nova instância do objeto será criada pelo metaobjeto. O estado desse objeto é restaurado usando o último estado salvo, e a chamada ao método é novamente efetuada, porém a um novo conjunto de objeto e metaobjeto (bloco alternativo). O estado do objeto do bloco alternativo é restaurado antes do início da execução, usando o estado já salvo.

As novas instâncias criadas após as falhas terem ocorrido e a recuperação de seus estados são realizadas para que se obtenha novamente um estado consistente e para que se prossiga com as atividades normalmente. Os objetos que falharam são mantidos para que o projetista tenha conhecimento dos problemas, fazendo um diagnóstico amplo e preciso das situações em que se apresentam falhas.

6.2 Pontos de Verificação

Os pontos de verificação são os instantes em que são feitas as validações e comparações entre os resultados das n-réplicas (votador), no caso do modelo com replicação ativa, e no teste de aceitação, no caso do modelo com replicação passiva.

Além de uma simples comparação entre os resultados fornecidos pelos objetos, pode-se também fazer uma comparação do estado interno através dos mecanismos de introspeção proporcionados pela reflexão computacional. Se as estruturas dos objetos forem diferentes, a comparação pode se tornar mais complexa.

Tanto no modelo usando a técnica de replicação ativa quanto no que usa replicação passiva, a implementação dos pontos de verificação é feita nos metaníveis da aplicação.

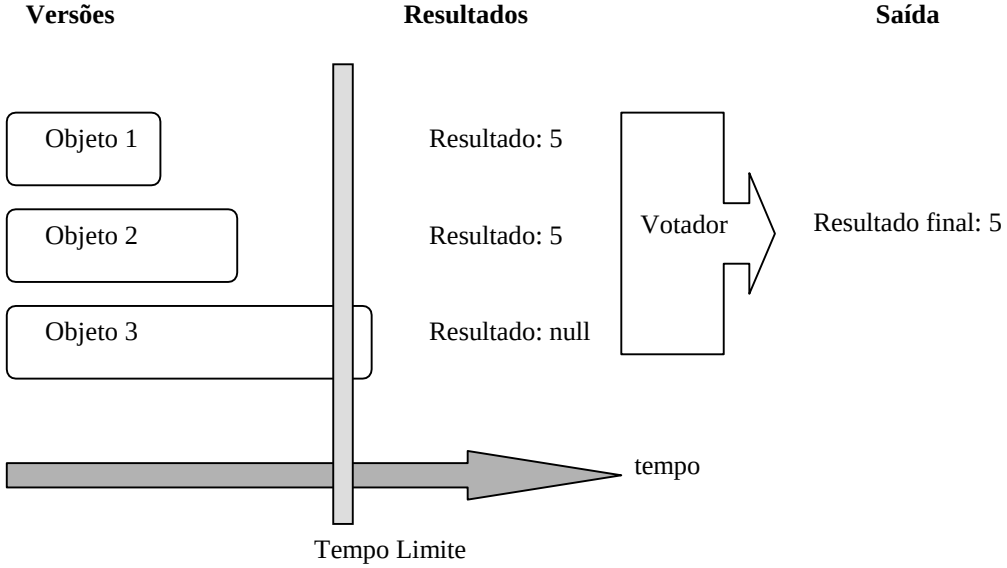

FIGURA 32 - Limite de tempo de execução.

No modelo que usa a técnica de replicação ativa para a inclusão de tolerância a falhas, encontra-se um problema em relação ao limite de tempo de execução. Durante o processamento dos objetos, o invocador da execução (metaobjeto principal) aguarda até que os demais metaobjetos retornem os seus resultados da computação. Isso leva a que o tempo estabelecido para a resposta possa não ser obedecido. Para resolver esse problema, se uma das réplicas não responder em um tempo-limite, deverá ser assumido um valor nulo a esse objeto, registrando a falha de limite de tempo em um *log*. Devem ser realizados todos os procedimentos junto a esse objeto, como se ele tivesse uma falha normal (FIGURA 32).

6.3 Mascaramento

A técnica de mascaramento de falhas e confinamento de danos é empregada quando um objeto apresenta algum resultado incorreto. O erro ocorrido no objeto não é propagado ao usuário, pois esse recebe sempre a resposta correta fornecida por outro objeto, no caso do uso de replicação passiva (bloco alternativo), ou pela votação entre as respostas de outros objetos (n-réplicas) que detenham a maioria. Os erros que podem vir a ocorrer são tratados pelos metaobjetos, os quais providenciam a recuperação.

6.4 Conclusões

Como foi visto, este capítulo mostrou uma proposta de modelo que se utiliza da reflexão computacional aliada às técnicas tradicionais de tolerância a falhas, contribuindo, dessa forma, para o tratamento de problemas tanto de confiabilidade como de disponibilidade em sistemas computacionais, de forma mais clara e com pouca complexidade.

O uso da reflexão computacional como técnica de programação possibilita a introspecção nos objetos que participam do processamento e a separação das atividades de gerenciamento das técnicas de TF e da aplicação em si.

Podem-se visualizar também o salvamento e a recuperação do estado da computação como os pontos mais importantes para o bom funcionamento do modelo, pois, sem um bom controle desses mecanismos, não se obtém sucesso na execução e no uso das técnicas de TF.

O mascaramento é a palavra que melhor resume o trabalho, pois busca-se a qualquer custo oferecer sempre respostas corretas, não importando as situações encontradas durante o processamento.

7 Estudo de Caso

Neste capítulo, descrevem-se a implementação e os experimentos realizados com um exemplo de sistema crítico que se utiliza de serviços que devem ser executados por objetos servidores remotos. Tal exemplo consiste em um sistema de controle de fluxo de trânsito em tempo real, para o qual se empregou a linguagem de programação Java em sua implementação.

A idéia inicial era de utilizar metaJava [GOL97] como ferramenta para a implementação de reflexão computacional. Isso, no entanto, não foi possível, pois metaJava não possui suporte à gravação em arquivos, tratamento de janelas, sendo pobre na parte de comunicação (ambiente distribuído).

Todos os programas foram desenvolvidos utilizando-se a linguagem de programação Java-padrão e RMI para a comunicação entre os nodos distribuídos. Java não possui bibliotecas ou ferramentas para reflexão comportamental, mas oferece suporte para todas as outras necessidades de implementação. Para realizar a reflexão, foram simulados os desvios aos métodos reflexivos através de chamadas simples a métodos, de forma transparente ao programador, pois os direcionamentos se dão no momento de informar o endereço IP de cada nodo da aplicação. A implementação das técnicas de tolerância a falhas deu-se pela criação de classes que são utilizadas entre as partes da aplicação sem que o programador se preocupe ou tenha conhecimento delas.

Para fazer o desenvolvimento e teste do estudo de caso, foram utilizadas três estações de trabalho da Sun Microsystems: uma UltraSPARC 10 300MHz e duas UltraSPARC 5 270MHz, ambas com 192Mb de memória, utilizando o sistema operacional Solaris versão 2.6 em uma rede Ethernet de 100 Mbps *full duplex*.

Foram utilizados também como plataforma de teste dez PCs Pentium-MMX 200 MHz com 32 Mb de memória interligados através de uma rede Ethernet de 10 Mbps. Esses PCs utilizam o sistema operacional Windows NT Workstation 4.0.

Ressalta-se que o foco do trabalho não é a validação dos sistemas utilizados para os cenários, mas, sim, a validação do modelo proposto, testando a aplicabilidade das técnicas de tolerância a falhas agregada com reflexão computacional.

7.1 Cenário 1

O cenário utilizado para desenvolver a aplicação e realizar os testes com as técnicas de TF com RC foi um controle de fluxo de trânsito com semáforo. O sistema utilizado possui características de um sistema de tempo real, o qual não pode sofrer interrupções em seu funcionamento sob pena de provocar problemas e prejuízos. Para melhor visualizar e analisar o uso das técnicas de TF juntamente com RC, o sistema é desenvolvido em uma arquitetura distribuída.

O exemplo aqui apresentado visa analisar os fluxos de veículos que se deslocam em vias que vão em direção a um cruzamento. Nesse cruzamento, existem semáforos que controlam a passagem permitindo-a ou não, com uma divisão de tempo para cada um dos sentidos. Utilizou-se nesse sistema um conjunto de sensores dispostos na via com o objetivo de contar a quantidade de veículos que passam por esse local (FIGURA 33). Os sensores são colocados no início e no fim de um quarteirão, buscando-se saber,

por meio de um cálculo matemático, quantos veículos se encontram naquele trecho. Reunindo as informações sobre todos os sensores dispostos, pode-se manusear o semáforo, de forma que exista um controle mais eficiente e inteligente sobre o fluxo de veículos.

O objetivo desse sistema é fazer com que o trânsito nessas vias flua sem problemas.

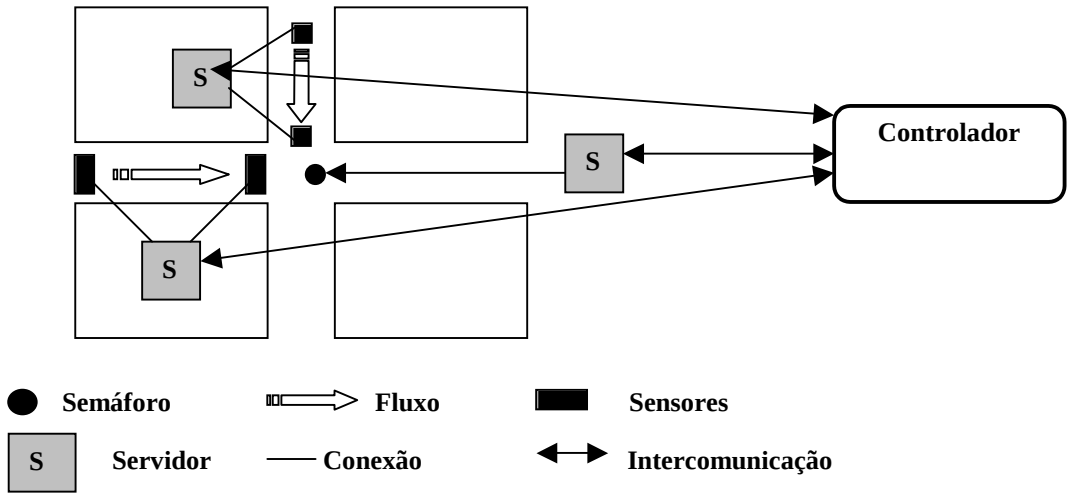

FIGURA 33 - Cenário de um controle de trânsito através de semáforos.

Analisando o cenário apresentado, é possível encontrar alguns problemas de confiabilidade e disponibilidade, caso um dos servidores venha a apresentar alguma falha. Para resolvê-los, podem-se fazer algumas adaptações no sistema, utilizando o modelo reflexivo proposto no capítulo anterior (FIGURA 34).

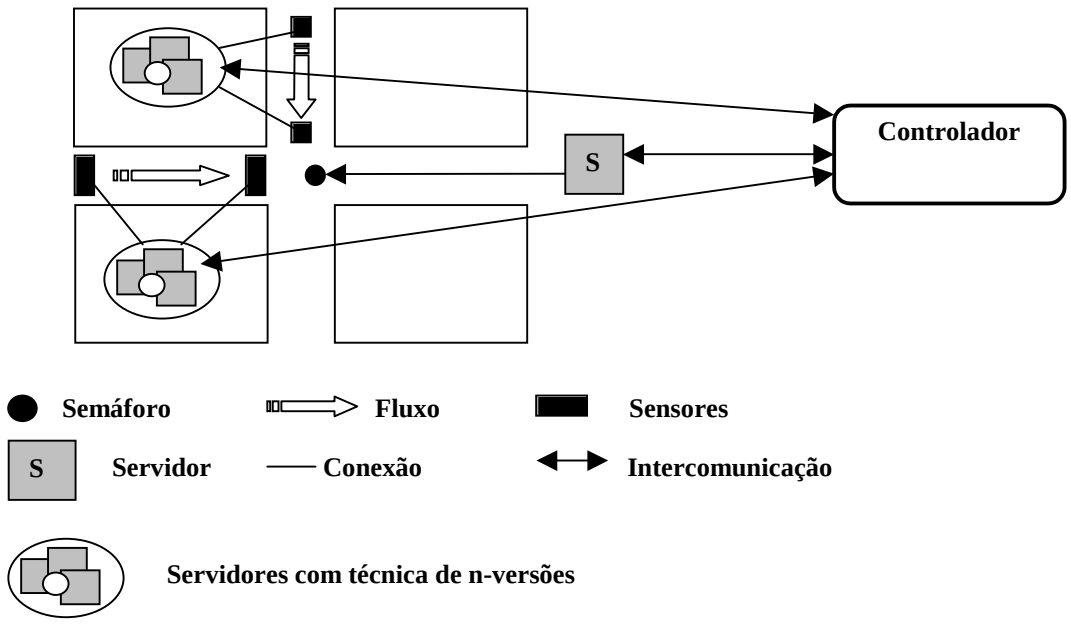

FIGURA 34 - Cenário de um controle de trânsito através de semáforos com TF.

Com essas adaptações, têm-se *n* servidores interligados a sensores, os quais fornecerão dados mais confiáveis, pois se utilizam de técnicas de TF.

Cada par de sensores (início e fim do quarteirão) está associado a um objeto que se encontra em um servidor, existindo *n* pares de sensores ligados a objetos. Os objetos, por sua vez, estão associados entre si através da característica de RC, como foi explicado na proposta, estando associados ao controlador. O controlador solicita a informação sobre o fluxo de veículos aos objetos, os quais realizam a leitura junto aos sensores; fazem a computação para determinar a quantidade de veículos que estão naquele trecho; aplicam as técnicas de TF e retornam com os resultados ao controlador. Os resultados obtidos podem ser considerados confiáveis pelo modo como foram obtidos.

De posse dessas informações, o controlador será capaz de analisar as informações, atualizar a base de dados e tomar decisões, como a de sinalizar ao servidor responsável pelo atuador as seguintes informações: 0, 1 e 2, que são, respectivamente, passagem livre (verde), passagem temporariamente interrompida (vermelha) e alerta (amarelo piscante). No caso de passar do estado livre para o estado interrompido, o controlador fará com que o sinal amarelo seja mostrado por um pequeno período de tempo antes de efetuar a troca.

O funcionamento básico dos objetos (FIGURA 35) que estão associados aos sensores pode ser resumido aos seguinte itens:

1. os objetos lêem um valor numérico de cada sensor a eles ligado, os quais correspondem ao número de veículos que foram registrados;
2. realiza uma computação para verificar a quantidade de veículos que se encontram entre os sensores (fluxo de veículos = sensor1 - sensor2);
3. os objetos utilizam-se da técnica de TF, implantada em metaobjetos, para validar a resposta;
4. as respostas são armazenadas em uma estrutura local (privada);
5. um valor correspondente à média das respostas também é armazenado;
6. as respostas são enviados ao controlador para que possa prosseguir com a computação: fluxo de veículos e a média das respostas das últimas leituras.

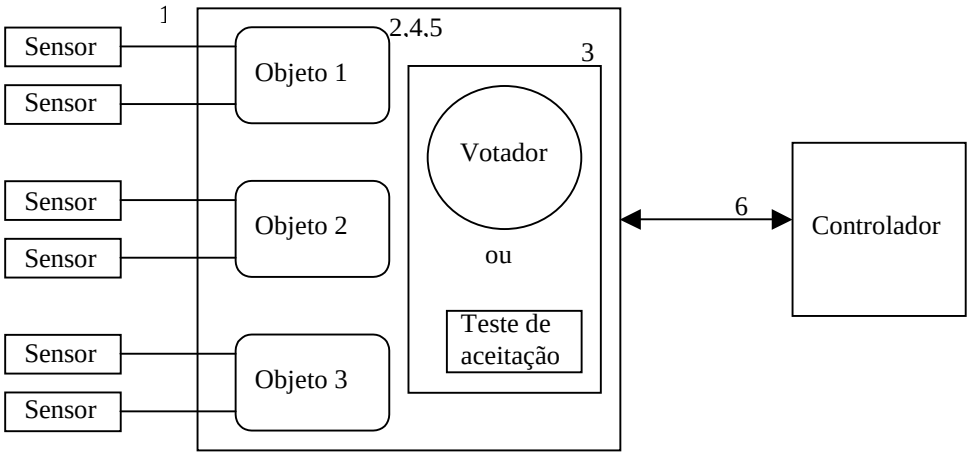

FIGURA 35 - Cenário da implementação com os objetos.

O programa controlador é um programa de tempo-real e não pode ser descontinuado; ele solicita informações sobre o fluxo de veículos para servidores que estão presentes nas duas vias. Esses servidores devem obter as informações sobre o fluxo junto aos sensores e enviar a resposta ao programa controlador para que este tome decisões e siga com o processamento.

Os sensores são componentes de *hardware* que armazenam informações referentes à quantidade de veículos que passam sobre eles. Quando é feita uma leitura nesses sensores, eles retornam o valor armazenado. Como todo componente de *hardware*, eles podem apresentar problemas, tais como não contar, contar a menos, contar a mais ou não responder. Esses se refletirão diretamente no controlador, que necessita dessas informações para tomar decisões sobre o semáforo.

7.1.1 O uso de tolerância a falhas

Com a implantação do modelo proposto, têm-se vários componentes replicados que proporcionarão a confiabilidade e a disponibilidade necessárias. A FIGURA 36 mostra o mapeamento dos componentes replicados, tanto de *hardware* como de *software*, que foram utilizados para a implementação.

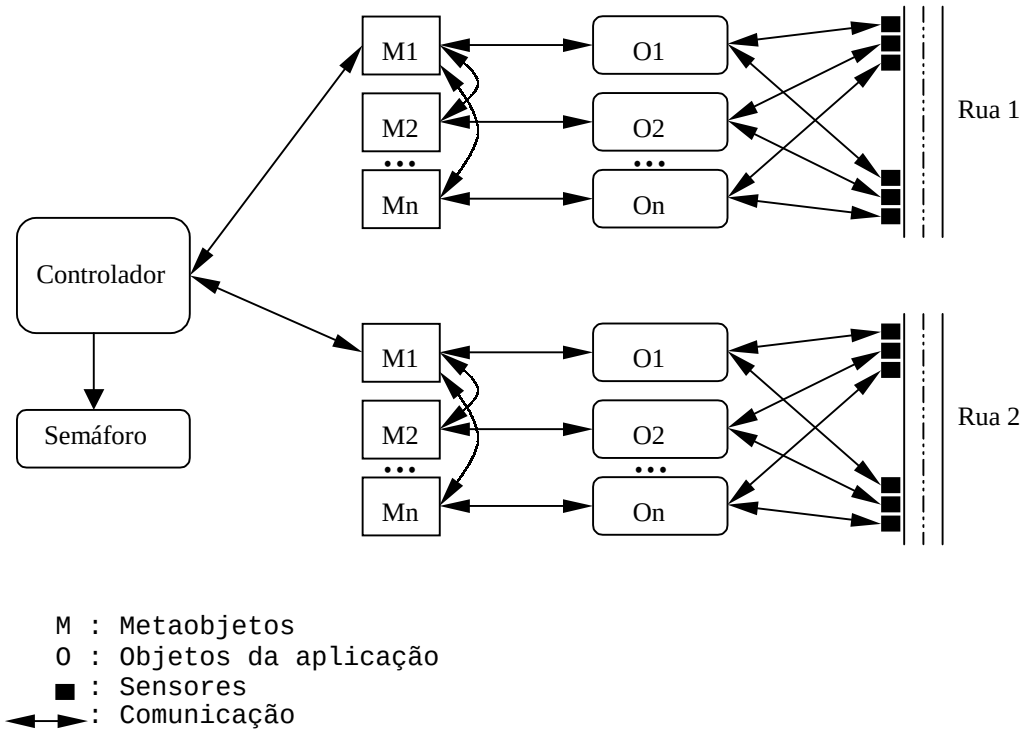

FIGURA 36 - Mapeamento dos componentes replicados.

Com o uso de uma técnica de tolerância a falhas, o fornecimento de uma resposta é mais confiável, não comprometendo o funcionamento da aplicação de tempo-real. Caso não seja utilizada uma técnica de tolerância a falhas, os riscos de que ocorram falhas são muito maiores, comprometendo o sistema e podendo gerar prejuízos.

A técnica utilizada para esse cenário foi a de replicação ativa, com a qual se podem utilizar quantos servidores/sensores forem necessários. A implementação da técnica se dá-se da seguinte forma: para cada par de sensores, há um objeto que solicita as informações sobre o fluxo de veículos (modelo normal) e, para cada objeto, há um metaobjeto que será responsável pela monitoração e controle do objeto. Um dos metaobjetos (metaobjeto1) é responsável pela comunicação entre eles e tem implementado na sua estrutura o mecanismo de votação.

Numa aplicação sem tolerância a falhas (FIGURA 37), o objeto controlador conhece apenas um objeto remoto, que é responsável pelo fornecimento das informações sobre fluxo de veículos. Da mesma forma, na aplicação com tolerância a falhas (FIGURA 36), o controlador conhece apenas um objeto, metaobjeto1, para o qual solicita as mesmas informações sobre fluxo de veículos. O metaobjeto1, por sua vez, conhece um objeto (objeto local) e *n-1*, metaobjetos. Quando o controlador necessita dos dados sobre o fluxo de veículos, solicita-os (invoca um método) ao suposto objeto (metaobjeto1). O metaobjeto1, por sua vez, solicita aos metaobjetos e ao objeto local que executem o método e retornem seus resultados. Cada metaobjeto solicita a seu objeto local que execute o método aguardando o resultado. Se um *timeout* ocorrer, é assumido o valor –2 (dois negativo) como resultado. Quando o metaobjeto1 já está com todos os resultados ou ocorrer um *timeout* (assume –1 como resultado), executa o votador e retorna o resultado. Observando que apenas um dos metaobjetos respondeu, o metaobjeto1 assume que a resposta está correta, mas avisa que o resultado pode não ser confiável, pois a aplicação passou para a situação de não-existência de tolerância a falhas (FIGURA 38, FIGURA 39 e FIGURA 40).

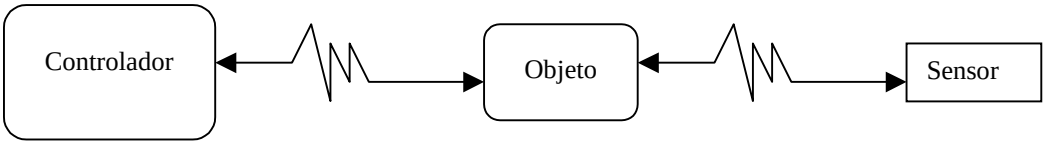

FIGURA 37 - Aplicação sem tolerância a falhas.

```
public class Controlador extends Frame implements Runnable{
        MetaObjetoSensorInterface m1,m2;
        public void run(){
                ...
                while(true){
                        ...
                        try {     qtd[0] = m1.Q_Carros();
                                  qtd[1] = m2.Q_Carros();
                        } catch (Exception e) { ... }
                         ...
                 }
                 ...
        }
}
```

FIGURA 38 - Controlador invoca objeto remoto (metaobjeto1).

```
class MetaObjetoSensor extends UnicastRemoteObject implements MetaObjetoSensorInterface{
        ObjetoSensor os;
        int timeout = 1000;
        public int[] Q_Carros(int tempo) throws RemoteException {
                //-- Solicita ao objeto os dados sobre o fluxo de veículos
                chamaOSQ_Carros c = new chamaOSQ_Carros(tempo);
                //-- Aguarda a finalização da solicitação ou detecta timeout
                ...
                return resultado;
        }
}
```

FIGURA 39 - Objetos remotos (metaobjetos) obtêm dados e retornam.

```
class MetaObjetoSensor1 extends UnicastRemoteObject implements MetaObjetoSensorInterface{
        ...
         public int[] Q_Carros() throws RemoteException {
                tempo=os.get_tempo(); // Obtem o tempo atual no objeto

                //-- Solicita a execução do método aos outros meta-objetos
                ObtemQtd[] c = new ObtemQtd[nmo];
                for(int k=0;k<nmo;k++) c[k] = new ObtemQtd(meta[k],tempo);

                //-- Aguarda o retorno ou timeout é detectado
                ...
                //-- Votador: verifica qual é o resultado da maioria e se não houve timeout
                ...
                //-- Salva o estado dos objetos consistente e invoca restauração dos objetos falhos
                ...
                //-- Retorna o resultado
                return qtd[p];
        }
}
```

FIGURA 40 - Objeto remoto (metaobjeto1) invoca outros metaobjetos.

7.1.2 Salvamento e recuperação do estado dos objeto

Para o salvamento do estado dos objetos, foi utilizada a técnica de serialização, que se caracteriza pela simplicidade por trabalhar diretamente com o sistema de arquivos. Os objetos são direcionados para *streams*, as quais são escritas em uma memória estável (disco rígido), podendo, posteriormente, ser lidas para que se reconstruam os objetos salvos. A escolha pela serialização deu-se também pelo fato de a aplicação trabalhar com um pequeno número de objetos a serem armazenados.

O salvamento do estado do objeto é efetuado após o término da execução do método invocado. O metaobjeto1, que é responsável pela comunicação entre os metaobjetos e pelo votador, é o que comanda o salvamento, a recuperação e a restauração dos objetos com o auxílio dos seus parceiros (outros metaobjetos), pois sabe exatamente quais são os objetos que estão com seu estado consistente e quais os que não estão (FIGURA 41).

```
class MetaObjetoSensor1 extends UnicastRemoteObject implements MetaObjetoSensorInterface{
        ...
         public int[] Q_Carros() throws RemoteException {
                ...
                //-- Salva o estado dos objetos consistente e invoca ressauração dos objetos falhos
                ...
        }
}
class MetaObjetoSensor extends UnicastRemoteObject implements MetaObjetoSensorInterface{
        ObjetoSensor os;
        public void salvar() throws RemoteException {
                FileOutputStream out;
                ObjectOutputStream  sout;
                os.thread.suspend();
                try {   out = new FileOutputStream("temp"+os.nObjeto);
                        sout = new ObjectOutputStream(out);
                        sout.writeObject(os);
                        sout.close();
```

```
                } catch (Exception e){...}
                out = null;    sout = null;    os.thread.resume();
        }
        public ObjetoSensor ler() throws RemoteException {
                FileInputStream in;
                ObjectInputStream sin;
                ObjetoSensor tos=null;
                try {       in = new FileInputStream("temp"+os.nObjeto);
                            sin = new ObjectInputStream(in);
                            tos = (ObjetoSensor)sin.readObject();
                            sin.close();
                } catch (Exception e){ return null; }
                in = null;    sin = null;       return tos;
        }
        public void ressaura(ObjetoSensor arg,int valorretorno) throws RemoteException { ... }
}
```

FIGURA 41 - Salvamento, recuperação e restauração de objetos.

7.1.3 Injeção de falhas

Para fazer a validação de sistemas computacionais que se utilizam de alguma técnica de tolerância a falhas, são utilizados os mecanismos de injeção de falhas. A técnica de injeção de falhas é utilizada para simular a ocorrência de falhas de modo que os componentes do sistema possam ser verificados, avaliados e validados durante a execução do sistema em um curto espaço de tempo. As falhas são introduzidas de forma controlada, auxiliando na validação das técnicas utilizadas.

Neste trabalho, foi utilizada a injeção de falhas por *software*, introduzidas através da alteração de variáveis que são utilizadas para o processamento (falha de resposta), assim como para provocar que o objeto não responda às solicitações de informações (falha de omissão).

A injeção de falha para a falha de resposta foi implementada no ObjetoSensor que faz a coleta de dados junto aos sensores e fornece os dados sobre o fluxo de veículos. Quando for efetuada uma chamada de método a esse objeto, a fim de obter os dados sobre o fluxo de veículos, e for feita uma solicitação por parte do usuário para que uma falha seja injetada, o valor retornado será alterado, provocando, dessa forma, a falha especificada (FIGURA 42). A solicitação de injeção de falha é feita pelo usuário através de um botão, sendo mostrada na interface do objeto (FIGURA 44), o que possibilita saber exatamente quando a falha vai ocorrer para que possam ser analisadas as suas conseqüências.

```
class ObjetoSensor extends Frame implements Serializable,Runnable{
        public int[] Q_Carros(int t){
                ...
                if(provocarerro==1){   // 1 = sim, provocar falha de resposta
                        ...
                        inf[0]=vetQ_Carros[t]=vetQ_Carros[t]+10;  // erro acrescentado: +10
                        ...
                }
                ...
        }
}
```

FIGURA 42 - Injeção de falha de resposta.

A injeção de falha para a falha de omissão também foi implementada no ObjetoSensor. Quando é efetuada uma chamada de método a esse objeto a fim de obter os dados sobre o fluxo de veículos e o usuário solicita que a execução do objeto seja cancelada através do botão fechar (FIGURA 44), o objeto entra em um *loop* infinito, o que acarreta que não haja retorno para a chamada (FIGURA 43). Dessa forma, pode-se analisar como a aplicação se comporta com a presença desse tipo de falha.

```
class ObjetoSensor extends Frame implements Serializable,Runnable{
        ...
        public int[] Q_Carros(int t){
                ...
                if(provocarerro==2){        // 2 = sim, provocar falha de omissão
                        thread.stop();
                        while(true) {}
                }
                ...
        }
}
```

FIGURA 43 - Injeção de falha de omissão.

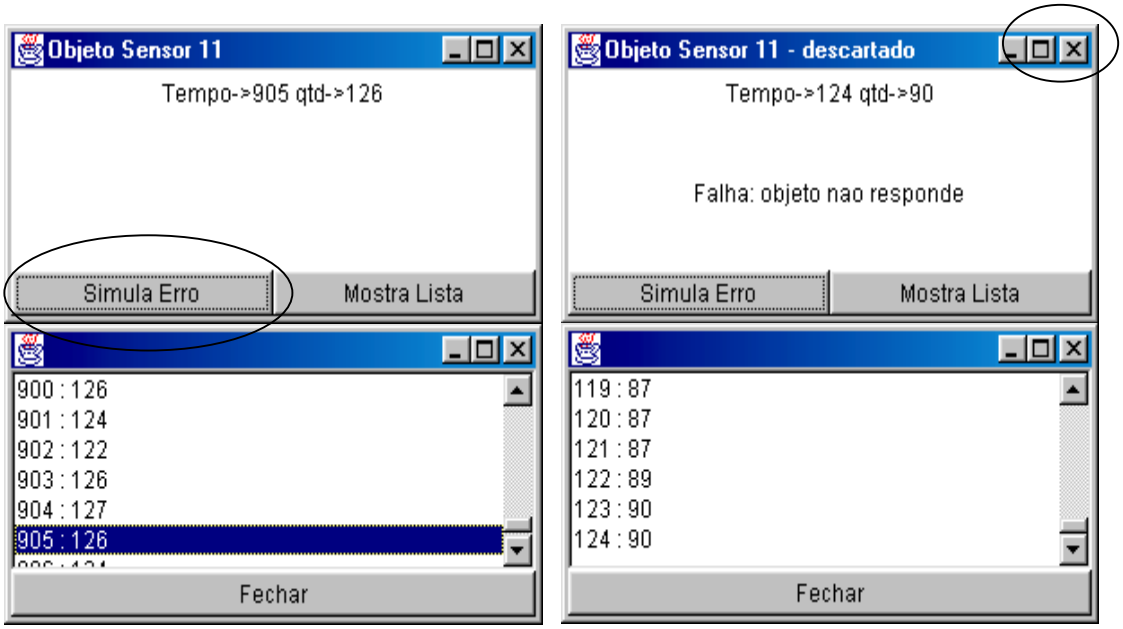

FIGURA 44 - Injeção de falha: resposta e omissão.

Tratamentos para as falhas de parada (fail-stop) e de particionamento de rede também foram implementadas na aplicação (FIGURA 44). Quando, em uma estação de trabalho, ocorre uma falha de parada, o restante do sistema toma conhecimento dessa falha com a perda da conexão. O sistema, nesse instante, assume um valor não-válido (-1) para o valor que se teria como resposta e, de tempos em tempos, tenta refazer a conexão para verificar se a estação já retornou ao seu funcionamento normal.

No caso da falha de particionamento de rede, o sistema também detecta se houve problemas pela perda da conexão. Uma diferença que se pode notar entre as duas é que

a estação que ficou particionada tem seu funcionamento interrompido até o momento da retomada da conexão, ao passo que a que teve problemas de falha de parada tem de ser reiniciada. Ambas, no momento da retomada da conexão, são forçadas a atualizar seus estados internos.

As falhas de parada e de particionamento de rede são suportadas pelo modelo, com exceção de quando se trata do metaobjeto1 e do controlador.

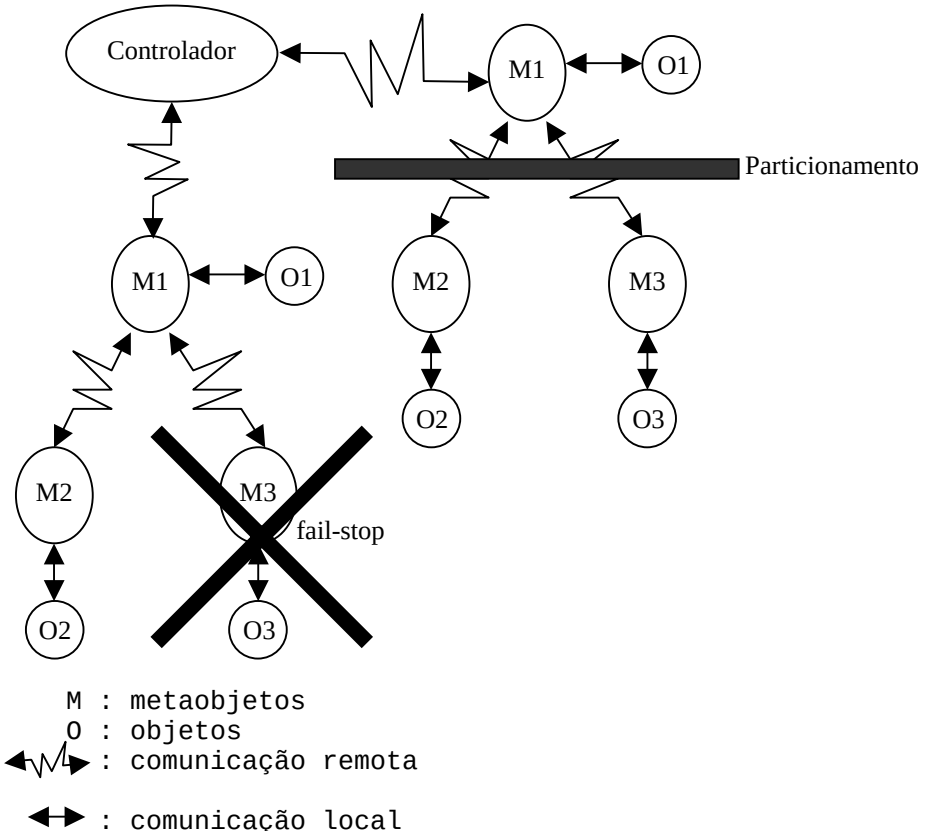

FIGURA 45 - Falhas de particionamento e de parada (*fail-stop*).

7.1.4 A simulação da reflexão computacional

Como já se mencionou, a reflexão computacional foi simulada. Os desvios aos métodos reflexivos foram realizados através de chamadas simples a métodos. A principal preocupação foi desenvolver a simulação de forma que ficasse transparente ao programador. Foram desenvolvidas novas classes que contêm os mesmos métodos do objeto que será monitorado, acrescentando-se a palavra *meta* ao nome das classes. Por exemplo, a classe do objeto chama-se *ObjetoSensor* e a classe responsável pela reflexão, *MetaObjetoSensor*.

As classes *meta* fazem a validação pela comparação (votação) dos resultados obtidos, retornando ao objeto que invocou o método o valor que obteve a maioria.

Quando um método do objeto é invocado, o método do metaobjeto é executado, solicitando a execução do método a todos os metaobjetos, os quais, por sua vez, solicitam o mesmo aos objetos (FIGURA 46). O método do metaobjeto aguarda até que todos retornem ou que um *timeout* definido seja observado. Como se podem ter *n-metaobjetos* replicados, fez-se a invocação dos métodos aos objetos através de *threads*, pois, o sendo o ambiente de processamento distribuído, não haveria uma justificativa para um processamento seqüencial, o que ocasionaria um gasto de tempo muito grande.

```
public class ObjetoSensor
{ ... }
public class MetaObjetoSensor {
        int timeout=3000;
        public int[] Q_Carros() throws RemoteException {
                ObtemQtd[] c = new ObtemQtd[n];

                for(int k=0;k<nmo;k++)
                        c[k] = new ObtemQtd(meta[k],tempo);      // ativa Threads

                //-- aguarda o retorno dos threads ou timeout
                ...
                //-- Votador: verifica qual e o resultado da maioria
                ...
                //-- Ressaura os objetos que falharam
                ...
                //-- Retorna quantidade
                ...
        }
}
```

FIGURA 46 - Desvio da execução de um método para o metaobjeto.

Como a aplicação foi desenvolvida para ambiente distribuído, no momento de executar algum componente, deve-se informar onde os outros componentes estão rodando através do endereço IP das máquinas (FIGURA 47). Dessa forma, pode-se obter a transparência necessária para a reflexão, pois, ao invés de informar onde está o objeto, informa-se onde está o metaobjeto. O metaobjeto, por sua construção, instancia o objeto e começa a controlá-lo.

```
...
java Controlador  192.168.115.14/MetaObjetoSensor11   192.168.115.18/MetaObjetoSensor12
...
```

FIGURA 47 - Execução do controlador informando o IP dos metaobjetos.

7.1.5 A comunicação entre o controlador e servidores

Os objetos remotos da aplicação comunicam-se através da utilização de um conjunto de classes e interfaces oferecidas pelo Java - RMI (*Remote Method Invocation*). O RMI integra a tecnologia de objetos distribuídos diretamente à linguagem Java, fazendo a chamada remota de forma quase transparente; apresenta uma interface de programação para rede de mais alto nível, mais conveniente e mais natural em muitos casos do que *sockets* e *streams*. O servidor, quando é executado, realiza o processamento enquanto aguarda que um método seja invocado (mensagem). Esse método é executado e, se for o caso, retorna a resultado do processamento (FIGURA 48).

```
import java.rmi.*;
import MetaObjetoSensorInterface;
public class Controlador extends Frame implements Runnable{
        MetaObjetoSensorInterface m1,m2;
        public Controlador(String meta1,String meta2,String sin) {
```

```
                while(true){
                        try {
                                m1 = (MetaOSensorInterface)Naming.lookup("rmi://"+meta1);
                                m2 = (MetaOSensorInterface)Naming.lookup("rmi://"+meta2);
                                break;
                        } catch (Exception e){ ... }
                }
                ...
        }
        public void run(){
                ...
                while(true){
                        ...
                        try {   qtd[0] = m1.Q_Carros();
                                qtd[1] = m2.Q_Carros();
                        } catch (Exception e) { ... }
                        ...
                }
                ...
        }
}
class MetaObjetoSensor extends UnicastRemoteObject implements MetaObjetoSensorInterface{
        ObjetoSensor os;
        ...
        public int[] Q_Carros() throws RemoteException {
                ...
        }
}
```

FIGURA 48 - Invocação de métodos de um objeto remoto.

7.1.6 Análise de desempenho

A aplicação foi executada em diferentes plataformas (rede de PCs e Estações de trabalho Sun) e com diferentes números de objetos para que se pudesse verificar como a aplicação se comportava na presença ou não de falhas, além de determinar o tempo gasto com processamento e comunicação. Foram feitas exaustivas execuções em diferentes horários. Para que a coleta dos dados sobre o tempo gasto entre a requisição de informações e o retorno feita pelo controlador não fosse prejudicada pelo tráfego existente na rede, foram feitos os levantamentos de dados em períodos em que não havia outras pessoas ou processos utilizando a rede.

Para o levantamento dos dados, foram feitas aplicações com um, três (FIGURA 49), cinco e nove objetos distribuídos pelas máquinas. Os dados foram obtidos através de arquivos de LOG que são gerados pela aplicação (TABELA 2). O tempo utilizado para cada execução foi de uma hora.

TABELA 2 - Média de tempo gasto para processamento com PNV

	Rede de PCs	Essações Sun
Um objeto (sem TF)	484	69
Três objetos	2884	243
Cinco objetos	3955	291
Nove objetos	14550	414

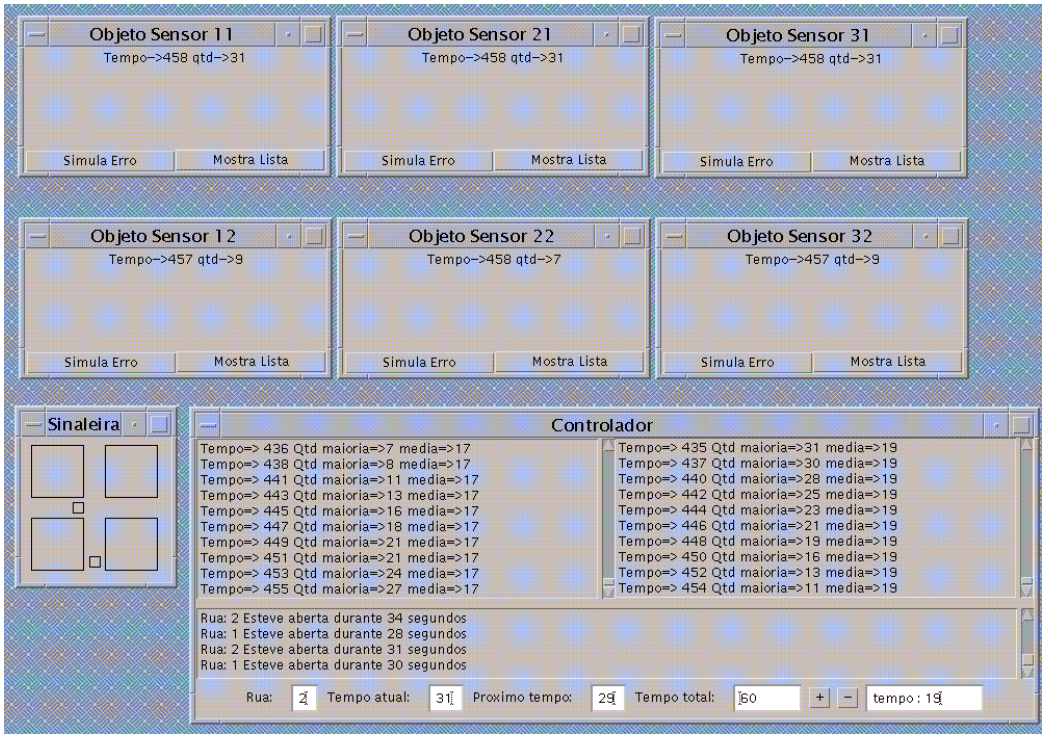

FIGURA 49 - Interface visual dos componentes com três objetos por rodovia (reunidos)

7.2 Cenário 2

O segundo cenário utilizado para desenvolver a aplicação e realizar os testes do uso das técnicas de TF com RC foi um monitoramento da temperatura de uma caldeira. Como no cenário 1, o sistema utilizado possui características de um sistema de tempo real, não podendo sofrer interrupções em seu funcionamento sob pena de provocar problemas e prejuízos. Para melhor visualizar e analisar o uso das técnicas de TF juntamente com RC, o sistema foi desenvolvido em uma arquitetura distribuída.

A simulação implementada busca fazer um monitoramento da temperatura de uma caldeira a fim de informar aos operadores, através de um gráfico, as oscilações que ocorreram em um período de tempo. Essas informações servirão como base para comandar o equipamentos responsáveis por aumentar ou diminuir a temperatura.

Há, nesse sistema, um sensor de temperatura instalado na caldeira, que está interligado a um servidor, o qual é responsável por informar ao controlador a temperatura obtida (FIGURA 50).

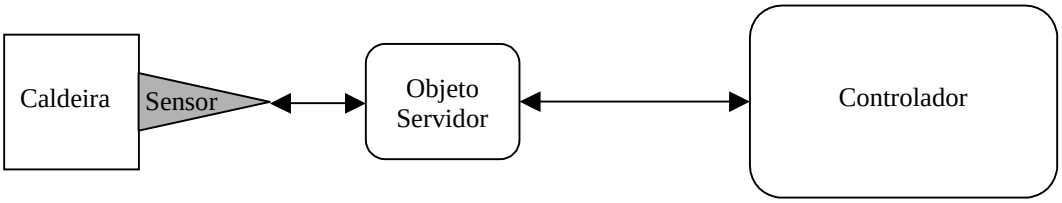

FIGURA 50 - Monitoramento de temperatura de uma caldeira.

Analisando o cenário apresentado, podem-se encontrar problemas de confiabilidade e disponibilidade caso o servidor venha a apresentar alguma falha. Para resolvê-los, podem-se fazer algumas adaptações no sistema (FIGURA 51), utilizando novamente o modelo reflexivo proposto no capítulo anterior (replicação passiva).

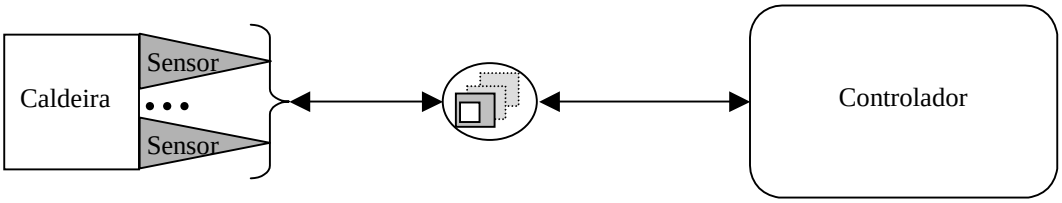

FIGURA 51 - Monitoramento de temperatura de uma caldeira com TF.

Com essa adaptação, ter-se-ão n objetos/servidores interligados a *n* sensores, que fornecerão dados mais confiáveis, pois utilizam-se de uma técnica da tolerância a falhas: replicação passiva.

Todos os objetos/servidores estão associados entre si pela característica de RC, como foi explicado na proposta, estando associados ao controlador. O controlador solicita informação sobre a temperatura ao objeto que se encontra no bloco primário. O objeto realiza a leitura junto ao sensor, executa o teste de aceitação e devolve o resultado ao controlador. Tal resultado pode ser considerado confiável por ter sido obtido através do uso de tolerância a falhas.

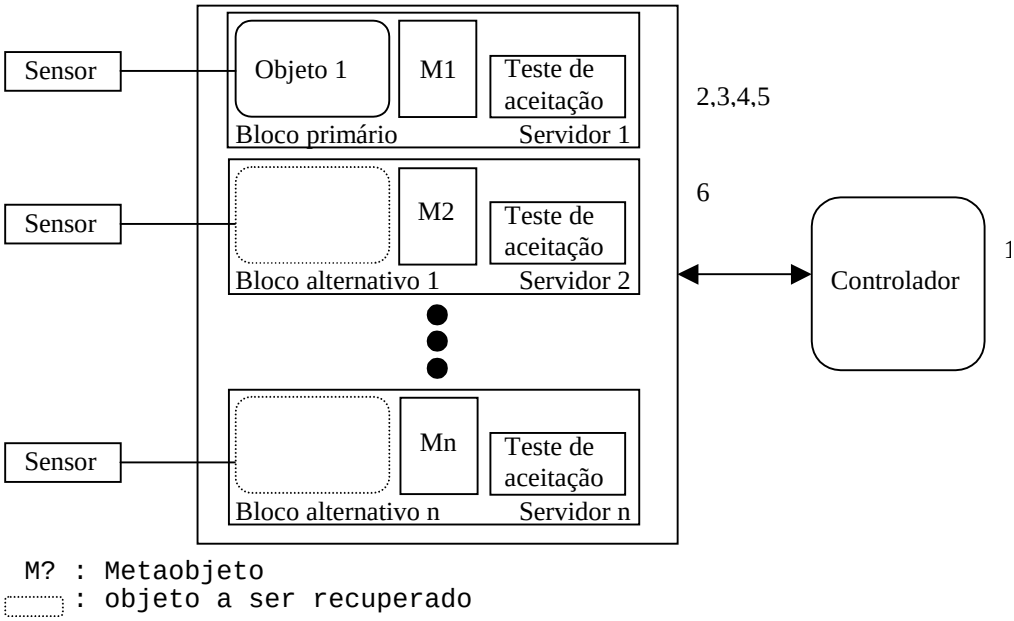

FIGURA 52 - Cenário da aplicação com os componentes.

O funcionamento básico dos componentes (FIGURA 52) da aplicação pode ser resumido nos seguintes itens:

1. controlador: solicita ao bloco primário a temperatura atual da caldeira;
2. bloco primário: armazena seu estado interno em uma memória estável;
3. bloco primário: obtém um valor numérico junto ao sensor que corresponde à temperatura da caldeira e encaminha esse valor ao teste de aceitação para que seja validado;
4. bloco primário: passando pelo teste, retorna o resultado, avisa ao controlador que ele é o bloco primário para a próxima requisição e aguarda até que seja solicitado novamente;

5. bloco primário: não passando pelo teste, restaura o estado salvo no próximo bloco alternativo *, instanciando um novo objeto;
6. bloco alternativo: torna-se o bloco primário e executa os itens 3,4 e 5.

* Se não existirem mais blocos alternativos, é retornado um valor nulo.

7.2.1 O uso de tolerância à falha

Com a implantação do modelo proposto, ter-se-ão vários componentes replicados que proporcionarão a confiabilidade e a disponibilidade necessárias. Como visto na FIGURA 52, os componentes são replicados tanto na parte de *hardware* como na de *software*.

A técnica de tolerância a falhas utilizada nesse cenário foi a técnica de replicação passiva, a qual trabalha com um bloco primário e *n* blocos alternativos. Podem-se ter tantos blocos alternativos quantos forem necessários para a aplicação.

Para cada bloco, além de um objeto normal, existe um metaobjeto que faz o monitoramento e o controle do objeto, sendo responsável pela interceptação das mensagens enviadas ao objeto, pelo salvamento do estado do objeto, pela execução do teste de aceitação sobre a resposta fornecida pelo objeto e pela comunicação entre eles. Cada metaobjeto conhece o endereço do metaobjeto que está no bloco alternativo seguinte.

Da mesma maneira que em uma aplicação sem tolerância a falhas, o controlador conhece apenas um objeto ao qual pode solicitar informações, que, no caso, é o metaobjeto. O metaobjeto envia ao objeto local a mensagem recebida, aguardando a resposta para aplicar sobre ela o teste de aceitação. Se a resposta não passar pelo teste, o metaobjeto solicita que o próximo metaobjeto restaure o estado salvo anteriormente, criando um novo objeto e executando a mensagem enviada pelo controlador. A partir desse momento, o bloco alternativo passa a ser o bloco primário, dando seqüência à execução. Se a resposta do objeto do bloco alternativo falha, o ciclo é retomado até que uma resposta seja aceita ou não haja mais blocos alternativos (FIGURA 53 e FIGURA 54).

Para a implementação do teste de aceitação, fez-se a seguinte suposição: se a temperatura obtida variasse mais do que 10% (+10% ou –10%) sobre a leitura anterior, uma falha havia ocorrido; se isso não ocorresse, a temperatura era considerada válida. Pode-se dizer que uma falha ocorria se a temperatura variava, de uma leitura a outra, de 100 a 150, por exemplo.

```
public class Controlador extends UnicastRemoteObject implements
ControladorInterface,Runnable{
        ObjetoInterface objeto;
        public void run (){
                while(true){
                        ...
                        try { temp = objeto.ObtemTemperatura(); }
                        catch(Exception e){...}
                        ...
                }
        }
}
```

FIGURA 53 - Controlador invoca objeto remoto (metaobjeto).

```
public class MetaObjeto extends UnicastRemoteObject implements  MetaObjetoInterface{
        Objeto objeto;
        MetaObjetoInterface ProximoMetaObjeto;
        ...
        public int ObtemTemperatura() throws RemoteException{
                ...
                Salvar();  // salva estado do objeto
                ...
                // Solicita execução do método ao objeto local
                ObtemTemperatura ot = new ObtemTemperatura(objeto);
                // Aguarda retorno ou timeout é detectado
                // Executa teste de aceitação
                if (teste_de_aceitacao()) {
                        // avisa controlador que ele é o bloco primário
                        // retorna resultado
                } else {
                        // ressaura estado salvo
                        // instancia novo objeto no próximo bloco
                        // solicita execução ao metaobjeto do próximo bloco e retorna o  resultado
                }
        }
}
```

FIGURA 54 - Objeto remoto (metaobjeto) invoca objeto local.

7.2.2 Salvamento e recuperação do estado dos objetos

Como no cenário 1, foi utilizada a técnica de serialização para o salvamento do estado dos objetos pelo fato de a aplicação trabalhar com um pequeno número de objetos que devem ser armazenados em memória estável.

No instante em que uma mensagem é recebida pelo metaobjeto para executar algum método, ele efetua o salvamento do estado do objeto local e solicita a esse objeto que execute o método solicitado, aguardando até que o objeto retorne o resultado ou um timeout seja percebido. Se um timeout for percebido ou alguma falha for observada através do teste de aceitação, o metaobjeto repassará a mensagem recebida ao próximo metaobjeto, informando-o de que deverá restaurar o estado do objeto e executar a mensagem. Caso não se detecte falha nem timeout, o metaobjeto avisará ao controlador que ele é o bloco primário e retornará o resultado obtido a quem solicitou. O metaobjeto do bloco alternativo executa os mesmos passos do metaobjeto do bloco primário (FIGURA 55).

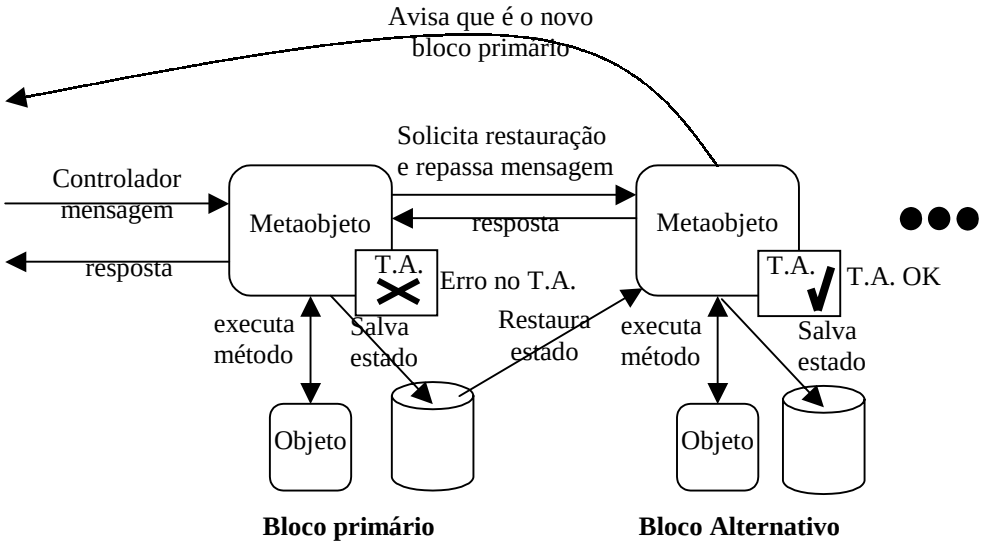

FIGURA 55 - Salvamento, recuperação e execução do objeto.

```
public class MetaObjeto extends UnicastRemoteObject implements  MetaObjetoInterface{
        Objeto objeto;
        MetaObjetoInterface ProximoMetaObjeto;
        ...
        public int ObtemTemperatura() throws RemoteException{
                // salva estado do objeto
                // solicita execução do método ao objeto local e aguarda retorno ou timeout
                // executa teste de aceitação
                if (teste_de_aceitacao())
                        // avisa controlador que ele é o bloco primário e retorna resultado
                else {  // restaura estado e solicita instanciação do novo objeto no próximo bloco
                        // solicita execução ao metaobjeto do próximo bloco e retorna o  resultado}
        }
        public void Salvar(){
                ...
        }
        public Objeto ler() throws RemoteException{
                ...
        }
        public void instancia (Objeto arg) throws RemoteException{
                ...
        }
}
```

FIGURA 56 - Salvamento, recuperação e instanciação de objetos.

7.2.3 Injeção de falhas

Nesse cenário, a injeção de falhas foi implementada para simular as falhas de resposta e de omissão.

Para as falhas de resposta, foi implementado, na interface do objeto, um botão de forma que o usuário pudesse solicitar que o valor da temperatura fosse alterado, gerando, dessa forma, um valor não aceitável. Assim, o usuário podia analisar as conseqüências desse tipo de falha.

Para as falhas de omissão, foi feita uma implementação para que nenhum valor fosse retornado quando uma chamada de método ao objeto fosse efetuada, isto é, o objeto entrava em um laço infinito, gerando, dessa forma, uma falha que seria detectada através de um *timeout* no metaobjeto.

```
public class Objeto extends Frame{
        ...
        public int ObtemTemperatura(){
                ...
                temperatura = sensor.ObtemTemperatura(chamada);
                if(provocaerro==1){         // provocar falha de resposta
                        // erro entre 50-100% a mais
                        temperatura=temperatura+temperatura/2+(int)(Math.random()*temperatura/2);
        }
                ...
                if(provocaerro==2){         // provocar falha de omissão
                        // laço infinito
                        while(true) {}
                }
                ...
        }
        public boolean handleEvent(Event e){
                if(e.id==Event.ACTION_EVENT)
                        if(e.arg.equals("Provoca erro"))
                                provocaerro=1;
                if(e.id==Event.WINDOW_DESTROY)
                        provocaerro=2;
                return false;
        }
}
```

FIGURA 57 - Injeção de falhas de resposta e omissão.

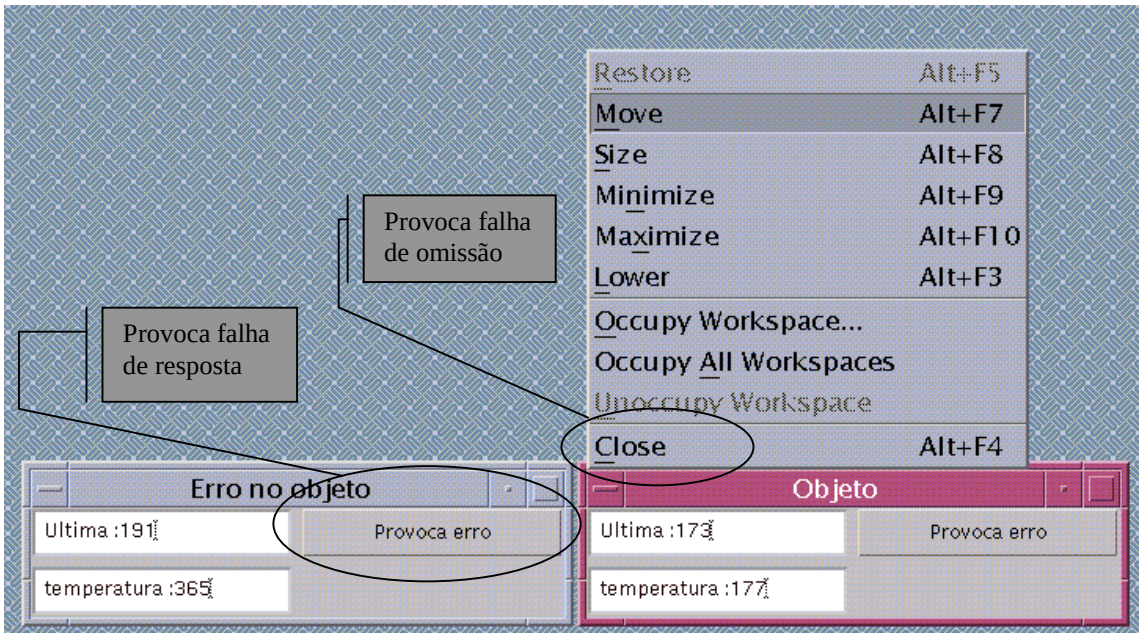

FIGURA 58 - Injeção de falhas de resposta e omissão.

7.2.4 A simulação da reflexão computacional

Como no cenário 1, a reflexão computacional foi simulada. Os desvios aos métodos reflexivos foram realizados através de chamadas simples a métodos. A aplicação de tolerância a falhas foi desenvolvida de forma que ficasse transparente ao programador. Classes foram escritas contendo métodos com os mesmos nomes dos métodos encontrados nos objetos, juntamente com novos métodos necessários para a reflexão. As novas classes receberam o mesmo nome das classes dos objetos, sendo acrescidas da palavra *Meta.*

As classes *Meta* fazem a validação através de um teste de aceitação. Caso a execução não passe pelo teste, o próximo bloco alternativo é preparado para execução e realiza todas as operações do bloco principal. Se a resposta obtida por esse bloco for aceita pelo teste, o bloco torna-se o novo bloco primário; em caso contrário, todo o processamento é repetido para o próximo bloco alternativo, até que um deles tenha seu resultado aceito ou não existam mais blocos alternativos. Caso isso ocorra, uma falha é sinalizada.

Desenvolvendo o presente cenário sem tolerância a falhas e sem reflexão computacional, tem-se uma aplicação com apenas três componentes remotos: o objeto controlador, o objeto responsável pela leitura e o sensor. Para executar essa aplicação, necessita-se informar o endereço do objeto ao controlador e o endereço do sensor ao objeto (FIGURA 59). A simulação da reflexão computacional utilizada aproveita exatamente essa necessidade de informar o endereço do componente, fazendo o desvio das chamadas realizadas pelo controlador ao objeto para o metaobjeto. Ao metaobjeto é informado o endereço do próximo metaobjeto (bloco alternativo) e o endereço do sensor, possibilitando, dessa forma, a inclusão de uma das técnicas de tolerância a falhas: a técnica de replicação passiva (FIGURA 60). O metaobjeto instancia um objeto que será por ele controlado.

```
...
java ObjetoR objeto 192.168.116.30/sensor &
java Controlador controlador 192.168.116.31/objeto &
...
```

FIGURA 59 - Linhas de comando para execução sem RC e TF.

Interpret./Classe	nome	próximo metaobjeto	controlador	Sensor	
java MetaObjeto	meta1	192.168.115.2/meta2	192.168.115.10/controlador	192.168.116.30/sensor	p
java MetaObjeto	meta2	192.168.115.3/meta3	192.168.115.10/controlador	192.168.116.30/sensor	
java MetaObjeto	meta3	192.168.115.4/meta4	192.168.115.10/controlador	192.168.116.30/sensor	
java MetaObjeto	meta4	192.168.115.1/meta1	192.168.115.10/controlador	192.168.116.30/sensor	

Interpret. / Classe	nome	metaobjeto principal			
java Controlador	controlador	192.168.115.1/meta1			

FIGURA 60 - Linhas de comando para execução com RC e TF.

7.2.5 Análise de desempenho

Como a aplicação do cenário 1, a aplicação foi executada em diferentes plataformas (rede de PCs e estações de trabalho Sun) e com diferentes números de

objetos para verificar como o aplicação se comportava na presença ou não de falhas e, também, verificar o tempo gasto com processamento e comunicação.

Para que a coleta dos dados sobre o tempo gasto entre a requisição de informações e o retorno feita pelo controlador não fosse prejudicada pelo tráfego existente na rede, foram feitos os levantamentos de dados em períodos nos quais não havia outras pessoas ou processos utilizando a rede.

Para o levantamento dos dados, foi utilizada uma aplicação com cinco metaobjetos distribuídos (FIGURA 61). Os dados foram obtidos através de arquivos de LOG que são gerados pela aplicação (TABELA 3). O tempo utilizado para cada execução foi de uma hora.

Como foi utilizada a técnica de replicação passiva nessa aplicação, a diferença de tempo de processamento de uma aplicação com três objetos para outra com cinco objetos não foi muito diferente, pelo fato de a técnica apenas executar os blocos alternativos se o bloco primário falhar. Por esse motivo, num primeiro momento, fez-se a execução da aplicação sem a presença de técnicas de TF e RC (apenas um objeto). A seguir, executou-se a aplicação com as técnicas de TF e RC, porém na presença de falhas que foram forçadas a ocorrer.

TABELA 3 - Média de tempo gasto para processamento com BR

	Rede de PCs	Estações Sun
Um objeto (sem TF)	39	17
Cinco objetos s/ falhas	73	30
Cinco objetos c/ 1 falha	18941	445
Cinco objetos c/ 2 falhas	30697	865
Cinco objetos c/ 3 falhas	41308	1216
Cinco objetos c/ 4 falhas	45144	1583

* Tempo em milissegundos.

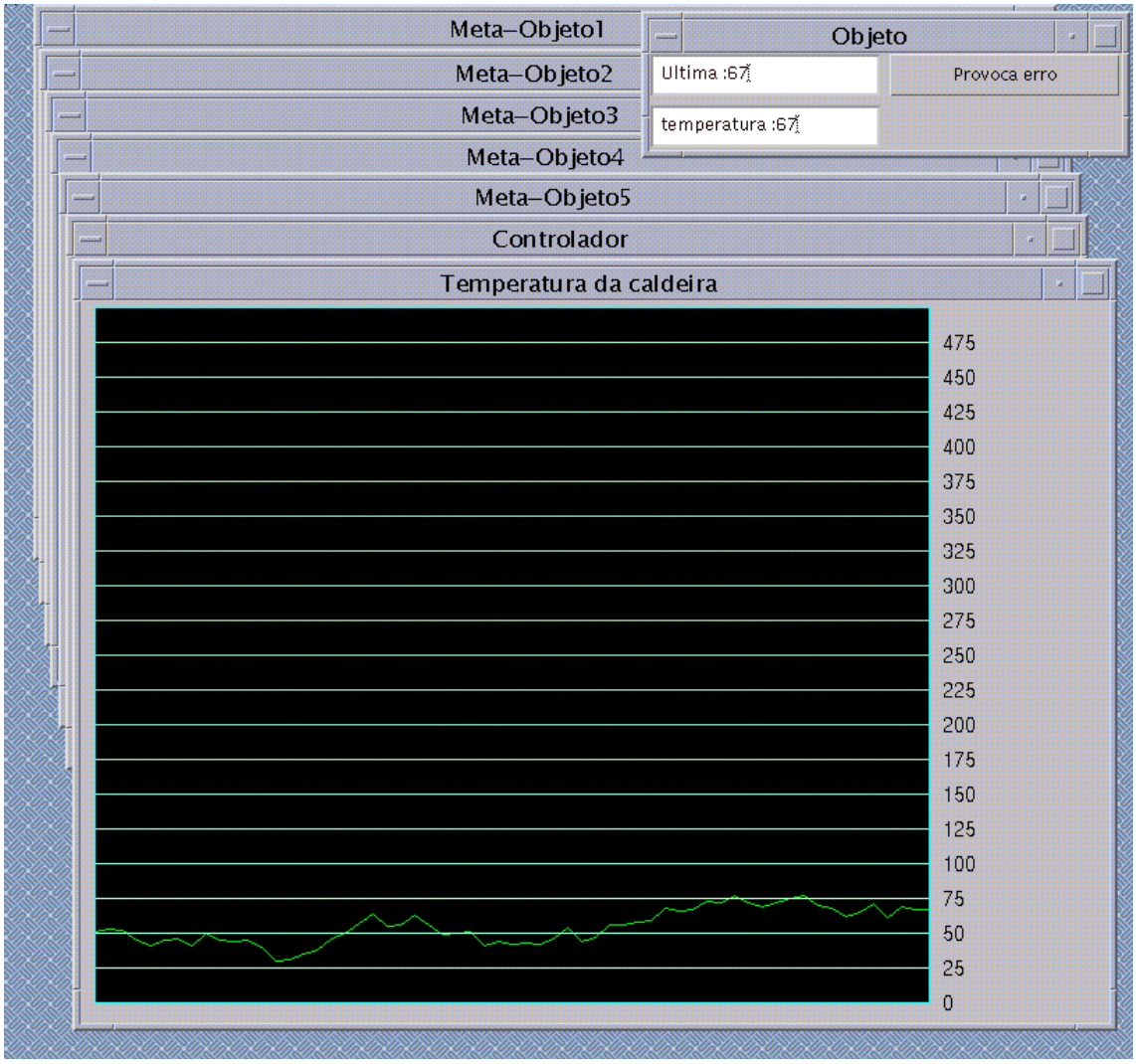

FIGURA 61 - Interface visual dos componentes com cinco metaobjetos (reunidos).

8 Conclusões

Este trabalho consiste num estudo sobre técnicas do domínio de tolerância a falhas, técnicas para salvamento e recuperação de estados de objetos, das características dos ambientes distribuídos e da arquitetura reflexiva, visando propor uma estratégia de união dessas técnicas, características e arquitetura para resolver o problema da falta de confiabilidade e disponibilidade dos sistemas computacionais de forma transparente ao programador e ao usuário.

O objetivo da pesquisa foi atingido na medida em que se obteve sucesso na implementação dos cenários do estudo de caso com a utilização da proposta apresentada. O trabalho ofereceu uma solução viável para a utilização em sistemas que necessitam de total confiabilidade dos dados e disponibilidade para processamento.

A proposta visou principalmente à implementação de um ambiente distribuído para a construção das aplicações, o qual possibilita a divisão das tarefas junto aos nodos, acelerando, dessa forma, o processamento para a obtenção dos resultados. Com pequenos ajustes, também se pode utilizar essa solução para ambientes centralizados. Tais modificações não são feitas no código da aplicação, mas, sim, na maneira como os módulos da aplicação são ativados.

A utilização de reflexão computacional foi de fundamental importância por possibilitar a separação da aplicação em dois níveis: nível-base e nível-meta. O nível-base é responsável pelos aspectos de funcionalidade da aplicação, e o nível-meta, pelo controle e administração. As técnicas de tolerância a falhas foram implementadas no nível-meta a fim de que o programador de um sistema qualquer não precise se preocupar com a implementação do controle sobre os componentes da aplicação, dedicando-se mais ao domínio da aplicação. Com a separação do nível-base do metanível, possibilita-se que uma aplicação seja mais facilmente compreendida, permitindo uma reutilização dos algoritmos desenvolvidos para o uso das técnicas de tolerância a falhas.

As técnicas de tolerância a falhas são utilizadas para validar os resultados obtidos por meio da comparação dos resultados, no caso de replicação ativa, ou com o uso de um teste de aceitação, no caso de replicação passiva.

A programação distribuída aliada à programação paralela também teve uma importante contribuição no modelo proposto, pois possibilitou que se incluíssem novas versões na aplicação, auxiliando na validação das respostas obtidas. Essa inclusão de novas versões não exige modificações no código-fonte.

Outro ponto de fundamental importância é o salvamento do estado dos objetos para uma futura possível restauração. Como foi visto, existem várias formas de armazenar o estado dos objetos em memória estável, contudo, para cada técnica de TF utilizada, as necessidades são diferentes. Isso implica que a escolha seja uma etapa muito importante para o bom funcionamento da aplicação.

Como trabalho futuro, propõe-se uma reestruturação das metaclasses implementadas neste trabalho a fim de que seja possível uma reutilização dos componentes básicos de TF na definição de novas metaclasses necessárias a novos tipos de aplicações. As metaclasses podem ser recusadas pelo mecanismo de herança.

Além da reestruturação, propõe-se a criação de um pré-processador para a linguagem Java, que seja capaz de realizar a reflexão comportamental sem que haja a perda de qualquer um dos componentes da linguagem, incluindo automaticamente o fornecimento de técnicas de TF e de salvamento e recuperação do estado dos objetos.

Como foi visto, a inclusão de reflexão computacional, aliada a técnicas de tolerância a falhas, pode tornar a aplicação um pouco pesada; por isso, essas técnicas devem ser utilizadas apenas em sistemas de tempo-real, nos quais as restrições temporais não são muito rígidas ou um alto nível de tolerância a falhas é exigido. Para amenizar esse problema, o uso de ambientes distribuídos é aconselhável.

Bibliografia

[AVI85] AVIZIENIS, A. The n-version approach to fault-tolerant software. **IEEE Transactions on Software Engineering**, New York, v. SE-11, n.12. Dec. 1985.

[AND76] ANDERSON, T.; KERR, R. Recovery blocks in action: a system supporting high reliability. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 2., 1976, San Francisco, CA. [S.l.: s.n.], 1976.

[BEC98] BECCENERI, José Carlos e FERNANDES, Clovis Torres. Conceitos fundamentais em sistemas de tempo real. In: WORKSHOP EM SISTEMAS DE TEMPO REAL, 1., 1998. **Anais...** Rio de Janeiro: Sociedade Brasileira de Computação, 1998.

[CAM96] CAMPOS, Alvaro E.; CASTILLO, Miguel Angel A. **Checkpointing through garbage collection**. Pontificia Universidad Católica de Chile, Santiago/Chile, 1996. Disponível por WWW em http://alesna.ing.puc.cl/~mca/chicago/paper.html (dezembro de 1998).

[CON97] CONCEIÇÃO, Sérgio Ricardo da. **Um protocolo para rastreamento de mensagens em sistemas com checkpointing assíncrono**. São Paulo: DCC-IME, 1997. Dissertação de Mestrado.

[GOL97] GOLM, Michael. **Design and implementation of a meta architecture for java**. Nürnberg: Institut für Mathematische Maschinen und Datenverarbeitung der Friedrich-Alexander-Universität, Erlangen-Nürnberg, Diplomarbeit, Jan. 1997.

[HAE98] HAETINGER, Werner. **Troca dinâmica de versões de componentes de programas no modelo de objetos**. Porto Alegre: CPGCC da UFRGS, 1998. Dissertação de Mestrado.

[HOM74] HOMING, J.J. et al. **A program structure for error detection and recovery**. Heidelberg: Springer-Verlag, 1974. p. 177-193 (Lecture Notes in Computer Science, v. 16).

[HOR98] HORSTMANN, Cay S.; CORNELL, Gary. **Core Java 1.1 v.II – Advanced features**. California: Sunsoft Press, 1998.

[HUA95] HUANG, Y.; KINTALA, C. Software fault tolerance in the application layer. In: LYU, Michael (Ed.). **Software fault tolerance**. Bellcore: John Wiley and Sons Ltd., 1995.

[JAL94] JALOTE, Pankaj. **Fault-tolerance in distributed systems**. New Jersey; Prentice-Hall, 1994.

[JAV99] JAVASOFT. **Java platform documentation**. Disponível em http://www.javasoft.com/docs/index.html (dezembro de 1998).

[KOO87] KOO, R.; TOUEG, S. Checkpointing and rollback-recovery for distributed systems. **IEEE Trans. on Software Engeneering**, New York, v. SE-13, n. 1, p. 23-31, Jan. 1987.

[LEE78] LEE, P.A. A reconsideration of the recovery block scheme. **Computer Journal**, [S.l.], v. SE-21, n. 4, p. 306-310, 1978.

[LIS95] LISBOA, Maria Lúcia Blanck. **MOTF**: metaobjetos para tolerância a falhas. Porto Alegre: CPGCC da UFRGS, 1995.

[LIS96] LISBOA, Maria Lúcia Blanck; RUBIRA, Cecília M.F.; BUZATO, Luiz E. Arquitetura reflexiva para o desenvolvimento de software tolerante a falhas. In: SEMINÁRIO INTEGRADO DE SOFTWARE E HARDWARE, SIMISH, 23., 1996. **Anais...** Recife: [s.n.], 1996.

[LIS98] LISBOA, Maria Lúcia Blanck. Reflexão Computacional no Modelo de Objetos. In: SEMINFO, 8., 1998. **Anais...** Salvador: [s.n.], 1998.

[MAH96] MAHMOUD, Qusay H. **Sockets programming in Java**: a tutorial. Disponível por www em http://www.javaworld.com/javaworld/jw-12-1996/jw-12-sockets.html publicado em dezembro 1996 (15/01/1999).

[OBR97] O'BRIEN, Patrick. **Making java objects persistent**. Technical Report, Object Design, Inc., 1997. Disponível por WWW em http://www.sigs.com/publications/docs/java/9701/obrien.html (21/08/1998).

[PAV97] PAVAN, Willingthon. **O uso de blocos de recuperação em tolerância a falhas:** trabalho individual. Porto Alegre: CPGCC da UFRGS, dezembro de 1997.

[PLA96] PLANK, J.S. et al. M**emory exclusion:** optimizing the performance of checkpointing systems. Knoxville: University of Tennessee, 1996.

[POR90] PORTO, Ingrid E. S. Jansch; WEBER, Taisy S.; WEBER, Raul F. Fundamentos de Tolerância a Falhas. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, JAI, 9., 1990. **Anais...**. Vitória: [s.n.], 1990.

[RAN75] RANDELL, B. System structure for software fault tolerance. **IEEE Trans. Soft. Eng**., New York, v. SE-I, n. 2, p.220-232, 1975.

[RAN94] RANDELL, B.; XU, J. **Recovery blocks**. Newcastle: Newcastle upon Tyne, 1994 (Technical report series, n. 479).

[REB98] REBONATTO, Marcelo Trindade; PASQUALOTTI, Adriano; BRUSSO, Marcos José. **Estudo de um algoritmo de pontos de recuperação assíncrono**. Informação por correio eletrônico em rebonatto@upf.tche.br. (dezembro de 1998).

[SUN99] SUN MICROSYSTEMS. **Java Remote Method Invocation Specification.** Disponível por www em

http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html (20/03/99).

[WOL99] WOLLRATH, Ann; WALDO, Jim. **Java Tutorial –RMI.** Disponível por www em http://www.java.sun.com/docs/books/tutorial/rmi/index.html (15/01/1999).

[XU93] XU, Jian; NETZER, Robert H. B. Adaptative Independent Checkpointing for Reducing Rollback Propagation. 1993. In: IEEE SYMPOSIUM ON PARALLEL AND DISTRIBUTED PROCESSING, December 1993, Dallas, TX. Disponível por WWW em http://pig.postech.ac.kr/~clotho/fault.html. (29 Set. 1998).

[ZAW92] ZAWAENEPOEL, W; ELNOZAHY, E. N.; ZWAENEPOEL, Willy**. The performance of consistent checkpointing**. Houston: Rice University, Department of Computer Science, 1992.