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

# Implementação de Mecanismo de Sincronismo Virtual: Experiência com Java

por

ROBSON SOARES SILVA

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

Profa. Dra. Ingrid Eleonora Schreiber Jansch-Pôrto
Orientadora

Profa. Dra. Maria Lúcia Blanck Lisbôa
Co-Orientadora

Porto Alegre, março de 2002

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Soares Silva, Robson
Implementação de Mecanismo de Sincronismo Virtual: Experiência com Java / por Robson Soares Silva. – Porto Alegre: PPGC da UFRGS, 2002.
100p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2002. Orientadora: Jansch-Pôrto, Ingrid Eleonora Schreiber; Co-orientadora: Blanck Lisbôa, Maria Lúcia.
1. Tolerância a Falhas. 2. Sistemas Distribuídos. I. Jansch-Pôrto, Ingrid Eleonora Schreiber. II. Lisbôa, Maria Lúcia Blanck. III. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Jaime Evaldo Fernsterseifer
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

# Agradecimentos

Agradeço a Deus por me dar saúde e força de vontade para o desenvolvimento deste trabalho.

Agradeço à UNIDERP, Universidade para o Desenvolvimento do Estado e da Região do Pantanal, pelo apoio financeiro e pela disponibilização de equipamentos para o estudo e desenvolvimento das minhas pesquisas.

Agradeço aos meus amigos e companheiros de mestrado Ana Fernanda, Edilene, Emerson, Fernando, Gláucia, João Carlos, Luiz Alberto, Lincoln, Terezinha, Roberto e Paulo, que sempre me apoiaram e me incentivaram em todos os momentos. Agradeço à minha namorada Handria Cristina pelo incentivo e apoio para a finalização da dissertação.

Agradeço à minha co-orientadora Profa. Maria Lúcia Blanck Lisbôa, que sempre me atendeu com muita atenção e pelos ensinamentos passados.

Um agradecimento especial à minha orientadora Profa. Ingrid, pelos valiosos ensinamentos que me passou durante o desenvolvimento deste trabalho, pela grande pessoa e profissional que é, e pelo exemplo de trabalho, dedicação e responsabilidade.

E sem dúvida nenhuma, agradeço especialmente aos meus pais Alice e Waldemar, por estarem juntos comigo sempre, e pela compreensão e apoio em todos os momentos da minha vida.

# Sumário

# Lista de Figuras

# Resumo

Este trabalho relata as atividades de estudo, projeto e implementação de uma aplicação distribuída que explora mecanismos básicos empregados em comunicação de grupo. O estudo é focado no desenvolvimento e uso dos conceitos de sincronismo virtual e em resultados aplicáveis para tolerância a falhas.

O objetivo deste trabalho é o de demonstrar as repercussões práticas das principais características do modelo de sincronismo virtual no suporte à tolerância a falhas. São preceitos básicos os conceitos e primitivas de sistemas distribuídos utilizando troca de mensagens, bem como as alternativas de programação embasadas no conceito de grupos.

O resultado final corresponde a um sistema Cliente/Servidor, desenvolvido em Java RMI, para simular um sistema distribuído com visões de grupo atualizadas em função da ocorrência de eventos significativos na composição dos grupos (sincronismo virtual). O sistema apresenta tratamento a falhas para o colapso (*crash*) de processos, inclusive do servidor (coordenador do grupo), e permite a consulta a dados armazenados em diferentes servidores. Foi projetado e implementado em um ambiente Windows NT, com protocolo TCP/IP. O resultado final corresponde a um conjunto de classes que pode ser utilizado para o controle da composição de grupos (*membership*).

O aplicativo desenvolvido neste trabalho disponibiliza seis serviços, que são: inclusão de novos membros no grupo, onde as visões de todos os membros são atualizadas já com a identificação do novo membro; envio de mensagens em *multicast* aos membros participantes do grupo; envio de mensagens em *unicast* para um membro específico do grupo; permite a saída voluntária de membros do grupo, fazendo a atualização da visão a todos os membros do grupo; monitoramento de defeitos; e visualização dos membros participantes do grupo. Um destaque deve ser dado ao tratamento da suspeita de defeito do coordenador do grupo: se o mesmo sofrer um colapso, o membro mais antigo ativo é designado como o novo coordenador, e todos os membros do grupo são atualizados sobre a situação atual quanto à coordenação do grupo.

**Palavras-chave:** sistemas distribuídos, primitivas de comunicação de grupo, tolerância a falhas, sincronismo virtual.

**TITLE:** "IMPLEMENTATION OF A VIRTUAL SYNCHRONY MECHANISM: A JAVA EXPERIENCE"

# Abstract

This work reports the study, the design and implementation of a distributed application based in group communication, which uses the virtual synchrony execution model.

The goal of this work is to put into practice the main characteristics of the virtual synchrony model, and to apply the concepts and primitives of distributed systems based on message passing. Some alternatives offered to the programmer on the use of group communication principles are explored.

The final result corresponds to a Client/Server system, developed in Java RMI, to simulate a distributed system where group views are updated under adequate events (Virtual Synchrony). It presents a fault tolerant behavior under process crashes including the server, and allows querying the data stored in different servers. We have used Windows NT platform, with protocol TCP/IP, for the implementation. The final result corresponds to a set of classes that may be used in the implementation of the membership handling.

The application developed in this work offers six services, that are: inclusion of new members in the group, where the views in all the active members are updated with the new member identification, including the new member (brought up to date with the current vision); sending of multicast messages to the participant members of the group; sending of unicast messages for a specific member of the group; leaving the group, which includes the view update to all members in the group; monitoring failures; and visualizing data concerning the group participants. An interesting point is the handling of suspected failures related to the group coordinator. Under crash suspicion, the oldest active member in the group is made the new coordinator and all the members are updated with the new view and coordinator identification.

**Keywords:** distributed systems, group communication primitives, fault tolerance, virtual synchrony.

# 1 Introdução

Hoje em dia, além da profusão de equipamentos ligados por redes locais em ambientes fisicamente limitados, os preços baixos dos equipamentos e a utilização da Internet estão tornando cada vez mais comum entre os usuários o uso compartilhado dos recursos e capacidade de processamento de uma grande rede corporativa, caracterizado como computação distribuída.

Tradicionalmente as aplicações ficavam confinadas em uma única máquina tornando-se limitadas ao poder de computação local; se esta máquina não estivesse disponível, todo o sistema pararia de funcionar. Outro inconveniente é que este modelo de processamento e armazenamento centralizado não possui uma boa escalabilidade, pois quanto maior o número de usuários do sistema, menor o desempenho da máquina. Já uma aplicação distribuída pode executar localmente em uma máquina ou utilizar vários computadores para efetuar as tarefas planejadas, trabalhando cooperativamente. Tal aplicação possui vantagens como: aumento do desempenho e da capacidade de processamento, boa escalabilidade e tolerância a falhas (um computador pode parar o processamento isoladamente, mas os 'clientes' continuam a executar os seus processos normalmente). Em compensação, cria novos problemas como, por exemplo, a dificuldade de gerência das informações replicadas em diversas máquinas, e a capacidade de tratar a ocorrência de falhas de forma transparente aos usuários.

Assim, a proliferação dos sistemas distribuídos fez com que fosse necessária a revisão dos mecanismos de implementação de tolerância a falhas com vistas ao seu uso adequado (confiável) neste novo contexto. Um sistema digno de confiança é caracterizado por ser: disponível (pronto para ser usado quando necessário), confiável (fornece continuamente um serviço adequado), seguro (incapaz de causar danos ao seu ambiente) e protegido (no sentido de preservar a sua confidencialidade) [PRA 96]. O desenvolvimento de aplicações de alta confiabilidade na área de programação é referido sob a denominação genérica de tolerância a falhas em software [BIR 87].

A busca por alternativas de funcionamento de aplicações distribuídas tolerantes a falhas é bastante motivadora; novos desafios surgem à medida que se busca cumprir as funcionalidades atingindo melhores parâmetros de desempenho e atendendo modelos de falhas mais genéricos.

O objetivo da área de tolerância a falhas em sistemas distribuídos (ponto de vista da computação) é garantir que algumas propriedades, ou serviços, sejam preservados mesmo na presença de falhas em alguns componentes físicos do sistema [JAL94]; entretanto, fornecer este tipo de garantia exige um esforço significativo dos programadores.

O desenvolvimento de aplicações distribuídas robustas exige a inserção, no programa da aplicação, de uma série de mecanismos de tolerância a falhas e de cooperação entre processos. Usados para estruturar o sistema e implementar o controle de concorrência entre os seus componentes, estes mecanismos influenciam fortemente na determinação do desempenho, da confiabilidade e da disponibilidade do sistema. Entretanto, a inserção destes mecanismos extras para driblar os problemas oriundos de defeitos típicos na comunicação implica num grande esforço do

programador. Para auxiliar no desenvolvimento destas aplicações, alguns ambientes de computação incluem, em suas camadas inferiores, ferramentas e/ou serviços que tornam a programação mais facilitada [NUN 98]. Os serviços ou as ferramentas de comunicação de grupo constituem-se em exemplos de suporte à programação. A designação comunicação de grupo tem sido usada para caracterizar programação orientada a grupos com o emprego de ferramentas que forneçam suporte a comunicação *multicast* confiável [LIA 90, BIR 93].

O paradigma de orientação a objetos tem sido reconhecido como de interesse, por seu suporte para reutilização de software. Além disto, a computação distribuída, com ênfase a objetos distribuídos, poderá se tornar o paradigma de desenvolvimento dominante em um futuro não muito distante.

Por conseqüência, passaram a ser desenvolvidas interfaces orientadas a objetos, que incorporam os serviços de tolerância a falhas e comunicação de grupo (implementados nas camadas mais baixas), imperceptíveis ao usuário. Exemplos de ferramentas deste tipo incluem o Arjuna [SHR 95], o Garf [GUE 97a] e o Electra [REN 96].

A partir destas constatações, começou a ser investigada a implementação de objetos usando como referência estes modelos, com características de reuso, transparência para o usuário e principalmente a confiabilidade e disponibilidade de uso. Um modelo de implementação interessante do ponto de vista de tolerância a falhas é a reflexão computacional [LIS 85], devido ao fato de que a reflexão permite separar os aspectos funcionais da aplicação, daqueles que deveriam estar embutidos no ambiente de suporte e que não estão diretamente relacionados à aplicação (não-funcionais).

Segundo Elbson Quadros e Cecilia Rubira [QUA 97], o conceito de *frameworks* foi proposto como alternativa para alcançar reutilização de *software* em grande escala. Ainda de acordo com estes autores, tem-se observado que a reutilização de classes ou mesmo de bibliotecas de classes (baseadas em herança) não provê um grau satisfatório de reutilização. Segundo eles, classes e objetos possuem granulosidade pequena para alcançar o nível de reutilização desejado. Entretanto, estas dificuldades serão encontradas na medida em que não houver suporte adequado para a integração de classes construídas através de uma biblioteca, ou de uma ferramenta que faça a inserção automática de classes em aplicações em desenvolvimento. Por outro lado, o desenvolvimento de classes isoladas pode ser reconhecido como uma boa forma de fazer o desenvolvimento detalhado e restrito aos aspectos de interesse.

Babaoglu [BAB 98] reporta uma investigação realizada na Universidade de Bologna, que é a implementação em Java de uma tecnologia de grupos com objetos distribuídos. O Jgroup, como foi denominado o projeto, suporta o desenvolvimento de serviços confiáveis e altamente disponíveis, com base em replicação. São consideradas falhas que resultam em partições do sistema, oferecendo opções para o tratamento referente à reunião das partições após as falhas. Este enfoque é o grande diferencial do projeto com relação à maior parte dos demais sistemas de comunicação de grupo atualmente existentes. Foi proposta a integração do serviço de comunicação em grupo resultante à tecnologia Java RMI, com o objetivo de projetar grupos de objetos, cada um composto de uma coleção de objetos remotos replicados. Segundo os autores, os clientes poderão acessar um grupo de objetos por invocação remota dos seus métodos com o mesmo nível de simplicidade de objetos únicos (não

replicados). Um protótipo da implementação do Jgroup está disponível para *download,* na página do grupo de Bologna, desde o final de 1999 [JGR 99].

O projeto coordenado por Babaoglu compreende a implementação de vários serviços tais como: mecanismos para o controle de sincronização de operações - sincronismo virtual [BIR 87, BIR 96, FRI 95, GUO 96], controle de replicação de dados [GUE 96] e gerência de grupos [LIA 90]. Do ponto de vista do presente trabalho, o ponto de interesse é que aquele projeto também está sendo desenvolvido sob forma de uma biblioteca de classes em Java, com o intuito de comprovar a funcionalidade de idéias propostas. Embora ele só tenha sido conhecido após o início do presente trabalho (em fase de coleta bibliográfica), por estar integrado a um grande projeto europeu chamado *Relacs Project* [REL 99], ele auxilia na defesa dos objetivos aqui propostos.

Atualmente a linguagem de programação Java vem despertando grande interesse no mundo científico e comercial, devido aos seus recursos e facilidades para programação de aplicativos distribuídos. Java é uma linguagem puramente orientada a objetos, pois o menor programa possível deve conter pelo menos uma classe. A estrutura de dados ou função não existe, ou não está acessível durante a execução, exceto como um elemento para definição de uma determinada classe [FAR 98].

Java foi desenvolvida pela Sun Microsystems. Modelada a partir de C++, a linguagem Java foi projetada para ser compacta, simples e portável a todas as plataformas e sistemas operacionais, tanto o código fonte como os binários. Esta portabilidade é obtida por ser interpretada, ou seja, o compilador gera um código independente de máquina (*byte-code*). No momento da execução, este *byte-code* é interpretado por uma máquina virtual instalada na mesma. Para portar Java para uma arquitetura de *hardware*/ sistema operacional específica, basta instalar a máquina virtual (interpretador). Além de ser integrada à Internet, Java também é uma excelente linguagem para desenvolvimento de aplicações em geral. Também fornece suporte ao desenvolvimento de *software* em larga escala [CES 98, DAM 96]. Java RMI permite a comunicação de objetos, através da chamada de métodos, em máquinas diferentes em uma aplicação distribuída.

Objetivou-se, através desta dissertação, estudar e implementar mecanismos de tolerância a falhas que auxiliem no desenvolvimento de aplicações distribuídas, e que assegurem o funcionamento adequado, mesmo diante da ocorrência de situações de falhas. O tema específico de estudo escolhido voltou-se para a implementação do mecanismo de sincronismo virtual e suas variações. O sincronismo virtual é definido a partir de um conjunto de regras funcionais que será detalhadamente apresentado no capítulo 3. A implementação deve assegurar a manutenção de todos os princípios funcionais.

Além dos objetivos gerais de fornecer facilidades para a programação de aplicações distribuídas tolerantes a falhas e de utilizar os recursos de orientação a objetos para suportar o desenvolvimento posterior de novas aplicações, foram definidos como objetivos específicos deste trabalho:

- dispor de meios eficientes e transparentes para aplicações tolerantes a falhas em sistemas distribuídos tais como o controle de réplicas, a gerência de um conjunto dinâmico de réplicas, a identificação de elementos falhos e sua remoção do grupo;

- estudo dos mecanismos de sincronização de eventos entre grupos de processos e objetos, com ênfase nos princípios de sincronismo virtual e suas variações;

- a modelagem e a implementação de mecanismos de sincronização virtual, através de uma biblioteca de classes, que possa ser empregada futuramente como suporte à programação de aplicações distribuídas.

Para atender aos objetivos propostos, foi efetuado o estudo dos conceitos e das propriedades do sincronismo virtual, além de suas repercussões práticas. Foram criados ou adaptados vários algoritmos para controle de *membership* e detecção de defeitos. Desenvolveu-se um aplicativo em Java com recursos do RMI, que disponibiliza serviços de controle de *membership* e de sincronismo virtual, disponibilizando assim classes e código fonte que podem ser utilizadas para estudo e trabalhos futuros sobre comunicação de grupo.

Visando atender a objetivos de portabilidade e modularidade, ficou logo definido o modelo de objetos para a definição dos componentes e a linguagem de programação Java. Desempenho não foi priorizado, tendo em vista a conotação inicial de "protótipo".

Assim, nesta dissertação, as atividades de projeto e implementação de elementos de suporte à programação têm como ambiente um sistema distribuído típico, sem compartilhamento de memória entre os diferentes nós, e cuja comunicação é baseada no conceito de troca de mensagens. O modelo de execução está baseado nos conceitos e primitivas de sincronismo virtual conforme proposta de Birman [BIR 96]. O sistema prevê aplicações tais como a gerência de dados replicados em diferentes membros (nós) de um determinado grupo ou a reconfiguração do sistema a partir da detecção de falhas nos nós. Por simplicidade, a premissa de execução de um processo por nó está sendo empregada. O modelo de falhas por colapso (*crash*) também foi adotado. Por hipótese, não há falhas nos *links* e não se permite a sobreposição de grupos.

O presente texto corresponde aos aspectos estudados no desenvolvimento deste trabalho: partindo do modelo inicial de sistemas distribuídos estuda-se também os conceitos de comunicação de grupo e os princípios da comunicação *multicast* confiável. Visando o uso de redundância em um ambiente distribuído, foram previstas formas de resolução: da gerência de grupos e seus objetivos, as propriedades e o tratamento de sincronismo virtual e suas particularidades, além de como controlar os mecanismos de replicação. A identificação das técnicas de replicação existentes faz parte deste estudo.

O trabalho que segue está dividido em seis capítulos cobrindo desde os conceitos estudados até a apresentação de aspectos da linguagem de programação utilizada, e a implementação propriamente dita do módulo resultante deste estudo. A ênfase maior está no mecanismo de sincronismo virtual e suas peculiaridades.

O capítulo 2 descreve os conceitos sobre comunicação de grupo a partir do modelo de sistemas distribuídos. São caracterizados os grupos, a forma de comunicação utilizada (*multicast* confiável), e a idéia de sincronismo de operações (visão).

No capítulo 3, serão apresentados os conceitos para realização do controle de membros do grupo (*membership*), o que é, como fazer e o que se precisa

para a sua execução. São mostrados também os modelos de execução (sincronismo virtual e suas variações), detecção de defeitos e os modelos de monitoramento.

No capítulo 4, serão apresentados os recursos da programação distribuída com enfoque à linguagem de programação Java e os conceitos sobre RMI.

No capítulo 5, aborda-se o sistema proposto propriamente dito. Mostra-se a estrutura geral e os módulos componentes do protótipo desenvolvido, os algoritmos utilizados e os testes de situações críticas do protótipo.

Finalmente, no capítulo 6, são apresentadas as considerações finais do trabalho, identificando-se as suas contribuições e perspectivas de trabalhos futuros.

A essência da contribuição deste trabalho está efetivamente descrita nos capítulos 5 e 6. Os capítulos 2, 3 e 4 reúnem os conceitos necessários à sua realização, que precisaram ser estudados pelo autor para desenvolver este trabalho. A escrita aqui funciona como uma forma de organizar idéias e demonstrar a compreensão dos aspectos estudados. Para o leitor que conhece a comunicação de grupo, sincronismo virtual e controle de replicação, sua leitura é dispensável.

# 2 Conceitos Básicos de Comunicação de Grupo

Neste capítulo, conceitua-se a idéia de trabalhar com os sistemas distribuídos organizados sob forma de grupos, apresentando-se em seguida a conceituação relacionada ao tema: os modelos de grupo existentes, as formas de comunicação utilizadas (*multicast* confiável, *multicast* uniforme), e a idéia básica do sincronismo de operações (visão).

Dentro do sentido aqui empregado, um grupo é composto a partir de um sistema distribuído que permita a comunicação entre os seus membros. O termo distribuído é usado para descrever um sistema computacional que possui as seguintes características: consiste de vários computadores que não compartilham memória nem relógio e os computadores comunicam-se uns com os outros através de troca de mensagens, usando canais de comunicação [SIN 97, SIN 97a]. Segundo [BIR 96], um sistema de computação distribuída é um conjunto de programas de computadores, sendo executados em um ou mais computadores, e coordenando ações por troca de mensagens. Uma rede de computadores é uma coleção de computadores interconectados por hardware que suporta diretamente troca de mensagens. Embora haja outras possibilidades, o modelo distribuído assim definido é o considerado no presente trabalho. A maioria dos sistemas distribuídos opera sobre redes de computadores, contudo isto não é obrigatório: pode-se construir sistemas de computação distribuída em que as informações fluem diretamente entre os diferentes computadores.

Têm sido usadas principalmente duas abordagens para um sistema distribuído: a definição dos componentes físicos do sistema, e a definição do ponto de vista de processamento ou computação. Chamam-se de modelo físico e de modelo lógico, respectivamente. A visão computacional é importante: nela, os serviços são definidos dentro de uma determinada perspectiva e confiabilidade. O modelo físico da rede é importante: através do mesmo, consegue-se identificar como a computação é efetuada na rede física, e os componentes da rede que são passíveis de falha [JAL 94]. Freqüentemente, a meta da tolerância a falhas em sistemas distribuídos é assegurar que algumas propriedades ou serviços, no modelo lógico, sejam preservados, apesar da ocorrência de falhas de alguns componentes do sistema físico.

A confiabilidade em sistemas distribuídos é um termo muito geral, e pode ter muitos significados (sentidos), incluindo: tolerância a falhas, alta disponibilidade, disponibilidade (utilização) contínua, recuperabilidade, consistência, segurança, privacidade, especificação correta, implementação correta, desempenho prognosticado.

Implícita em muitos conceitos de confiabilidade em sistemas distribuídos [SIN 97, SIN 97a], existem questões de tolerância a falhas. Falha também pode ter muitos significados: falhas de parada, falhas temporárias, falhas de omissão no envio (*send-omission*), falhas de omissão no recebimento (*receive-omission*), falhas de rede, falhas de particionamento de redes, falhas bizantinas, falhas do mundo real. A seguir, são apresentados os componentes de sistemas confiáveis de computação distribuída e suas características.

Sistemas confiáveis de computação distribuída podem ser reunidos a partir de blocos básicos de construção. É comum pensar em sistemas distribuídos como operações sobre um conjunto de camadas de serviços de rede. Cada camada

corresponde a uma abstração de *software* ou aspecto de *hardware*, e pode ser implementada dentro de programas de aplicação, em bibliotecas de procedimentos empregadas na construção de programas, em sistemas operacionais, ou igualmente em dispositivos de comunicação de *hardware*. Como um exemplo, tem-se os diferentes níveis/camadas propostas pela *International Organization for Standardization* (ISO), referentes ao modelo *Open System Interconnection* (OSI) [BIR 96]:

- aplicação: o programa de aplicação propriamente dito até o ponto onde ele realiza as operações de comunicação;
- apresentação: *software* usado para codificar os dados da aplicação em mensagens e decodificá-los na recepção;
- sessão: a lógica associada que garante as propriedades fim-a-fim (*end-to-end*) tais como a confiabilidade;
- transporte: *software* destinado a fragmentar mensagens grandes em pacotes pequenos;
- rede: funcionalidade de roteamento e de controle de fluxo, usualmente limitada a pacotes pequenos ou pacotes de tamanho fixo;
- *link* de dados: o protocolo responsável pelo envio e pela recepção de pacotes;
- físico: o protocolo usado para representar pacotes na linha (tecnologia de hardware).

Para que uma computação distribuída seja confiável, deve-se observar entre outros, o grau de tolerância a falhas do *hardware* da mesma. Para isso, mostra-se, a seguir, como são definidos os modelos de rede e suas principais características.

A rede física de um sistema distribuído consiste de muitos computadores (freqüentemente referenciados como nós), que podem estar geograficamente em localizações distantes, mas que estão conectados por uma rede de comunicação. Pode-se dizer que um sistema distribuído consiste de vários nós. Cada nó consiste de um processador, que tem memória volátil inacessível aos outros nós, e um relógio privado, que governa a execução das instruções neste processador. Todos os nós são autônomos, e podem comunicar-se entre si. Cada nó possui uma interface para rede através da qual o processador é conectado. Dois nós quaisquer em um sistema de comunicação trocam mensagens para se comunicar. As propriedades-chave do sistema distribuído são a separação geográfica e a natureza autônoma dos vários nós [JAL 94].

Sistemas distribuídos são diferentes de sistemas paralelos, onde os nós apresentam forte acoplamento, ou seja, eles não atuam com autonomia de processamento. Em contraste, a computação dos elementos de um sistema distribuído é fracamente acoplada. Outro fator comum nos sistemas distribuídos é a ausência de memória compartilhada entre os diferentes nós. Com isso, não há memória no sistema que possa ser acessível de forma generalizada. A memória local pertence a um nó e somente os componentes do nó tem acesso a ela. Nos sistemas paralelos e num subconjunto específico de sistemas distribuídos que possuem memória compartilhada, diferentes nós podem comunicar-se diretamente através da memória compartilhada[1]. Finalmente, existe um *software* que governa a seqüência de instruções que podem ser

[1] Os sistemas distribuídos com memória compartilhada (*Distributed Shared Memory* – DSM) têm restrições diversas dos aqui considerados, necessitando alterações nos algoritmos e hipóteses empregadas.

executadas no nó. Com isso, os componentes básicos de um sistema distribuído são: processadores, comunicação de rede, relógios, *software* e o armazenamento não-volátil. Em um sistema distribuído, consideram-se estes componentes como atômicos, e raramente considera-se a estrutura física dos mesmos.

Freqüentemente, um sistema distribuído é modelado apenas a partir dos seus nós e da comunicação como componentes básicos, sem considerar a estrutura interna dos nós. Neste modelo, a falha do nó e a falha da comunicação de rede são os principais elementos representativos dos componentes falhos.

Em uma rede ponto a ponto, a estrutura de comunicação consiste de um conjunto de conexões, cada uma delas unindo dois nós através de suas respectivas interfaces de rede. Neste modelo, um sistema distribuído pode ser representado por um grafo, onde os nós da rede são igualmente referenciados como nós do grafo representativo do sistema, e suas arestas representam as conexões da comunicação do sistema. A maneira com que as diferentes conexões são organizadas entre os diversos nós é chamada de topologia de rede.

O envio de mensagens em uma rede ponto-a-ponto requer protocolos de comunicação. Protocolos são necessários à adequação da natureza autônoma dos nós e a separação geográfica entre eles. Uma vez que os nós podem ficar distantes uns dos outros, os dados enviados através de um *link* entre dois nós podem ser perdidos na transmissão. E uma vez que os nós não compartilham relógio (*clock*) e podem trabalhar em diferentes velocidades, dentro de um protocolo, um nó remetente pode sobrecarregar o nó recebedor. Muitos protocolos de comunicação, tais como TCP/IP e UDP, têm sido propostos para garantir uma comunicação confiável[2] entre os nós de uma rede. Tipicamente, há partes do protocolo sendo executadas no *hardware* da interface da rede, e no *software* do sistema [JAL 94].

No modelo totalmente conectado (conforme diagrama da figura 2.1, onde são representados cinco processos), existe um canal de comunicação entre cada par de processos que compõem o sistema. Este modelo é viável apenas em redes muito pequenas, devido ao seu custo. Entretanto, este é um modelo ideal do ponto de vista da conectividade devido à sua eficiência na comunicação entre processos.

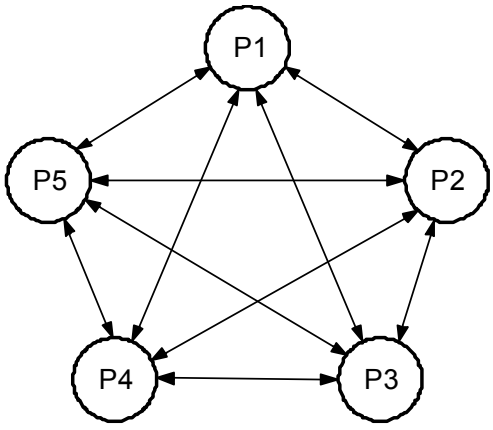


FIGURA 2.1- Modelo totalmente conectado

[2] O termo confiável é empregado aqui em sentido amplo, empregado na área de comunicação de dados, podendo se referir apenas ao descarte de mensagens corrompidas, por exemplo. Não tem o sentido formal empregado no capítulo 5 do livro do Mullender [MUL96], com entrega garantida de mensagens válidas.

O modelo em barra (figura 2.2), por outro lado, por ter um custo de implementação (construção) bem menor que aquele da rede totalmente conectada; é um modelo real para muitas redes existentes em instituições públicas e privadas. A rede onde o protótipo foi criado e testado usa um modelo em barra.

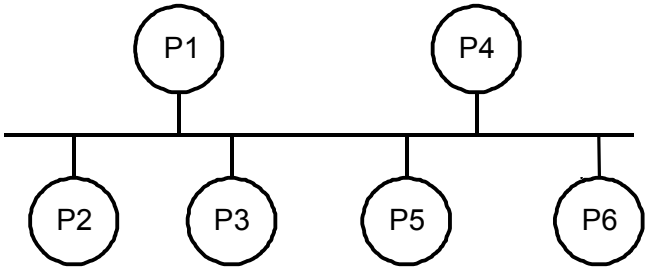


FIGURA 2.2 - Modelo em barra

Um grupo é um conjunto de processos ou objetos que agem juntos, de maneira especificada pelo sistema ou por um usuário. Cada participante de um grupo é chamado membro ou elemento. Cada objeto (estação, processo) possui sua própria memória local e não tem acesso direto aos dados dos demais. Em decorrência disto, processos e objetos trocam informações através de mensagens, para realizar a requisição de serviços. Processos ou objetos clientes requisitam serviços para os servidores. A cada requisição, o cliente aguarda por uma resposta, enquanto o servidor realiza o serviço [PAS 99].

Segundo citado por Leite, Amaral e Pasin, Birman define que o conceito de grupos permite que conjuntos de objetos sejam tratados como uma simples abstração, facilitando o desenvolvimento de aplicações confiáveis [PAS 98]. O modelo de grupos precisa garantir que todos os objetos pertencentes a um grupo recebam e processem as mensagens na mesma ordem, quando isto for necessário à aplicação.

Em um sistema de arquivos replicados, os servidores que contêm as mesmas réplicas de um arquivo constituem um grupo de replicação, ou simplesmente grupo. Um objeto cliente que deseja comunicar uma operação para o grupo, ao invés de enviar uma mensagem individual para cada servidor, pode usar uma primitiva de comunicação de grupo. O cliente não precisa saber quais servidores têm cópias de um determinado arquivo, nem se preocupar em nomear cada servidor independentemente [GAR 91].

Segundo Tanenbaum *apud* Nunes [NUN 98], para modelar uma aplicação, não é suficiente confinar os servidores em um grupo. É preciso estruturar este grupo. A estrutura interna de um grupo pode assumir as formas:

- grupos fechados e grupos abertos – quanto à relação entre membros internos e objetos externos ao grupo;
- grupos hierárquicos e não-hierárquicos – quanto à relação entre os membros de um grupo;
- grupos estáticos e grupos dinâmicos – quanto à permanência ou não de objetos no grupo; as quais são detalhadas na seqüência, com base principalmente nas informações extraídas de [TAN 95], [GUE 97] e [PAS 98].

Grupos fechados e grupos abertos: esta estruturação de grupos é relativa à troca de mensagens com elementos externos e especifica quais objetos podem enviar mensagens. Em grupos fechados, somente podem enviar mensagens para o grupo os objetos que são membros do mesmo. Objetos que não pertencem ao grupo não podem enviar mensagens para o grupo, embora possam fazê-lo para membros individuais do grupo. Contrastando, na estruturação através de grupos abertos não há esta restrição, pois qualquer objeto do sistema pode enviar mensagens para o grupo. Grupos fechados são naturalmente adequados ao processamento paralelo; por exemplo: um conjunto de objetos trabalhando juntos em um programa de jogo de xadrez, onde todos os objetos possuem seus próprios objetivos e não interagem com o mundo exterior. Entretanto, se a utilização do modelo de grupos for com o propósito de suportar servidores replicados, podem ser usados grupos abertos para permitir que objetos não-membros do grupo enviem mensagens para o grupo, oferecendo transparência, do ponto de vista externo, às situações de falha de membros do grupo [GUE 97].

Estes exemplos, entretanto, não definem uma pré-associação obrigatória, por exemplo, entre grupos abertos/uso de replicação. Embora o sistema Ensemble ofereça apenas suporte para grupos fechados, um sistema de réplicas ativas pode ser implementado reunindo clientes e servidores em grupo [PAS 98].

Grupos hierárquicos e grupos não-hierárquicos: esta estruturação de grupos trata da relação entre os seus membros. Em alguns grupos, há igualdade entre os membros – nenhum é superior – e todas as decisões são tomadas coletivamente. Esta estruturação representa grupos não-hierárquicos.

Em grupos hierárquicos, os membros seguem uma ordem ou hierarquia. Nestes grupos pode haver um coordenador para gerenciar as tarefas dos demais membros do grupo ou uma estruturação em níveis, onde membros pertencentes a determinados níveis têm prioridades sobre membros de outros níveis. O protótipo implementado nesta dissertação utiliza os conceitos de grupos hierárquicos, pois existe sempre uma máquina que funciona como coordenador do grupo.

Quando a aplicação utilizar grupos hierárquicos, é permitida uma abordagem na qual o coordenador do grupo está capacitado a atender o pedido do cliente, ou a escolher outro membro para atender o cliente.

Tanto a estrutura de grupos hierárquicos quanto a de não-hierárquicos possuem suas vantagens e desvantagens. Um grupo não-hierárquico é simétrico e não possui um ponto único de falha. Quando um dos membros do grupo está inoperante, o funcionamento do grupo não é interrompido, já que os demais são capazes de desempenhar a função do membro falho. Uma desvantagem de grupos não-hierárquicos, entretanto, é o aumento de complexidade nos algoritmos para tomada de decisões.

Em grupos hierárquicos, a perda do membro coordenador pode interromper o serviço do grupo ou gerar atraso para a tomada de decisões, se houver uma implementação tolerante a falhas. Por outro lado, enquanto estiver operacional, o coordenador é capaz de tomar decisões por si mesmo, repassando-as aos demais membros, tornando mais simples o controle destas ações.

Grupos estáticos e grupos dinâmicos: um outro aspecto relacionado com a estruturação dos grupos é a variação da composição de membros. O conjunto de membros de um grupo pode variar ou ser constante.

Em grupos estáticos, os membros não podem deixar o grupo e nem outros novos podem entrar no grupo. O número de elementos, durante sua existência, é fixo. Em um grupo dinâmico, entretanto, a constituição do grupo pode ser alterada, durante a vida útil do grupo, pela entrada e saída de membros.

Uma desvantagem para grupos dinâmicos, é a necessidade de identificar quais objetos fazem parte do grupo em dado instante, já que os membros destes grupos podem estar em constante mudança dentro do sistema distribuído. Esta dinâmica é embutida no conceito de visão do grupo. Visão do grupo é uma representação de quais elementos são membros do grupo em determinado instante. A visão do grupo é modificada sempre que um elementoentra no grupo, através de uma operação *join*, ou que um membro abandona o grupo, através de uma operação *leave*, de modo voluntário ou involuntário; este último modo pode ser devido à ocorrência de falha.

Para finalizar, um grupo dinâmico é bem mais flexível no uso do que um grupo estático e também fornece mais facilidades ao programador de aplicação. Entretanto, a complexidade na implementação de grupos dinâmicos é maior que a implementação de grupos estáticos.

Grupos dinâmicos podem ser implementados com auxílio de um servidor de grupos. O servidor de grupos gerencia todos os envolvidos mantendo uma base de dados completa de todos os grupos e de seus membros. Este método é simples, eficiente e fácil de ser implementado. A desvantagem é o fato de possuir um único ponto de falha, se não for previsto tratamento para este tipo de evento. Se o servidor de grupos tornar-se inoperante, não haverá como gerenciar os grupos. Quando o servidor se recuperar, os grupos precisam ser recompostos. Para controlar esta desvantagem, uma outra metodologia possível seria gerenciar os grupos de maneira distribuída que, apesar de ocasionar atrasos, tem a vantagem de não possuir um ponto único de falha. Ainda uma opção intermediária, mais simples, é a replicação do servidor [CAR 97].

Em um sistema, em geral, os grupos são entidades que apresentam comportamento dinâmico: novos grupos podem ser construídos e grupos antigos podem ser destruídos. Portanto, este sistema precisa prover métodos para construir e destruir grupos, bem como para lidar com as implicações das possíveis sobreposições de grupos.

## 2.1 Mecanismos de comunicação de grupos

A propriedade fundamental dos grupos é que, quando uma mensagem é enviada para o grupo, todos os membros deste grupo devem recebê-la, em uma forma de comunicação do tipo um-para-muitos. A comunicação um-para-muitos pode ser de dois tipos [PAS 98]:

a) *multicast* – quando uma estação envia uma mensagem para estações específicas;

b) *broadcast* – quando uma estação envia uma mensagem para todas as estações, sem a possibilidade de indicar destinatários.

Estes dois tipos de comunicação contrastam com a comunicação ponto-a-ponto ou *unicast*, onde um objeto envia uma mensagem para outro único objeto especificado.

A implementação das primitivas de comunicação de grupo depende fundamentalmente da estrutura da rede existente para *broadcast* ou *multicast*. Caso o sistema operacional não possua esta capacidade, a aplicação ou o próprio sistema operacional deverá implementar a difusão através de múltiplos envios de uma mensagem para cada nó destino (múltiplos envios do tipo *unicast* ou um-para-um).

Do ponto de vista de um *broadcast*, a existência de suporte real para esta função permite que uma rede em barra (figura 2.2) tenha, quanto ao tempo de transmissão de uma mensagem, o mesmo comportamento que uma rede totalmente conectada (figura 2.1). Se a rede permitir o uso de *multicast* por *hardware*, irá facilitar a tarefa de comunicação de grupo. Nestas redes é possível a criação de endereços especiais que podem ser vistos por várias estações. Quando uma mensagem é enviada para um destes endereços, é automaticamente enviada a todas as estações com acesso a este endereço. Nesse ambiente, a implementação dos grupos torna-se simples, bastando atribuir a cada grupo um endereço especial para a comunicação.

Em redes que não implementam diretamente *multicast*, *broadcast* pode ser usado na comunicação dos grupos, apesar de ser menos eficiente que *multicast*. Usar *broadcast* é mais oneroso, pois todas as estações da rede receberão a mensagem, mesmo aquelas que não sejam destinatários próprios, obrigando o *software* destas estações a descartar mensagens. Além disso, usar *broadcast* corrompe a semântica de segurança (mensagens do grupo podem ser "vistas" por estações que não fazem parte do grupo).

Por último, mesmo que a rede não disponha de facilidades de *multicast* e nem de *broadcast*, é possível implementar grupos fazendo com que o transmissor envie – através de múltiplas operações de *unicast* – uma mensagem para cada um dos membros do grupo. Supondo que um grupo possua *n* membros, então são necessários *n* envios de mensagens em vez de apenas um, como no caso do *multicast* ou *broadcast*, além de outros cuidados (a mensagem é uma só, semanticamente, apesar da multiplicidade de destinatários).

Cechin [CEC 98] analisou a relação entre a topologia e os diferentes tipos de transmissão de mensagens, tendo em vista seu uso em ambientes de recuperação. Nos próximos parágrafos, são reproduzidas aquelas idéias, adequando-as ao contexto de comunicação.

Pode-se perceber que, no caso da transmissão de uma mensagem de um nó para vários outros, usando *unicast*, o custo da transmissão, avaliado através do tempo empregado para sua realização, será particular a cada topologia de rede. No caso da rede totalmente conectada, este custo é aproximadamente o mesmo que o custo de transmissão para um único nó-destino, ou seja, uma unidade de tempo para enviar a mensagem a todos os nós-destino, já que a rede suporta mensagens simultâneas, mas o transmissor precisa coordenar os destinos diferentes (afeta a fase de empacotamento). Por outro lado, no caso de uma rede conectada em barra, este custo sobe para um valor proporcional ao número de mensagens. Este aumento deve-se à necessidade da serialização das mensagens no canal.

Do ponto de vista do custo de comunicação imposto pelos limites de conexão, os dois modelos podem ser ditos opostos: o modelo totalmente conectado impõe limites mínimos ao envio de mensagens simultâneas; no modelo em barra, o envio das mensagens é limitado a uma por vez.

Usar *unicast* possui as desvantagens de aumentar a quantidade de mensagens na rede e de exigir a gerência, pelo remetente, da lista dos possíveis destinatários. Apesar de ser menos eficiente, ainda assim é possível utilizar este esquema para a implementação de grupos, sobretudo se o grupo possuir poucos membros [PAS 98].

## 2.2 Comunicação *multicast* confiável e ordenada

Um grupo de *multicast* é uma coleção de processos identificados como destino de uma dada (ou de uma mesma seqüência de) mensagem, ou seja, quando uma mensagem for enviada em *multicast* ao grupo, ela destina-se a todos os membros do grupo. Estas mensagens podem ser originadas de um ou mais nós e o processo destino pode ser executado em um ou mais nós, não necessariamente distintos [GAR 91]. Cada mensagem de origem é endereçada para o grupo de *multicast* (que, neste caso, se opõe a processos ou nós/endereços individuais). O protocolo de *multicast* deve garantir que as mensagens serão transmitidas para os destinatários apropriados. Ainda a partir de Garcia-Molina [GAR 91], Cheriton e Deering estudaram o uso de *multicast* em redes e alegam que um *multicast* eficiente facilitado é necessário, porque o *broadcast* não é genericamente proveitoso em termos de facilidade, pois existem poucasrazões para comunicar com todos os nós. Segundo citado por Garcia-Molina, Gray concorda afirmando que a escalabilidade de *multicast* é utilizada por muitas redes de trabalho de larga escala e preferida pelos profissionais.

Para algumas aplicações, é necessário que o protocolo de *multicast* forneça garantias quanto à ordem com que as mensagens serão entregues para os processos-destino. As propriedades são usualmente as seguintes, organizadas pelo aumento de complexidade [GAR 91]:

a) ordenamento de origem simples: se as mensagens *m1* e *m2* foram originadas a partir de um mesmo nó, e se elas são endereçadas para um mesmo grupo *multicast*, então todos os destinatários recebe-las-ão na mesma ordem relativa.

b) ordenamento de origens múltiplas: se as mensagens *m1* e *m2* são endereçadas para um mesmo grupo *multicast*, então todos os destinatários recebe-las-ão na mesma ordem relativa (igualmente se elas vierem de diferentes origens).

c) ordenamento de grupos múltiplos: se as mensagens *m1* e *m2* são entregues para dois processos, elas são entregues na mesma ordem relativa (até mesmo se elas vierem de diferentes origens e forem endereçadas para destinatários diferentes, porém com sobreposição dos grupos de *multicast*).

Nem todas aplicações requerem todas estas propriedades. Mas existem aplicações onde o processamento de mensagens sem controle sobre a ordem destas operações conduzirá para problemas de inconsistência e até mesmo paralisação completa (*deadlock*). Para ilustrar esse problema, pode-se supor um banco com dois computadores principais. Cada computador possui uma cópia do banco de dados completo da instituição e processará todas as transações dos escritórios das filiais (a segunda máquina é necessária para recuperar um possível problema). Os dois computadores principais constituem um grupo de *multicast*, e cada filial do escritório é um local (*site*) potencial de origem. As transações deveriam ser executadas na mesma

ordem (propriedade b), nos computadores principais, senão a base de dados corre o risco de ficar inconsistente.

Considere-se, por exemplo, duas operações financeiras, tais como um depósito e uma retirada sobre uma mesma conta, onde o saldo atual incluindo o valor do depósito permitirá a conta ter o saldo suficiente para a futura retirada. Se a retirada for feita primeiro, pode ocorrer uma inconsistência no saldo da conta: se o valor a ser retirado depender do depósito, que já deveria ter sido efetuado, poderá gerar inclusive uma penalidade para o cliente. Com o depósito efetuado primeiro, nenhuma penalidade será incumbida, e o resultado do balanço final da conta será adequando ao cliente.

Neste mesmo exemplo bancário, pode-se considerar agora um segundo grupo de *multicast* para distribuir uma nova atualização de um novo *software* ou tabelas do sistema (e.g. definindo penalidades incidentes ao ultrapassar limites da conta). Este segundo grupo inclui dois computadores principais, mas em adição outras máquinas de desenvolvimento. Igualmente, embora os dois grupos de *multicast* estejam separados, é importante processar todas as informações na mesma ordem nas máquinas que fazem parte da intersecção dos grupos (propriedade c).

Estes exemplos ilustram porque as propriedades de ordenação para *multicast* são importantes.

Na comunicação através de *multicast,* o destino da mensagem corresponde a todos os membros do grupo; da mesma forma como outras modalidades de comunicação ela é suscetível a falhas de comunicação e de estações da rede. Isto pode ser um problema para algumas aplicações; assim, *multicast* confiável é um bloco de construção básico para aplicações tolerantes a falhas. Informalmente, o *multicast* confiável requer que todos os processos corretos no grupo de *multicast* entreguem ao processamento ou para as aplicações o mesmo conjunto de mensagens, que este conjunto inclua todas as mensagens em *multicast* para processos corretos do grupo, e não perca mensagens que são transmitidas [KAR 97].

Quando um grupo de processos interage com o mundo externo, necessita garantir a consistência entre os eventos antes de atualizá-los. Informalmente, a entrega confiável de uma mensagem a qualquer membro do grupo deve garantir a entrega da mesma a todos os membros ativos do grupo.

Um exemplo de entrega não confiável de mensagens é dado na figura 2.3. O processo *p1* envia uma mensagem *m1* em *multicast* para o grupo e então falha. A mensagem *m1* é recebida e entregue somente pelo processo *p2*, sendo que a mesma perde-se na rede, no caminho para *p3* e *p4*. Se, por hipótese, o processo *p2* também falhar, então pode ocorrer que a mensagem *m1* nunca seja recuperada, isto é, a mensagem *m1* pode nunca ser entregue para os membros não falhos do grupo. Se o processo *p2* executar qualquer ação externa ao grupo, de tal modo que responda para o cliente ou escreva em disco, antes de sua falha e depois de ter entregue *m1*, então isto produzirá uma inconsistência no grupo.

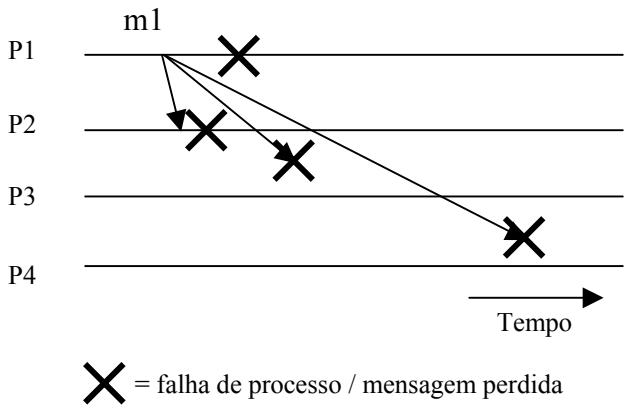


FIGURA 2.3 – Entrega não confiável de mensagens [KAR 97]

Segundo Hadzilacos e Toueg [HAD 94], a especificação do *multicast* confiável é uma generalização da especificação de *broadcast* confiável: se, o grupo corresponde ao conjunto de todos os processos, as propriedades de *multicast* confiável tornam-se equivalentes às de *broadcast* confiável. A grande diferença com relação ao *broadcast* confiável é que no *multicast* confiável somente processos pertencentes ao grupo podem entregar *m*. Assim, o *multicast* confiável é definido em termos de primitivas de entrega e *multicast* que satisfazem as seguintes propriedades [HAD 93]:

- Validade (*Validity*) – se um processo correto difunde (*multicast*) uma mensagem *m* ao grupo, então algum processo correto do grupo irá entregar *m* num tempo finito, ou nenhum processo do grupo é correto.
- Acordo (*Agreement*) – se um processo correto entrega uma mensagem m, então todos os processos corretos pertencentes ao grupo irão entregar *m* num tempo finito.
- Integridade (*Integrity*) – para qualquer mensagem *m*, cada processo correto entrega m somente uma vez, e somente se algum processo difundiu *m* anteriormente.

Em síntese, a propriedade de validade indica que o conjunto das mensagens entregues inclui todas as mensagens enviadas ao grupo pelos processos corretos; a propriedade de acordo requer que todos os processos corretos pertencentes ao grupo entreguem o mesmo conjunto de mensagens; e a propriedade de integridade indica que não deve haver entrega de mensagens espúrias.

Se o remetente não falhar, todos os membros do grupo que não falharam entregam a mensagem. Se o remetente falhar então, ou todos os membros do grupo que não falharam entregam a mensagem, ou nenhum deles o faz. Esta é uma propriedade de atomicidade. Um *multicast* confiável assegura que todos os membros do grupo que estão "vivos" (ativos) entreguem a mensagem ou nenhum deles o faça. O *multicast* confiável não impõe nenhuma restrição na ordem em que as mensagens são entregues.

## 2.3 Ordenação causal e total de mensagens

A ordenação causal de mensagens provê garantias de sincronização que tornam mais fácil a programação distribuída cooperativa. Informalmente, os requisitos da ordenação causal são: se o envio da mensagem *m1* precede a mensagem *m2*, então a mensagem *m1* deve ser entregue antes da mensagem *m2*, caso o destino de ambas as mensagens *m1* e *m2* seja o mesmo [KAR 97].

Um exemplo da violação da ordenação causal para mensagens do tipo *unicast* é mostrado a seguir. Por hipótese, *P1*, *P2* e *P3* são três processos, em um sistema distribuído. O processo *P1* envia uma mensagem *m1* para *P3* e então envia uma mensagem *m2* para *P2*. *P2* recebe a mensagem *m2* e então envia uma mensagem *m3* para *P3*. Pela relação "aconteceu-antes" (*happened-before)* de Lamport, que é uma relação de ordem parcial, o envio da mensagem *m1* precede o envio da mensagem *m3* e então deveria ser entregue antes de *m3* por *P3*. Mas a mensagem *m3* foi recebida antes; se for entregue imediatamente, leva à violação da ordenação causal (figura 2.4).

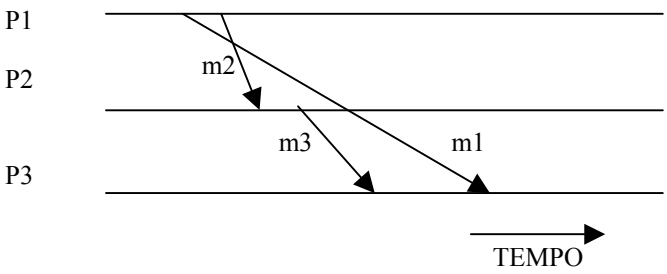


FIGURA 2.4 - Violação de entrega causal de mensagens *unicast* [KAR 97]

A figura 2.5 ilustra um exemplo da violação da entrega causal para mensagens em *multicast*. Por hipótese, o processo *p1* envia uma mensagem *m1* em *multicast* para o grupo que consiste dos processos *P1*, *P2*, *P3* e *P4*; o processo *P3*, ao receber a mensagem *m1*, envia uma mensagem *m2* em *multicast* para o mesmo grupo. Novamente, usando a relação de ordem parcial de Lamport, o envio da mensagem *m2* aconteceu após a recepção da mensagem *m1* e então deveria ser entregue após *m1*. Mas em *P4* a mensagem *m2* é recebida antes de *m1*, podendo levar para a violação do ordenamento causal.

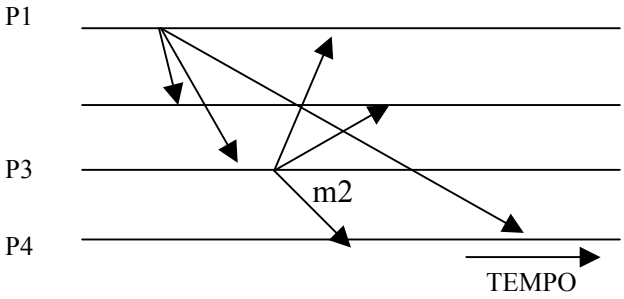


FIGURA 2.5 - Violação de entrega causal para mensagens em *multicast* [KAR 97]

Na figura 2.6, supõe-se que *p1*, *p2*, *p3*, *p4* e *p5* são processos pertencentes ao mesmo grupo. Os processos *cliente1* e *cliente2* enviam em *multicast* as mensagens *m1* e *m2* respectivamente, para o grupo de forma concorrente. Logo, não existe relacionamento causal entre *m1* e *m2*; *p1* poderia receber a mensagem *m1* e então *m2* e fazer sua entrega nessa ordem, enquanto que *p5* poderia receber a mensagem *m2* e então m1 e manter essa ordem relativa para as operações de entrega. Desse modo, há dois processos do mesmo grupo ordenando mensagens de uma forma inconsistente. Esta situação pode conduzir a inconsistências para aplicações tais como uma réplica de servidor de grupo, onde todos os membros do grupo estão gerenciando uma cópia do mesmo dado.

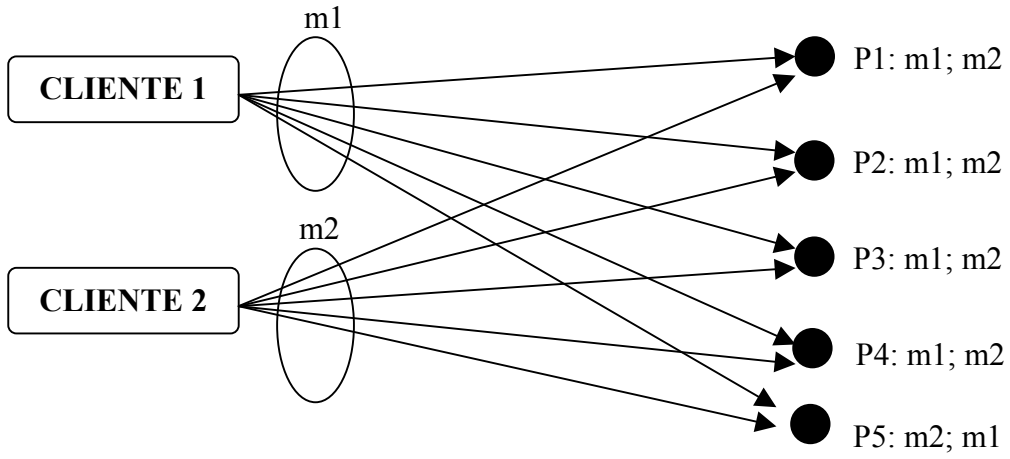


FIGURA 2.6 - Violação de ordenação total [KAR 97]

A ordenação total visa assegurar que todos os membros no grupo entreguem as mensagens na mesma ordem relativa. Usualmente, a ordenação causal é uma ordem parcial que é estendida para fazer a ordenação total de mensagens, a partir da causalidade e ordem total na entrega de mensagens. Ela também é chamada de ordem combinada (*agreed order*). Quando esta ordenação é feita em combinação com entrega confiável, ela é chamada de *multicast* atômico [KAR 97].

O *multicast* atômico permite que processos enviem mensagens para o grupo de forma confiável, visto que os membros concordam sobre um conjunto de mensagens que eles desejam entregar e a ordem da entrega das mensagens. Informalmente, o *multicast* atômico garante que:

- ✓ todos os processos corretos entreguem o mesmo conjunto de mensagens que foram enviadas para o grupo;
- ✓ todas as mensagens enviadas pelos processos corretos sejam entregues;
- ✓ apenas as mensagens não falsificadas sejam entregues;
- ✓ requer que todos os processos corretos entreguem as mensagens na mesma ordem.

O *broadcast* atômico (AB) satisfaz as seguintes propriedades:

Validade: Se um processo correto envia uma mensagem *m*, então ele entrega *m* em um tempo finito.

Acordo: Se um processo correto entrega uma mensagem *m*, então todos os processos entregam *m* em um tempo finito.

Integridade: Para qualquer mensagem *m*, todos os processos corretos entregam *m*, no máximo uma vez se, e somente se, *m* foi uma mensagem anteriormente enviada por um membro do grupo.

Ordenamento Total: Se dois processos corretos *p* e *q* entregam duas mensagens *m0* e *m1*, então *p* entrega *m0* antes de *m1*, se e somente se, *q* entrega *m0* antes de *m1*.

A ordenação total e as propriedades de acordo que definem o *broadcast* atômico asseguram que todos os processos corretos entregam a mesma seqüência de mensagens. O *broadcast* atômico é um poderoso paradigma de comunicação para computação distribuída tolerante a falhas [KAR 97]. Usado como abstração, libera o desenvolvedor da aplicação de detalhes de sincronização, garante a entrega no sistema, e ajuda-o a concentrar-se apenas na lógica da aplicação.

# 3 Realização de *Membership*

Neste capítulo, são apresentados os conceitos básicos de gerência de membros de grupo (*membership*), o que é preciso para a sua execução, além de abordar os principais modelos de execução existentes (sincronismo virtual e suas variações), e as técnicas para detecção de defeitos.

O termo *membership* é utilizado para designar um conjunto de protocolos para a gerência dos membros de um grupo. Esta gerência é responsável pelo controle de entrada e saída de membros [PAS 99]. Sua execução parte da composição ou da existência de um grupo de membros participantes. Um exemplo de realização de *membership* é o sistema Ensemble [ENS 98 *apud* PAS 99]. Este sistema possui várias funções para gerenciar membros de um grupo.

Uma ferramenta de programação (ou *middleware*) que oferece comunicação *multicast* confiável entre grupos de processos/objetos, operando de forma ordenada em relação às mudanças nos membros do grupo (importante para obtenção de consistência), possibilita a construção de aplicações tolerantes a falhas de forma mais facilitada, principalmente porque simplifica o controle de concorrência entre os membros do grupo. Nestas ferramentas, as políticas adotadas para implementar as relações entre a entrega de mensagens (comunicação) e as mudanças na composição do grupo, considerando semânticas e cenários de falhas, são denominadas de modelos de execução, os quais têm em seu cerne as políticas de *membership*. Naturalmente estes modelos são voltados para grupos dinâmicos [NUN 99].

Num grupo dinâmico, a qualquer momento, novos membros podem juntar-se voluntariamente ao grupo (*join*), ou membros do grupo podem ser excluídos voluntária (*leave*) ou involuntariamente (defeito ou suspeita de defeito). Um Serviço de *Membership* de Grupo (GMS) abstrai da aplicação os eventos de mudanças no conjunto de membros do grupo, provendo, ao invés disto, uma visão local, ou simplesmente visão do grupo, composta dos membros alcançáveis no grupo, num dado instante. Naturalmente, as visões dos diversos membros do grupo devem ser atualizadas de uma maneira coerente, ou seja, os membros que as instalam (que desejam permanecer ou juntar-se ao grupo) devem concordar (*agree*) com sua composição.

De forma geral, e usando uma linguagem informal, um GMS deve fornecer as seguintes propriedades [BIR 96]:

1. partindo de uma visão inicial, a cada adição (*join*) ou exclusão (voluntária, por defeito ou suspeita de defeito), o GMS relata a nova visão do grupo. Dependendo do protocolo adotado, o relato das mudanças ao processo da aplicação pode ocorrer imediatamente ou após a estabilização das mensagens difundidas na visão que se encerra;

2. a visão do grupo não é alterada involuntariamente. Um processo é adicionado ao grupo somente se ele estiver ativo e tentando se juntar ao grupo (*join*); ele é excluído somente devido à sua própria vontade, ou por estar falho, ou sob suspeita de falha por outro processo;

3. todos os membros do grupo observam uma subseqüência contínua da mesma seqüência de visões do grupo, iniciando com a visão em que o membro foi

adicionado ao grupo e terminando com aquela em que ele foi excluído (saída voluntária, defeito ou suspeita de defeito);

4. o GMS não atrasa indefinidamente uma mudança de visão associada a um evento, isto é, se o serviço está ativo, um evento de *join*, saída voluntária, defeito ou suspeita, causa uma mudança de visão, num tempo finito, que corresponda ao evento;

5. ou o GMS permite o progresso somente num componente primário de uma rede particionada ou, se ele permite o progresso em componentes não-primários, todas as visões do grupo são entregues com uma sinalização indicando sua associação, ou não, à partição majoritária da rede.

Como pode-se perceber, o problema da impossibilidade de obtenção do consenso deterministicamente em sistemas assíncronos sob falhas [FIS 85] também reflete na busca de uma solução aceitável para um GMS sob as mesmas condições, pois a propriedade 4 exige o cumprimento de tempo finito [NUN 99].

Analisando estas propriedades para o funcionamento de um grupo dinâmico, verifica-se que, sozinhas, elas não são suficientes para o suporte de aplicações distribuídas consistentes, pois não há referência à ordem das mensagens em relação às mudanças de visão. Esta omissão permite que uma dada mensagem destinada a vários membros seja entregue à aplicação, por membros diferentes, em visões diferentes, causando inconsistências na aplicação. A percepção deste problema, bem como de problemas de reunificação de partições e desempenho, dentre outros, gerou a proposição de modelos de execução bem definidos [NUN 99].

Um exemplo de um sistema que realiza *membership*, implementado em Java RMI, é o Jgroup (figura 3.1), que está baseado em duas abstrações fundamentais: grupo de objetos remotos e objetos remotos replicados. Do ponto de vista do servidor, um grupo de objetos remotos consiste em uma coleção de objetos remotos replicados (às vezes chamados de réplicas) que implementam o mesmo conjunto de interfaces remotas, e coordenam as suas execuções para parecerem objetos remotos não replicados. As réplicas que formam um grupo de objetos remotos cooperam usando um serviço de comunicação de grupo particionável, cuja tarefa é simplificar o desenvolvimento de consistência de protocolos necessários para oferecer um serviço altamente disponível e confiável [MON 99].

No Jgroup, os clientes não têm nenhum acesso a um objeto remoto simples, e só interagem com o grupo de objetos remotos. Do ponto de vista do cliente, grupos de objetos remotos não são distingüíveis de objetos remotos simples, ficando transparente para o cliente. Cada grupo implementa uma ou mais interfaces remotas, onde podem ser invocados métodos usando o mecanismo de RMI: clientes obtêm um *stub* local que apresenta o mesmo conjunto de interfaces e age como um substituto do grupo de objetos remotos. Cada invocação de método no *stub* local corresponderá a uma invocação remota em uma ou mais réplicas que formam o grupo, dependendo da invocação particular e da semântica adotada [MON 99].

O resto desta seção é dedicada a dois componentes fundamentais: a arquitetura do Jgroup: no lado do servidor, o GCS (*Group Communication Service*) particionável; no lado de cliente, a transparência e a tolerância a falhas no mecanismo de invocação.

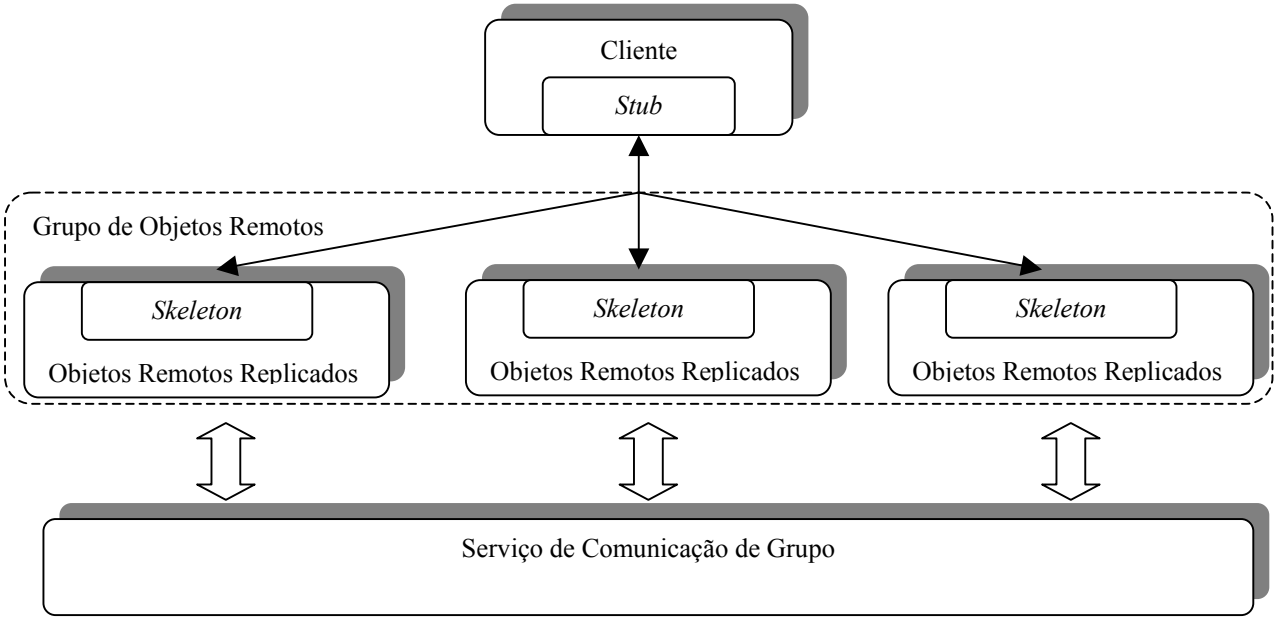


FIGURA 3.1- A arquitetura do Jgroup [MON 99]

Um objeto remoto replicado torna-se o membro do grupo de objetos remotos g por meio de um pedido, através de um método *join*(g). Depois de ter se juntado ao grupo g, os objetos remotos replicados podem deixar o grupo por meio de um pedido a um método *leave*(g). O serviço de membros de grupo particionável notifica os membros sobre as mudanças no grupo g, por um evento *vchg*(g,v) que corresponde à instalação de uma visão v. Cada visão é determinada por um identificador único e consiste de uma coleção de objetos remotos replicados [MON 99]. Estas características também foram implementadas no protótipo desta dissertação.

Como exemplo, é considerada a implementação de um quadro-negro compartilhado para apoiar sessões de trabalho cooperativas entre grupos de usuários. A instalação de uma nova visão, devido a um particionamento, pode requerer a exibição de uma mensagem de advertência que informa aos usuários que o estado do quadro-negro vai ficar inconsistente nas partições distintas. Por outro lado, a instalação de uma nova visão devido à fusão de duas partições, pode causar a execução de um protocolo de reconciliação de estados no conteúdo do quadro negro [MON 99].

## 3.1 O modelo com sincronismo virtual

O modelo de sincronismo virtual considera processos que utilizam a semântica *fail-stop*[n], e incorpora um detector de defeitos (FD-*Failure Detector*). Processos com defeito são detectados pelo envio de mensagens para eles e a espera por uma resposta dentro de um determinado intervalo de tempo [KAR 97, FEL 98]. O FD envia informações sobre o colapso e a recuperação de processos, os quais serão usados na revisão de suas visões, que são os conjuntos de processos considerados como ativos

[n] Na semântica *fail-stop*, adicionalmente ao colapso do processo (sem a realização de ações incorretas), os processos que interagem com os processos falhos tem uma forma precisa de detectar tais defeitos [SCH 83].

pelo detector. Em geral, e na prática, o detector de defeitos não é perfeito em todas as suas constatações: um processo que está ativo mas que, por algum motivo, demore no envio de uma mensagem, pode ser considerado como um defeito, o que não é verdade. Porém, é desejável que o FD seja completo no senso de que, se um processo falhar, então este defeito seja detectado com certeza. O FD pode ser visto como o remetente de informações para atualização de visões para os processos, onde se deve considerar explicitamente uma distinção importante entre recepção e entrega da visão por um processo, que é comum para protocolos de comunicação ordenados relatados em eventos de sistemas distribuídos [SCH 93].

O sincronismo virtual pode ser definido conforme segue.

Considere-se um grupo *g*, uma visão $v_i(g)$ e uma mensagem *m* enviada em *multicast* confiável para $v_i(g)$. O modelo de sincronismo virtual assegura as seguintes propriedades: se existe p pertencente a $v_i(g)$ que tenha entregue *m* na visão $v_i(g)$, e tenha entregue a visão $v_{i+1}(g)$, então todos os processos q pertencentes a *vi(g)*, os quais tenham recebido a visão $v_{i+1}(g)$, devem ter entregue *m* antes de $v_{i+1}(g)$. Ou seja, se um determinado processo *p* envia uma mensagem a ser entregue por todos os processos do grupo, o administrador do grupo deve verificar se os processos ativos correspondem exatamente aos processos que compunham a visão no envio da mensagem *m*, confirmando ou cancelando a entrega da mesma. A entrega confiável de uma determinada mensagem *m* na visão $v_i(g)$, significa que *m* deve ser entregue por todos os processos não falhos em $v_i(g)$, ou por nenhum.

Segundo Birman, o modelo de sincronismo virtual foi assim definido [BIR 96]:

1. cada grupo de processos tem associado a si uma visão do grupo, na qual os membros são listados pela ordem em que eles se juntaram ao grupo;
2. mudanças na visão do grupo são relatadas na mesma ordem para todos os membros;
3. qualquer *multicast* enviado para o grupo é realizado entre duas visões e para todos os membros do grupo. A gerência dos membros do grupo é feita com a última visão do grupo recebida pelo processo que enviou a mensagem (envio *multicast* com visão síncrona);
4. quando um processo junta-se ao grupo e a visão do grupo é relatada aos outros membros, este novo processo pode obter o estado corrente do grupo a partir de algum membro ou conjunto de membros pré-existentes;
5. os *multicasts* são atômicos e ordenados, podendo esta ordenação ser total ou causal;
6. os defeitos são tratados segundo o modelo *fail-stop*, ou seja, se um defeito é relatado a qualquer processo, todos os processos vêem o mesmo evento;

7. Um processo torna-se defeituoso devido a um colapso ou devido a um particionamento de rede. Neste último caso, quando a partição é reparada, o processo que se tornou falho, somente pode se reunir ao grupo com um identificador de processo diferente, e após executar um protocolo especial de desconexão, disparado quando o processo perceber que pertence a uma partição primária (majoritária no caso do Isis).

Dentre os principais tópicos estudados no projeto, destaca-se o conceito de visões de grupo. Uma nova visão é criada quando existe uma falha em um determinado processo no grupo, ou ainda quando ocorre a entrada ou a saída voluntária de um processo do grupo, forçando com isso a reorganização do grupo com a atualização dos processos ativos. Este conceito é extremamente importante no ordenamento das ações que serão realizadas em diferentes processos ou objetos pertencentes aos grupos.

No modelo de sincronismo virtual, a abstração simplificada vista pelo desenvolvedor é que, no conjunto de processos que fazem parte do grupo (membros do grupo), todos veêm os mesmos eventos na mesma ordem [BIR 96].

Sincronismo virtual dá um significado preciso de lista de transferência. Entre quaisquer duas mudanças de visão consecutivas, um conjunto *G* de mensagens é *multicast* se os remetentes são restritos aos processos da visão. Então, se um processador *p* é removido da visão, os processos remanescentes na visão podem assumir que *p* falhou. O sincronismo virtual garante que nenhuma mensagem de *p* será transmitida no futuro. Adicionalmente, o sincronismo virtual supõe a transmissão de mensagens de forma confiável [CHO 97]. Com isso identifica-se, por exemplo, uma situação em que dois processos *p* e *q* foram descontinuados. Logo, na mudança de visão, estes dois processos não farão mais parte do grupo. Isto é mostrado na figura 3.2. Intuitivamente, todos os processos da visão *v* que não falharam recebem todas as mensagens de G. Um processador *p* que falha não poderia receber todas as mensagens de G. Mas, desde que a transmissão de mensagem é sincronizada com mudanças de visão, se *p* for recuperado ele pode voltar a pertencer ao grupo novamente com um novo identificador.

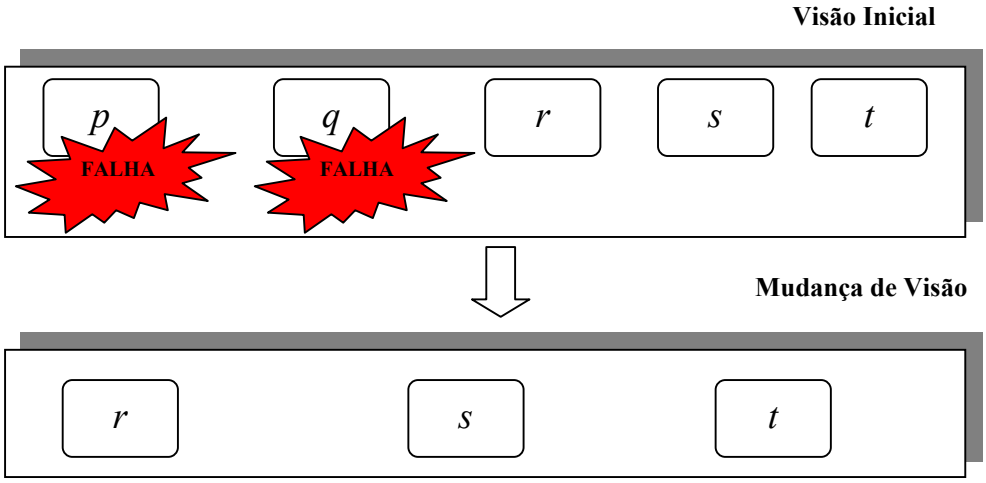


FIGURA 3.2- Mudança de visão no sincronismo virtual [CHO 97]

Por hipótese, deseja-se usar um grupo de processos (ou um conjunto de grupos de processos) como um bloco de construção em uma aplicação distribuída. Os membros do grupo unir-se-ão para uma proposta de cooperação, talvez para controlar réplicas de dados ou para realizar alguma operação tolerante a falhas e, portanto, redundante. A questão agora é como projetar ou escolher algoritmos que cumpram adequadamente sua função, garantindo que estes grupos ou seus membros possam operar corretamente; e que mantenham todos os componentes do grupo com os dados atualizados, onde todos têm a mesma visão. Dentro desse estudo existem vários modelos, mas a base de gerência dos membros dos grupos e de visões correntes entre eles está no sincronismo virtual [BIR 96, FRI 95a].

A máquina de estados exige que a atualização seja síncrona, que significa a necessidade de que os membros do grupo fiquem sincronizados para que a atualização possa proceder. Isto não é bom, pois além de todos terem que parar suas atividades para que a atualização seja feita, se um dos membros do grupo falhe, isto compromete a atualização de todos os outros membros [BIR 96,GUO 96].

O sincronismo virtual é uma técnica que permite a um sistema trabalhar com recursos que façam os dados serem atualizados em todos os membros do grupo, de uma maneira assíncrona e dinâmica. Caso um dos membros do grupo falhe, ele é automaticamente descartado [BIR 96].

Em [FRI 95], foram especificadas duas formas de implementação do sincronismo virtual, com diferentes restrições: sincronismo virtual forte e sincronismo virtual fraco. Essas opções foram implementadas no sistema Horus. No sincronismo virtual forte é definido que o programa de aplicação deve bloquear as mensagens durante as mudanças de visão. Já o sincronismo virtual fraco, não bloqueia mensagens durante as mudanças de visão. Friedman [FRI 95] mediu os reflexos do uso das duas abordagens e concluiu que a latência do envio de mensagens durante as mudanças de visão no modelo fraco é maior que a latência do modelo normal. Demonstra-se esta diferença pelo gasto de tempo extra (*overhead*) associado à manipulação de mensagens em segundo plano, causando aumento (no sincronismo virtual forte) no tráfego de mensagens. Observa-se também que as mensagens enviadas durante a troca de visão no modelo forte sobrecarrega o sistema de forma significativa, com relação ao modelo fraco e normal. Esta diferença pode ser visualizada nos gráficos mostrados nas figuras 3.3 e 3.4, através de medidas máximas e médias, respectivamente.

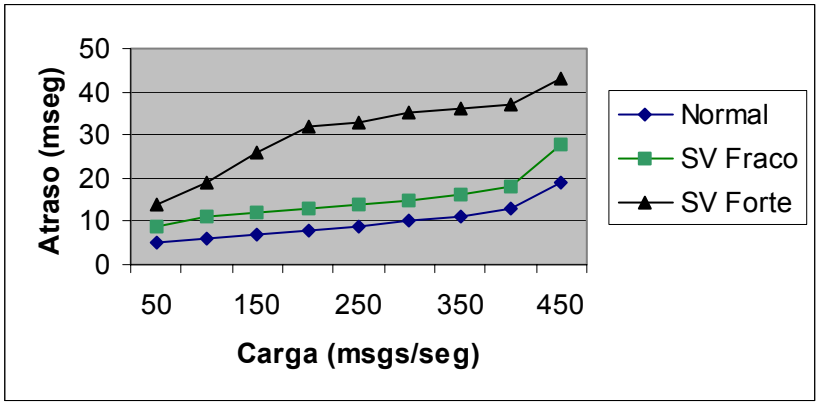


FIGURA 3.3- Medidas máximas de latência [FRI 95]

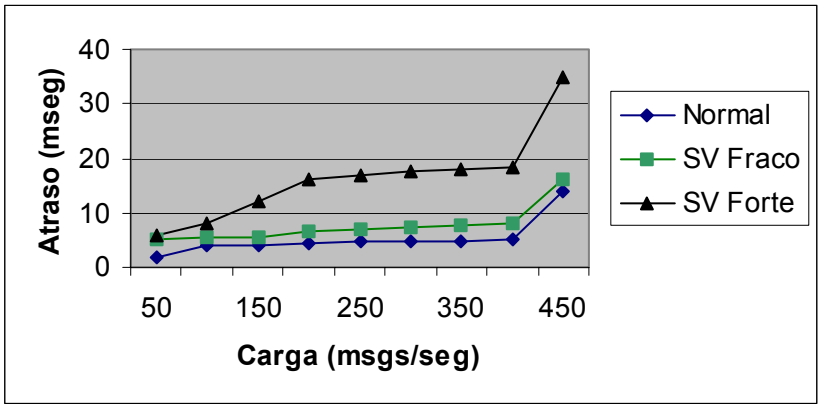


FIGURA 3.4- Medidas médias de latência [FRI 95]

Conforme citado por Nunes [NUN 98], Birman [BIR 96] afirma que o modelo com sincronismo virtual foi proposto para permitir computação cooperativa. Para isto, o modelo preocupa-se com o suporte a: processos que devem ser altamente confiáveis, gerenciamento de tarefas compartilhadas, ações sincronizadas, dados replicados, reconfiguração dinâmica frente a mudanças de carga, defeitos ou recuperações, entre outros.

Um exemplo expressivo do uso de sincronismo virtual foi o sistema ISIS [BIR 93], que usa ordenamento causal (envio assíncrono de mensagens) para conseguir otimizar o desempenho de um sistema completamente síncrono, como o encontrado no ordenamento total. O controle da concorrência também é feito usando mecanismos de sincronização.

O ponto chave para a denominação “virtual” deste tipo de sincronismo é que a princípio todas as aplicações são projetadas assumindo um ordenamento total (síncrono), mas o fluxo de mensagens ao invés de ser síncrono é assíncrono, com controle síncrono somente nos momentos de acesso a recursos compartilhados. O resultado final é idêntico ao de um sistema completamente síncrono, mas com a vantagem de que, na maioria dos casos, a solução é mais rápida.

Existe uma série de modelos embasados no sincronismo virtual, mas com diferenças que fazem com que os mesmos recebam novas definições, como apresentado na seção 3.3.

## 3.2 Sincronismo de operações

No sistema Phoenix [MAL 96], para implementar o paradigma do protocolo de mudança de visão, foi adotado um protocolo de consenso com alguns tratamentos adicionais. O protocolo de consenso de Chandra e Toueg [CHA 95] é baseado sobre o paradigma da rotatividade do coordenador. A computação ocorre através de rodadas assíncronas onde, em cada rodada, um dos processos presentes assume o papel de coordenador (figura 3.5). Supõe-se a não ocorrência de defeitos. As regras para o funcionamento deste protocolo são as seguintes [MAL 96]:

**fase 1** - os participantes enviam suas propostas para o coordenador;

**fase 2** - o coordenador reúne a maioria das propostas e propõe um valor de saída de acordo com os critérios descritos abaixo:

**fase 3** - (1) um participante recebe a proposta do coordenador, a aceita, e envia sua concordância(*acknowledgement* ou simplesmente *ack*) para o coordenador, ou (2) o participante suspeita de falha do coordenador e envia um sinal de caráter negativo (*nack*) para o coordenador. Em ambos os casos, o participante segue para outra rodada;

**fase 4** - o coordenador reúne a maioria das respostas; se existir maioria de *acks* positivos, o coordenador decide e envia a decisão para os participantes.

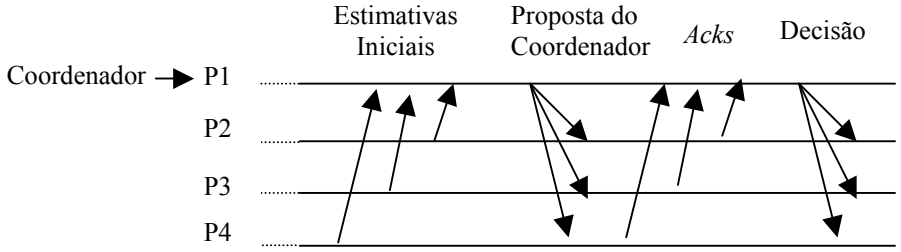


FIGURA 3.5– Protocolo de consenso sem a ocorrência de defeitos [MAL 96]

Diferentemente do protocolo de consenso justamente por operar com defeitos possíveis, o coordenador no protocolo de mudança de visão aguarda pela maioria das propostas, mas sim pela proposta de todos os processos não suspeitos. Com isto, nota-se que o Phoenix é baseado no modelo de sincronismo virtual, também conhecido como visão síncrona[4] [NUN 99].

O protocolo de mudança de visão possui as mesmas quatro fases do protocolo de consenso, porém possui algumas diferenças que são as seguintes:

1. na proposta inicial: na fase 1, um participante não envia uma proposta simples ao coordenador, mas sim o conjunto de mensagens não estáveis;
2. na geração de uma proposta: na fase 2, o coordenador não aguarda a maioria das propostas, mas todas as propostas de todos os processos não suspeitos de falha;
3. na decisão: um processo recebendo uma decisão, entrega todas as mensagens contidas na decisão que ainda não foram entregues e então instala a nova visão.

[4] Alguns autores estabeleceram este aspecto de sinônimo com sincronismo virtual, mas as idéias não são essencialmente iguais. A visão síncrona indica a possibilidade de existirem visões concorrentes, o que não é especificado no sincronismo virtual. Mais detalhes são discutidos na monografia desenvolvida por Nunes [NUN 99].

O protocolo de mudança de visão utiliza as mensagens não estáveis para compor a proposta a ser enviada pelo coordenador. Mensagens não estáveis são aquelas que, no instante de uma troca de visão, já foram enviadas, mas que ainda não foram entregues por todos os membros, podendo ser necessário retransmiti-las. [GUO 96].

No protocolo de mudança de visão, cada processo define sua proposta inicial. Esta proposta é composta do conjunto de mensagens que ainda não são estáveis neste processo e o conjunto de processos que deveria definir a nova visão. O valor inicial da nova visão proposta é o conjunto de processos correntes no momento em que o protocolo é iniciado (lembrar que a camada de direcionamento (*routing*) é que faz a entrega desta informação) [MAL 96].

A geração da proposta é feita pelo coordenador, quando o mesmo recebe dos participantes da rodada corrente as propostas contendo o conjunto de mensagens dos processos. Ele gera uma proposta baseada nas que foram recebidas. Em contraste ao protocolo de consenso, não se espera apenas pela maioria, mas por todas respostas dos processos corretos, sem exceção. As figuras 3.6 e 3.7 mostram o porquê da importância desta modificação, apresentando um caso onde a maioria é exigida e outro caso onde é necessária a participação de todos processos corretos (não suspeitos).

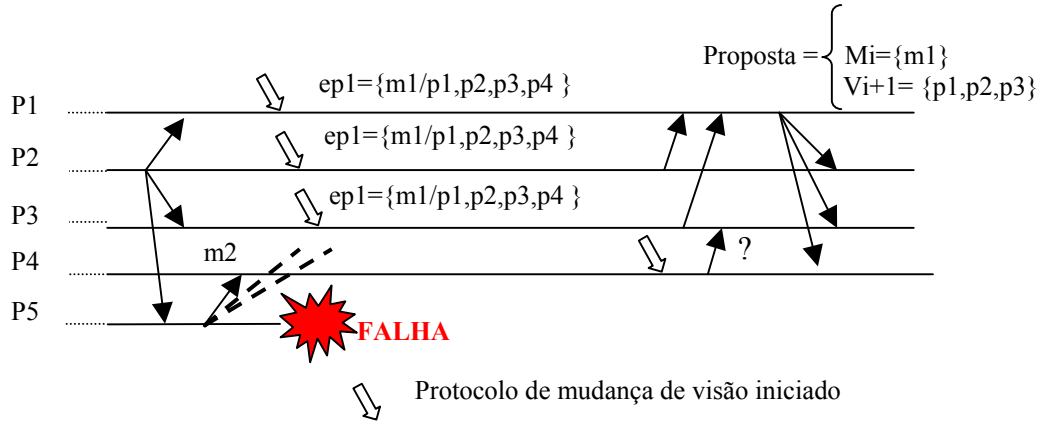


FIGURA 3.6 – Esperando somente a maioria [MAL 96]

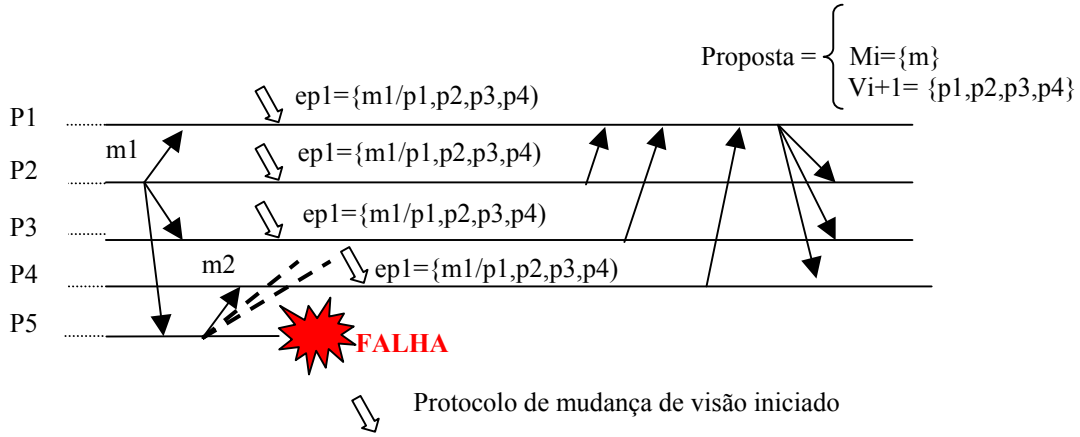


FIGURA 3.7 – Esperando por todos os processos corretos [MAL 96]

O coordenador da figura 3.6 aguarda somente pela maioria das propostas; assim não espera pela mensagem do processo *p4*. Ele não inclui *p4* na nova

visão, porque *p4* poderia ter entregue uma mensagem *m2* que não foi entregue pelos outros processos. Desta forma, seria violada a propriedade de visão síncrona, onde a proposta (*m1*/*p1,p2,p3*) deveria satisfazê-la. Já na figura 3.7, ocorre a espera pelas mensagens de todos os processos ativos, respeitando a propriedade correspondente.

## 3.3 Variações de sincronismo virtual

A forma básica do sincronismo virtual foi implementado no ISIS onde, quando acontece um particionamento da rede, somente a partição que contém o fragmento do grupo considerado oficial ou majoritário fica ativa (partição primária). A outra partição (ou outras) é suspensa e seus processos disparam procedimentos de desconexão compulsória. Quando o sistema restabelece a conexão, há atualização nos estados dos novos membros simplesmente pelo recebimento do estado a partir do armazenado na partição primária (aqueles que haviam sido induzidos a sair, reconectam-se com um novo indicador de processo) [NUN 98].

No sincronismo virtual, todas as entidades no sistema observam os eventos do sistema na mesma ordem relativa (entretanto não no mesmo tempo físico). Isto permite a aplicação ver a ocorrência de eventos no grupo de forma síncrona, ainda que o ambiente seja assíncrono. Sincronismo Virtual requer que, se um processo *p* e *q* são ambos membros de duas configurações consecutivas C e D no mesmo grupo, então *p* e *q* determinam exatamente a mesma ordem total (ordem concordante) de mensagens em C. Sincronismo Virtual estende os requisitos de ordenamento total para reconfigurações no grupo [KAR 97].

Uma partição da rede pode ocorrer quando o grupo é dividido em dois ou mais subconjuntos, cada processo pode comunicar com todos os processos do subconjunto, mas não pode se comunicar com qualquer processo fora deste subconjunto. Um exemplo de partição de rede está na figura 3.8. Neste exemplo, os processos *p*, *q*, *r* e *s* são executados sobre hospedeiros Hi (i = 1,2,3,4). H1 e H2 pertencem a um subconjunto e H3 e H4 pertencem a outro subconjunto. Os dois subconjuntos estão conectados por um roteador R. Se todos os processos pertencem a um mesmo grupo e o roteador falha, então tem-seuma rede particionada, onde o grupo é dividido em dois subconjuntos, um contendo *p* e *q* e outro contendo *r* e *s*.

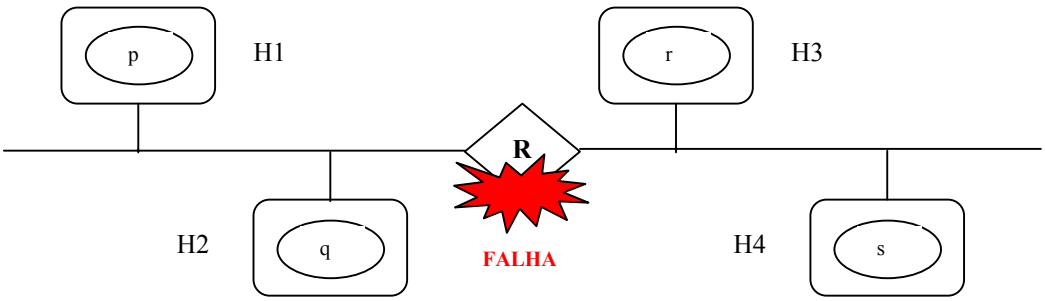


FIGURA 3.8 - Exemplo de uma partição de rede [KAR 97]

Um outro enfoque é o Sincronismo Virtual Estendido (*Extended Virtual Sinchrony* – EVS), o qual, como o próprio nome indica, constitui-se em uma

extensão do modelo básico de sincronismo virtual. Essa extensão foi criada para satisfazer a necessidade imposta pelo uso de redes *wide-area* (larga-escala), onde a baixa confiabilidade dos canais de comunicação e os grandes atrasos na comunicação resultam em partições da rede com uma freqüência muito maior do que em redes locais. Segundo citado por Nunes [NUN 99], os responsáveis pelos projetos Transis [LOT 95] e Totem [DOV 95], respectivamente Dolev e Agarwal, apontaram para aplicações que poderiam tolerar inconsistências geradas pelo progresso de um componente não-primário num sistema particionado. Assim, qualquer partição que possa alcançar um acordo interno no seu controle de membros (*membership*) recebe permissão de continuar operando. Aplicações que só são seguras se executadas numa única partição (primária) devem fazer a suspensão (de forma mais restrita) de seus serviços na(s) partição(ões) não-primária(s), agrupando seus estados quando o sistema eliminar o defeito.

Quando o funcionamento prevê a possibilidade de particionamento, significa que a operação permite partições na rede e o tratamento posterior, ou a reunião após desaparecer a causa do defeito. Isto pode ser conduzido com o uso de um modelo de execução acima referenciado como sincronismo virtual estendido. Se pode ocorrer a partição da rede, e a união das partições da rede (*rejoin*), então os membros do grupo devem permitir a entrega de mensagens sob certas condições.

Embora o modelo EVS aparentemente tenha um grande potencial de uso, na prática, ele apresenta um sério problema ao fazer a união (*merging*) das partições primária e secundárias, pois o algoritmo de união necessita resolver, de algum modo, as inconsistências que podem ter ocorrido durante o tempo em que o sistema se manteve particionado, o que pode nem sempre ser possível, conforme foi afirmado por Birman [BIR 96] e citado por Nunes [NUN 98].

Existe também um modelo misto que está sendo mais aceito [NUN 98]. Este modelo é denominado *two-tiered model.* Este modelo sugere a divisão de uma grande rede *wide-area* em várias redes locais (LAN) interconectadas por uma pequena rede *wide-area* (WAN). Esta divisão permite estruturar a rede em duas camadas: uma camada LAN e uma camada WAN.

Ainda conforme citado por Nunes, Babaoglu [BAB 96] desenvolveu um modelo chamado Visão Síncrona Enriquecida. Ele permite também que a partição não-primária continue funcionando, entretanto aceita somente composições de grupos não sobrepostos. A sobreposição poderia fazer com que um mesmo processo residisse em ambos os lados de uma partição, causando um comportamento inconsistente.

Existe também o SV Estruturado, que surgiu a partir de problemas de escalabilidade de grupos, e é explicado a seguir, a partir da motivação para sua proposta. Nos modelos de extensão do sincronismo virtual, quando ocorre uma mudança de visão no grupo, várias mensagens são geradas pelos algoritmos de controle de membros, tanto no modelo básico com sincronismo virtual como no modelo estendido. Estas mensagens extras trazem limitações no número de membros que podem pertencer a um grupo, ou mesmo no número de grupos que podem coexistir num dado momento [GUO 96].

No modelo com sincronismo virtual, tal como implementado no Horus, cada membro de um grupo controla uma visão privada (visão particular do grupo, onde cada membro tem uma visão atualizada sobre os membros ativos no mesmo) de todos os membros correntemente conectados, e uma mudança na visão do

grupo é vista como uma sucessão de visões, garantindo a propriedade de *multicast* com visão síncrona.

No Horus, quando não há alteração na composição do grupo, uma mensagem do tipo *multicast* é entregue a todos os membros na visão do remetente. Porém quando há mudanças nos membros, uma nova visão é instalada, e todas as mensagens enviadas em *multicast* na visão antiga (anterior à nova visão) são entregues aos membros sobreviventes antes da nova visão ser instalada. Este é o modelo de Sincronismo Virtual implementado na camada VS do Horus [SCH 93].

Quando ocorre uma mudança de visão, um membro designado coordenador faz um *multicast* de uma mensagem FLUSH. Após receber esta mensagem, todos os membros sobreviventes suspendem o envio de mensagens e retornam ao coordenador uma mensagem informando quais membros podem não ter recebido suas mensagens (mensagens instáveis). Em seguida, cada membro sobrevivente envia ao coordenador uma mensagem FLUSH_OK. Ao receber todas as respostas FLUSH_OK, o coordenador faz um *multicast* das mensagens instáveis. Após todas se tornarem estáveis (mensagens que o coordenador reenviou aos destinos desejados, e que por algum motivo não haviam sido entregue antes), o coordenador finalmente envia um *multicast* contendo a nova visão do grupo. Este sincronismo virtual é instalado sobre uma camada de *multicast* FIFO confiável ou sobre um sistema de comunicação ponto-a-ponto.

Segundo citado por Nunes, Guo [GUO 96] demonstra medidas de desempenho realizadas na Universidade de Cornell sobre uma rede de estações de trabalho Sparc 10 onde, quando o tamanho do grupo ultrapassa 40 membros, os *buffers* de entrada dos membros “estouram” e inicia-se um processo de retransmissão, tal como o efeito “bola de neve”. O modelo com sincronismo virtual estruturado ataca este problema, ou seja, a escalabilidade dos sistemas de gerência de grupos.

A estruturação divide o método de gerência de membros do grupo em dois níveis. Esta divisão define um conjunto de membros locais e um conjunto de membros controladores. Toda e qualquer troca de mensagens entre os membros locais, vinculadas a um mesmo controlador, fica restrita a este nível. Somente as comunicações entre membros locais de controladores distintos são feitas através dos controladores. Isto tem um impacto muito grande no efeito “bola de neve” que pode ser causado pelas mensagens extras geradas pelo sistema, quando ocorre uma troca de visão no grupo. Desta maneira, o número máximo de membros de um grupo cresce de N para $N^2/2$, sendo muito importante quando se pensa em utilizar o modelo de grupo de processos para construir aplicações em redes de larga escala como a Internet.

O sincronismo virtual está ligado com a replicação de dados e objetos, pois o mesmo está diretamente ligado ao controle na realização coordenada de atividades em todas as réplicas e com o controle de alterações diante de situações de falha em nós e da rede de comunicação. Assim, para os programas de aplicação, sem o conhecimento detalhado da localização e número de réplicas, garante-se a realização de forma correta.

A escolha por uma das diferentes opções de sincronismo virtual vai depender das exigências da aplicação e dos sistemas.

## 3.4 Atualização distribuída no modelo sincronismo virtual

Considere-se um grupo de processos *g* e uma visão corrente *vi(g)*. A atualização distribuída pode ser resolvida em *g* usando a primitiva de *multicast* uniforme confiável [SCH 93].

Existem algumas aplicações construídas com base em troca de mensagens via *multicast*, porém necessitam da garantia de entrega na visão em que foram enviadas. Com isso, pode-se usar *multicast* uniforme, que é uma típica primitiva "todos ou nenhum" (*all-or-nothing)*. Ela assegura que, se qualquer processo de uma determinada visão *vi* (quer seja correto ou não) entrega uma determinada mensagem *m*, todos os processos corretos devem entregar esta mensagem *m*, na mesma visão em que a mesma foi enviada, em um tempo finito [MAL 96].

Malloth compara um envio em *multicast* não uniforme com um em *multicast* uniforme, através das figuras 3.9 e 3.10. Em ambos, o processo p1 falha durante o envio de uma mensagem m em *multicast*, e somente o processo p2 recebe a mensagem enviada. O processo p2 fica em uma partição diferente dos outros processos e com isto não é incluído na nova visão vi+1 (p2 é suposto como um processo incorreto). Na figura 3.9, a mensagem é entregue pelos processos p1 e p2, mas não por p3, p4 e p5, ou seja, a propriedade de entrega uniforme foi violada. Já na figura 3.10, a mensagem não é entregue por nenhum processo (correto ou não) [MAL 96].

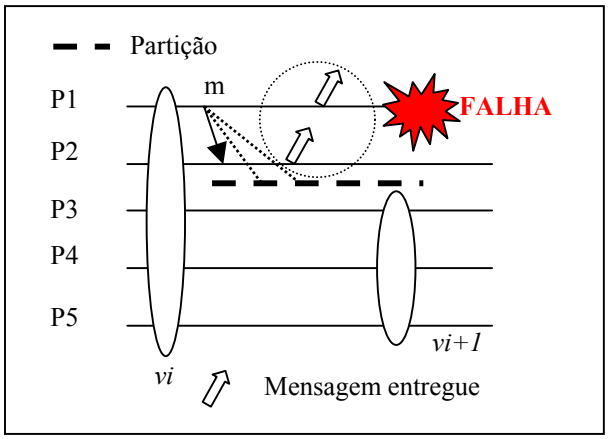


FIGURA 3.9 -*Multicast* não uniforme [MAL 96]

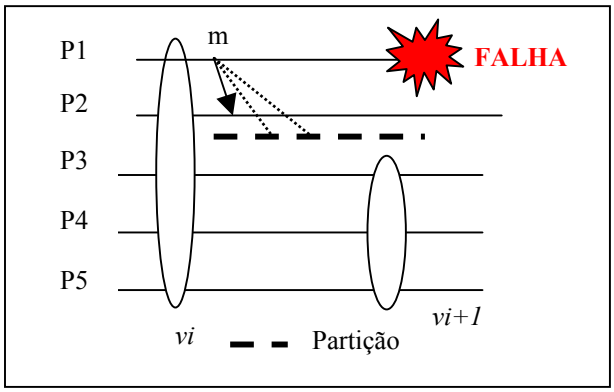


FIGURA 3.10- *Multicast* uniforme [MAL 96]

*Multicast* uniforme é a primitiva adequada para implementar encerramento atômico de transações. Como exemplo, suponha-se que *p1* é o coordenador para o encerramento atômico. Se o processo *p1* enviar uma mensagem de atualização *c* com a primitiva *multicast* uniforme, existirão duas situações a serem levadas em consideração: (1) se qualquer processo entregar *c* na visão *vi*, então todo processo correto também deverá entregar *c* na visão *vi*, ou (2) nenhum processo entregará *c* em qualquer visão. No caso (1), a entrega da mensagem *c* precede o ato de encerrar a transação, enquanto que no caso (2), um novo coordenador tem de ser definido para que haja a tentativa de se encerrar a transação na visão *vi+1*.

Para exemplificar o uso do *multicast* uniforme confiável, supõe-se uma mensagem m *multicast* confiável para *vi(g)*. O *multicast* uniforme existe se e somente se:

Se $\exists p \in vi(g)$ o qual tenha transmitido *m* na visão *vi(g)*, então todos os processos $q \in vi(g)$ que tenham transmitido a visão *vi+1(g)* devem ter transmitido m antes de *vi+1(g)*.

O *multicast* uniforme confiável de uma mensagem m para *vi(g)* pode ser implementado por um protocolo de duas fases, e deve seguir os seguintes passos:

1. o remetente p inicia por uma mensagem m enviada através de *multicast* confiável, sendo que a confirmação ocorre com uma indicação de não entrega de m;

2. qualquer processo q que receba m envia um sinal de recepção para o remetente na forma de *ack*(id(m)) para p;

3. tão logo quanto p tenha recebido *ack*(id(m)) de todos os processos de vi(g), p ativa a segunda fase;

4. na segunda fase, p envia para cada processo em *vi(g)*, uma mensagem para entrega de m, na forma de *deliver*(id(*m*)). Na recepção de *deliver*(id(*m*)) por cada processo *q*, *q* entrega a mensagem *m*.

Se uma falha ocorrer durante o *multicast*, a finalização do protocolo é necessária. O comentário característico da implementação baseada no *multicast* confiável, no modelo de sincronismo virtual, é que a finalização do protocolo não requer o envio de qualquer mensagem. Um processo *p* tendo recebido *m*, mas sem autorização para entregá-la, e certo de entregar a visão *vi+1(g)*, pode certamente entregar m antes de *vi+1(g)*. Detalhes do protocolo e provas podem ser encontrados em [SCH 93].

## 3.5 Serviços para gerenciar comunicação de grupo

O objetivo principal para o serviço de gerência de grupos é controlar a entrada e saída de membros do grupo, disponibilizando serviços em benefício dos processos que o compõem. O serviço de grupos de objetos fornece o suporte adequado para grupos dinâmicos, isto é, a composição dos grupos podem mudar de acordo com o tempo. Novos membros podem juntar-se a grupos existentes, outros membros podem deixar o grupo, ou podem ser explicitamente removidos do grupo pela ocorrência de

alguma falha. Objetos que desejem juntar-se ao grupo, devem fazê-lo pelo contato com o serviço de membros, o qual atualiza a lista dos componentes do grupo. Uma vez admitido que para um grupo, um objeto pode interagir com outros membros do grupo. Finalmente, se o objeto falha ou abandona o grupo, o serviço de membros deverá novamente atualizar a lista de membros do grupo [FEL 98]. Grupos dinâmicos envolvem dois gêneros de protocolos, descritos a seguir:

Protocolo de mudança de visão: é executado a cada tempo em que a composição do grupo muda. Ele assegura que qualquer membros corrente do grupo receba uma notificação de mudança de visão, indicando a nova composição do grupo como uma lista de membros do grupo como uma posição mutalmente consistentes. Mudanças de visão devem ser totalmente ordenadas.

Protocolo de Transferência de Estados: é uma operação atômica que acontece durante a mudança de visão, quando um novo membro junta-se a um grupo existente. Consiste em obter o estado dos membros de grupos correntes, entregando-a para o novo membro. Este protocolo assegura que os estados de todos os membros do grupo sejam consistentes sobre a mudança de membros. O protocolo de mudança de visão pode terminar somente após a transferência de estados ter sido completada.

## 3.6 Detecção de defeitos

Muitas aplicações distribuídas atuais necessitamrecursos de alta disponibilidade e de suporte a tolerância a falhas. Estas aplicações são baseadas em protocolos que possuem recursos para avaliação de componentes falhos como, por exemplo, a utilização de um detector de defeitos [FEL 98].

Processos defeituosos podem ser detectados pelo envio de mensagens a eles, e a espera por uma resposta dentro de um determinado intervalo de tempo [KAR 97].

Um detector de defeitos (FD) é um serviço, usualmente local a cada processo, que provê informações sobre componentes falhos. Ele monitora um subconjunto de componentes do sistema, e mantém uma lista dos processos suspeitos de terem sofrido um colapso.

A implementação do detector de defeitos é geralmente baseada sobre mecanismos de *timeout*. A escolha de valores para o tempo de espera são cruciais para o cumprimento adequado de seus objetivos. Pequenos *timeouts* permitem a um processo detectar defeitos rapidamente, mas induz um número grande de suspeitas incorretas. Com isto, existe um meio termo entre a latência (*timeouts* curtos) e precisão (*timeouts* longos). Se um sistema envolver mais que uma LAN, a opção para o valor de *timeouts* entre dois objetos depende de ambas as suas localizações no sistema e sobre a característica da rede existente.

Um sistema que provê facilidades de monitoramento envolve diferentes grupos de objetos. Clientes usam as facilidades de monitoramento para as interfaces completamente bem definidas para estes objetos. Em adição para clientes, existem geralmente três categorias de objetos no monitoramento do sistema [FEL 98]:

- ✓ **Monitores (ou detectores de defeitos):** são objetos que coletam informações sobre os componentes defeituosos. Em muitos sistemas, mecanismos de

monitoramento são diretamente implementados no código do cliente, e eles não aparecem como objetos separados.

✓ **Objetos Monitorados:** são objetos que podem ser monitorados, isto é, o defeito pode ser detectado pelo sistema de detecção.

✓ **Objetos Notificados:** são objetos que podem ser registrados pelo serviço de monitoramento, e que são assincronamente notificados sobre objetos falhos. Nem todos os sistemas de monitoramento provêm notificações assíncronas de defeitos assíncrono, e com isto não provém objetos notificados.

Objetos monitorados e notificados são geralmente específicos às aplicações, enquanto monitores são implementados pelo serviço. Existem dois tipos de interações entre esses componentes:

✓ **Monitor ↔ cliente, monitor ↔ objeto notificado:** este relacionamento permite a aplicação obter informação sobredefeitos dos componentes.

✓ **Monitor ↔ objetos monitorados:** esta interação é realizada pelo serviço de monitoramento para continuamente manter a informação referente ao estado do monitoramento de objetos.

## 3.7 Modelos de monitoramento

Vários modelos podem ser usados para monitoramento de objetos. Estes modelos diferem dependendo do caminho em que as informações sobre os componentes falhos são propagadas para o sistema. Isto corresponde à política de fluxo sobre os processos falhos (*flow policy*). Existem duas formas básicas de fluxos unidirecionais, *push* e *pull* [FEL 98].

Modelo *push*: O controle de fluxo é no sentido do fluxo da informação. Com este modelo, são monitorados objetos que estão ativos. Eles periodicamente enviam mensagens (*heartbeat*) para informar a outros objetos que eles ainda estão ativos. Se o detector de defeitos não receber o *heartbeat* dos objetos monitorados dentro de um limite específico de tempo, ele inicia uma suspeita sobre o objeto, senão ele envia uma informação ao cliente indicando que o objeto monitorável está ativo, conforme são mostradas nas figuras 3.11 e 3.12.

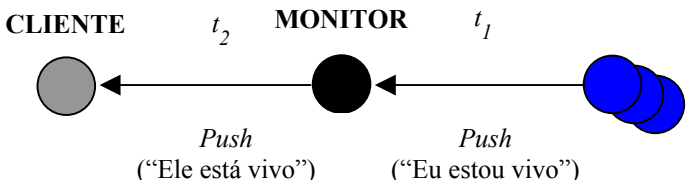


FIGURA 3.11 - O modelo *push* para monitorar objetos [FEL 98]

Observa-se que a troca de mensagens entre o monitor e o cliente é diferente das mensagens *heartbeat* enviadas dos objetos monitorados. O monitor geralmente notifica o cliente somente quando o objeto monitorado altera seu estado, enquanto mensagens *heartbeat* são enviadas continuamente [FEL 98].

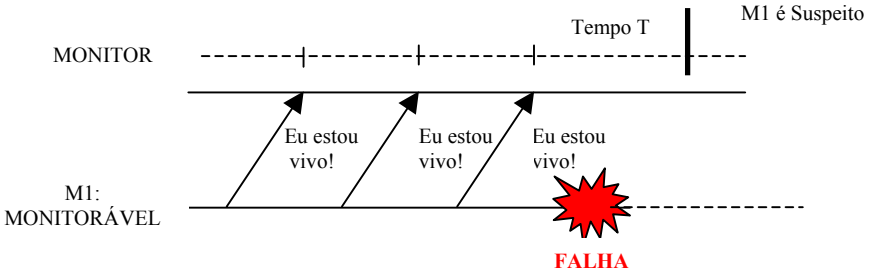


FIGURA 3.12 - Monitorando mensagens com o modelo *push* [FEL 98]

Modelo *pull*: No modelo *pull,* o fluxo das informações é no sentido oposto ao fluxo de controle, isto é, somente quando requisitado pelo cliente. Com este modelo, os objetos monitorados são passivos. O detector de defeitos envia periodicamente mensagens de requisição de estado (*liveness requests*) para os objetos monitorados. Se o objeto monitorado responde, ele indica que está ativo. Este modelo pode ser menos eficiente do que o modelo *push* visto que mensagens do tipo *one-way* são enviadas para objetos monitorados. Ainda assim é mais fácil para o desenvolvedor da aplicação, uma vez que os objetos monitorados são passivos, e não precisam ter qualquer conhecimento sobre tempo (isto é, eles não tem que conhecer a freqüência com a qual o detector de defeitos espera para receber as mensagens), como é mostrado nas figuras 3.13 e 3.14.

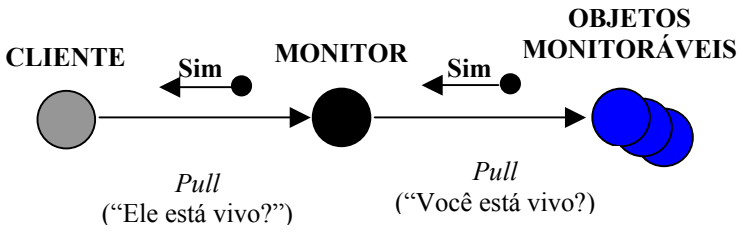


FIGURA 3.13 - O modelo *pull* para monitorar objetos [FEL 98]

O monitor envia periodicamente requisições de estado (*liveness requests*) para os objetos monitoráveis, e aguarda resposta. Se não obtiver uma resposta, dentro de um determinado intervalo de tempo, gera-se uma suspeita de defeito sobre o objeto que está sendo monitorado (figura 3.14).

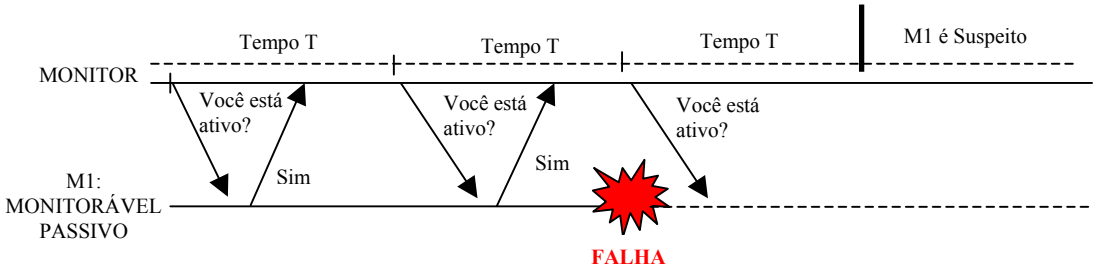


FIGURA 3.14 - Monitorando mensagens com o modelo *pull* [FEL 98]

Modelo duplo (*dual*): Para prover mais flexibilidade para o monitoramento de sistema, foi proposta por Felber uma extensão do modelo anterior, chamada modelo *dual,* em que ambos os modelo *push* e *pull* podem ser usados ao mesmo tempo com o mesmo conjunto de objetos. A idéia básica é usar dois gêneros de mensagens para o monitoramento de objetos. O detector de defeitos envia mensagens para os objetos monitoráveis, e mais tarde recebe uma resposta com uma mensagem indicando que o objeto está ativo (modelo *pull*). Os objetos monitoráveis podem também enviar uma mensagem indicando que estão ativos, sem terem sido requisitados para isto, o que corresponde ao modelo *push*. O detector de defeitos envia requisição de atividade somente se os objetos monitoráveis não enviarem uma mensagem dentro de algum limite de tempo específico. Este modelo trabalha com quaisquer tipos de objetos monitoráveis, sem a necessidade do monitor conhecer qual modelo é suportado por qualquer objeto simples.

Como exemplo, pode ser suposta uma situação em que dois objetos estão sendo monitorados, sendo eles *M1* e *M2*. O primeiro objeto, *M1* é *push-aware* (tem a iniciativa do envio de mensagens), portanto é ativo, e periodicamente envia mensagens indicando que está ativo. O segundo, *M2*, é passivo: ele somente envia mensagens indicando estar ativo quando é requisitado. O monitor usa dois períodos de tempos de espera *T1* e *T2*. Ele espera mensagens sobre a atividade do objeto monitorável *push-aware* durante *T1*. Depois de *T1*, o monitor envia uma requisição de estado (*liveness)* para cada objeto monitorável dos quais ele não recebeu mensagem de atividade, esperando uma resposta durante *T2*. Depois de *T2*, o monitor suspeita de todo objeto de quem ele não recebeu uma mensagem. Neste exemplo, *M1* envia mensagens durante *T1* na primeira fase, e trava pouco tempo após um defeito. Na segunda fase, o monitor envia uma requisição de mensagem de atividade para *M1*, mas não recebe uma mensagem antes de finalizar *M2*. Assim, ele inicia uma suspeita de que *M1* sofreu um defeito, conforme é mostrado na figura 3.15.

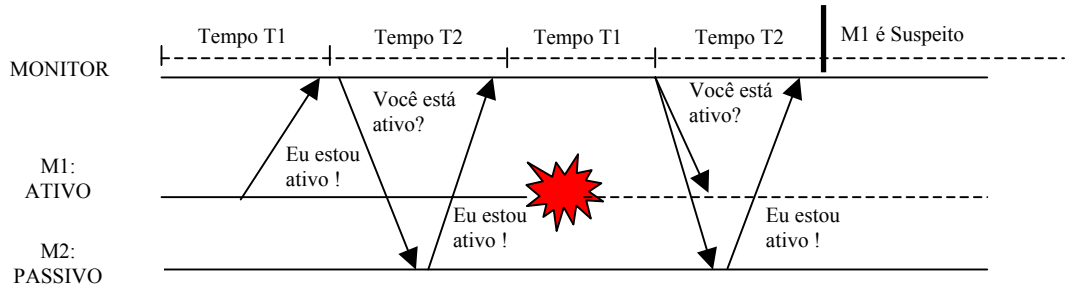


FIGURA 3.15 - Monitorando mensagens com o modelo *dual* [FEL 98]

O modelo usado no trabalho desenvolvido nesta dissertação é o modelo *pull*, onde o qualquere um dos membros ativos pode disparar a execução da opção Monitorar Falhas que efetua a detecção de defeitos.

## 3.8 Controle de Replicação

Segundo citado por Pasin, Leite e Amaral, Schneider define que em um sistema de arquivos replicados, as estações podem ser estruturadas em termos de clientes e servidores, onde os clientes requisitam serviços – operações de leitura ou de escrita em determinados arquivos – aos servidores através de troca de mensagens.

Embora associar cada servidor a um arquivo facilite a implementação do serviço, restringe a capacidade de tolerância a falhas do sistema distribuído. Se é desejado um serviço tolerante a falhas, os arquivos podem ser replicados em diferentes servidores, que representam pontos de falha independentes. Se uma das réplicas é perdida, por falha ou desconexão de algum servidor, os clientes podem dispor de uma outra réplica do mesmo arquivo, localizada em outro servidor operacional. A gerência destas réplicas, bem como o mascaramento de falhas e controle de consistência, deve ser realizado de forma transparente para os clientes [PAS 98].

Além de permitir tolerância a falhas, a replicação de arquivos tem sido empregada para aumentar o *throughput*, quando réplicas locais são criadas à medida que o serviço as torna necessárias. A desvantagem da replicação é a dificuldade na gerência das réplicas dispersas na rede e de suas atualizações, o que precisa ser realizado por protocolos específicos. Os protocolos específicos de gerência de réplicas encaixam-se em duas principais classes: réplicas ativas (ou máquina de estados) e cópia primária (ou primário-backup), as quais são extensivamente estudadas por Guerraoui e Schiper [GUE96] e [GUE97].

No atual trabalho, a replicação ocorre com relação ao histórico de mudanças
de visão armazenado num arquivo criado para tal fim. A técnica utilizada engloba algumas características da primário-backup, onde os participantes do grupo enviam mensagens para o coordenador e o mesmo é o responsável pela atualização e repasse das informações aos demais membros do grupo. Quando ocorre mudança de visão, o coordenador repassa aos participantes o arquivo inteiro da atualização, o que não respeita uma das características relativas ao primário-backup, porém, o mesmo engloba a filosofia de controle de forma centralizada, pois o coordenador do grupo mantém o arquivo com o histórico de mudanças de visão, e repassa aos demais membros participantes este arquivo, mantendo todos sincronizados com a mesma visão (sincronismo virtual). Uma referência que define as atualizações dos clientes, repassando o arquivo inteiro é a de [ZOU 98].

Quando um novo membro deseja se unir ao grupo, ele faz uma invocação ao coordenador pedindo permissão para juntar-se ao mesmo. Caso o novo membro seja aceito, onde para isso sua identificação deve ser diferente de todos os que já pertencem ao grupo, o coordenador atualiza a sua visão com o novo membro, e repassa a visão atualizada a todos os membros já pertencentes ao grupo, e também ao novo participante. Existem várias outras técnicas de replicação existentes, como a técnica de réplica ativa, onde não há controle centralizado, e o cliente faz invocações a todas as réplicas, cada réplica processa o pedido, atualiza seu estado e envia a resposta de volta para o cliente, dentre outras técnicas existentes que combinam primário-backup e réplica ativa com algumas propriedades a mais. Para maiores informações sobre técnicas de replicação, uma referência em português é a dissertação recente de Ferreira Filho [FER 2000]. Para um estudo bastante completo, uma referência adequada é [GUE 96 GUE 98].

# 4 Programação Distribuída em Java

Neste capítulo, é abordada a linguagem de programação utilizada para o desenvolvimento do protótipo no que se refere a características gerais, algumas particularidades relativas à Internet e a programação distribuída.

## 4.1 Linguagem de programação Java

Para o desenvolvimento do protótipo visando aplicar os conceitos de programação distribuída estudados, foi utilizada a linguagem de programação Java. Diversas razões amparam a escolha de Java como linguagem de implementação do protótipo, destacando-se:

1) facilidades oferecidas para o desenvolvimento de aplicações distribuídas (ver seção 4.3);

2) excelente portabilidade de programas fonte e objeto, possibilitando a sua fácil integração com outros trabalhos desenvolvidos no âmbito do PPGC ([BER 2000], [FER 2000], [AMA 2000], entre outros);

3) como representante típica do modelo de objetos, Java força a modularização de programas, visto que a classe é a estrutura básica de projeto e a unidade de compilação;

4) Java oferece segurança de execução na forma de um sistema de tratamento de exceções, não permitindo que exceções e erros detectados pelo ambiente de execução sejam desconsiderados pelo programador.

A portabilidade da linguagem Java é obtida através da geração de código não nativo de alguma plataforma específica. Seu compilador gera um código independente de máquina, chamado *bytecode* que, no momento da execução, é interpretado por uma máquina virtual a qual oferece um ambiente de execução conhecido como JRE – *Java Runtime Enviroment*. Programas Java também podem ser transportados via redes de computadores e executados através de navegadores (*browsers*) comerciais que possuam os respectivos interpretadores. Além de ser integrada à Internet, Java também é uma excelente linguagem para desenvolvimento de aplicações em geral. Dá suporte ao desenvolvimento de software em larga escala ([CES 98] e [DAM 96]).

## 4.2 Invocação remota de métodos ou RMI

A programação distribuída em Java possui diferentes mecanismos como *sockets*, integração com CORBA, *Servlets*, dentre outros. Neste trabalho, usou-se essencialmente Invocação Remota de Métodos (RMI), que possui os recursos necessários para a implementação e aplicação dos conceitos estudados. O RMI é comparável às chamadas remotas de procedimentos (ou *Remote Procedure Call*, RPC) usadas para as linguagens imperativas, sendo o RMI aplicável às linguagens orientadas a objetos. Um objeto faz a chamada a um método de um objeto em outra máquina e obtém um resultado de forma similar a uma chamada local. Assim como a maioria dos sistemas de RPC, o RMI requer que o objeto cujo método está sendo invocado (o servidor) esteja ativo e em execução ([WUT 98], [DEI 2000]).

Aplicações RMI são freqüentemente compostas por dois programas separados: um servidor e um cliente. Uma aplicação típica de servidor cria alguns objetos remotos, faz referências para obter acesso a eles, e aguarda que clientes invoquem métodos destes objetos remotos. Uma aplicação típica de cliente possui referências remotas para um ou mais objetos remotos no servidor e então invoca os métodos destes objetos remotos como se fossem locais. Estas aplicações são referenciadas como aplicações de objetos distribuídos ([CLEM 99], [SUN 99]).

Aplicações de objetos distribuídos envolvem objetos remotos e comunicação entre estes objetos como elementos principais:

Objetos remotos: estes objetos devem ser capazes de disponibilizar serviços a objetos residentes em diferentes máquinas. Java RMI possibilita um serviço de registro/localização de referências a objetos remotos. O registro é feito pelo servidor e a localização refere-se a consultas de clientes que desejam obter as referências remotas que possibilitem a invocação de métodos.

Para a comunicação com objetos remotos, quando um método remoto é invocado, há necessidade de passagem de parâmetros e/ou recebimento de resultados. Esta comunicação é feita pelo Java RMI de forma transparente ao programador da aplicação.

Métodos remotos são definidos por interfaces remotas, isto é, uma interface remota define um conjunto de métodos que podem ser chamados remotamente. Qualquer objeto que permita o acesso remoto a alguns dos seus métodos, deve torná-los disponíveis através de uma ou mais interfaces remotas.

Para criar um objeto cujos métodos possam ser chamados remotamente (um objeto servidor), deve ser criada uma interface remota. Todas as interfaces remotas devem estender a interface `java.rmi.Remote`. Cada método em uma interface remota pode devolver uma exceção `java.rmi.RemoteException`. Esta exceção é devolvida pelo sistema RMI subjacente sempre que houver um erro no envio ou recebimento de informação.

Um objeto que dispõe de uma interface remota é chamado de servidor. Um objeto que chama um método remoto é chamado de cliente. Um objeto pode ser ao mesmo tempo cliente e servidor. Esses nomes apenas indicam quem está chamando em um determinado momento e quem está sendo chamado.

Depois de definir a interface remota e criar um objeto correspondente, ainda é necessário saber o modo pelo qual o cliente pode invocar os métodos no servidor. Para isto, é necessário criar um *stub* para o cliente. Um *stub* de objeto é uma visão remota do objeto que contém somente os métodos remotos do mesmo. O *stub* é executado no lado do cliente e representa o objeto remoto no espaço de dados do cliente.

O cliente invoca métodos no *stub* e ele então invoca os métodos no objeto remoto. Isto permite a qualquer cliente invocar métodos remotamente por meio de invocação de métodos normal do Java. Um *stub* é também chamado de *proxy*. A figura 4.1 ilustra a relação entre um cliente, um servidor e um *stub*.

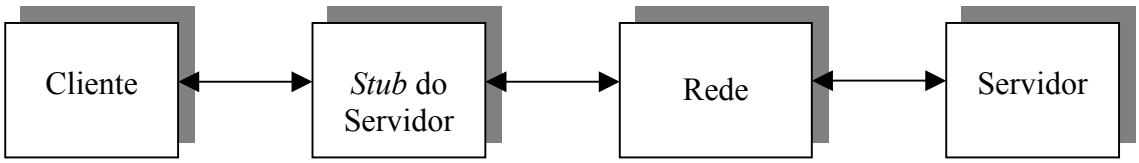


FIGURA 4.1- Invocação de métodos pelo *stub* a partir do cliente [WUT 98]

O RMI por ser um mecanismo implementado de acordo com o modelo de objetos, acrescenta um recurso extra que a maioria dos sistemas RPC não possui. Os objetos remotos podem ser passados como parâmetros nas chamadas de métodos remotos. Quando se passa um objeto remoto como um parâmetro, na verdade é passado um *stub* para o objeto.

Um servidor pode tornar-se um cliente desde que receba um *stub* que permita fazer invocações remotas. A figura 4.2 mostra como um cliente passa um *stub* para um servidor, de modo que o servidor possa invocar métodos no cliente.

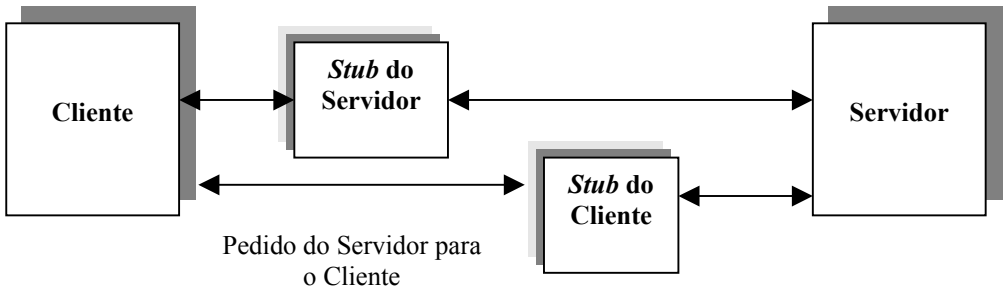


FIGURA 4.2- Um cliente passa um *stub* para o servidor [WUT 98]

Em um sistema distribuído, figura 4.3, os clientes devem encontrar os servidores que precisam. O RMI fornece um objeto de busca de nome simples que permite ao cliente obter um *stub* para um determinado servidor baseado no nome do servidor. O serviço de nomeação que vem com o sistema RMI é bastante simples, porém muito útil em vários casos. A figura 4.3 mostra como um cliente usa o serviço de nomeação para encontrar um servidor [WUT 98].

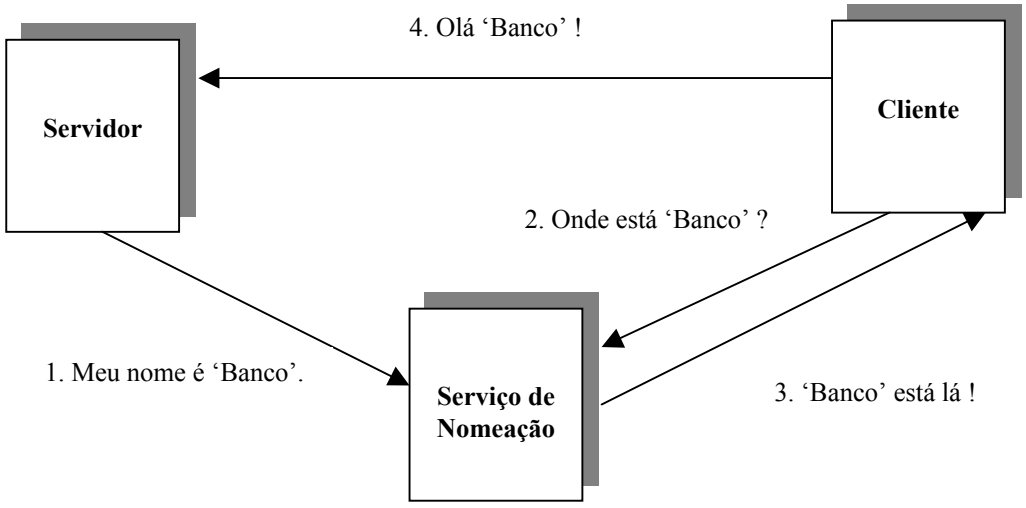


FIGURA 4.3- O serviço de nomeação para localizar um servidor [WUT 98]

# 5. Sistema Proposto

Neste capítulo demonstra-se a estrutura geral do sistema proposto e seus módulos componentes, os algoritmos utilizados, além de mostrar a avaliação e os detalhes de implementação, e também os testes efetuados. Trabalha-se com a hipótese de um ambiente síncrono, com falhas de colapso.

## 5.1 Estrutura geral e seus módulos componentes

O protótipo desenvolvido, denominado SVgroup, faz o controle de membros de um grupo de processos distribuídos em uma rede de computadores. Este grupo é hierárquico e dinâmico. A hierarquia existe devido a um processo chamado de coordenador, que faz o controle da entrada e da saída de membros no grupo, além de monitorar as falhas do grupo. Ele possui também a característica de replicação de objetos, onde todos os processos (membros) que fazem parte do grupo possuem as mesmas características, permitindo com isso fazer com que qualquer membro possa se tornar o coordenador do grupo (caso ocorra um defeito no mesmo).

Este protótipo pode ser executado em qualquer rede de computadores, onde os computadores podem possuir qualquer nome, sendo que no momento da execução do aplicativo deve-se especificar o nome da máquina que deve exercer a função de coordenador do grupo. No primeiro instante de execução, a aplicação faz um questionamento se a máquina em uso é o coordenador ou não. Caso ela seja, então a aplicação mostra um menu de opções; porém, caso a máquina não seja o coordenador, então ela deve informar sua identificação, para que o coordenador possa armazenar o seu nome.

O coordenador é que faz a gerência dos serviços de comunicação do grupo. Os clientes interagem apenas com o coordenador; e excepcionalmente um cliente interage com outro cliente (setas tracejadas), mas isto acontece apenas quando da suspeita de defeito do coordenador, forçando a designação de um novo líder. Este procedimento de designação exige com isto que exista a troca de mensagens entre os clientes que estão ativos no grupo, conforme é mostrado na figura 5.1. No monitor de vídeo do coordenador são mostradas mensagens que indicam a inclusão de novos membros no grupo e as identificações dos novos membros. Além disto, mostram também mensagens indicando a identificação de membros que, por algum motivo, desejaram abandonar o grupo.

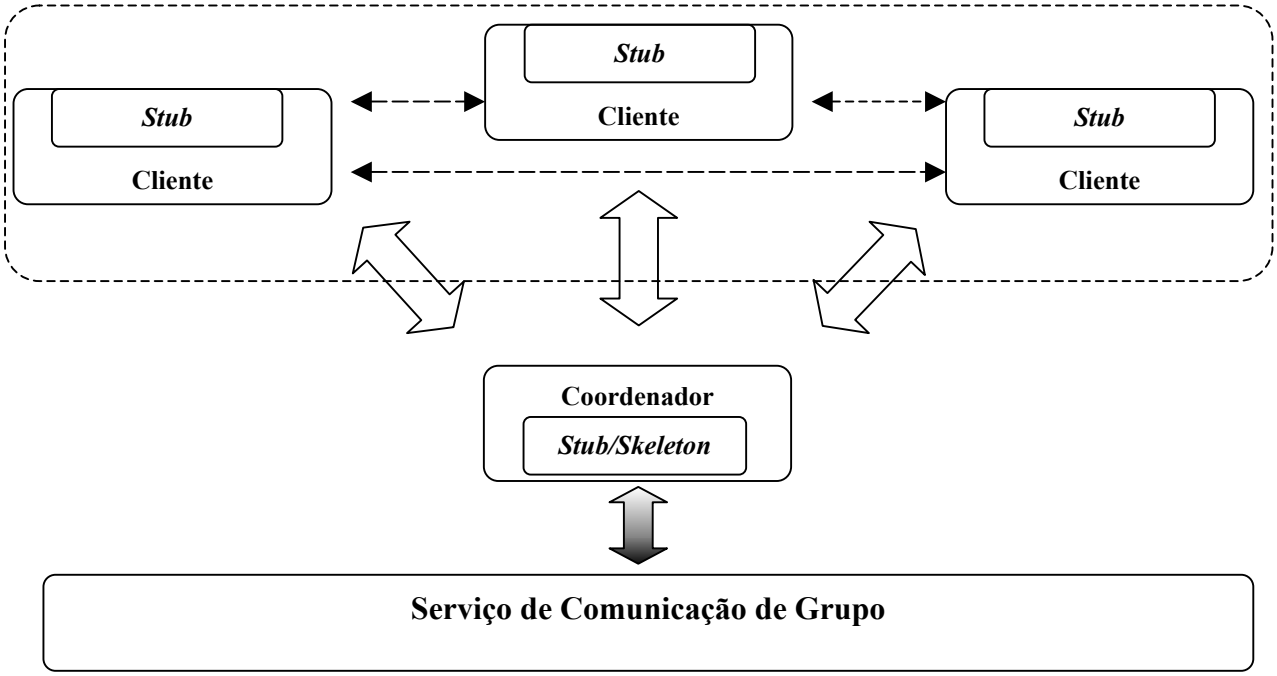


FIGURA 5.1- A arquitetura do protótipo SVgroup

O objetivo do desenvolvimento do protótipo é o de demonstrar o cumprimento das propriedades definidas pelo sincronismo virtual, realizando a atualização de visão do grupo em todos os membros pertencentes ao mesmo, além de tolerar falhas em qualquer dos membros do grupo, incluindo o seu coordenador.

Cada membro do grupo possui um arquivo texto com as informações correspondentes ao histórico de visões do grupo. Este arquivo é sempre atualizado pelo coordenador quando ocorre alguma troca de visão, ou seja, por razões que podem ser: falha, saída de um dos membros já pertencentes ao grupo, ou pela união de um novo.

Caso o coordenador falhe, outro membro assume a coordenação do mesmo, avisando todos os membros ativos que é o novo coordenador. A figura 5.2 dá uma visão geral do funcionamento do serviço de gerência de membros do SVgroup. Os elementos gráficos representados por retângulos identificados pelas letras A, B e C, representam os processos que compõem o grupo de membros. O processo A tem uma hierarquia superior em relação a B e C, ele tem o papel de coordenador do grupo, seguindo a definição do sincronismo virtual usado no ISIS [GUO 96]. Cada membro possui uma visão local dos membros que estão ativos no grupo. Qualquer operação a ser realizada por qualquer membro do grupo deve ser sempre enviada ao coordenador, que tem a responsabilidade de repassar para os outros membros do grupo. Isto assegura o determinismo nas alterações realizadas pelos membros do grupo e ordenação total na seqüência de visões.

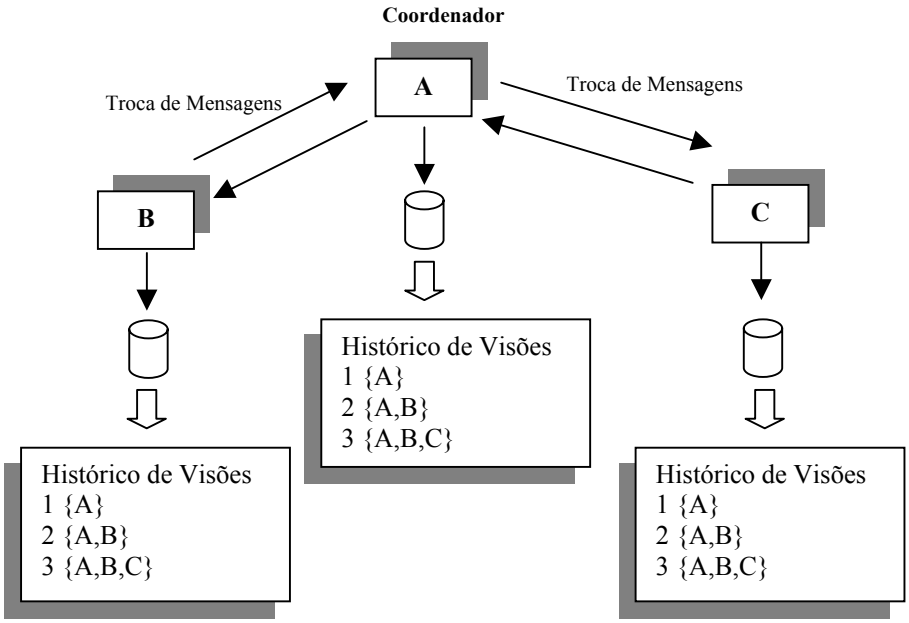


FIGURA 5.2 - Visão geral do protótipo

## 5.2 Algoritmos utilizados

Neste capítulo são descritos 13 (treze) algoritmos que foram elaborados para a estrutura lógica implementada em Java RMI.

O primeiro algoritmo a ser mostrado é o `Processo` (figura 5.3), que é o algoritmo principal do projeto. Através dele, são dadas as opções pelas quais se pode acessar todos os serviços disponibilizados pelo sistema.

No início do algoritmo, é verificado se o gerente de segurança do sistema já está ativo. Logo após, configura-se a variável `chamada_remota` com o nome da estação que foi especificada como parâmetro na linha de comando. Se o valor da variável `argumentos` for diferente de 1 é porque, ou não foi especificado o nome do coordenador do grupo, ou então foi passado mais de um parâmetro, acarretando a emissão de uma mensagem de erro e a finalização do sistema. Caso o número de parâmetros esteja correto, é questionado se a máquina é a coordenadora ou não. Se a mesma funcionar como coordenador, é executado o método `unir_ao_grupo` passando como parâmetro a identificação da máquina.

Caso a identificação não faça parte do grupo, então permite-se a entrada, e é exibido um menu de opções com alguns serviços disponíveis para os usuários do grupo. O menu contém algumas opções, que serão escolhidas a partir da entrada de um número que corresponda a uma delas:

1) Enviar mensagem em *Multicast*

2) Enviar mensagem em *Unicast*

3) Sair do Grupo

4) Visualizar Membros do Grupo

5) Monitorar Falhas

6) Finalizar

A primeira opção permite o envio de mensagens a todos os processos ativos do grupo. A segunda opção permite o envio de uma mensagem a um membro específico do grupo, e é usada quando se deseja passar uma mensagem que não interessa a todos os membros do grupo.

A terceira opção é utilizada quando um membro participante deseja sair do grupo. A próxima opção (4ª) é utilizada para visualizar os membros ativos no grupo, e também mostra o histórico de mudanças de visão.

A quinta opção faz o monitoramento de falhas, enviando mensagens aos membros ativos para poder detectar suspeitas de falhas, e se necessário atualizar o histórico de visões. Por fim a última opção deve ser utilizada para finalizar a aplicação.

Em todos os algoritmos, existem as palavras-chave que estão sempre em negrito e sublinhadas. Deve-se dar destaque à palavra chave `Tentar` que é utilizada para proteger determinadas linhas de código. Caso ocorra uma exceção, a mesma é capturada e a execução é desviada para `Capturar Erro`, onde efetua-se o tratamento do defeito.

```
Algoritmo Processo
    Declare identificação Literal
    Declare valor Numérico
    Declare Chamada_remota Serviço de Acesso Remoto a Métodos
    Declare visao[10], linhas [100] literal
    Se Gerenciador de Segurança do Sistema = vazio então
    |   Gerenciador de Segurança do Sistema (new RMISecurityManager());
    Fim Se
    Processo svr = new Processo(); {Cria uma instância dos serviços disponíveis }
    Naming.bind ("Servicos", svr); {e liga-o com o rmi registry }
    visao[0] ← argumento da linha de comando [0]
    Chamada_remota = (Servicos) Naming.lookup
            ("rmi://" + visao[0] + "/Servicos");
    // Checar a informação do argumento que referencia o Coordenador
    Se (argumentos.tamanho != 1)
    |  então  Escreva "Entrada Inválida. Deve-se informar o nome do Coordenador"
    |                 Sair do Sistema()
    Fim se
    Escreva "Processo Coordenador (S-Sim/N-Não) ? ");  Leia identificacao
    Se (Maiúscula(identificacao) = "S")  então
    |    Escreva "Coordenador do Grupo PANTANAL Ativo!"
    |    Escreva "Iniciado o Grupo: " +  chamada_remota.unir_ao_grupo(args[0])
    |  Senão
    |    Escreva "Membro do Grupo PANTANAL!"
    |    Escreva "Digite a Identificacao do Processo: " , Leia identificacao
    |    Enquanto (chamada_remota.unir_ao_grupo(identificacao)
    |    |                = "ABORT")
    |    |                Escreva "IDENTIFICACAO JA EXISTE NO GRUPO";
    |    |                Escreva "Digite outra Id p/ o Processo: " ; Leia identificacao
    |    Fim Enquanto
    Fim se
    Escreva "CONEXÃO EFETUADA COM SUCESSO"
    Escreva "Digite a Opção Desejada:"
    Enquanto (Verdadeiro) Faça { Laço infinito – Em Java for (;;) }
    |    Escreva "1 - Enviar Mensagem em Multicast"
    |    Escreva "2 - Enviar Mensagem em Unicast"
    |    Escreva "3 - Sair do Grupo"
    |    Escreva "4 - Visualizar Membros do Grupo"
    |    Escreva "5 – Monitorar Falhas"
    |    Escreva "6 - Exit"
    |    Escreva "Digite a Opção Desejada: " ; Leia  valor
    |    Escolha (valor)
    |    |    Caso 1: Algoritmo Enviar_Multicast
    |    |    Caso 2: Algoritmo Enviar_Unicast
    |    |    Caso 3: Algoritmo Sair_do_Grupo
    |    |    Caso 4: Algoritmo Visualizar_Membros_do_Grupo
    |    |    Caso 5: Algoritmo Monitor_de_Falhas
    |    |    Caso 6: Sair()
    |    Fim Escolha
    Fim Enquanto
Fim Algoritmo
```

FIGURA 5.3- Algoritmo Principal - Processo

Caso seja escolhida a opção 1 do algoritmo `Processo`, é então executado o algoritmo `Enviar_Multicast` (figura 5.4), que faz a leitura de uma determinada mensagem. Logo após é executada a chamada remota ao método enviar_mensagem; caso ocorra alguma exceção, significa que o coordenador é suspeito de defeito. Em consequência é efetuada a reconfiguração da variável `chamada_remota` com o nome da próxima máquina armazenada na posição 1 do vetor. É decrementado o `Número_de_membros_do_grupo`, e é efetuada a chamada ao método `redirecionar_coordenador` para que o novo Coordenador possa repassar a todos os membros do grupo sua identificação.

```
Algoritmo Enviar_Multicast
    Declare mensagem Literal
    Declare Chamada_remota Serviço de Acesso Remoto a Métodos
    Declare visao[10] literal
    Se Gerenciador de Sergurança do Sistema = vazio então
        Gerenciador de Segurança do Sistema (new RMISecurityManager());
    // Cria uma instancia dos servicos disponiveis no servidor (habilita o acesso a seus
métodos)
    Processo svr = new Processo();
    // ... e liga-o com o rmi registry
    Naming.bind ("Servicos", svr);
    { visao[0] ← argumento da linha de comando [0] }
    Chamada_remota = (Servicos) Naming.lookup
            ("rmi://" + visao[0] + "/Servicos");
    mensagem  ← vazio
    Escreva "Entre com a mensagem a ser enviada aos membros do grupo:"  ; Leia
mensagem
            Tentar
                Escreva "Enviando Mensagem ao Coordenador: "
                Escreva "Resposta do Coordenador: ",
                Chamada_remota.Enviar_mensagem(mensagem)
                Capturar Erro
                        Escreva "Coordenador Suspeito de Falha"
                        Escreva "Redirecionamento do Coordenador"
                        { Chamada_remota recebe como novo Coordenador, o nome da
                        máquina da Segunda posição do vetor (Nome_novo_coordenador))
                        Chamada_remota ← (Servicos) Naming.lookup
                        ("rmi://" + visao[1] + "/Servicos");
                        Escreva "Redirecionar chamadas remotas "
                        Remover (0 , Numero_de_Membros_do_Grupo)
                        Incrementa Numero_de_Membros_do_Grupo
                        Chamada_remota.redirecionar_coordenador()
                Fim Capturar Erro
            Fim Tentar
Fim Algoritmo
```

FIGURA 5.4- Algoritmo Enviar em *Multicast*

O algoritmo `Enviar_Mensagem(m)` recebe, em uma variável de nome "**m"**, a mensagem enviada como parâmetro na chamada do método. Este método faz a chamada aos métodos **Monitor_de_Falhas** e **Entregar_Mensagem**, conforme é mostrado na figura 5.5.

```
Algoritmo Enviar_Mensagem(m)
    Monitor_de_Falhas(Numero_de_Membros, visao[tamanho])
    Entregar_Mensagem_aos_Membros_do_Grupo (m)
Fim Algoritmo
```

FIGURA 5.5- Algoritmo Enviar Mensagem

O algoritmo `Entregar_Mensagem_aos_Membros_do_Grupo` tem por objetivo receber uma mensagem como parâmetro e processá-la, e reenvia a mensagem enviada a todos os membros participantes do grupo. Uma variável de nome `contador` é declarada como numérico, e é inicializada com o valor 0 (zero). Uma estrutura de repetição tem como condição de execução que o conteúdo de contador seja menor que o `Numero_de_Membros_do_Grupo`. Se contador for menor que `Numero_de_Membros_do_ Grupo` então é inicializado o objeto `serviço` com o nome da estação que está armazenada na posição referente ao conteúdo de contador. Escreve-se uma mensagem que é composta dos literais (em cadeia) "Entrega ao processo" concatenada com o conteúdo retornado pela execução remota do método processar_mensagem. O contador é incrementado e, se o seu valor continuar sendo menor que o número de membros do grupo, repete-se todo o processo para o nome da estação armazenada na posição correspondente do vetor, conforme é mostrado na figura 5.6.

```
Algoritmo Entregar_Mensagem_aos_Membros_do_Grupo (m)
    Escreva "Mensagem Enviada: " , m
    Processa_Mensagem(m) { Efetua Gravação em Arquivo Texto, por exemplo}
    // Confiabilidade do multicast confiável 19/01/2001
    j = 1; // j comeca valendo 1 para nao envio da mensagem ao coordenador
    enquanto (j < Numero_de_Membros_do_Grupo) faça
        Tentar
            Servicos servico = (Servicos) Naming.lookup("rmi://" +
                                visao[j] + "/Servicos");
            System.out.println ();
            System.out.println (servico.reenviar(m));
        Capturar
            System.out.println("Objeto Monitorado " + visao[j] + "
                                            Suspeito de Falha");
            Incrementa j
        Fim Capturar
    Fim Tentar
    Fim enquanto
    Retorne ("Mensagem Entregue com Sucesso")
Fim Algoritmo
```

FIGURA 5.6 - Algoritmo Entregar Mensagens aos Membros do Grupo

O algoritmo `Processar_Mensagem(m)`, visto na figura 5.7, recebe como parâmetro uma mensagem que é recebida em uma variável chamada `m`, faz o processamento da mesma, e retorna uma cadeia (string) indicando a entrega.

```
Algoritmo Processar_Mensagem (m)
    Escreva "Mensagem Enviada: " , m
    Processa_Mensagem(m)
    Retorne ("Mensagem Entregue com Sucesso")
Fim Algoritmo
```

FIGURA 5.7 - Algoritmo Processar Mensagem

O algoritmo `Monitor_de_Falhas(Numero_de_Membros_do_Grupo ,visao[tamanho])`, executa uma chamada ao método **sinalizar** em cada uma das estacões que fazem parte do grupo, cujos nomes estão armazenados em cada uma das posições do vetor. Se ocorrer uma exceção, é indicada a suspeita de falha e o nome da estação é retirado, atualizando a visão nas outras estações da rede, conforme é mostrado na figura 5.8.

```
Algoritmo Monitor_de_Falhas (Numero_de_Membros_do_Grupo,visao[tamanho])
    Declare contador Numérico
    Contador ← 0
    Enquanto (Contador < Numero_de_Membros_do_Grupo) Faça
        Tentar
            Servicos servico = (Servicos) naming.lookup("rmi://"+visao[contador]+
            "/Servicos"
            Escreva "Processo " + visao[contador] + " está ativo ? " ,
        servico.sinalizar()
            Capturar Erro
                Escreva "Processo " + visao[contador] + " Suspeito de Falha"
                Remover (contador , Numero_de_Membros_do_Grupo)
                Incrementa Numero_de_Membros_do_Grupo
                Gravar em arquivo Texto (visao)
                Atualiza_Visao_aos_Membros(visao)
                Atualiza_visao()
            Fim Capturar Erro
        Fim Tentar
        Contador ← Contador + 1
    Fim Enquanto
Fim Algoritmo
```

FIGURA 5.8 - Algoritmo Monitor de Falhas

O algoritmo `Remover(Indice)` remove o nome da estação que foi suspeita de falha, de acordo com o valor do índice que é passado como parâmetro. Descreve-se o mesmo na figura 5.9.

```
Algoritmo Remover(Indice)
    Declare contador Numérico
    Contador ← indice
    Enquanto (Contador < Numero_de_Membros_do_Grupo) Faça
        Visao[contador] ← Visao[contador + 1]
        Incrementa Contador
    Fim Enquanto

Fim Algoritmo
```

FIGURA 5.9 - Algoritmo Remover

O algoritmo `Redirecionar_Coordenador(nome_do_novo_coordenador)`, executa a chamada ao método `novo_caminho`, que configura o nome do novo coordenador do grupo para as outras estações pertencentes ao grupo, ver figura 5.10.

```
Algoritmo Redirecionar_Coordenador (nome_do_novo_coordenador)
    Declare contador Numérico
    Contador ← 0
    Enquanto (Contador < Numero_de_Membros_do_Grupo) Faça
        Tentar
            Servicos servico =
            (Servicos) naming.lookup("rmi://"+ visao[contador] + "/Servicos")
            Escreva "Processo " + visao[contador] + " está ativo ? " ,
            servico.novo_caminho(nome_do_novo_coordenador)
        Capturar Erro
            Escreva "Processo " + visao[contador] + " Suspeito de Falha"
            Remover (contador , Numero_de_Membros_do_Grupo)
            Incrementa Numero_de_Membros_do_Grupo
            Gravar em arquivo Texto (visao)
            Atualiza_Visao_aos_Membros(visao)
        Fim Capturar Erro
    Fim Tentar
    Incrementa Contador
    Fim Enquanto
Fim Algoritmo
```

FIGURA 5.10 - Algoritmo Redirecionar Coordenador

O algoritmo `Unir_ao_Grupo (Identificacao)`, verifica, através da identificação, se a máquina que deseja se unir ao grupo já não pertence ao mesmo; se já pertencer, retorna uma mensagem indicando o impedimento de nova entrada no grupo. Caso a identificação não corresponda a algum membro do grupo, então a mesma é armazenada em uma nova posição do vetor, incrementa-se o contador de mudanças de visão e o `Número_de_membros_do_grupo`, faz-se a gravação em arquivo texto da nova visão, já com o novo membro, efetua a chamada ao método para atualizar a nova visão nos outros membros, e retorna a mensagem "Elemento unido ao grupo com sucesso !", e pode-se observar sua estrutura sequencial na figura 5.11.

```
Algoritmo Unir_ao_Grupo(Identificacao)
    Declare n, Numero_de_Membros Numérico
    n=0;
    Enquanto (n< Numero_de_Membros_do_Grupo) Faça
        Se (visao[n].equals(identificacao)) = Verdadeiro
            então
                Escreva "PROCESSO JA EXISTENTE";
                Retorne ("Cancelada a junção ao grupo")
        Fim Se
        Incrementa n
    Fim Enquanto
    visao[n] = identificacao;
    Incrementa n; // Incrementa o contador de mudanças de visao
    Incrementa Numero_de_Membros_do_Grupo
    Gravar em arquivo Texto (visao)

    Atualiza_Visao_aos_Membros(visao)
    Retorne ("Elemento unido ao grupo com sucesso !")
Fim Algoritmo
```

FIGURA 5.11 - Algoritmo Unir ao Grupo

O algoritmo `Novo_Caminho (novo)` reconfigura a variável `chamada_remota` com o nome do novo coordenador (figura 5.12).

```
Algoritmo Novo_Caminho(novo)
    Declare Chamada_remota Serviço de Acesso Remoto a Métodos
    Tentar
        Chamada_remota = (Servicos) Naming.lookup
        ("rmi://" + novo + "/Servicos");
    Capturar Erro
        Escreva "Objeto Suspeito de Falha"
    Fim Capturar Erro
    Fim Tentar
Fim Algoritmo
```

FIGURA 5.12 - Algoritmo Novo Caminho

O algoritmo `Sair_do_Grupo(identificacao)`, figura 5.13, recebe como parâmetro a identificação do membro que deseja abandonar o grupo, verifica se a mesma corresponde a alguma identificação realmente existente no grupo, armazena no seu lugar a identicação armazenada na próxima posição do vetor. Em seguida, procede à atualização de visão correspondente.

```
Algoritmo Sair_do_Grupo (identificacao)
    Declare encontrou, n, i  numérico;
    encontrou = 0
    N = 0
    enquanto (n < Numero_de_Membros_do_Grupo) faça
        se (visao[n].equals(identificacao)) então
            Escreve "PROCESSO ENCONTRADO!"
            visao [n] = null
           // Remover
            i = n ;
            enquanto (i < Numero_de_Membros_do_Grupo) Faça
                visao[i] = visao[i+1]
                Incrementa i
            Fim Enquanto
            //
            Decrementa Numero_de_Membros_do_Grupo
            encontrou = 1;

        Fim se
        Incrementa n
    Fim enquanto

    se (encontrou = 0) então
        retorne "IDENTIFICACAO NAO ENCONTRADA !"
    senão
        Gravar em arquivo Texto (visao)
        Atualiza_Visao_aos_Membros(visao)
        retorne "Processo Removido do Grupo com Sucesso!"
    Fim se

Fim Algoritmo
```

FIGURA 5.13 - Algoritmo Sair do grupo

O algoritmo `Visualizar_Membros_do_Grupo`, figura 5.14, permite a visualização de todos os membros ativos do grupo, e também o conteúdo armazenado no arquivo texto, que corresponde ao histórico de mudanças de visão.

```
Algoritmo Visualizar_Membros_do_Grupo
    Declare n, Numero_de_Membros Numérico
    J ← 0;
    Escreva "Membros do Grupo"
    J ← 0
    Enquanto (j < Numero_de_Membros_do_Grupo) Faça
        Escreva visao[j]
        Incrementa J
    Fim Enquanto
    Escreva "Histórico de Visões"
    Mostrar_Conteúdo_Arquivo_Texto

Fim Algoritmo
```

FIGURA 5.14 - Algoritmo Visualizar Membros do Grupo

O algoritmo `Enviar_Unicast(String mensagem, String visao[], int n_membros, String identificacao)`, figura 5.15, recebe como parâmetros a mensagem a ser enviada, a visão atual do grupo, o número de membros e a identificação de quem deve receber a mensagem. Uma variável de nome `servico` é configurada com a identificação do destinatário da mensagem. Depois, é efetuada a chamada a uma função remota de nome `entregar_mensagem`, que retorna um conjunto de caracteres indicando que a mensagem foi entregue com sucesso. Caso não seja possível enviar a mensagem, ocorre uma exceção, e é emitida uma mensagem indicando que o processo para o qual se desejava enviar a mensagem é suspeito de falha.

```
Algoritmo Enviar_Unicast (String mensagem, String visao[], int n_membros,
                          String identificacao)

  Tentar
      Servicos servico = (Servicos) Naming.lookup("rmi://" + identificacao + "/Servicos");
      Escreva servico.entregar_mensagem(mensagem)
    Capturar Erro
        Escreva "Não foi possível enviar a mensagem"
        Escreva "Processo " + identificacao + " Suspeito de Falha";
        Retorne ("Não foi possível efetuar o envio da mensagem !");
    Fim Capturar Erro
  Fim Tentar

  Retorne ("Ok");

Fim Algoritmo
```

FIGURA 5.15 - Algoritmo para Envio de Mensagem em *Unicast*

O algoritmo `Reenviar (String m)` recebe uma mensagem, e escreve que a mensagem foi reenviada, e retorna uma mensagem que a mesma foi entregue, figura 5.16.

```
Algoritmo Reenviar(String m)
          Escreva "Reenvio de Mensagem Multicast Enviada para o Grupo
                   PANTANAL: " + m);
          Retorne ("Mensagem Entregue! " + m);
Fim Algoritmo
```

FIGURA 5.16 - Algoritmo reenviar

Pode-se expressar a aplicação destes algoritmos através de diagramas de estado (figuras 5.17 e 5.18), onde pode-se demonstrar a troca de mensagens entre os membros participantes do grupo. Na figura 5.17 demonstra-se através de estados para entrada/saída de um membro do grupo. A figura 5.18 demonstra os estados dos membros participantes do grupo quando ocorre requisição para o envio de mensagens e também o monitoramento sobre a suspeita de falhas.

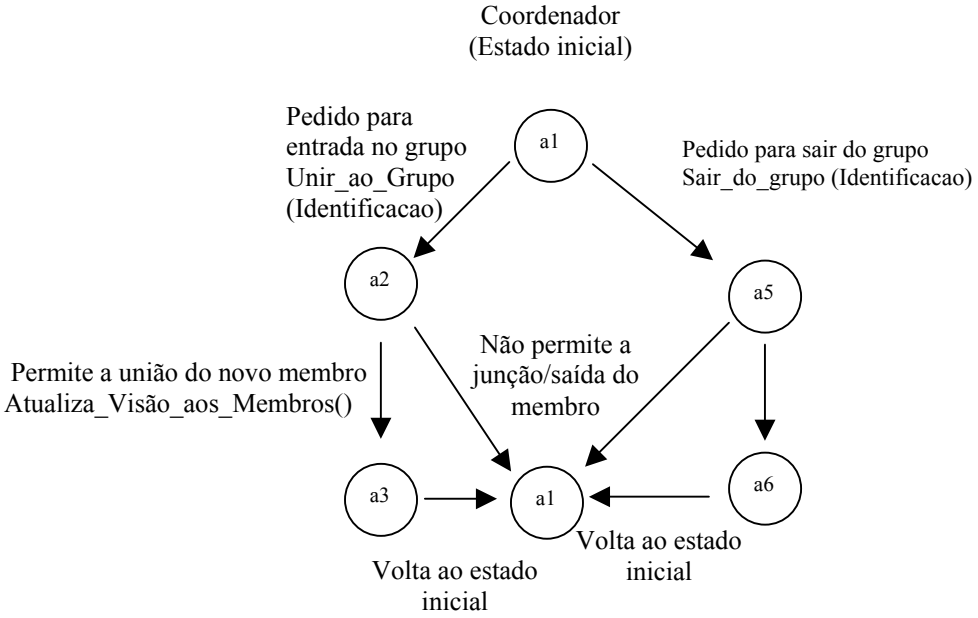


FIGURA 5.17 - Diagrama de estados de entrada/saída do grupo

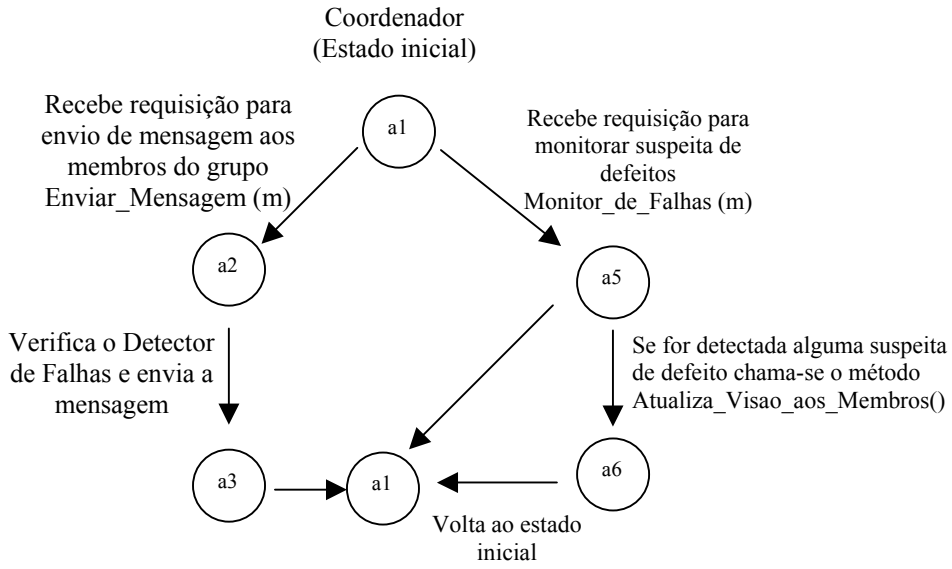


FIGURA 5.18 - Diagrama de estados de controle de mensagens e defeitos

## 5.3 Comentários sobre a implementação

A aplicação foi projetada para funcionar em um ambiente de rede de computadores, onde as máquinas são nomeadas. Deve-se escolher qualquer uma delas para funcionar como coordenadora do grupo: a única restrição é que ela deve ser iniciada em primeiro lugar. O sistema identifica a escolha do coordenador pelo nome que vem na linha de comando para a execução do aplicativo. Por exemplo, se a aplicação for executada em uma máquina de nome UFRGS, digita-se, na linha de comando:

C:\> java Processo **UFRGS**

As outras máquinas que desejarem participar do grupo, devem executar a mesma linha de comando. Por exemplo, supondo que existam mais três máquinas que desejam se unir ao grupo, uma chamada POA, outra chamada SANTA_MARIA, e por fim uma outra chamada PANTANAL, estas três máquinas deveriam executar a linha de comando mostrada anteriormente. Assim elas indicam que desejam se unir ao grupo cujo coordenador é a estação UFRGS. A figura 5.19 esboça graficamente a estrutura física e os comandos executados em cada uma das estações existentes.

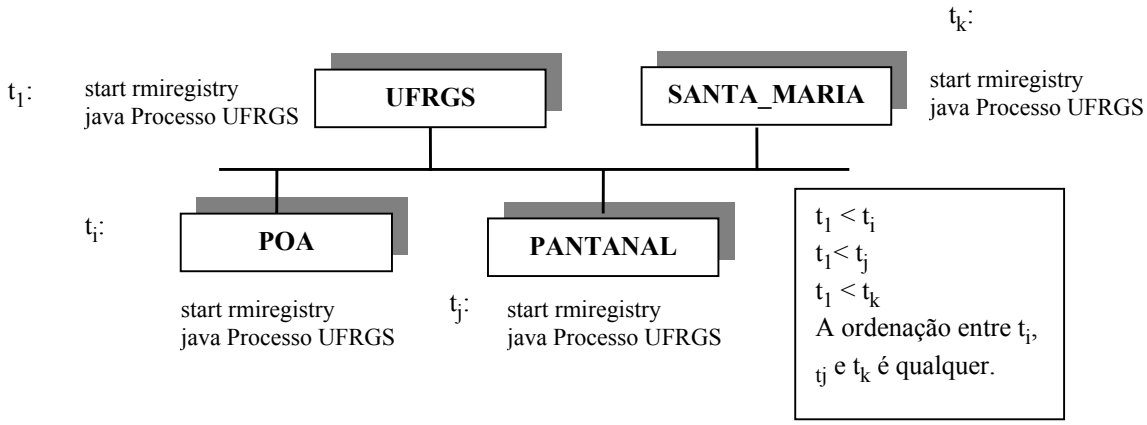


FIGURA 5.19 – Estrutura física utilizada para os testes do protótipo

O objetivo inicial proposto foi alcançado através da especificação de classes que resultaram em um sistema de monitoramento de troca de mensagens entre membros de um grupo, e de queda de qualquer um dos membros ou do coordenador. Estes processos são controlados através de visões atualizadas nos diversos membros do grupo, disponibilizando um histórico atualizado e replicado.

Na implementação do protótipo foram programadas execuções de métodos remotos de forma mútua, onde uma determinada máquina "***m1***" pode ser servidor enquanto outras "***m2,m3,m4,...,mn***" executam seus métodos. Porém, m1 pode se tornar cliente e efetuar chamadas aos métodos das outras máquinas, que passam do papel de cliente ao de servidor.

Também conseguiu-se com sucesso a replicação de objetos, em todos os membros do grupo. Esta replicação é necessária caso haja algum problema com o coordenador, possibilitandoque uma outra máquina possa assumir as suas funções, já que possui tudo que o coordenador contém.

Foram implementados tratamentos de falhas para o colapso do coordenador do grupo durante o envio de mensagens aos membros do grupo.

O *software* necessário para a execução do protótipo é composto por: Windows 9x e JDK 1.2.1. Foram desenvolvidos quatro programas fonte (4 arquivos), com extensão .java, sendo que estes programas são carregados em todas as máquinas utilizadas para os testes:

- processo.java (É o programa fonte onde estão todos códigos do conteúdo dos métodos que podem ser acessados de forma remota)
- servicos.java (Protótipo dos métodos que podem ser chamados remotamente)
- implementa_carteiro.java (É o programa fonte onde está o código dos métodos prototipados no arquivo Carteiro.java)
- carteiro.java (Protótipo de dois métodos que podem ser chamados remotamente)

Uma observação a ser feita é que para executar o arquivo processo.class no JDK 1.2.1, deve-se configurar um arquivo que define a política de acesso das estações remotas, podendo-se restringir a ação de acesso como somente leitura ou escrita. Na linha de comando deve-se informar a localização do arquivo de política de segurança, como mostrado logo a seguir:

java **-Djava.security.policy=c:\java.policy** processo <id_coordenador>

O arquivo java.policy inclui como conteúdo as seguintes linhas, sendo que neste caso o arquivo foi previamente configurado como acesso total:

```
grant {
    // Permissão total de acesso
    permission java.security.AllPermission;
};
```

Um dos problemas encontrados, durante a implementação, foi o fato dos processos executarem métodos no coordenador, e o mesmo (coordenador) executar métodos remotos nos membros do grupo (teoricamente clientes de seus serviços) para, por exemplo, poder verificar quais processos estão ativos e quais são suspeitos de falha. A implementação exige apenas que sejam executados métodos remotos que retornem uma mensagem; caso ocorra uma exceção, significa que a máquina é suspeita de defeito. Com isso, verificou-se a necessidade de iniciar um arquivo de registro para acesso a métodos localizados não só no coordenador, como também nos processos que funcionam como membros do grupo. Isto foi conseguido da forma relatada a seguir.

Para inicializar o coordenador do grupo, que é obrigatoriamente o primeiro , deve-se executar os dois comandos abaixo:

- `start rmiregistry` (ativa uma janela em segundo plano (*background*), que permite o acesso aos serviços disponibilizados pelos métodos localizados no coordenador).
- `java Processo identificação_do_coordenador_do_grupo` (executa a classe `processo.class`, que inicia o funcionamento do grupo, especificando como parâmetro a identificação da máquina que atuará como coordenadora do grupo).

Para iniciar os outros membros participantes do grupo, executam-se os mesmos comandos utilizados para iniciar o coordenador, à medida em que chegam os pedidos.

A linha a seguir trata da configuração de uma variável em Java RMI denominada "**chamada_remota",** como forma de acesso aos métodos definidos em um arquivo chamado Servicos.java, e que está localizado em uma determinada máquina remota:

- ✓ **chamada_remota** = (Servicos) Naming.lookup ("rmi://" + "**nome da máquina remota**" + "/Servicos");

Para executar uma chamada remota a um método, basta colocar o nome da variável, seguido do ponto, e do nome do método; por exemplo:

- ✓ **chamada_remota**.**unir_ao_grupo** (identificacao);

Após a formação inicial do grupo, as operações disponíveis podem ser realizadas, como exemplificado a seguir.

a) Mudança de coordenador

Quando, por exemplo, uma determinada máquina deseja enviar uma mensagem ao grupo, ela efetua uma chamada ao método `enviar_mensagem(m)` no coordenador, que verifica a visão atual e efetua a entrega da mensagem. Se por algum motivo o coordenador sofrer um colapso (*crash*), isto é detectado pelo tratamento de erros (*try* { } .. *catch* { }). Em consequência, ocorre um redirecionamento na execução das chamadas remotas aos métodos, ficando responsável a máquina cuja identificação está localizada na segunda posição do vetor chamado *visao*. Apenas para lembrar, este vetor contém o nome das estações pertencentes ao grupo, na ordem em que as mesmas uniram-se ao grupo.

b) Inserção de novo membro

Quando ocorre a execução do aplicativo, qualquer membro deve pedir permissão ao coordenador para fazer parte do grupo. O coordenador verifica se a identificação do novo membro ainda não faz parte do grupo. Caso seja realmente um novo membro, a sua identificação é armazenada, e é efetuada a gravação em disco da nova visão. É enviada uma mensagem a todos os membros ativos sobre a nova visão, fazendo com que todos mantenham a visão atualizada dos membros ativos do grupo, com o histórico de todas as visões, inclusive a visão mais atual. Caso a identificação informada pelo membro já faça parte do grupo, é enviada uma mensagem "PROCESSO JÁ FAZ PARTE DO GRUPO" (figura 5.20).

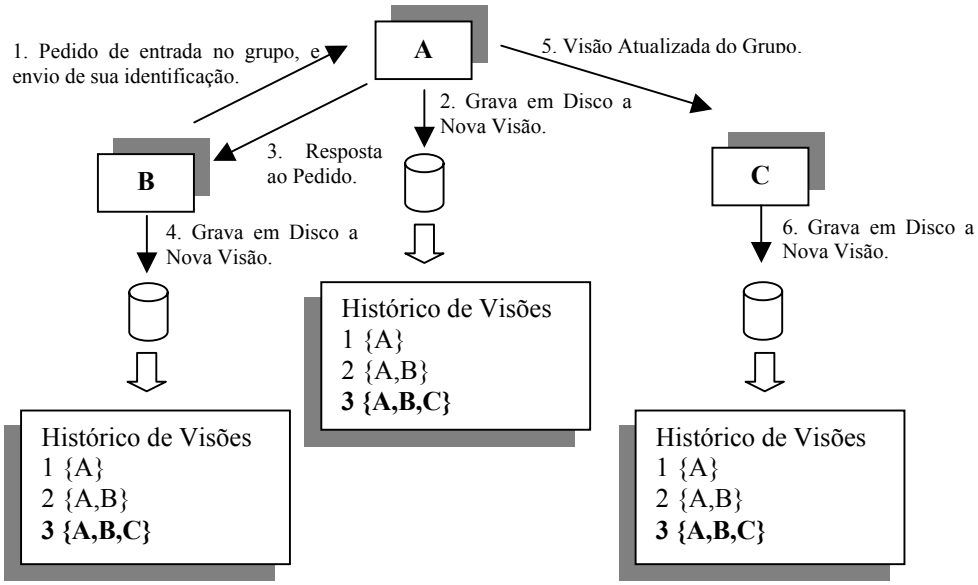


FIGURA 5.20 – Entrada de um novo membro no grupo

c) Saída do grupo

Quando um membro do grupo deseja abandonar o mesmo, ele deve informar a sua saída ao coordenador, através do envio de uma mensagem. O coordenador verifica a sua identificação, o retira da lista de membros do grupo, e atualiza a visão dele (Coordenador) e também dos demais membros do grupo (figura 5.21).

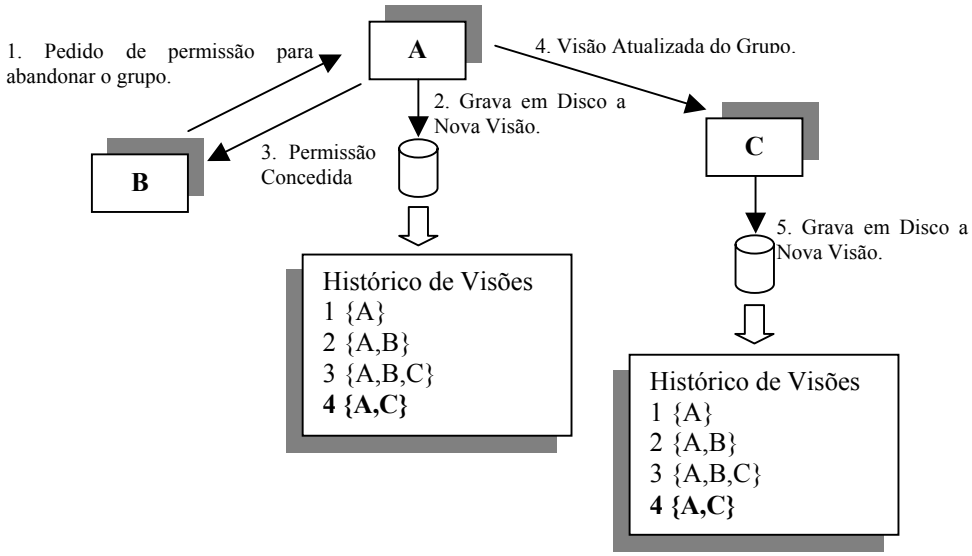


FIGURA 5.21 - Saída de um membro do grupo

d) Envio de mensagens em *multicast*

Quando um membro do grupo deseja enviar uma mensagem ao grupo, ele deve enviá-la ao coordenador, que verifica os membros ativos do grupo, e faz a entrega da mensagem a todos os membros ativos (figura 5.22). O algoritmo que resolve este problema foi demonstrado na figura 4.9.

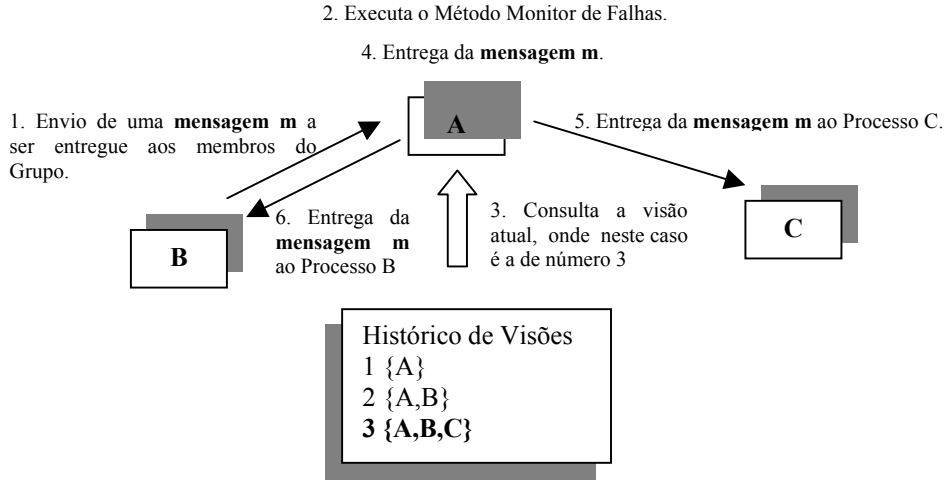


FIGURA 5.22- Envio de mensagens *multicast*

e) Envio de mensagem em *unicast*

Quando um membro do grupo deseja enviar uma mensagem a um outro membro específico do grupo, ele deve enviá-la ao coordenador através da chamada a um método remoto de nome enviar_unicast, que depois a envia para o processo que deve receber a mensagem. A sequência de passos pode ser visualizada na figura 5.23.

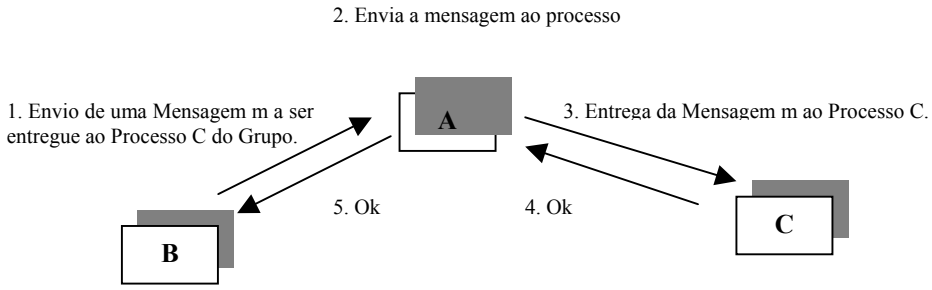


FIGURA 5.23- Envio de mensagem *unicast*

f) Monitor de Falhas

Quando o coordenador deseja verificar se houve falha dentre os membros do grupo, ele envia uma mensagem a cada membro, e espera por uma resposta. Caso o membro tenha falhado, sua identificação é retirada do grupo. Esta técnica adotada é baseada no modelo *pull*, conforme foi descrito no capítulo 3, e pode ser visualizado na figura 5.24. Depois de ter efetuado a verificação de todos os membros ativos, a visão do grupo é atualizada e repassada a todos os membros do grupo.

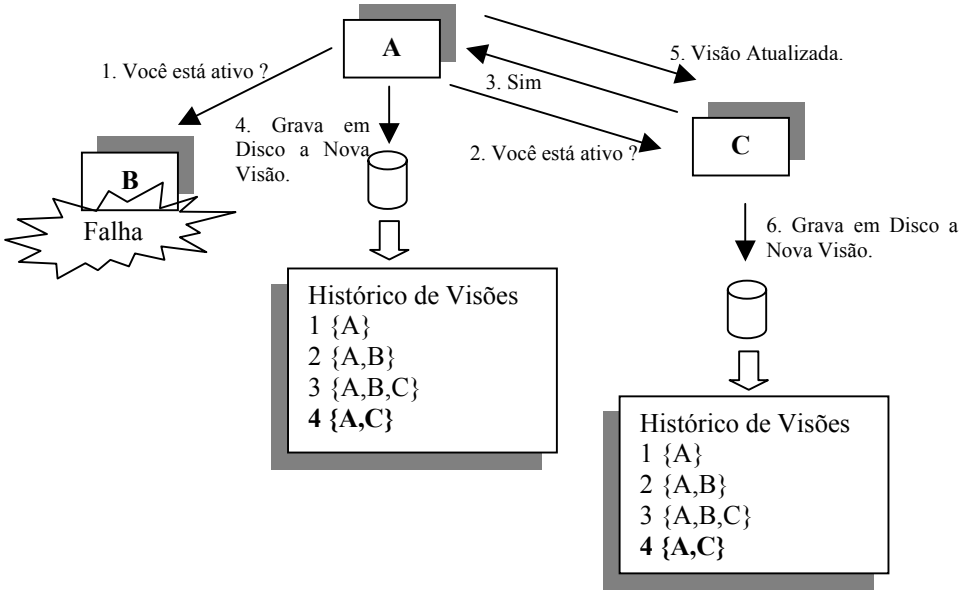


FIGURA 5.24- Monitor de falhas no grupo

g) Tratamento de falha do coordenador do grupo

Quando algum membro tenta enviar uma mensagem ao coordenador, caso o mesmo sofra um falha, é necessário designar um novo coordenador. Feito isto, todos os membros do grupo são avisados e ocorre a configuração do redirecionamento de chamadas remotas a serem efetuadas para o novo coordenador do grupo (figura 5.25).

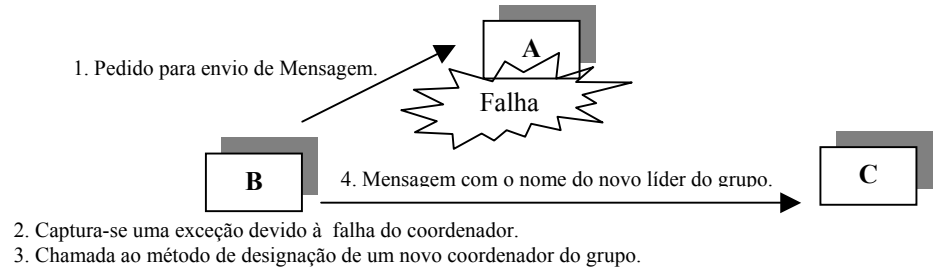


FIGURA 5.25 - Tratamento para a falha do coordenador

h) Visão de membros

Cada processo pode utilizar um recurso que permite a visualização da seqüência de mudanças de visão que aconteceram no grupo, inclusive a mais atual. Cada processo possui uma visão local atualizada e idêntica a todos os demais processos ativos no grupo. A figura 4.26 mostra a consulta hipotética ao conteúdo dos arquivos-texto armazenados nos respectivos discos rígidos. Não existe nenhuma ordem seqüencial: a qualquer momento, qualquer membro do grupo pode fazer uma consulta ao histórico de visões. A partir dos resultados obtidos, entretanto, pode-se deduzir que todas as consultas, em 4.26, ocorreram durante a mesma visão.

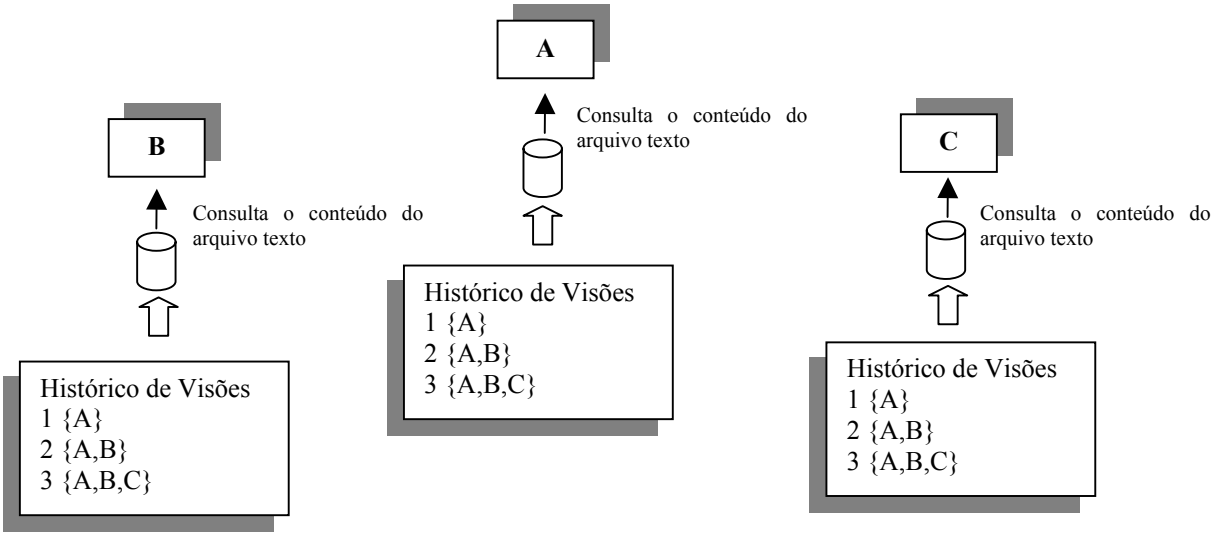


FIGURA 5.26 - Visão dos membros do grupo

O protótipo construído em Java possui as classes Servicos, Processo, Carteiro e ImplementaCarteiro, e estão relacionadas conforme mostrado na figura 5.27, que segue a metodologia UML. A classe Servicos possui os métodos que podem ser executados de forma remota no coordenador do grupo, onde os mesmos são

implementados na classe Processo. A classe Processo possui uma série de variáveis que são utilizadas para fazer o controle de visões e de membros pertencentes ao grupo. A classe Carteiro possui dois métodos que possibilitam o envio de mensagens para todos os membros do grupo (*multicast*), ou algum membro específico (*unicast*).

Neste diagrama, não foram especificados os parâmetros para facilitar a visualização do modelo conceitual, porém os mesmos podem ser verificados nos Anexos 1, 2, 3 e 4, que contém a listagem dos programas-fonte de todos os arquivos, com seus respectivos métodos e parâmetros. A interface do protótipo pode ser visualizada na figura 5.28.

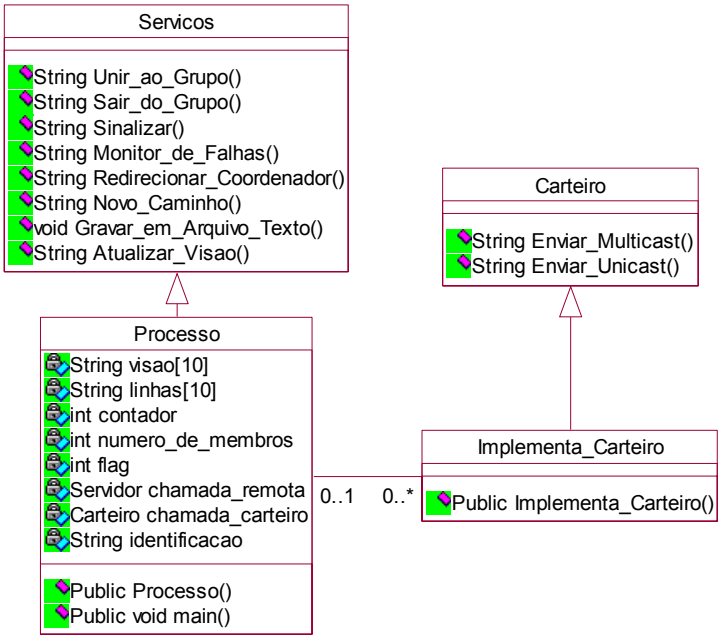


FIGURA 5.27 - Modelo de classes do protótipo

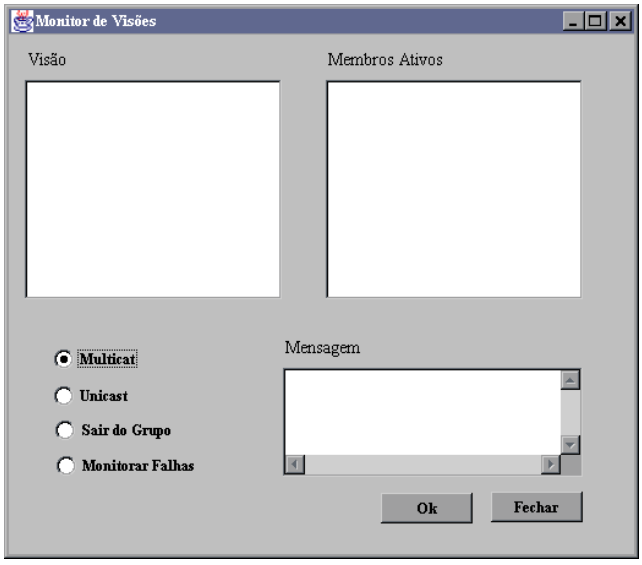


FIGURA 5.28 - Interface do protótipo

## 5.4 Testes de situações críticas do protótipo

O protótipo construído respeita as características do sincronismo virtual proposto inicialmente por Birman [BIR 96], e usado no Isis. Porém não trata a ordenação causal ou total de mensagens. No modelo com sincronismo virtual, tal como implementado no Isis, cada membro de um grupo controla uma visão local de todos os membros correntemente conectados; uma mudança de visão do grupo é vista como uma sucessão de mudanças de visões, de tal forma que seja garantida a propriedade de que a mensagem enviada em *multicast* seja entregue dentro da mesma visão.

Trabalha-se com a hipótese de falhas por colapso das estações envolvidas, com características síncronas do sistema (*timeouts* máximos adequados para detecção de falhas) e com detectores confiáveis. A estrutura do grupo implementado é fechado, hierárquico e dinâmico, e não se permite a sobreposição de grupos.

O protótipo possui uma característica forte que corresponde ao coordenador do grupo, onde qualquer membro que deseje enviar mensagens a um membro (*unicast*) ou a todos os membros do grupo (*multicast*), deve enviá-la ao coordenador, que gerencia o repasse ao(s) destino(s) desejado(s).

Os testes foram efetuados numa rede de computadores com 4 estações de trabalho, o sistema operacional Windows NT, sendo definida como identificação das mesmas: Estação 01, Estação 02, Estação 03 e Estação 04. A primeira máquina a ser ligada foi a Estação 01, que inicialmente funcionou como coordenadora do grupo. Em seguida foram ligadas as máquinas 02, 03 e 04, nesta ordem. O primeiro teste foi efetuado para ver o funcionamento das máquinas, fazendo o envio de uma mensagem *multicast* da Estação 02 para o coordenador, e o coordenador a repassou aos demais membros do grupo, conforme é mostrado na figura 5.29.

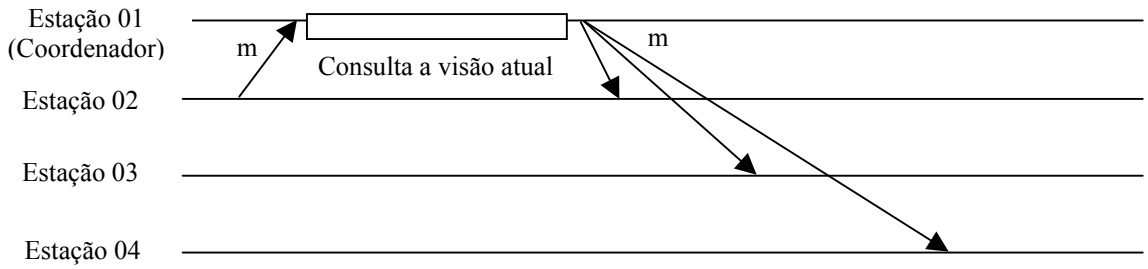


FIGURA 5.29 - Envio de mensagem *multicast*

O próximo teste foi gerar um defeito na máquina que funcionava como coordenador, através de uma reinicialização forçada (*reset)* na mesma. A visão inicial do grupo *vi* estava composta pelas Estações 01, 02, 03 e 04. A Estação 02 tentou enviar uma mensagem ao coordenador, tendo sido detectada uma falha no mesmo. O sistema designou um novo coordenador, que no caso era a própria Estação 02; foi atualizada a visão *vi+1,* e disparadas as mensagens de redirecionamento do coordenador para as Estações 03 e 04 (figura 5.30).

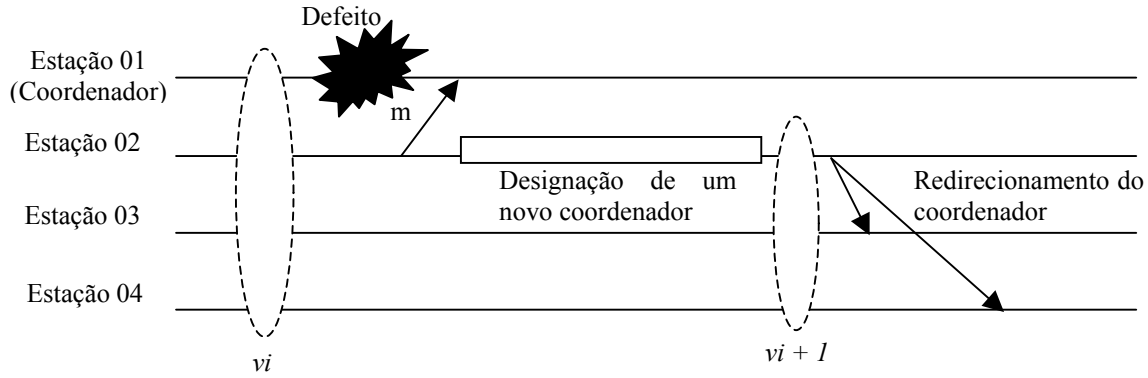


FIGURA 5.30 - Procedimentos pós-falha do coordenador

O próximo teste foi o de "forçar" o colapso da Estação 04 e executar a opção de monitoramento de defeitos a partir do coordenador, onde dispara-se o método para sinalizar a situação de cada uma das estações ativas no grupo. Durante a tentativa de execução do método na Estação 04 foi gerada uma exceção; logo após, foi atualizada a visão no coordenador *vi+1*, sendo repassada aos demais membros do grupo (figura 5.31).

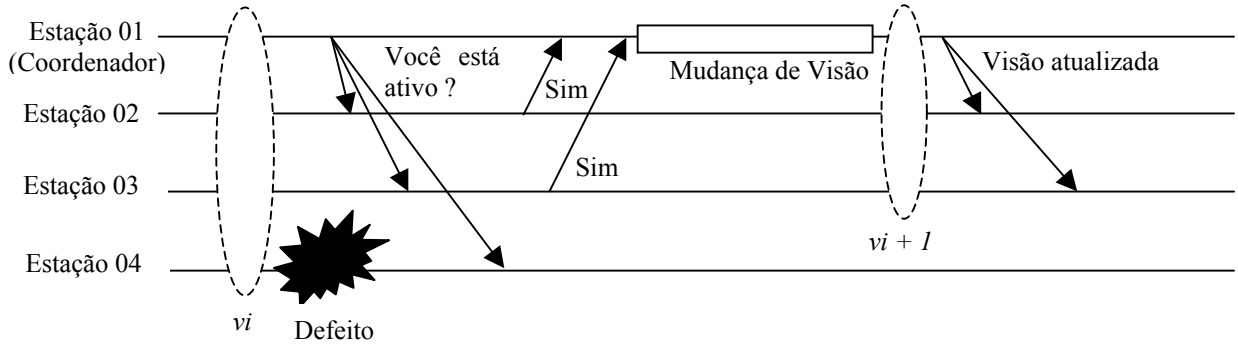


FIGURA 5.31- Monitoramento de defeitos no protótipo

Uma outra situação gerada foi a de forçar a queda dos processos da Estação 01 e da Estação 02. A Estação 04 tenta enviar uma mensagem ao coordenador, é gerada uma exceção, com isso dispara-se o monitoramento de falhas, que detecta a queda da Estação 02 também. Designa-se um novo coordenador, atualizando-se a visão *vi+1*, e são repassadas estas informações à Estação 03 (figura 5.32).

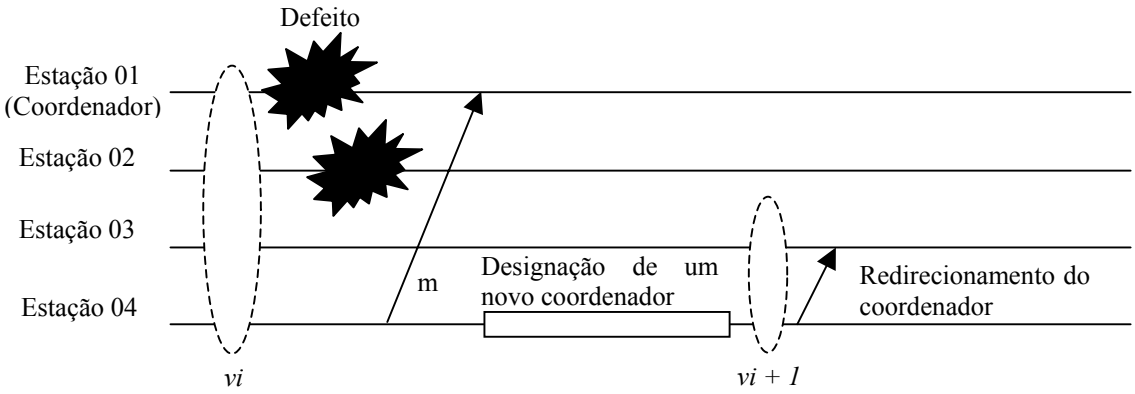


FIGURA 5.32 -Tratamento de defeitos em mais de um membro

Outro teste efetuado foi o de forçar a queda do Processo na Estação 03. Ao ser enviada uma mensagem da Estação 02 para o coordenador, durante a tentativa de entrega da mensagem *m* aos membros indicados como ativos, foi detectado um defeito na Estação 03. Logo após a entrega da mensagem aos membros ativos, efetuou-se a mudança de visão e a atualização da mesma em todos os membros ativos no grupo (figura 5.33).

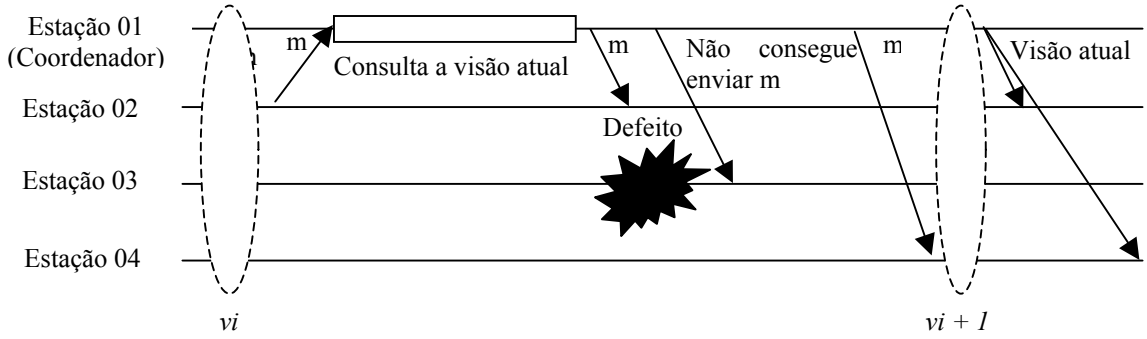


FIGURA 5.33 - Detecção de defeito durante a entrega de mensagens

# 6 Conclusões

A computação distribuída e a comunicação de grupo despertam grande interesse da comunidade científica devido à globalização das informações e a busca em acessá-las de forma segura e confiável, reduzindo a complexidade que seria exigida dos programadores.

O modelo de sincronismo virtual mantém uma visão atualizada sobre os membros (não-falhos) do grupo para fins de comunicação. As suas variações adequam-se a maneiras diferentes de se tratar mudanças de visão. O conhecimento adequado delas torna possível definir aquela que será a mais eficaz possível de acordo com a situação a ser tratada, como é o caso de redes *wide-area* que se utiliza do Sincronismo Virtual Estruturado, e do tratamento para partição de grupo que utiliza-se do com Sincronismo Virtual Estendido onde somente a partição que contém o fragmento do grupo considerado oficial ou majoritário fica ativa.

A implementação do sistema de comunicação utilizando Java RMI permitiu exercitar os conhecimentos de programação de grupo, além de poder implementar chamadas a métodos remotos, e verificar na prática o seu funcionamento e recursos.

O objetivo geral básico desta dissertação era o de conseguir o embasamento necessário para os conceitos de sistemas distribuídos, suas características e peculiaridades. Dentre os pontos positivos da idéia pode-se destacar que a programação distribuída é, sem dúvida, fundamental para os sistemas em uso na atualidade. Utilizar uma linguagem como Java, e usufruir os recursos de orientação a objetos e computação distribuída, é outro ponto de destaque.

Dentre os pontos atendidos peloprojeto físico pode-se citar a gerência dos membros do grupo com alguns serviços, como os citados a seguir. A tolerância a falhas para a queda do coordenador foi tratada de forma que mantenha total transparência aos participantes do grupo; foi programada a implementação da designação de um novo coordenador, que demonstra a funcionalidade do tratamento de exceções em Java RMI. Esta funcionalidade é totalmente confiável, proporcionando com isso um tratamento eficaz para esta situação. O tratamento da saída de membros do grupo também foi um ponto trabalhado: o coordenador retira o nome do membro do grupo da lista de membros ativos, e atualiza a nova visão nos membros remanescentes. Existe ainda um serviço que permite a visualização dos membros ativos no grupo e também permite o monitoramento de falhas através do modelo *pull*. Ainda outro serviço disponível é a possibilidade da troca de mensagens entre os membros do grupo, efetuada através do envio da mensagem pelo membro ao coordenador, o qual a distribui aos demais membros do grupo.

Os passos seguidos para a construção deste projeto foram os seguintes: o embasamento conceitual sobre sistemas distribuídos, e também sobre comunicação de grupo e suas peculiaridades (tipos de grupos, sincronismo, etc); depois foi feito um estudo sobre a linguagem Java e os recursos de programação distribuída através da mesma, chegando por fim aos estudos em RMI, com testes de pequenos exemplos na versão JDK 1.1.7; finalmente partiu-se para a construção do protótipo.

Na implementação do projeto, foi feito primeiro o controle de entrada de membros no grupo e a atualização em disco da visão atual do grupo. Depois foram

implementadas a opção para envio de mensagens em *multicast*, o tratamento de falhas do coordenador, e a designação de um novo líder. Por fim, iniciou-se a programação de novos métodos para com isso disponibilizar mais serviços para os membros do grupo como, por exemplo, verificar quais são os membros ativos, e gerenciar a saída voluntária de qualquer um dos membros do grupo.

Depois de construído o protótipo, foram efetuados vários testes, com procedimentos que incluíam desde a ligação das máquinas, a composição dos grupos, com a análise da visão atual; com entrada posterior de outras máquinas e análise da visão decorrente dos demais membros quanto à sua consistência. Foram efetuados testes "forçando" o colapso do coordenador, utilizando-se do procedimento padrão de inicialização (*reset* ou *Ctrl+Break*) da máquina. Fez-se a análise da designação do novo coordenador e a atualização da visão nos membros ativos do grupo, observando-se que os serviços disponibilizados pelo coordenador do grupo funcionavam de forma transparente com relação à queda do coordenador do grupo.

## 6.1 Contribuições do trabalho

Com o desenvolvimento deste trabalho, pode-se citar uma série de atividades realizadas que representam a contribuição do autor ao grupo de tolerância a falhas. Elas são brevemente destacadas a seguir.

Entendimento dos conceitos e propriedades do sincronismo virtual e suas repercussões práticas na implementação do mecanismo: foram criados ou adaptados vários algoritmos, com destaque para os de envio de mensagens aos membros do grupo, o de monitoramento de defeitos e o de designação de um novo coordenador do grupo (caso ocorra um defeito no mesmo). Embora o mecanismo seja conhecido e conceitualmente bem definido, a "tradução" de suas exigências ou propriedades em aspectos de implementação exigiu do autor um aprofundamento nos estudos e na compreensão plena de suas repercussões.

Desenvolvimento de um aplicativo para demostrar o controle de *membership* implementado em Java com recursos do RMI, utilizando recursos de execução remota de métodos, e disponibilizando serviços de envio/recebimento de mensagens e monitoramento de participantes do grupo suspeitos de falha. Foi efetuado o tratamento para o controle de visão atualizada do grupo em todos os processos corretos.

Destaca-se também o tratamento de falha no coordenador, além de tratar também a falha em qualquer um dos membros participantes do grupo. Neste caso o funcionamento do aplicativo fica transparente para os processos ativos, não ocasionando perda de informações, pois as mesmas encontram-se replicadas, garantindo o funcionamento do sistema através da designação de um novo coordenador.

Com a aplicação dos conceitos de sincronismo virtual, demonstra-se que manter a visão dos membros participantes de um grupo atualizada, é de grande importância, pois dessa forma cada um dos membros pode ter permanentemente o conhecimento da composição do grupo do qual participa, sem o risco de possuir membros cujas visões são inconsistentes com os demais.

## 6.2 Trabalhos futuros

Devido ao tempo para o desenvolvimento do trabalho ser curto, considerando toda a etapa necessária como embasamento ao autor, surgiram alguns pontos que são sementes para futuros trabalhos, abordando tópicos que poderiam ser bastante úteis para as pesquisas realizadas nas áreas de sistemas distribuídos e tolerância a falhas.

Um primeiro trabalho seria **implementar o mecanismo de sincronismo virtual estendido, provendo o tratamento de múltiplas partições de grupo.** Neste caso, quando acontecerem partições na rede, somente a partição que contém o fragmento do grupo considerado oficial ou majoritário fica ativa (partição primária), a outra partição (ou partições) é suspensa e seus processos disparam um procedimento de desconexão compulsória. Para tomar esta decisão, segundo Birman, existem duas políticas que podem ser utilizadas para definir a partição primária: escolher a partição majoritária, ou seja, com a maioria dos membros, independentemente de propriedades específicas dos componentes do sistema, ou escolher a partição essencial, ou seja, aquela que mantenha algum serviço imprescindível para o funcionamento do sistema [BIR 94]. Quanto às partições não-primárias ou não essenciais, elas podem ser bloqueadas ou não.

**Implementar um *chat* com monitoramento de defeitos.** Com o desenvolvimento deste novo trabalho, a visão dos membros do grupo vai estar atualizada em relação à falha de algum dos participantes. O método de monitoramento a ser aplicado poderia ser o *push,* ao invés do modelo *pull*, usado na presente dissertação. Supõe-se que, neste caso, a detecção seria mais rápida pois há um acompanhamento permanente da atividade dos membros do sistema. Com o desenvolvimento deste trabalho, não aconteceria de usuários ficarem enviando mensagens a outros que não estão mais no sistema, pois a visão estaria sempre atualizada, não contendo o nome dos que saíram do grupo de forma voluntária ou involuntária.

**Tratar a inconsistência da visão do coordenador perante a visão da maioria dos outros membros do grupo.** Esta situação pode ocorrer quando, por hipótese, o coordenador possui uma visão diferente da maioria dos membros ativos do grupo. É possível supor, neste caso, que o elemento com defeito é ele, e que deve sere designado um novo coordenador. No caso, este seria o membro ativo que reconhece o maior número de elementos ativos no grupo, e o coordenador "suspeito" perderia a função de coordenar para ser apenas um membro participante do grupo. Observe-se que a situação acima descrita poderia ocorrer em situações onde o sistema esteja particionado ou o coordenador esteja com falhas mais abrangentes que colapso. Ambas estão fora do escopo deste trabalho.

**Implementar a comunicação entre controladores de grupos diferentes, aplicando assim o conceito de sincronismo virtual estruturado.** No trabalho desenvolvido nesta dissertação, existe um controlador e os membros participantes do grupo subordinados a ele. Pode –ser estudada, como um trabalho futuro, a possibilidade de manter em funcionamento vários grupos, e um grupo de membros controladores. Pode-se implementar a criação de um recurso que possibilite que membros de qualquer um dos grupos, enviem mensagens a membros de outros grupos, onde o membro participante de um grupo específico envia a mensagem para o coordenador do seu grupo, e este envia a mensagem aos coordenadores dos outros

grupos, que fazem a entrega nos grupos locais respectivos de cada um.

**Fazer a implementação de um projeto utilizando outras tecnologias para programação distribuída**, ao invés de RMI, e analisar o desempenho entre as elas. Esta análise de desempenho seria muito útil pois assim poderiam ser detectadas peculiaridades que tornam o acesso remoto mais rápido ou mais lento nas plataformas envolvidas. E, como este trabalho utilizou essencialmente RMI, não há como fazer este tipo de análise.

# Anexo 1

## Código do arquivo Carteiro.java

A interface Carteiro possui o protótipo dos métodos que podem ser executados remotamente, e que são implementados no arquivo Implementa_Carteiro.java. Estes métodos tratam o envio de mensagens *multicast* e *unicast* aos membros participantes do grupo.

```
import java.rmi.*;

interface Carteiro extends java.rmi.Remote {

        //Envio de mensagem em multicast
        public String enviar_mensagem (String mensagem, String visao[], int n_membros)
        throws RemoteException;



        // Envio de mensagem em unicast
        public String enviar_unicast (String mensagem,
        String visao[],
        int n_membros,
        String identificacao) throws RemoteException;
}
```

# Anexo 2

## Código do arquivo Servicos.java

A interface Servico possui o protótipo dos métodos que podem ser executados remotamente, e que são implementados no arquivo Processo.

```
//
// Nome do Arquivo: Servicos.java
// Autor: Robson Soares Silva
//
// Objetivo: Interface para um serviço RMI que envia e entrega mensagens

import java.math.BigInteger;
import java.rmi.*;
//
public interface Servicos extends java.rmi.Remote
{
//Método para Uniao (join) do processo ao Grupo
public String unir_ao_grupo (String identificacao) throws RemoteException;

//Método que o servidor executará no cliente
public String sinalizar() throws RemoteException;

//Método que o servidor executará no cliente
public String entregar_mensagem(String m) throws RemoteException;

//Método para Sair (leave) do Grupo
public String sair_do_grupo (String identificacao) throws RemoteException;

// Método para Monitorar Falhas (Failure Detector - Pull (Objetos Passivos) do Grupo
public String Monitor_de_Falhas () throws RemoteException;

// Método para acessar o Histórico de Visões
public String[] Historico_de_Visoes () throws RemoteException;


// Método para Redirecionar o coordenador do grupo
public String redirecionar_coordenador
(String visao[], int Numero_de_Membros_do_Grupo,
String linhas[], int contador) throws RemoteException;

// Método para Configurar o Novo Caminho do Coordenador
public String novo_caminho(
String visao[], int Numero_de_Membros_do_Grupo,
String linhas[], int contador) throws RemoteException;

// Método para Atualizar a visão do grupo
public String atualizar_visao (
      String nova_visao[], int n_membros,
      String historico[], int contador_mudancas) throws RemoteException;
}
```

# Anexo 3
# Código do arquivo Processo.java

Código fonte do arquivo Processo.java, que disponibiliza uma série de serviços de *membership*.

```
/*
Nome do Arquivo: Processo.java
Objetivo: Servidor para um serviço RMI que envia e entrega mensagens, armazena o histórico de visões do
          grupo e monitora falhas.
*/
import java.applet.*;

import java.sql.*;
import java.io.*;
import java.awt.*;
import java.math.*;
import java.rmi.*;
import java.rmi.server.*;
import java.net.*;

import java.rmi.Naming;
import messagebox;

public class Processo extends UnicastRemoteObject implements Servicos
{
      static String visao[] = new String[10]; // Armazena Processo a Processo
      // um em cada indice do vetor
      static String linhas[] = new String[10];

      static int contador=0; // {Conta o número de mudanças de visão existentes}
      static int Numero_de_Membros_do_Grupo=0; // {Conta o número de membros ativos no grupo}
      static int n = 0;
      static int flag = 0;
      static int i = 0;
      static int j = 0;
      static int k = 0;

      static String identificacao;

static Servicos chamada_remota;

      static Carteiro chamada_carteiro;

      static String url = "jdbc:odbc:visao.mdb";
      static String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
      static String login = "dba";
      static String passwd = "javabank";
      static Connection curConn;

      static messagebox m1;

      public Processo () throws RemoteException
      {
        super();
      }
```

```
// Método que efetua a gravação em arquivo texto
public void Gravar_em_arquivo_texto()
{
   FileOutputStream f1;
   PrintStream o1;

   contador ++;

   System.out.println("Contador foi incrementado " + contador);

    try
            {
              f1 = new FileOutputStream("Visao.txt");
              o1 = new PrintStream(f1);

              o1.print("< " + contador + " , { ");

  linhas[contador-1] = "< " + contador + " , { ";

  n = 0;

  while (n < Numero_de_Membros_do_Grupo)
     {
      o1.print(visao[n]);
      linhas[contador-1] = linhas[contador-1] + visao[n];
      n++;
      if (n < Numero_de_Membros_do_Grupo)
         {
           o1.print(" , ");
           linhas[contador-1] = linhas[contador-1] + " , ";
         }
     }
     o1.println(" } >");
     linhas[contador-1] = linhas[contador-1] + " } >";
}
 catch (Exception e)
{
 System.out.println("Problemas na Gravação do Texto");
}

   // Atualiza o arquivo texto com o histórico de visões
   // armazenado no vetor linhas
   try
    {
     f1 = new FileOutputStream("Visao.txt");
     o1 = new PrintStream(f1);

     o1.println ("Histórico de Visões");
     j = 0;
     while (j < contador)
        {
          o1.println (linhas[j]);
          j = j + 1;
        }
    }
  catch (Exception e)
    {
     System.out.println("Problemas na gravação do texto no unir_ao_grupo");
    }
```

```
        System.out.println("Linhas ---> " + linhas[contador-1]);

        // Atualização da visão nos membros
        j = 1;
        while (j < Numero_de_Membros_do_Grupo)
           {
            try {
            Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
            // Mudanca_de_Visao servico_visao= (Mudanca_de_Visao) Naming.lookup("rmi://" + visao[j] +
"/Mudanca_de_Visao");
            System.out.println ("Atualizando Visao na Estacao " + visao[j] + " ");
            System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
            }
            catch (Exception e)
            {
              System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
            }
            j = j + 1;
          }
        System.out.println("Ok.");
       }

      public String redirecionar_coordenador(String visao[], int Numero_de_Membros_do_Grupo, String
linhas[], int contador) throws RemoteException
        {
        int j;
        // Atualização da visão nos membros
        j = 1;
        while (j < Numero_de_Membros_do_Grupo)
           {
            try {
            Mudanca_de_Visao servico = (Mudanca_de_Visao) Naming.lookup("rmi://" + visao[j] +
"/Mudanca_de_Visao");
            System.out.println ("Atualizando Visao na Estacao " + visao[j] + " ");
            System.out.println ("Testando aqui - Ponto 3");
            System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
            }
            catch (Exception e)
            {
              System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
            }
            j = j + 1;
          }
         return ("Ok. Redirecionador Finalizado com Sucesso!");
      }

// Método para atualizar a visão do grupo
public String atualizar_visao (String nova_visao[], int n_membros, String historico[], int contador_mudancas)
throws RemoteException
{
            FileOutputStream f1;
            PrintStream o1;

            System.out.println("Executando o método atualizar visao!");

            int j = 0;
            Numero_de_Membros_do_Grupo = n_membros;
            j = 0;
            while (j < n_membros)
            {
```

```
                visao[j] = nova_visao[j];
                j = j + 1;
        }
        System.out.println("Atualizando o Histórico de Visões !");
        contador = contador_mudancas;
        j = 0;
        while (j < contador_mudancas)
        {
                linhas[j] = historico[j];
                j = j + 1;
        }

        System.out.println("Membros do Grupo Atualizados com a Nova Visao !");
        System.out.println("Número de membros ativos no grupo: ");
        System.out.println(Numero_de_Membros_do_Grupo);

       // Atualiza o arquivo texto com o histórico de visões armazenado no vetor linhas

       try
        {
          f1 = new FileOutputStream("Visao.txt");
          o1 = new PrintStream(f1);

          o1.println ("Histórico de Visões");
          j = 0;
          while (j < contador)
            {
              o1.println (linhas[j]);
              j = j + 1;
            }
        }
      catch (Exception e)
        {
          System.out.println("Problemas na Gravação do Texto no Atualizar_Visao");
        }
        return ("Visão Atualizada com Sucesso !");
}

public String novo_caminho (String visao[], int Numero_de_Membros_do_Grupo, String linhas[], int contador)
throws RemoteException
{
        chamada_remota = (Servicos) Naming.lookup ("rmi://" + visao[0] + "/Servicos");
        return ("Ok");
}

//Método para Monitorar Falhas (Failure Detector - Pull (Objetos Passivos) do Grupo
public String Monitor_de_Falhas ()
{
        FileOutputStream f1;
        PrintStream o1;
        i = 0;
        while (i < Numero_de_Membros_do_Grupo)
        {
                try {
                        Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[i] + "/Servicos");
                        System.out.println ("Processo " + visao[i] + ". Vocˆ est  ativo ?");
                        System.out.println (servico.sinalizar());
                    }
                 catch (Exception e)
                 {
```

```
System.out.println("Processo " + visao[i] + " Suspeito de Falha");
// Remover(i);
while (i < Numero_de_Membros_do_Grupo-1)
{
        visao[i] = visao[i+1];
        visao[i+1] = "";
        i = i + 1;
}
//
Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo - 1;
//
// Gravar_em_arquivo_texto();
contador++; // Incrementa o contador de mudanças de visao
System.out.println("Membros do Grupo Atualizados com a Nova Visao !");
try
{
        f1 = new FileOutputStream("ViewChange.txt");
        o1 = new PrintStream(f1);
        flag = 1;
        o1.print("< " + contador + " , { ");
        linhas[contador-1] = "< " + contador + " , { ";
        n = 0;

        while (n < Numero_de_Membros_do_Grupo)
        {
                o1.print(visao[n]);
                linhas[contador-1] = linhas[contador-1] + visao[n];
                n++;
                if (n < Numero_de_Membros_do_Grupo)
                {
                        o1.print(" , ");
                        linhas[contador-1] = linhas[contador-1] + " , ";
                }
        }
        o1.println(" } >");
        linhas[contador-1] = linhas[contador-1] + " } >";
}
catch (Exception e1)
{
        System.out.println("Problemas na Gravacao do Texto");
}
```

**// Atualiza o arquivo texto com o histórico de visões armazenado no vetor linhas**

```
try
{
         f1 = new FileOutputStream("Visao.txt");
        o1 = new PrintStream(f1);

        o1.println ("Histórico de Visões");
        j = 0;
        while (j < contador)
        {
                o1.println (linhas[j]);
                j = j + 1;
        }
}
catch (Exception em)
{
        System.out.println("Problemas na Gravacao do Texto no Monitorar_Falhas");
}
```

```
                        // Atualização da visão nos membros
                        j = 1;
                        while (j <= Numero_de_Membros_do_Grupo)
                        {
                                try {
                                    Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] +
                                    "/Servicos");
                                    System.out.println ("Atualizando visão na estação " + visao[j] + " ");
                                    System.out.println
                                    (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas,
                                    contador));
                                }
                                catch (Exception e2)
                                {
                                    System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de
                                    Falha");
                                }
                            j = j + 1;
                        }
                       // Gravar_em_arquivo_texto();
                        //
                    }
                i = i + 1;
            }
            return ("Finalizada a tarefa do Monitor de Falhas !");
}

    // Método para Saida (leave) do processo do Grupo
    public String sair_do_grupo (String identificacao) throws RemoteException
    {
     FileOutputStream f1;
     PrintStream o1;

     int encontrou = 0;

     try
     {
      // Verificar se aquela IDENTIFICACAO ja existe no grupo
      n=0;
      while (n < Numero_de_Membros_do_Grupo)
         {
          if (visao[n].equals(identificacao))
             {
              System.out.println("PROCESSO ENCONTRADO!");
              visao [n] = "";
              // Remover(n);
              i = n ;
              while (i < Numero_de_Membros_do_Grupo-1)
              {
               visao[i] = visao[i+1];
               i = i + 1;
              }
              //
              Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo - 1;
              encontrou = 1;
              break;
             }
          n++;
         }
```

```
  contador ++;

  System.out.println("Contador foi incrementado " + contador);

  if (encontrou == 0)
    {
      return("IDENTIFICAÇÃO NÃO ENCONTRADA !");
    }

   try
          {
            f1 = new FileOutputStream("visao.txt");
            o1 = new PrintStream(f1);

            o1.print("< " + contador + " , { ");


 linhas[contador-1] = "< " + contador + " , { ";

 n = 0;

 while (n < Numero_de_Membros_do_Grupo)
    {
     o1.print(visao[n]);
     linhas[contador-1] = linhas[contador-1] + visao[n];
     n++;
     if (n < Numero_de_Membros_do_Grupo)
        {
          o1.print(" , ");
          linhas[contador-1] = linhas[contador-1] + " , ";
        }
    }
    o1.println(" } >");
    linhas[contador-1] = linhas[contador-1] + " } >";
}
catch (Exception e)
{
 System.out.println("Problemas na Gravacao do Texto");
}
}
catch (Exception e)
{
 System.out.println("Problemas na Gravacao do Texto");
}

System.out.println("Linhas ---> " + linhas[contador-1]);

 // Atualiza o arquivo texto com o histórico de visões
 // armazenado no vetor linhas

   try
    {
     f1 = new FileOutputStream("Visao.txt");
     o1 = new PrintStream(f1);

     o1.println ("Histórico de Visões");
     j = 0;
     while (j < contador)
        {
```

```
            o1.println (linhas[j]);
            j = j + 1;
          }
      }
    catch (Exception e)
       {
        System.out.println("Problemas na Gravacao do Texto no unir_ao_grupo");
       }
 // Atualização da visão nos membros
 j = 1;
 while (j < Numero_de_Membros_do_Grupo)
     {
      try {
      Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
      System.out.println ("Atualizando Visao na Estacao " + visao[j] + " ");
      System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
      }
      catch (Exception e)
      {
        System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
      }
      j = j + 1;
    }
 return ("Processo removido do grupo com sucesso, e atualizacao da visão nos membros do grupo !");
}

// Método que o remove um membro do grupo
public void Remover (int i)
 {
 while (i < Numero_de_Membros_do_Grupo-1)
  {
   visao[i] = visao[i+1];
   visao[i+1] = "";
   i = i + 1;
  }
}

 // Método que o servidor executará nos clientes
 public String sinalizar() throws RemoteException
 {
   return("Sim");
 }
```

```
 // Método para entrega de mensagens nos clientes
 public String entregar_mensagem(String m) throws RemoteException
 { int j;
  System.out.println ();
  System.out.println ("Mensagem Multicast Enviada para o Grupo PANTANAL: " + m);
  //
  // Tratamento para garantir a confiabilidade de entrega da mensagem 19/01/2001
  //
  j = 1; // j pode comecar valendo um para nao envio da mensagem ao coordenador
  while (j < Numero_de_Membros_do_Grupo)
     {
      try {
      Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
      System.out.println ();
      System.out.println (servico.reenviar(m));
      }
      catch (Exception e)
      {
        System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
      }
      j = j + 1;
   }
  //
  // Tratamento para garantir a confiabilidade de entrega da mensagem 19/01/2001
  //
  return("Mensagem Entregue!");
 }

 //Método para entrega de mensagens nos clientes
 public String reenviar(String m) throws RemoteException
 {
  System.out.println ();
  System.out.println ("Reenvio de Mensagem Multicast Enviada para o Grupo PANTANAL: " + m);
  return("Mensagem Entregue! "  + m);
 }

 //Método para saber a visão atual
 public String[] Historico_de_Visoes() throws RemoteException
 {
  return (visao);
 }

 //Método para União (join) do processo ao Grupo
 public String unir_ao_grupo (String identificacao) throws RemoteException
 {
  int j;

  FileOutputStream arquivo5;
  PrintStream fila5;

  FileOutputStream ff1;
  PrintStream oo1;

try
{
  Class.forName(driver);
  curConn = DriverManager.getConnection(url, login, passwd);
}
catch(SQLException ex)
{
```

```
  System.out.println(ex);
}
catch(java.lang.Exception ex)
{
  m1 = new messagebox("Erro na Abertura do Banco de Dados");
  m1.show();
}
    try
       {
         int Cod = 1;
         Statement curStmt = curConn.createStatement();
         boolean res = curStmt.execute("INSERT INTO BD VALUES('1','UNIDERP07')");
         curStmt.close();
       }
       catch(SQLException ex)
       {
         ex.printStackTrace();
       }
       catch(java.lang.Exception ex)
       {
         m1 = new messagebox("Erro de Gravação");
         m1.show();
       }

    // Verificar se aquela IDENTIFICACAO já existe no grupo
    n=0;
    while (n < Numero_de_Membros_do_Grupo)
       {
        if (visao[n].equals(identificacao))
           {
            System.out.println("TENTATIVA DE UNIAO AO GRUPO. PROCESSO JA EXISTENTE");
            System.out.println(identificacao);
            return("ABORT");
           }
        n++;
       }

    visao[Numero_de_Membros_do_Grupo] = identificacao;
    contador++; // Incrementa o contador de mudanças de visão
    // Incrementa o Contador de Numero de Membros do Grupo
    Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo + 1;

    System.out.println("Número de Membros do Grupo Atualizados com a Nova Visao !");
    System.out.println("Membros Ativos no Grupo : " + Numero_de_Membros_do_Grupo);
    try
      {
       arquivo5 = new FileOutputStream("Visao.txt");
       fila5 = new PrintStream(arquivo5);
       flag = 1;

    fila5.print("< " + contador + " , { ");
    linhas[contador-1] = "< " + contador + " , { ";
    System.out.println(linhas[contador-1]);

    n = 0;
    while (n < Numero_de_Membros_do_Grupo)
       {
        // System.out.println(visao[n]);
        fila5.print(visao[n]);
        linhas[contador-1] = linhas[contador-1] + visao[n];
```

```
      System.out.println(linhas[contador-1]);
      n++;
      if (n < Numero_de_Membros_do_Grupo)
        {
          fila5.print(" , ");
          linhas[contador-1] = linhas[contador-1] + " , ";
          System.out.println(linhas[contador-1]);
        }
     }
     fila5.println(" } >");
     linhas[contador-1] = linhas[contador-1] + " } >";
     System.out.println("Visao Atual: " + linhas[contador-1]);
    }
    catch (Exception e)
    {
     System.out.println("Problemas na Gravação do Texto no unir_ao_grupo");
    }

   // Atualiza o arquivo texto com o histórico de visões
   // armazenado no vetor linhas

   try
    {
     arquivo5 = new FileOutputStream("Visao.txt");
     fila5 = new PrintStream(arquivo5);

     System.out.println ();
     System.out.println ("Histórico de Visões");
     j = 0;
     while (j < contador)
       {
         System.out.println (linhas[j]);
         j = j + 1;
       }
    }
  catch (Exception e)
    {
     System.out.println("Problemas na Gravação do Texto no unir_ao_grupo");
    }

// Atualização da visão nos membros
j = 1;

System.out.println("Numero_de_Membros_do_Grupo: ");
System.out.println(Numero_de_Membros_do_Grupo);

// Na primeira vez o j não vai ser menor que 1, com isso
// não efetua a execução do loop.

while (j < Numero_de_Membros_do_Grupo)
   {
    System.out.println(j);
    try {
    Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
    System.out.println ("Atualizando a visao na Estação " + visao[j] + " ");
    System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
    }
    catch (Exception e)
    {
      System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
```

```
        }
        j = j + 1;
      }
    return ("Ok");
   }
```

**/* Programa principal, onde ativa-se o Gerenciador de Segurança verifica-se o número de argumentos informados na execução do aplicativo, e disponibiliza-se um menu de opções que permite o envio de mensagens e também monitorar falhas e visões do grupo. */**

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

     FileOutputStream f1;
     PrintStream o1;

     FileOutputStream a2;
     PrintStream b2;

     Processo Objeto;

     String id; // Identificacao da máquina destinatário para mensagens em unicast

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

      // Cria uma instancia do servidor de servicos ...
      Processo svr = new Processo();

      Implementa_Carteiro svr2 = new Implementa_Carteiro();

      Implementa_Mudanca_de_Visao svr3 = new Implementa_Mudanca_de_Visao();

      // ... e liga-o (bind) com o rmi registry
      Naming.bind ("Servicos", svr);
      Naming.bind ("Carteiro", svr2);
      Naming.bind ("Mudanca_de_Visao", svr3);

      // Atualiza (Assign) o gerenciador de segurança
      if (System.getSecurityManager() == null)
        {
         System.setSecurityManager(new RMISecurityManager());
        }
      // Configuração das variáveis chamada_remota e chamada_carteiro
      // Se args[0] contém "UFRGS", configura as variáveis para
      // chamarem os métodos nesta máquina.
      chamada_remota = (Servicos) Naming.lookup
                    ("rmi://" + args[0] + "/Servicos");

      chamada_carteiro = (Carteiro) Naming.lookup
                    ("rmi://" + args[0] + "/Carteiro");

       // Verifica o número de argumentos, que corresponde ao hostname
                if (args.length != 1)
                {
             System.out.println ("Digite o nome do Coordenador ...");
             System.exit(1);
                }

      DataInputStream din = new DataInputStream (System.in);
```

```
System.out.print ("Processo Coordenador (S/N) ? ");
identificacao = din.readLine();

if (identificacao.toUpperCase().equals("S"))
   {
    System.out.println ("Coordenador do Grupo PANTANAL Ativo!");
    System.out.println ("Iniciado o Grupo: " + chamada_remota.unir_ao_grupo(args[0]));
   }
else
   {
    System.out.println ("Membro do Grupo PANTANAL!");
    System.out.print ("Digite a Identificacao do Processo: ");
    identificacao = din.readLine();

    //Método para Uniao (join) do processo ao Grupo
    while (chamada_remota.unir_ao_grupo(identificacao).equals("ABORT"))
    {
     System.out.println("IDENTIFICACAO JA EXISTE NO GRUPO");
     System.out.println("Digite outra Identificacao para o Processo: ");
     identificacao = din.readLine();
    }

    // Nº de membros , atualizado na estação que conectou
    // Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo + 1;

    System.out.println("CONEXÃO DO PROCESSO AO GRUPO EFETUADA COM SUCESSO");
    System.out.println("================================================");
   }
  System.out.println ("Digite a Opcao Desejada:");
  //
  //
for (;;)
{
            System.out.println();
            System.out.println("======================================");
            System.out.println ("1 - Enviar Mensagem Multicast ao Grupo");
            System.out.println ("2 - Enviar Mensagem Unicast");
            System.out.println ("3 - Sair do Grupo");
            System.out.println ("4 - Visualizar Membros do Grupo");
            System.out.println ("5 - Monitorar Falhas");
            System.out.println ("6 - Exit"); System.out.println();
            System.out.println("======================================");
            System.out.print ("Digite a Opção Desejada: ");

                    String line = din.readLine();
                    Integer choice = new Integer(line);

                    int value = choice.intValue();

                    switch (value)
                    {
              case 1:
                    System.out.print ("Digite a Mensagem : ");
                                    line = din.readLine();
                    System.out.println("=======");
                    // choice = new Integer (line);
                    // value  = choice.intValue();

                    // Chamada ao m,todo remoto de envio de mensagem para o coordenador do grupo
```

```
                        try
                        {
                         System.out.println ("Resposta : " +
chamada_carteiro.enviar_mensagem(line,visao,Numero_de_Membros_do_Grupo));
                        }
                        catch (Exception e)
                        {
                          System.out.println("Coordenador Suspeito de Falha");
                          System.out.println("Redirecionar Coordenador !!!");

                         System.out.println("Processo " + visao[i] + " Suspeito de Falha");
                         // Remover(i);
                         k = i;
                         while (k < Numero_de_Membros_do_Grupo)
                         {
                              visao[k] = visao[k+1];
                              visao[k+1] = "";
                              k = k + 1;
                         }
                         // Remover(i);
                         Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo - 1;
                         //
                         // Gravar_em_arquivo_texto();

                         contador ++;
                         System.out.println("Contador foi incrementado " + contador);
                         try
                         {
                            f1 = new FileOutputStream("visao.txt");
                            o1 = new PrintStream(f1);
                            o1.print("< " + contador + " , { ");

                            linhas[contador-1] = "< " + contador + " , { ";

                            n = 0;

                            while (n < Numero_de_Membros_do_Grupo)
                            {
                                o1.print(visao[n]);
                                linhas[contador-1] = linhas[contador-1] + visao[n];
                                n++;
                                if (n < Numero_de_Membros_do_Grupo)
                                {
                                  o1.print(" , ");
                                  linhas[contador-1] = linhas[contador-1] + " , ";
                                }
                             }
                             o1.println(" } >");
                             linhas[contador-1] = linhas[contador-1] + " } >";
                            }
      catch (Exception e1)
      {
       System.out.println("Problemas na Gravação do Texto");
      }

         // Atualiza o arquivo texto com o histórico de visões
         // armazenado no vetor linhas

         try
          {
```

```
            f1 = new FileOutputStream("Visao.txt");
            o1 = new PrintStream(f1);

            o1.println ("Histórico de Visões");
            j = 0;
            while (j < contador)
               {
                 o1.println (linhas[j]);
                 j = j + 1;
               }
            }
          catch (Exception ee)
            {
             System.out.println("Problemas na Gravação do Texto no unir_ao_grupo");
            }

        // Atualizaão da visão nos membros
        j = 1;
        while (j < Numero_de_Membros_do_Grupo)
           {
            try {
            Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
            // Mudanca_de_Visao servico_visao= (Mudanca_de_Visao) Naming.lookup("rmi://" + visao[j] +
"/Mudanca_de_Visao");
            System.out.println ("Atualizando Visao na Estacao " + visao[j] + " ");
            System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
            }
            catch (Exception ee3)
            {
              System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
            }
            j = j + 1;
          }

                              System.out.println("Designação do novo coordenador efetuada, o eleito foi: " +
visao[0]);

                              chamada_remota = (Servicos) Naming.lookup
                                        ("rmi://" + visao[0] + "/Servicos");
                              chamada_carteiro = (Carteiro) Naming.lookup
                                         ("rmi://" + visao[0] + "/Carteiro");

                              System.out.println("Redirecionar Coordenador: " +
                              chamada_remota.redirecionar_coordenador  (visao,Numero_de_Membros_do_Grupo,
linhas , contador));
                           }

                           break;

                      case 2:
                           System.out.print ("Digite a Mensagem : ");
                                        line = din.readLine();
                           System.out.print ("Identificacao do Destinatário : ");
                           id = din.readLine();

                           System.out.println("=======");

                           // Chamada ao método remoto de envio de mensagem para o coordenador do grupo
                           try
                           {
```

```
                                System.out.println ("Resposta : " +
chamada_carteiro.enviar_unicast(line,visao,Numero_de_Membros_do_Grupo,id));
                               }
                               catch (Exception e)
                               {
                                 System.out.println("Coordenador Suspeito de Falha");
                                 System.out.println("Redirecionar Coordenador !!!");

System.out.println("Processo " + visao[i] + " Suspeito de Falha");
                                // Remover(i);
                                k = i;
                                while (k < Numero_de_Membros_do_Grupo)
                                {
                                     visao[k] = visao[k+1];
                                     visao[k+1] = "";
                                     k = k + 1;
                                }
                                // Remover(i);
                                Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo - 1;
                                //
                                // Gravar_em_arquivo_texto();
                                 contador ++;
                                System.out.println("Contador foi incrementado " + contador);
                                try
                                {
                                   f1 = new FileOutputStream("visao.txt");
                                   o1 = new PrintStream(f1);
                                   o1.print("< " + contador + " , { ");

                                   linhas[contador-1] = "< " + contador + " , { ";

                                   n = 0;

                                   while (n < Numero_de_Membros_do_Grupo)
                                   {
                                       o1.print(visao[n]);
                                       linhas[contador-1] = linhas[contador-1] + visao[n];
                                       n++;
                                       if (n < Numero_de_Membros_do_Grupo)
                                       {
                                         o1.print(" , ");
                                         linhas[contador-1] = linhas[contador-1] + " , ";
                                       }
                                    }
                                    o1.println(" } >");
                                    linhas[contador-1] = linhas[contador-1] + " } >";
                                   }
      catch (Exception e1)
      {
       System.out.println("Problemas na Gravação do Texto");
      }

         // Atualiza o arquivo texto com o histórico de visões
         // armazenado no vetor linhas

         try
          {
           f1 = new FileOutputStream("Visao.txt");
           o1 = new PrintStream(f1);
```

```
            o1.println ("Histórico de Visões");
            j = 0;
            while (j < contador)
              {
                o1.println (linhas[j]);
                j = j + 1;
              }
           }
         catch (Exception ee)
           {
            System.out.println("Problemas na Gravação do Texto no unir_ao_grupo");
           }

       // Atualização da visão nos membros
       j = 1;
       while (j < Numero_de_Membros_do_Grupo)
          {
           try {
           Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
           // Mudanca_de_Visao servico_visao= (Mudanca_de_Visao) Naming.lookup("rmi://" + visao[j] +
"/Mudanca_de_Visao");
           System.out.println ("Atualizando Visao na Estacao " + visao[j] + " ");
           System.out.println (servico.atualizar_visao(visao,Numero_de_Membros_do_Grupo, linhas, contador));
           }
           catch (Exception ee3)
           {
             System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
           }
           j = j + 1;
         }

                              System.out.println("Designação do novo coordenador efetuada, o eleito foi: " +
visao[0]);

                              chamada_remota = (Servicos) Naming.lookup
                                        ("rmi://" + visao[0] + "/Servicos");
                              chamada_carteiro = (Carteiro) Naming.lookup
                                         ("rmi://" + visao[0] + "/Carteiro");

                              System.out.println("Redirecionar Coordenador: " +
                              chamada_remota.redirecionar_coordenador(visao,Numero_de_Membros_do_Grupo,
linhas , contador));
                            }

                          break;
                      case 3:
                          System.out.print ("Digite a sua identificacao: ");
                          line = din.readLine();
                          System.out.println("=======");
                          try
                          {
                           System.out.println ("Resposta ao pedido de sa¡da do grupo: " +
                           chamada_remota.sair_do_grupo(line));
                          }
                          catch (Exception e)
                          {
                            System.out.println("Coordenador Suspeito de Falha");
                            System.out.println("Redirecionar Coordenador !!!");
                            // Call registry for PowerService
                            chamada_remota = (Servicos) Naming.lookup
```

```
                    ("rmi://" + visao[1] + "/Servicos");
                  System.out.println("Redirecionar Coordenador: " +
chamada_remota.redirecionar_coordenador(visao,Numero_de_Membros_do_Grupo, linhas , contador));
                  // Remover(0);
                    i = n ;
                    while (i < Numero_de_Membros_do_Grupo)
                    {
                     visao[i] = visao[i+1];
                     i = i + 1;
                    }
                  //
                    Numero_de_Membros_do_Grupo = Numero_de_Membros_do_Grupo - 1;
                 }
                                    System.exit(0);
                 // break;
            case 4: j = 0;
                 System.out.println ("Membros do Grupo");
                 System.out.println ();

                 while (j < Numero_de_Membros_do_Grupo)
                    {
                     System.out.println (visao[j]);
                     j = j + 1;
                    }
                 System.out.println ();
                 System.out.println ("Histórico de Visões");
                 j = 0;
                 while (j < contador)
                    {
                     System.out.println (linhas[j]);
                     j = j + 1;
                    }
                 System.out.println ();

                 break;
            case 5:
                    try {
                    Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[0] + "/Servicos");
                    System.out.println ();
                    System.out.println ("Resposta do Monitor de Falhas " +
servico.Monitor_de_Falhas());
                    }
                    catch (Exception e)
                     {
                      System.out.println("Eu estou morto");
                     }
                    break;
            case 6:
                                    System.exit(0);
                          default :
        System.out.println ("Opção Inválida. Digite 1, 2, 3 ou 4.");
                                    break;
                  }
              }
         }
}
```

# Anexo 4
## Código do arquivo Implementa_Carteiro.java

Código fonte do arquivo Implementa_Carteiro.java, que implementa o envio de mensagens a todos ou a um único membro do grupo.

```
import java.rmi.server.*;
import java.net.*;
import java.rmi.Naming;
import java.io.*;
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Hashtable;
import java.util.Enumeration;
import java.util.Calendar;

public class Implementa_Carteiro extends UnicastRemoteObject implements Carteiro {

        // Construtor ----------------------------------------------------------

    public Implementa_Carteiro() throws RemoteException {
                super();
    }

    // Envio de mensagem em multicast
    public String enviar_mensagem (String mensagem, String visao[], int n_membros) throws
RemoteException
    {

     FileOutputStream f1;
     PrintStream o1;

     int j;
     // Atualização da visão nos membros
     j = 0;
     while (j < n_membros)
        {
         try {
         Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
         // Carteiro servico_visao = (Carteiro) Naming.lookup("rmi://" + visao[j] + "/Carteiro");
         System.out.println ();
         System.out.println ("Voce esta ativo " +  visao[j] + " ?  Resposta: " + servico.sinalizar());
         }
         catch (Exception e)
         {
           System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");

           System.out.println("Processo Suspeito de Falha");
           try {
               Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[0] + "/Servicos");
               System.out.println ();
               System.out.println ("Resposta do Monitor de Falhas " + servico.Monitor_de_Falhas());
              }
           catch (Exception ee2)
             {
              System.out.println("Coordenador suspeito de falha");
             }
```

```
            }
            j = j + 1;
          }

        j = 0;
        while (j < n_membros)
           {
            try {
            Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
            System.out.println ();
            System.out.println (servico.entregar_mensagem(mensagem));
            }
            catch (Exception ee3)
            {
              System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
            }
            j = j + 1;
          }
        return ("Ok");
           }

       // Envio de mensagem em unicast
       public String enviar_unicast (String mensagem, String visao[], int n_membros, String identificacao)
throws RemoteException
       {

        FileOutputStream f1;
        PrintStream o1;

        int j;
        // Atualização da visão nos membros
        j = 0;
        while (j < n_membros)
           {
            try {
            Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[j] + "/Servicos");
            // Carteiro servico_visao = (Carteiro) Naming.lookup("rmi://" + visao[j] + "/Carteiro");
            System.out.println ();
            System.out.println ("Voce esta ativo " +  visao[j] + " ?  Resposta: " + servico.sinalizar());
            }
            catch (Exception e)
            {
              System.out.println("Objeto Monitorado " + visao[j] + " Suspeito de Falha");
              System.out.println("Processo Suspeito de Falha");
              try {
                  Servicos servico = (Servicos) Naming.lookup("rmi://" + visao[0] + "/Servicos");
                  System.out.println ();
                  System.out.println ("Resposta do Monitor de Falhas " + servico.Monitor_de_Falhas());
                 }
              catch (Exception ee2)
                 {
                  System.out.println("Coordenador suspeito de falha");
                 }
            }
            j = j + 1;
          }

            try {
            Servicos servico = (Servicos) Naming.lookup("rmi://" + identificacao + "/Servicos");
            System.out.println ();
```

```
            System.out.println (servico.entregar_mensagem(mensagem));
            }
            catch (Exception ee3)
            {
              System.out.println("Objeto Monitorado " + identificacao + " Suspeito de Falha");
            }

           return ("Ok");
           }

}
```

# Anexo 5
# Mensagens da Universidade de Cornell

Foram solicitadas informações através da lista do Horus, sobre a identificação de processos utilizados no sistema, onde foram recebidas respostas de três pesquisadores de Cornell, envolvidos no projeto. As respostas foram enviadas por Tim Clark (quadro A), Robbert van Renesse (quadro B) e Mark Haiden (quadro C).

Subject: Re: Identifier of Processes
Date: Tue, 31 Aug 1999 10:18:57 -0400 (EDT)
From: Tim Clark <tclark@CS.Cornell.EDU>
To: rss@pintado.msinternet.com.br

Robson,

I believe you are asking about endpoint ids?

Each communication endpoint (can have many per process) in Ensemble is assigned an endpoint id, which is generally available from the accepted_view() callback. (They are
printed out by the hot_test2 demo program, see hot_test2.c for more details).

They are of the form:

{Endpt:128.84.254.30:40252:572:0}
{Endpt:128.84.218.5:40120:139:0}

Other than that, every process under Unix is assigned a process ID (pid) which you can find using "ps".

--Tim

Subject: RE: HELP
Date: Tue, 31 Aug 1999 09:05:54 -0400
From: Robbert van Renesse <rvr@cs.cornell.edu>
To: "'rss@pintado.msinternet.com.br'" <rss@pintado.msinternet.com.br>

You'll probably will get a better response to your second message of the mailing list, but basically, the identifier consists of

Transport protocol type (e.g., TCP/IP)
Transport protocol address (e.g., 128.84.218.1:3456)
Time of address creation (in seconds since beginning of 1970)

Other information, such as the host name, may be included, but will not make the identifier more unique. In some cases, an extra demultiplexer is added for further precision.

Robbert

-----Original Message-----
From: Robson Soares Silva [mailto:rss@pintado.msinternet.com.br]
Sent: Monday, August 30, 1999 11:18 PM
To: rvr@CS.Cornell.EDU
Subject: HELP

Hello:

My name is Robson Soares Silva, and I am studying na UFRGS (University of Rio Grande do Sul - Brazil).

Please, help me.

What is used to identifier of processes in the Horus and Ensemble ?

Thank you, and I wait for reply.

Robson

-----Mensagem original-----
De: hayden@pa.dec.com <hayden@pa.dec.com>
Para: rss@pintado.msinternet.com.br <rss@pintado.msinternet.com.br>
Cc: horus-L@cornell.edu <horus-L@cornell.edu>
Data: Sexta-feira, 3 de Setembro de 1999 21:03
Assunto: Re: Identifier of Processes
>
>Horus uses endpoint identifiers which contain an IP address and UDP port (for the UDP transport). They also contain MUX values and incarnation numbers to guarantee uniqueness.
>
>Ensemble splits "identifiers" into two pieces: endpoint identifiers and addresses. Endpoint identifiers are opaque data structures that are "guaranteed" to be unique for a particular endpoint. (Actually, we sometimes put IP addresses and port numbers into Ensemble endpoints, but they are only used to help guarantee uniqueness.)
>Ensmble addresses contain information used for sending messages to na associated endpoint. For the UDP transport, addresses contain just an IP address and a UDP port number.
>
>A "logical endpoint" in Ensemble will have one endpoint identifier and one or more addresses associated with it, and the addresses associated with the "logical endpoint" may change over its lifetime, although the endpoint identifier usually will not.
>
>--Mark
>
>Date: Tue, 31 Aug 1999 00:14:39 -0300
>To: horus-L@cornell.edu
>From: Robson Soares Silva <rss@pintado.msinternet.com.br>
>Subject: Identifier of Processes
>Reply-To: rss@pintado.msinternet.com.br
>Precedence: bulk
>X-Ph: V4.1@router2.mail.cornell.edu (Cornell Modified)
>X-Listprocessor-Version: 7.2(a) -- ListProcessor by CREN
>>Hello:
>>My name is Robson Soares Silva, and I am studying na UFRGS (University
>>of Rio Grande do Sul - Brazil).
>>
>>Please, help me.
>>What is used to identifier of processes in the Horus and Ensemble ?
>>Thank you, and I wait for reply.
>>
>>Robson

# Bibliografia


[ABB 89] ABBADI, A. El; TOUEG, S. Maintaining availability in partitioned replicated databases. **ACM Transactions on Database Systems**, New York, v. 14, n. 2, p.264-290, June 1989.

[AMA 2001] AMARAL, Jeferson B. **Definição de classes para comunicação em *unicast* e *multicast*.** 2001. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[BAB 98] BABAOGLU, Özalp; BARTOLI, Alberto; DINI, Gianluca. **Enriched view synchrony:** a programming paradigm for partitionable asynchronous distributed systems. Bologna: Department of Computer Science of University of Bologna, 1996.

[BER 2000] BERTAGNOLLI, S.C. **Ambiente visual para o desenvolvimento de aplicações Java reflexivas.** 2000. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

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

[BIR 93] BIRMAN, K. The process group approach to reliable distributed computing. **ACM Transactions on Database Systems,** New York, v.36, n.12, p.37-53,103, Dec. 1993.

[BIR 96] BIRMAN, K. **Building secure and reliable network applications.** Greenwich: Manning Publ., 1996. 591p.

[BIR 99] BIRMAN, K. **A review of experiences with reliable multicast.** Ithaca: John Wiley & Sons, 1999.

[BUR 97] BURNS, Alan; WELLINGS, Andy. **Real time systems and programming languages.** Harlow: Addison Wesley, 1997. 611p.

[CAR 97] CARDOSO, Afonso Jorge Ferreira. **Servidores de arquivos duplicados em redes locais com ambiente UNIX.** 1997. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CEC 98] CECHIN, Sérgio Luis. **Avaliação teórica do desempenho de algoritmos de recuperação por retorno do tipo síncrono e assíncrono.** 1998. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[CES 98] CESTA, André Augusto. **Tutorial:** a linguagem de programação Java – orientação a Objetos. Campinas: IC – UNICAMP, 1996.

[CHA 95] CHANDRA, T.D.; TOUEG, S. **Unreliable failure detectors for reliable distributed systems.** Ithaca, NY: Departamento de Ciência da Computação da Universidade de Cornell, 1995.

[CHO 97] CHOW, Randy; JOHNSON, Theodore. **Distributed operation systems & algorithms.** California: Addison-Wesley, 1997. 569p.

[CLEM 99] CLEMSON UNIVERSITY. **Transfering an object via remote method call.** Disponível em: <http://www.cs.clemson.edu/~ihcho/java-rmi/objtransfer.html>. Acesso em: 29 fev. 1999.

[COA 98] COAD, Peter; MAYFIELD, Mark. **Projeto de sistemas em Java:** construindo aplicativos e melhores applets. São Paulo: Makron Books, 1998.

[DEI 2000] DEITEL, H. M.; DEITEL, P. J. **Java, como programar.** 3.ed. Porto Alegre: Bookman, 2000.

[DAM 96] DAMASCENO JUNIOR, Américo. **Aprendendo JAVA:** programação na Internet. São Paulo: Érica, 1996.

[DOV 95] DOVEV, D; MALKI, D. The design of the Transis System. In: DAGSTUHL WPRKSHOP ON UNIFYING THEORY AND PRACTICE IN DISTRIBUTED COMPUTING, 1995. **Proceedings…** [S.l.: s.n.], 1995.

[ENS 98] ENSEMBLE Home Page. Disponível em: <http://simon.cs.cornell.edu/ Info/Projects/Ensemble>. Acesso em: 1998.

[FAR 98] FARLEY, Jim. **Java distributed computing.** California: O'Reilly & Associates, 1998. 365p.

[FEA 97] FEATHER, Stephen. **Javascript em exemplos.** São Paulo: Makron Books, 1997.

[FEL 98] FELBER, Pascal. **The CORBA object group service.** A service approach to object groups in CORBA. 1998. These (Docteur ès Sciences) – École Polytechnique Fédérale de Lausanne, Lausanne.

[FEL 98a] FELBER, Pascal; GUERRAOUI, Rachid; SCHIPER, André. **The implementation of a CORBA object group service.** Lausanne: École Polytechnique Fédérale de Lausanne, Département d´ Informatique, 1998.

[FER 2000] FERREIRA FILHO, João Carlos. **Implementação de objetos replicados usando Java.** 2000. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[FRI 95] FRIEDMAN, Roy; RENESSE, Robert Van. **Strong and weak virtual syncrony in Horus.** Ithaca, NY: Cornell University, 1995. Technical Report.

[FRI 95a] FRIEDMAN, Roy. **Using virtual synchrony to develop efficient fault tolerant distributed shared memories.** Ithaca, NY: Department of Computer Science – Cornell University, 1995.

[GAR 91] GARCIA-MOLINA, Hector; SPAUSTER, Annemarie. Ordered and reliable multicast communication. **ACM Transactions on Database Systems**, New York, v. 9, n. 3, p. 242-269, August 1991.

[GAR 98] GARBINATO, Benoit. **Protocol objects and patterns for structuring reliable distributed systems.** 1998. These (Docteur ès Sciences) – École Polytechnique Fédérale de Lausanne, Lausanne.

[GIF 79] GIFFORD, D. K. **Weigthed voting for replicated data**. In: SYMPOSIUM ON OPERATING SYSTEMS PRINCIPLES, 7., 1979, Pacific Grove. **Proceedings…** New York: ACM/SIGOPS, 1979. p. 150-159.

[GOS 91] GOSCINCSKI, Andrzej; BEARMAN, Mirion. **Resource management in large distributed systems.** New South Wales: The University of New South Wales Australian Defense Force Academy Canberra, 1991.

[GUE 96] GUERRAOUI, R.; SCHIPER, A. Fault-Tolerance by replication in distributed systems. In: ADA – EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 1996. **Proceedings…** Berlin: Springer-Verlag, 1996. p. 38-57. (Lecture Notes in Computer Science, v. 1088).

[GUE 97] GUERRAOUI, R.; SCHIPER, A. Software-based replication for fault tolerance. **Computer**, New York, v. 30, n. 4, p. 68-74, April 1997.

[GUE 97a] GUERRAOUI, R.; GARBINATO, B.; MAZOUNI, K. R. Garf: a tool for programming reliable distributed applications. **IEEE Concurrency**, Los Alamitos, v. 5, n. 4, p32-39, Oct.-Dec. 1997.

[GUE 98] GUERRAOUI, R.; FELBER, P.; GARBINATO, B.; MAZOUNI, K. System suport for object groups. **SIGPLAN Notices**, New York, v. 33, n. 10, Oct. 1998. Trabalho apresentado na Conference on object oriented programming, Systems, Languages, and Applications, OOPSLA, 1998.

[GUO 96] GUO, K.; VOGELS, W.; RENESSE, R.V. Structured virtual synchrony: exploring the bounds of virtual synchronous group communication. In: EUROPEAN WORKSHOP, 1996. **Proceedings…** [S:l.: s.n.], 1996.

[HAD 94] HADZILACOS, V.; TOUEG, S. **A modular approach to faul-tolerant broadcasts and related problems.** Ithaca, NY: Dept. of Computer Science at Cornell Univerity, 1994. (Technical Report n. 94-1425).

[HER 86] HERLIHY, M. A quorum-consensus replication method for abstract data types. **ACM Transactions on Computer Systems**, New York, v. 4, n. 1, p.32-53, Feb. 1986.

[JAL 94] JALOTE, P. **Fault tolerance in distributed systems.** New Jersey: Prentice-Hall, 1994. 432p.

[JGR 99] THE JGROUP Project. Disponível em: <http://www.cs.unibo.it/projects/jgroup>. Acesso em: ago. 1999.

[KAR 97] KARMARKAR, Anish. **A framework for communication in object oriented distributed systems.** 1997. These (Docteur ès Sciences) – Texas A&M University, Texas.

[LIA 90] LIANG, L.; CHANSON, S.; NEUFELD, G. Process groups and group communications: classifications and requirements. **IEEE Computer**, New York, p.56-66, Feb. 1990.

[LIS 95] LISBÔA, M.L. **MOTF:** Meta-objetos para tolerância a falhas. 1995. Tese (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[LOT 95] LOTEM, Esti Yeger. **Transis documentation.** Disponível em: < http://www.cs.huji.ac.il/labs/transis/lab-projects/guide/intro.html>. Acesso em: ago. 1999.

[MAL 96] MALLOTH, Christoph Peter. **Concetion and implementation of a toolkit for building fault-tolerant distributed applications in large scale networks.** 1996. Thèse. (Docteur ès Sciences) – Ingénieur informaticien diplômé de I´École Polytechnique Fédérale de Lausanne, Lausanne.

[MON 99] MONTRESOR, Alberto. **The Jgroup reliable distributed object model.** Bologna: University of Bologna, March 1999. (Technical Report UBLCS-9-12).

[MUL 96] MULLENDER, Sape. **Distributed systems.** 2nd ed. New York: ACM Press Books, 1996. 600p.

[NUN 98] NUNES, R. C. **Programação orientada a grupos:** o ponto de vista das aplicações. 1998. Trabalho Individual (Mestrado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[NUN 99] NUNES, R. C. **Suporte a partições de rede no serviço de comunicação de grupo.** 1999. Exame de Qualificação (Doutorado em Ciência da Computação) – Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre.

[PAS 98] PASIN, Marcia; LEITE, Fábio; AMARAL, Jeferson Botelho. **Um sistema de arquivos replicado e distribuído para um ambiente com comunicação de grupo confiável.** Porto Alegre: CPGCC-UFRGS, 1998. (Projeto de Pesquisa, n.CMP301).

[PAS 99] PASIN, Marcia; JANSCH-PÔRTO, Ingrid; LEITE, Fábio Olivé; AMARAL, Jeferson Botelho. Implementando réplicas de arquivos através de uma ferramenta de comunicação de grupo confiável. In: SIMPÓSIO DE COMPUTAÇÃO TOLERANTE A FALHAS, 8., 1999, Campinas. **Anais...** Campinas: Unicamp, 1999. p.84-98.

[PRA 96] PRADHAN, D. **Fault-tolerant computer system design.** New Jersey: Prentice Hall, 1996.

[QUA 97] QUADROS, Elbson; RUBIRA, Cecilia. Construção de um framework para sistemas controladores de trens utilizando padrões de projeto e metapadrões. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, SBES, 11., 1997. **Anais…** Fortaleza: UFC, 1997. p. 99-114.

[REL 99] RELACS Project. Disponível em: <http://www.cs.unibo.it/projects/ relacs/>. Acesso em: 1999.

[REN 96] RENESSE, R. V.; BIRMAN, K.; MAFFEIS, S. Horus: a flexible group communication system. **ACM Transactions on database systems**, New York, v.39, n.4, p.76-83, Apr. 1996.

[SCH 83] SCHLICHTING, R. D.; SCHNEIDER, F.B. Fail-stop processors: An aproach to designing fault-tolerant computing systems. **ACM Transactions on Computer Systems** (TOCS), New York, v. 1, n.3, p. 222-238, 1983.

[SCH 93] SCHIPER, A.; SANDOZ, A. Understand the power of the virtually synchronous model. In: EUROPEAN WORKSHOP ON DEPENDABLE COMPUTING, 5. 1993. **Proceedings...** [S.l.: s.n.], 1993.

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

[SHR 95] SHRIVASTAVA, S.K. Lessons learned from building and using the arjuna distributed programming system. Invited talk at the 6th Symp. on Fault Tolerant Computers, Aug. 1995.

[SIN 97] SINGHAL, R.; SHIVARATRI, A. **Advanced concepts in operating systems.** New York: McGraw-Hill, 1994. 522 p.

[SIN 97a] SINGHAL, R.; SHIVARATRI, A. Distributed systems. In: ADA-EUROPE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE TECHNOLOGIES, 1997. **Reliable Software Technologies:** proceeding. Berlin: Springer-Verlag, 1997. p. 38-57. (Lecture Notes in Computer Science, v. 1088).

[SUL 97] SULEIMAN, Lalani; JAMSA, Kris. **Java:** biblioteca do programador. São Paulo: Makron Books, 1997.

[SUN 99] SUN MICROSYSTEMS. **An overview of RMI applications.** Disponível em: <http://java.sun.com/docs/tutorial/rmi/overview.html>. Acesso em: 1999.

[TAN 95] TANENBAUM, Andrew. **Sistemas operacionais modernos.** Rio de Janeiro: Prentice Hall do Brasil, 1995. 493 p.

[WUT 98] WUTKA, Mark. **Java técnicas profissionais.** São Paulo: Berkeley, 1997.

[ZOU 98] ZOU, Hengming; JAHANIAN, Farnam. Real-time primary-backup (RTPB) replication with temporal consistency guarantees. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS, ICDCS, 18., 1998. **Proceedings…** [S.l.: s.n.], 1998.