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

Definição de Classes para Comunicação *Unicast* e *Multicast*

por

JEFERSON BOTELHO DO AMARAL

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

Profa. Ingrid E. S. Jansch-Pôrto
Orientadora

Porto Alegre, março de 2001.

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Amaral, Jeferson Botelho do

Definição de Classes para Comunicação Unicast e Multicast / Jeferson Botelho do Amaral – Porto Alegre: PPGC da UFRGS, 2001.

129p.:il.

Dissertação (Mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, Brasil-RS, 2001. Orientadora: Profa. Dra. Ingrid E. S. Jansch-Pôrto

1. Orientação a objetos 2. Java 3. *Framework* 4. Protocolos de comunicação 5. *Unicast* 6. *Multicast*. I. Jansch-Pôrto, Ingrid E. S. II. Título

Agradecimentos

Se eu tivesse que citar nominalmente cada uma das pessoas que merecem o meu agradecimento, indubitavelmente acabaria esquecendo-me de alguém. Desta forma, quero agradecer a todas as pessoas que direta ou indiretamente colaboraram na execução desta dissertação.

Em especial, entretanto, agradeço aos colegas do grupo de Tolerância a Falhas: Ceretta, Cechin, Emerson, Fabio, Silvia, Lenise, Roger, Marcia; agradeço à orientação imprescindível e paciente da Profa. Dra. Ingrid E. S. Jansch-Porto; agradeço ao CNPq pela bolsa, com a qual consegui incentivo; agradeço ao meu pai e aos meus irmãos, pelo que representam em minha vida; e, principalmente, agradeço à Liane, por estar sempre ao meu lado e por fazer-me feliz.

Oferecimento

Ofereço esta dissertação a quem me ofereceu calor, alimento e afeto durante 9 meses; a quem, feliz, me viu nascer e dar os primeiros passos na vida; a quem me levou ao médico, quando estava doente; a quem me levou à escola e me ensinou humildade; a quem me incentivou sempre a não desistir de alcançar um objetivo e desejava, acima de tudo, que eu fosse feliz.

Mãe, ofereço esta dissertação a você, por tudo que fez por mim e pelo que representa em minha vida.

À Marina Botelho do Amaral,
in memoriam.

Sumário

Lista de Figuras 7

Lista de Tabelas 9

Resumo 10

Abstract 11

1 Introdução 12

1.1 Sistemas Distribuídos 14
1.2 Serviços e Protocolos 16
1.3 Orientação a Objetos 16
1.4 Estrutura desta Dissertação 20

2 Infra-estruturas para o Desenvolvimento de Aplicações Distribuídas 22

2.1 Ordenação 22
2.2 TCP 24
2.3 UDP 25
2.4 Multicast IP 25
2.5 Comunicação em Grupo 26
2.6 Frameworks 28

3 Projeto de Serviços de Comunicação Confiável 30

3.1 Tipo da Rede de Comunicação 33
3.2 Topologia da Rede de Comunicação 33
3.3 Relacionamento dos Eventos 36
3.4 Sincronismo do Sistema 37
3.5 Modelo de Falhas de Nodos e de Canais de Comunicação 38
3.6 Detecção de Defeitos 42
3.7 Métodos de Obtenção de Confiabilidade 44
3.7.1 Enfoque Baseado em Redundância 44
3.7.2 Enfoque Orientado a Emissor 45
3.7.3 Enfoque Orientado a Receptor 47
3.7.4 Enfoque Combinado: Orientado a Emissor e Orientado a Receptor 50
3.7.5 Enfoque Hierárquico 52

4 Protocolos de Comunicação 54

4.1 Unicast Não-Confiável 54
4.2 Unicast Confiável 55
4.2.1 Exemplos de Execução do Algoritmo 57
4.3 Multicast Não-Confiável 58
4.4 Multicast Confiável 60
4.4.1 Multicast Confiável com Múltiplos Unicasts 61
4.4.2 Multicast Confiável com Primitivas de Multicast 63

5 Classes para Comunicação 66

5.1 Modelagem em UML 66
5.2 Descrição das Classes Propostas 70
5.2.1 Interface DataHandler 70
5.2.2 Classe Refs 71
5.2.3 Classe Buffer 73
5.2.4 Classe Node 73
5.2.5 Classe SequenceNumber 74
5.2.6 Classe Member 75
5.2.7 Classe Group 75
5.2.8 Classe Serial 76
5.2.9 Classe Unicast 77
5.2.10 Classe RUnicast 78
5.2.11 Classe Multicast 79
5.2.12 Classe RMulticast 80
5.2.13 Classe Register 81
5.2.14 Classe Communication 83
5.2.15 Classe NameServer 85

6 Utilização das Classes 86

6.1 Envio em Unicast Não-Confiável 86
6.2 Envio em Unicast Confiável 89
6.3 Envio em Multicast Não-Confiável 92
6.4 Envio em Multicast Confiável 95
6.5 Utilizando um Servidor de Nomes 102

7 Conclusões 107

7.1 Trabalhos Futuros 108

Anexo 1 Código Fonte 110

Bibliografia 126

Lista de Figuras

FIGURA 2.1 - Comunicação sem ordenação das mensagens 23
FIGURA 3.1 - Camada de comunicação confiável 30
FIGURA 3.2 - Comunicação confiável em um sistema distribuído tolerante a falhas 32
FIGURA 3.3 - Topologia do tipo barramento 34
FIGURA 3.4 - Topologia do tipo estrela 35
FIGURA 3.5 - Topologia do tipo anel 35
FIGURA 3.6 - Relacionamento entre as primitivas *send*, *receive* e *deliver* 37
FIGURA 3.7 – Classificação do Modelo de Defeitos 40
FIGURA 3.8 - Enfoque baseado em redundância 45
FIGURA 3.9 - Enfoque orientado a emissor sem perda de mensagens 46
FIGURA 3.10 - Enfoque orientado a emissor com perda de mensagens 47
FIGURA 3.11 - Primeira variação do enfoque orientado a receptor 48
FIGURA 3.12 - Segunda variação do enfoque orientado a receptor 49
FIGURA 3.13 - Terceira variação do enfoque orientado a receptor 50
FIGURA 3.14 - Enfoque combinado sem defeitos 51
FIGURA 3.15 - Enfoque combinado com perda de mensagem 51
FIGURA 3.16 - Exemplo de uma estrutura hierárquica 52
FIGURA 3.17 - Enfoque hierárquico sem defeitos 53
FIGURA 4.1 - Algoritmo para comunicação *unicast* não-confiável 54
FIGURA 4.2 - Comunicação *unicast* não-confiável 55
FIGURA 4.3 - Algoritmo para comunicação *unicast* confiável 56
FIGURA 4.4 - *Unicast* confiável sem defeito 57
FIGURA 4.5 - *Unicast* confiável na ocorrência de perda de mensagem 57
FIGURA 4.6 - *Unicast* confiável na ocorrência de duplicação de mensagem 58
FIGURA 4.7 - *Unicast* confiável na ocorrência de defeito no receptor 58
FIGURA 4.8 - Algoritmo para comunicação *multicast* não-confiável via múltiplos *unicasts* 59
FIGURA 4.9 - *Multicast* não-confiável utilizando *unicasts* 59
FIGURA 4.10 - Algoritmo para comunicação *multicast* não-confiável com *multicast* 59
FIGURA 4.11 - *Multicast* não-confiável utilizando *multicast* 60
FIGURA 4.12 - Algoritmo para *multicast* confiável utilizando *unicast* 62
FIGURA 4.13 - *Multicast* confiável (*unicast*) sem defeito 63
FIGURA 4.14 - *Multicast* confiável (*unicast*) com defeito do emissor 63
FIGURA 4.15 - Algoritmo para *multicast* confiável utilizando suporte real de *multicast* 64
FIGURA 4.16 - *Multicast* confiável (*multicast*) sem defeito 64
FIGURA 4.17 - *Multicast* confiável com defeito em um dos receptores 65
FIGURA 5.1 – Diagrama das classes propostas 68
FIGURA 5.2 - Interface *DataHandler* 70
FIGURA 5.3 - Exemplo de uso de *DataHandler* 71
FIGURA 5.4 - Classe Refs 71
FIGURA 5.5 - Classe Buffer 73
FIGURA 5.6 - Classe Node 73
FIGURA 5.7 - Classe SequenceNumber 74
FIGURA 5.8 - Classe Member 75
FIGURA 5.9 - Classe Group 76
FIGURA 5.10 - Classe Serial 76
FIGURA 5.11 - Classe *Unicast* 78
FIGURA 5.12 - Classe *RUnicast* 78
FIGURA 5.13 - Classe *Multicast* 79

FIGURA 5.14 - Classe *RMulticast* 80
FIGURA 5.15 - Classe Register.....82
FIGURA 5.16 - Classe Communication 84
FIGURA 5.17 - Classe NameServer.....85
FIGURA 6.1 - Envio de *unicast* não-confiável de O_e para O_r 86
FIGURA 6.2 - Código Java da classe do objeto emissor O_e (*unicast* não-confiável).....87
FIGURA 6.3 - Código Java da classe do objeto receptor O_r (*unicast* não-confiável) 88
FIGURA 6.4 - Relacionamento entre camadas durante *unicast* não-confiável 89
FIGURA 6.5 - Formato das mensagens do tipo *unicast* não-confiável 89
FIGURA 6.6 - Envio de *unicast* confiável de O_e para O_r 89
FIGURA 6.7 - Código Java da classe do objeto emissor O_e (*unicast* confiável).....90
FIGURA 6.8 - Código Java da classe do objeto receptor O_r (*unicast* confiável) 91
FIGURA 6.9 - Relacionamento entre camadas durante *unicast* confiável 92
FIGURA 6.10 - Formato das mensagens do tipo *unicast* confiável 92
FIGURA 6.11 - Envio de *multicast* não-confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE} 93
FIGURA 6.12 - Código Java da classe do objeto emissor O_e (*multicast* não-confiável).....93
FIGURA 6.13 - Código Java da classe do objeto receptor O_r (*multicast* não-confiável) 94
FIGURA 6.14 - Relacionamento entre camadas durante *multicast* não-confiável 95
FIGURA 6.15 - Formato das mensagens do tipo *multicast* não-confiável 95
FIGURA 6.16 - Envio de *multicast* confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE} via *unicast*.....96
FIGURA 6.17 - Envio de *multicast* confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE} via *multicast* IP 96
FIGURA 6.18 - Código Java da classe do objeto emissor O_e (*multicast* confiável/*unicast*) 97
FIGURA 6.19 - Código Java da classe do objeto emissor O_e (*multicast* confiável/*multicast*) 98
FIGURA 6.20 - Código Java da classe do objeto receptor O_r (*multicast* confiável) 99
FIGURA 6.21 - Relacionamento entre camadas durante *multicast* confiável via *unicast*.....100
FIGURA 6.22 - Formato das mensagens do tipo *multicast* confiável (via *unicasts*).....100
FIGURA 6.23 - Relacionamento entre camadas durante *multicast* confiável via *multicast*.....101
FIGURA 6.24 - Formato das mensagens do tipo *multicast* confiável (*multicast* IP).....101
FIGURA 6.25 – Funcionamento interno do servidor de nomes 102
FIGURA 6.26 - Código Java do objeto emissor O_e que utiliza o servidor de nomes 104
FIGURA 6.27 - Código Java da classe do objeto receptor O_r (*multicast* confiável) 105
FIGURA 6.28 - Relacionamento entre camadas utilizando a classe Communication.....105

Lista de Tabelas

TABELA 3.1 - Oito classes de detectores de defeitos não-confiáveis 43

Resumo

No projeto de arquiteturas computacionais, a partir da evolução do modelo cliente-servidor, surgiram os sistemas distribuídos com a finalidade de oferecer características tais como: disponibilidade, distribuição, compartilhamento de recursos e tolerância a falhas. Estas características, entretanto, não são obtidas de forma simples. As aplicações distribuídas e as aplicações centralizadas possuem requisitos funcionais distintos; aplicações distribuídas são mais difíceis quanto ao projeto e implementação. A complexidade de implementação é decorrente principalmente da dificuldade de tratamento e de gerência dos mecanismos de comunicação, exigindo equipe de programadores experientes. Assim, tem sido realizada muita pesquisa para obter mecanismos que facilitem a programação de aplicações distribuídas.

Observa-se que, em aplicações distribuídas reais, mecanismos de tolerância a falhas constituem-se em uma necessidade. Neste contexto, a comunicação confiável constitui-se em um dos blocos básicos de construção.

Paralelamente à evolução tanto dos sistemas distribuídos como da área de tolerância a falhas, foi possível observar também a evolução das linguagens de programação. O sucesso do paradigma de orientação a objetos deve-se, provavelmente, à habilidade em modelar o domínio da aplicação ao invés da arquitetura da máquina em questão (enfoque imperativo) ou mapear conceitos matemáticos (conforme o enfoque funcional). Pesquisadores demonstraram que a orientação a objetos apresenta-se como um modelo atraente ao desenvolvimento de aplicações distribuídas modulares e tolerantes a falhas.

Diante do contexto exposto, duas constatações estimularam basicamente a definição desta dissertação: a necessidade latente de mecanismos que facilitem a programação de aplicações distribuídas tolerantes a falhas; e o fato de que a orientação a objetos tem-se mostrado um modelo promissor ao desenvolvimento deste tipo de aplicação.

Desta forma, nesta dissertação definem-se classes para a comunicação do tipo *unicast* e *multicast*, nas modalidades de envio confiável e não-confiável. Além destes serviços de comunicação básicos, foram desenvolvidas classes que permitem referenciar os participantes da comunicação através de nomes. As classes estão organizadas na forma de um pacote, compondo um *framework*. Sua implementação foi desenvolvida usando Java. Embora não tivessem sido requisitos básicos, as opções de projeto visaram assegurar resultados aceitáveis de desempenho e possibilidade de reuso das classes. Foram implementados pequenos trechos de código utilizando e testando a funcionalidade de cada uma das classes de comunicação propostas.

Palavras-chave: orientação a objetos, Java, *framework*, protocolos de comunicação, *unicast*, *multicast*.

TITLE: “DEFINITION OF CLASSES FOR UNICAST AND MULTICAST COMMUNICATION”

Abstract

The distributed systems appeared from the evolution of the client-server model, aiming at offering characteristics such as: availability, distribution, resources sharing and fault tolerance. However, these characteristics are not simple to obtain. The distributed and centralized applications have different functional requirements; it is more difficult to design and implement distributed applications. The difficulty related to handling and managing the communication mechanisms explains most of this complexity and demands a team of senior programmers. Researchers have made significant efforts to obtain mechanisms that facilitate the distributed application programming.

It is possible to observe that fault tolerance mechanisms have become a need in existing distributed applications. In this context, to achieve this goal, reliable communication is one of the basic building blocks.

Together with the evolution of distributed systems and fault tolerance domain, a third point is the development of the programming languages. The success of the object-oriented programming is probably due to the ability in modeling the domain of the application instead of the architecture of the machine (imperative model) or to map mathematical concepts (according to the functional approach). Researchers have already demonstrated that the object-oriented programming is an adequate model for the development of fault-tolerant and modular distributed applications.

In this context, two main reasons have motivated the definition of this dissertation theme: the need of mechanisms to facilitate the programming of fault-tolerant distributed applications, and also the fact that the object-oriented programming has been shown to be an adequate model to the development of the same type of applications.

In this way, this dissertation defines classes for unicast and multicast communication services, exploring both reliable and unreliable sending approaches. Besides these basic communication services, classes were developed to allow reference by name to the participants of the communication. The classes are organized in the form of a package, which composes a framework. The implementation was developed in Java. The design options have considered adequate results concerning performance and reuse parameters, although these elements have not been defined as basic requirements. Small programs were implemented to show the usage and to test the functionality of each one of the proposed communication classes.

Keywords: object-orientation, Java, framework, communication protocols, unicast, multicast.

1 Introdução

O avanço tecnológico na área da computação tem sido surpreendentemente rápido, quando comparado ao de outras áreas. Inicialmente, os sistemas de computadores eram altamente centralizados. Uma empresa de porte médio ou universidade poderia ter um ou dois computadores, enquanto as grandes instituições tinham no máximo uma dúzia.

Comparado a um sistema centralizado com terminais remotos, um único computador (*standalone*) possui muitas vantagens do ponto de vista do usuário: os dados são acessados localmente e o usuário tem o poder computacional inteiramente para si. Esta independência permite trabalhar de forma mais rápida, pois não há a sobrecarga ocasionada por outros usuários; entretanto, não é possível a cooperação direta com outros usuários na realização de tarefas computacionais complexas e/ou descentralizadas. As **redes de computadores** surgiram desempenhando um importante papel na sobreposição dessas limitações, permitindo a integração de dois ou mais computadores.

Uma rede de computadores é formada por um conjunto de módulos processadores (nodos) capazes de trocar informações e compartilhar recursos, interligados por um sistema de comunicação. O sistema de comunicação vai se constituir de um arranjo topológico interligando os vários nodos através de enlaces físicos (meios de transmissão) e de um conjunto de regras com o fim de organizar a comunicação (protocolos). Através das redes, uma dada impressora passou a poder ser compartilhada por vários usuários (evitando-se o desperdício de recursos) que, por sua vez, passaram a poder utilizar os mesmos programas (que podem ser padronizados através da rede) e os mesmos dados (evitando-se redundância desnecessária e versões diferentes de um mesmo documento).

É preciso se ter absolutamente claras as razões que levam as empresas a precisarem utilizar as redes. A primeira razão é o **compartilhamento de periféricos**: em um ambiente onde os computadores são interligados através de uma rede, todos os usuários podem compartilhar uma mesma impressora, economizando-se brutalmente nos investimentos e na manutenção dos equipamentos. Outra razão é o **compartilhamento de dados**: se os usuários estão em sistemas independentes, não trabalharão simultaneamente nos mesmos documentos, havendo a necessidade de se copiar os mesmos documentos em diversos computadores (ou mídias) e, pior ainda, a possibilidade de se ter numerosas versões diferentes dos mesmos documentos, já que os usuários irão modificá-los localmente. Uma terceira vantagem do uso das redes é a de que elas possibilitam a **padronização dos aplicativos utilizados** pelos usuários, o que diminui custos de manutenção e suporte. A quarta vantagem é a **possibilidade de os usuários poderem se comunicar através dos computadores**, trocando mensagens ou mesmo documentos de diversos tipos, como planilhas, gráficos ou textos, além de também poderem compartilhar suas agendas para a marcação de eventos e reuniões.

A evolução desta estrutura de rede deu origem ao paradigma **cliente-servidor**, que serviu como base para o desenvolvimento dos sistemas de computação distribuídos. A idéia fundamental da arquitetura de um sistema cliente-servidor é a de particionar uma aplicação em um conjunto de **serviços** (que, por sua vez, fornecem um conjunto de operações a seus usuários) e em **programas-cliente** (implementando aplicações e distribuindo pedidos para os serviços de acordo com as necessidades da aplicação). Neste modelo, os processos da aplicação não cooperam diretamente um com o outro, ao invés compartilham dados e coordenam ações interagindo com um conjunto de servidores comuns, e pela ordem em que os programas são executados [BIR96].

Dentro deste contexto da interligação dos computadores em rede, conceitos como **serviço distribuído** e **aplicações distribuídas** são duas visões diferentes da mesma realidade: **sistemas distribuídos**.

Em sistemas distribuídos, as aplicações possuem requisitos funcionais diferentes das aplicações projetadas para sistemas centralizados. Estes requisitos as tornam mais difíceis de serem projetadas e implementadas do que as aplicações centralizadas. Em ambiente distribuído, a complexidade de implementação se deve, em grande parte, à dificuldade de tratamento e de gerenciamento dos mecanismos de comunicação. Tal fato, indubitavelmente, requer uma equipe de programadores experientes.

A busca de infra-estruturas, visando facilitar a programação de aplicações cujos processos não se localizam em uma mesma máquina, foi sempre uma constante desde que os computadores passaram a se interligar através de estruturas de rede. Neste sentido, originaram-se abstrações como a RPC (*remote procedure call*) [BIR84], cuja idéia básica era de que a programação de aplicações em rede fosse transparente (o programador pode colocar em seu código chamadas a procedimentos localizados em máquinas remotas, de forma similar às chamadas de procedimentos localizados na mesma máquina).

Ainda na década de 80, surgiram tecnologias fazendo uso de RPC, como DCE e ONC. Estas tecnologias representaram propostas de padronização da computação distribuída, introduzindo arquiteturas dentro das quais os componentes maiores de um sistema de computação distribuída teriam comportamentos e interfaces bem definidas, bem como as aplicações poderiam cooperar através da RPC [BIR96]. Em particular, DCE se tornou relativamente um padrão, estando disponível em muitas plataformas [OPE94].

Já na década de 90, entretanto, surgiu uma nova geração de tecnologias com base em RPC, através do OMG (*Object Management Group*), que buscou a padronização da computação que fosse baseada no paradigma orientado a objetos. Em um curto período de tempo, foi possível observar que CORBA [OMG95], uma das tecnologias propostas pelo OMG, ultrapassou as tecnologias que tinham como base a RPC.

Paralelamente, observa-se que foram feitas pesquisas no sentido de desenvolver aplicações distribuídas com confiabilidade. Sistemas de computação distribuída confiáveis são construídos partindo-se de blocos básicos, que (em termos mais simples) são apenas processos e mensagens. É bastante comum observar um sistema distribuído como operando sobre um conjunto de camadas de serviços de rede, com cada camada correspondendo a uma abstração de programação ou característica física. Além disso, a

implementação pode ser feita ao nível da aplicação propriamente dita, por exemplo em uma biblioteca para qual o programa é ligado, no sistema operacional ou mesmo no *hardware* dos dispositivos de comunicação. Neste contexto, por exemplo, surgiram sistemas de comunicação em grupo, como inicialmente pesquisado por Birman [BIR91], que funcionam como *middleware*[1], auxiliando na obtenção de sistemas tolerantes a falhas [JAL94].

Com o que foi até agora descrito, é possível observar uma necessidade latente por mecanismos que facilitem a programação de aplicações distribuídas. Estes mecanismos, por sua vez, podem constituir blocos de construção básicos, que servem para a construção de aplicações ou de outros blocos. Além disso, tem-se observado uma evolução constante das linguagens de programação orientadas a objetos, bem como sua utilização na construção de aplicações distribuídas.

Tendo estes aspectos motivadores, esta dissertação tem o intuito de fornecer serviços de comunicação básicos visando facilitar a programação de aplicações distribuídas, bem como, utilizando estes serviços como base, possibilitar a construção de outros serviços. Como a idéia é a de fornecer blocos de construção, os serviços propostos formam um pacote com os seguintes tipos básicos de comunicação não-confiável e confiável: ***unicast* não-confiável**, ***unicast* confiável**, ***multicast* não-confiável** e ***multicast* confiável**.

A seguir, descrevem-se muitos dos conceitos que antes foram comentados de forma resumida. A seção 1.1 contém uma introdução aos sistemas distribuídos; a seção 1.2 expõe a relação entre serviços e protocolos; a seção 1.3 justifica a utilização da orientação a objetos e da linguagem Java nesta dissertação; e a seção 1.4 descreve como estão organizados os capítulos desta dissertação.

1.1 Sistemas Distribuídos

Há uma considerável confusão na literatura na definição de **rede de computadores** e de **sistema distribuído**. A distinção fundamental é que, em um sistema distribuído, a existência de computadores autônomos é transparente (não visível) para o usuário, que não percebe a existência de múltiplos processadores; tudo parece como um monoprocessador virtual. Efetivamente, um sistema distribuído é um caso especial de rede, cujo *software* lhe fornece maior grau de coesão e transparência. Desta forma, a distinção entre uma rede e um sistema distribuído está no *software* (especialmente no sistema operacional) e não no *hardware* [TAN94].

Um sistema distribuído consiste de duas ou mais entidades de computação (ou nodos), que não compartilham um espaço de endereçamento de memória física, nem possuem um relógio global, e se comunicam através de troca de mensagens. Os nodos são conectados por uma rede de comunicação (*Local Area Network* - LAN, *Metropolitan Area Network* - MAN ou *Wide Area Network* - WAN) e executam cooperativamente **aplicações distribuídas** para fornecer **serviços distribuídos**. A *Internet* (*World Wide Web* - WWW) é hoje exemplo ativo de ambiente distribuído, onde

[1] Ambientes ou ferramentas de programação localizadas entre a aplicação e a infra-estrutura de comunicação.

as redes de comunicação têm se desenvolvido e interligado cada vez mais as empresas, universidades, instituições governamentais, enfim, as pessoas pelo mundo todo.

A *Internet* pode ser vista como um imenso agregado de serviços e de aplicações distribuídas. Os usuários da *Internet* têm liberdade na publicação de materiais e podem se beneficiar dos materiais que outros publicam. De alguma forma esta evolução não é tão surpreendente, visto que as pessoas são naturalmente distribuídas e, com elas, as informações. Ferramentas que tentam ocultar a natureza distribuída inerente, por exemplo através de uma interface gráfica (*Graphical User Interface* - GUI), são um fator importante no crescimento exponencial da *Internet*; um navegador (*web browser*) tenta fornecer ao usuário a ilusão de possuir apenas recursos locais.

Embora nossa distribuição natural e nosso desejo de ocultá-la possam parecer contraditórios, ambos contribuem para adaptar as ferramentas computacionais à nossa forma de pensar. As pessoas construíram os computadores com tecnologias avançadas, mas estão mais familiarizadas com imagens, sons e textos legíveis; em outras palavras, o que a maioria das pessoas quer dos computadores e das redes é a possibilidade de interagir com eles de modo **transparente** [BIR96].

Do ponto de vista do usuário, transparência implica que a natureza distribuída dos serviços utilizados esteja oculta; o usuário tem a impressão de que tudo está em âmbito local. Vale lembrar, entretanto, a citação de Leslie Lamport [LAM78]: "Um sistema distribuído é aquele no qual o defeito de um computador, do qual você desconhecia a existência, pode tornar o seu computador inutilizável".

Desta citação, conclui-se que o projeto de aplicações em sistemas distribuídos não é uma tarefa simples; muitos fatores contribuem para dificultar este projeto [JAL94]:

a) **Defeitos parciais**: uma vez que um sistema distribuído consiste de múltiplos componentes, alguns destes podem falhar, mas é esperado que as aplicações executando neste sistema distribuído sobrevivam a tais defeitos parciais; este requisito de tolerância a falhas aumenta bastante o número de estados possíveis do sistema.
b) **Execução concorrente**: em um sistema distribuído, há múltiplas *threads* de controle que são fracamente conectadas; a comunicação entre elas pode ser assíncrona e não-confiável, o que pode ocasionar indeterminismo.
c) **Ausência de controle central**: tipicamente, um sistema distribuído possui um controle decentralizado; pode ser necessário à aplicação distribuída implementar um mecanismo de sincronização.
d) **Ausência de relógio global**: isto dificulta bastante o projeto e a implementação de sistemas distribuídos; é difícil observar um estado do sistema e relações temporais entre eventos no sistema, pois cada nodo possui seu relógio local que, via de regra, registra um tempo diferente dos relógios dos outros nodos.
e) **Dependência do desempenho da rede**: o desempenho de um sistema ou aplicação distribuída depende muito do desempenho da rede que conecta os nodos processadores; este desempenho pode ser difícil de prever.

A necessidade do desenvolvimento de aplicações cada vez mais complexas, fornecendo os mais diversos serviços e voltadas aos mais variados campos, juntamente com o avanço tecnológico das redes de computadores, motivam a implementação de aplicações distribuídas. Mas, em decorrência dos fatores acima expostos e das diversas

funcionalidades que podem ter as aplicações distribuídas, conclui-se que a sua implementação exige bastante experiência dos programadores.

Um fator que certamente poupa esforços de programadores é a existência (ou a disponibilidade) de serviços fornecendo alguma abstração de programação. Convém ressaltar que serviços e protocolos são conceitos distintos, embora possam ser confundidos com certa freqüência.

1.2 Serviços e Protocolos

Um **serviço** é um conjunto de primitivas (operações) que uma camada oferece à camada acima dela. O serviço define quais operações a camada está preparada para realizar em nome de seus usuários, mas não diz nada sobre como essas operações são implementadas. Um serviço se refere a uma interface entre duas camadas, com a camada inferior sendo a provedora do serviço e a camada superior sendo a usuária do serviço [TAN94].

Cada serviço pode ser caracterizado por aspectos de qualidade. Por exemplo, um serviço que fornece comunicação confiável pode ser implementado fazendo com que o receptor confirme a recepção de cada mensagem; neste caso, o transmissor da mensagem obtém a informação necessária para certificar que a mensagem chegou ao destinatário. O processo de confirmação degrada o desempenho e introduz retardos que freqüentemente valem a pena, mas às vezes são indesejáveis.

Um **protocolo**, em contraste, é um conjunto de regras que governa o formato e significado de quadros, pacotes ou mensagens que são trocados entre entidades[2] parceiras dentro de uma determinada camada. As entidades usam protocolos para implementar suas definições de serviços. Elas têm a liberdade de mudar o protocolo como bem desejarem, desde que não mudem o serviço visível aos seus usuários. Dessa forma, serviços e protocolos são completamente desvinculados [TAN94].

Em relação a como os protocolos implementam os serviços, a experiência de utilização do paradigma orientado a objetos tem mostrado vantagens; tal fato é notado no decorrer das diversas etapas de projeto, bem como durante a manutenção e a reutilização do código.

1.3 Orientação a Objetos

A área de orientação a objetos é extensa e crescente. O modelo de objetos representa, em *software*, objetos que encontramos no mundo real. Esses objetos podem ser de vários tipos, representando entidades físicas (aviões, robôs, etc.) ou abstratas (listas, pilhas, filas, etc.). O sucesso da orientação a objetos se deve, provavelmente, à habilidade em modelar o domínio da aplicação ao invés da arquitetura da máquina em questão (**enfoque imperativo**) ou mapear conceitos matemáticos (**enfoque funcional**). O conceito de objeto pode ser visto como algum tipo de abstração "camaleônica" ou

[2] Os elementos ativos em cada camada, podendo ser *software* (processo) ou *hardware* (*chip*).

genérica: pode fornecer alguma abstração que se adapta melhor às semânticas da aplicação.

A característica mais importante (e diferente) da abordagem orientada a objetos para o desenvolvimento de *software* diz respeito à **unificação** (através do conceito de objetos) de dois elementos que, tradicionalmente, estão separados nos paradigmas de programação tradicionais: **dados** e **funções**. As linguagens imperativas levam a soluções baseadas em duas abstrações diferentes (algoritmos + estruturas de dados), ao passo que os objetos encapsulam ambas para formar entidades que são relevantes ao domínio da aplicação. Linguagens funcionais forçam a programação em termos de funções puras, ao passo que os objetos podem representar tais funções puras (se este for o intuito). Em outras palavras, objetos capacitam os programadores a pensar em termos do **seu** domínio do problema. **Elementos-chave** para esta tarefa são a experiência no projeto de *software* e no conhecimento profundo do domínio particular para o qual as abstrações são construídas [BUZ98].

Um **objeto** é uma entidade que possui uma identidade, um estado e uma interface/métodos para acessar este estado. Todo objeto é **instância** de uma classe geral. Uma **classe**, por sua vez, é a descrição de um molde que especifica as propriedades e o comportamento para um conjunto de objetos similares. Toda classe tem um nome e um corpo que define o conjunto de atributos e operações que suas instâncias possuem. O mundo externo interage com um objeto invocando-lhe métodos ou passando-lhe mensagens [MEY88].

As vantagens decorrentes da utilização de objetos na construção de aplicações são muitas, mas podem ser citadas as seguintes:

- **Simplicidade**: os objetos escondem a complexidade do código. Pode-se criar uma complexa aplicação gráfica usando botões, janelas, barras de rolagem, etc., sem conhecer a complexidade do código utilizado para criá-los.
- **Reutilização de código**: um objeto, depois de criado, pode ser reutilizado por outras aplicações, ter suas funções estendidas e ser usado, em combinação com outros, como bloco básico para a construção de sistemas mais complexos.
- **Inclusão Dinâmica**: objetos podem ser inseridos dinamicamente no programa, durante a execução. Isso permite que vários programas compartilhem os mesmos objetos e classes, reduzindo o seu tamanho final.

Os autores divergem quanto às características que fazem uma linguagem ser orientada a objetos, mas a maioria concorda que o paradigma se baseia em quatro princípios básicos: **abstração**, **encapsulamento**, **herança** e **polimorfismo** [BUZ98].
Abstração é o processo de extrair as características essenciais de um objeto real; é necessária para se ter um modelo fiel da realidade sobre o qual se possa operar. O conjunto de características resultante da abstração forma um tipo de dados abstrato com informações sobre seu estado e comportamento. A abstração nem sempre produz os mesmos resultados, depende do contexto onde é utilizada; a abstração de um objeto por uma pessoa pode ser diferente na visão de outra.

Na prática, as abstrações são fortemente acopladas com algum modelo de execução, que descreve o comportamento destas em tempo de execução [GAR98]. O poder de uma abstração de programação pode ser expresso através das seguintes questões:

- quando utilizada para a construção de *software* no seu domínio de aplicação, esta abstração resulta em implementações que se adequam às expectativas?
- quando utilizada em contextos para os quais não foi originalmente planejada, ainda será adequada ou, pelo menos, poderá ser facilmente adaptada?

Se a primeira questão é facilmente respondida através de experimentação direta, a segunda sempre permanece um tanto em aberto. Não se pode garantir que as abstrações de programação existentes serão adequadas em novos contextos ou que serão facilmente adaptáveis a estes. De alguma forma, a definição de abstrações poderosas para alcançar reutilização de *software* tentam arcar com o previsível e o imprevisível. A orientação a objetos é uma tentativa de resposta à segunda questão: os objetos são suficientemente gerais para modelar uma grande diversidade de domínios de aplicação.

Encapsulamento é o processo de combinar tipos de dados, dados e funções relacionadas em um único bloco de organização e só permitir o acesso a eles através de métodos determinados; é definido como sendo uma técnica para minimizar as interdependências entre módulos programados independentemente através de interfaces externas restritas. Uma das principais vantagens do encapsulamento é esconder a complexidade do código; outra, é proteger os dados. Permitindo o acesso a eles apenas através de métodos, evita que seus dados sejam corrompidos por aplicações externas, ou seja, o encapsulamento funciona tanto para proteger os dados, como para simplificar o uso de um objeto. Por exemplo, aplicativos conhecidos e que executam no sistema operacional *Windows* têm grande semelhança, porque usam os mesmos objetos.

Observa-se que cada objeto possui um número limitado de funções (**métodos**) que podem ser invocados sobre ele. Se, em alguma parte do programa, for decidido utilizar dois botões (por exemplo, **OK** e **Cancela**), não é preciso entender como o código é escrito, nem saber quais as variáveis internas, basta saber como construí-los, mudar o texto que contêm e depois incluí-los no programa. Há uma grande vantagem resultante disso: não há como corromper a estrutura interna dos botões.

Herança é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento. Uma classe derivada herda a representação de dados e operações de sua classe base, mas pode, seletivamente, adicionar novas operações, estender a representação de dados ou **redefinir** a implementação de operações já existentes. Uma classe base proporciona a funcionalidade que é comum a todas as suas classes derivadas, enquanto que uma classe derivada proporciona a funcionalidade adicional que especializa o seu comportamento.

A herança permite a formação de uma hierarquia de classes, onde cada nível é uma especialização do nível anterior, que é mais genérico. É comum desenvolver estruturas genéricas para depois ir acrescentando detalhes, pois isto simplifica o código e permite uma organização maior de um projeto; também favorece a reutilização de código. Sendo necessário, por exemplo, implementar um botão cor-de-rosa e já dispondo de uma classe que define as características de um botão cinza, pode-se fazer com que a nova implementação seja uma **subclasse** de `BotãoCinza`, e depois estender suas características. Desta forma, o programador concentra-se apenas no necessário (a cor do botão) e evita ter que definir tudo a partir do zero.

Polimorfismo é a propriedade de se utilizar apenas um nome para fazer coisas diferentes. No contexto de orientação a objetos, significa que diferentes tipos de objetos podem responder a uma mesma mensagem de formas diferentes, dependendo dos argumentos que foram passados e do objeto que a receber. Por exemplo: o envio de uma instrução **`desenhe`** para uma subclasse de **`polígono`**, poderá desenhar um triângulo, retângulo, pentágono, etc. dependendo da classe que receber a instrução. Isto é muito útil para escrever programas versáteis, que possam lidar com vários tipos diferentes de objetos.

Pesquisas têm demonstrado que a orientação a objetos apresenta-se como um modelo promissor para o desenvolvimento de *software* mais confiável e modular, devido a características inerentes ao próprio modelo de objetos. A utilização de técnicas orientadas a objetos facilita o controle da complexidade do sistema, porque promove uma melhor estruturação de seus componentes, e também permite que componentes já verificados/validados sejam reutilizados [KAR97] [BUZ98] [GAR98] [BAN2001].

A maioria das linguagens orientadas a objetos (como por exemplo: Smalltalk-80 [GOL83], CLOS [BOB88], Eiffel [MEY88], C++ [STR92] e Java [GOS96]) baseia-se no desenvolvimento de aplicações a partir de classes e objetos, não fornecendo apoio direto para a construção de "blocos de *software*" de granulosidade maior. Entre as linguagens citadas, com exceção de Eiffel, que dá apoio à noção de *clusters*, e de Java, que dá apoio à noção de *packages* (pacotes), nenhuma outra dá suporte direto à implementação de módulos[3].

Para o desenvolvimento de aplicações confiáveis, é importante que a linguagem de programação forneça suporte para verificação estática de tipos[4]. Levando-se este aspecto em consideração, apenas C++, Eiffel e Java são adequadas; C++ e Java apresentam algumas vantagens em relação a Eiffel: foram longamente difundidas e contêm mecanismos de tratamento de exceções[5] bem projetados. Java apresenta certas vantagens em relação a C++: apoio para *packages*, eliminação de *friends*[6], hierarquia de tipos separada da hierarquia de classes e coleta automática de lixo [BUZ98].

Foram observadas vantagens decorrentes da utilização do paradigma da orientação a objetos e também foi observado que, entre as linguagens orientadas a objetos, Java tem lugar de destaque pois, além dos aspectos já mencionados, é [NAU96]:

- **simples e poderosa**: os métodos para realizar determinada tarefa são claros e em número reduzido;
- **segura**: os programas Java não podem chamar funções globais e ter acesso a recursos arbitrários de sistema, há um certo nível de controle que pode ser exercido pelos *runtimes* de Java e que não é abordado pelos outros sistemas;
- **robusta**: libera o programador da preocupação com muitas das maiores causas de erros encontrados em outras linguagens. Java verifica o código enquanto é escrito e verifica-o novamente antes de executá-lo;

---

[3] Agrupamento de classes em conjunto com restrições de acesso.
[4] Linguagens estaticamente tipadas são aquelas onde o tipo de uma variável que aponta para um objeto, bem como a validade das operações sobre os objetos, é definido em tempo de compilação.
[5] Tratamento das respostas anormais (ou excepcionais) de um componente.
[6] Tipo de visibilidade onde as classes declaradas como *friends* de outras possuem o mesmo direito das classes originais (ou seja, há quebra de encapsulamento).

- **interativa**: foi criada para atender a requisitos do mundo real, entre eles a criação de programas interativos e em rede;
- **neutra em relação à arquitetura**: fornece longevidade e portabilidade de código entre plataformas diferentes;
- **interpretada e de alto desempenho**: embora Java seja interpretada, seu *bytecode* foi cuidadosamente projetado para que seja fácil traduzi-la diretamente para o código da máquina nativa e tenha um bom desempenho; e
- **de fácil aprendizagem**: os recursos da linguagem se assemelham à maneira natural de fazer as coisas e incentivam a uma programação correta.

Estão surgindo pesquisas no sentido de melhorar o desempenho de Java, já que este aspecto tem pesado negativamente sobre a linguagem. Estas pesquisas apontam para uma mudança de paradigma na área de linguagens de programação, particularmente com relação aos compiladores. Esforços estão sendo colocados no sentido de se conseguir uma compilação em tempo de execução. Assim, Java poderá obter um desempenho melhor do que linguagens como C++, pois o compilador Java passa a ter acesso a informações não disponíveis a um compilador C++. É claro que esta modificação também poderia ser feita sobre compiladores C++, mas Java, indubitavelmente, tem recebido maior atenção por parte dos pesquisadores e do mercado comercial [REI2000].

As observações descritas no decorrer desta seção motivaram a utilização da orientação a objetos e, em particular, da linguagem Java nesta dissertação.

1.4 Estrutura desta Dissertação

Neste capítulo introdutório, foram descritos aspectos relativos à evolução dos sistemas computacionais até a composição de sistemas em rede e de sistemas distribuídos. Já que em sistemas distribuídos a troca de mensagens é peça fundamental, foram elucidados os conceitos relativos a serviços e a protocolos. Por fim, comentou-se sobre as características do paradigma da orientação a objetos.

Na seqüência, o capítulo 2 enfoca algumas infra-estruturas relevantes na construção de aplicações distribuídas. Estas infra-estruturas podem estar constituídas na forma de requisitos (como ordenação), como protocolos de comunicação básicos (TCP, UDP, *Multicast* IP) ou compondo sistemas completos na forma de ferramentas de programação (sistemas de comunicação em grupo) ou na forma de *frameworks*.

O capítulo 3 tem o intuito de elucidar aspectos relacionados ao projeto de serviços de comunicação confiável. Para tanto, são abordadas questões como o tipo da rede de comunicação, a topologia da rede de comunicação, o modelo de falhas de nodos e de *links*, o sincronismo do sistema, o relacionamento dos eventos, a detecção de defeitos e o enfoque de confiabilidade.

O capítulo 4 aborda os algoritmos dos protocolos de comunicação que são propostos na dissertação: *unicast* não-confiável, *unicast* confiável, *multicast* não-confiável, *multicast* confiável.

No capítulo 5, é feita a descrição das classes propostas. Inicialmente, comenta-se um pouco sobre a linguagem UML e é ilustrado um diagrama com as classes propostas. Na seqüência, cada classe e os códigos correspondentes são ilustrados através de figuras.

O capítulo 6 explica como utilizar as classes propostas para compor os serviços de comunicação a que se propõe a dissertação. As classes são agrupadas na forma de um pacote, conforme as características da linguagem orientada a objetos adotada (Java). Observa-se, além disso, que este pacote forma um *framework*, ou seja, com as classes propostas podem ser criadas aplicações ou outros serviços.

Por fim, o capítulo 7 apresenta as conclusões desta dissertação, bem como direcionamentos para trabalhos futuros.

2 Infra-estruturas para o Desenvolvimento de Aplicações Distribuídas

Por aplicações distribuídas subentendem-se as aplicações que são executadas em dois ou mais nodos computacionais interligados em rede e que não compartilham memória. Desta forma, uma aplicação distribuída consiste de diversas ações realizadas de forma distribuída. Estas aplicações apresentam diversidade expressiva de exigências; algumas podem tolerar a perda de mensagens ou retardos de tempo na entrega das mensagens, enquanto que outras não. Pode-se afirmar que, devido às diferentes características exigidas pelas aplicações, não há uma infra-estrutura de comunicação capaz de atender a todas as classes de aplicações. Infra-estruturas genéricas não atendem aos requisitos de certas aplicações, por isso há muitos protocolos, fornecendo infra-estruturas de comunicação, que buscam eficiência em aplicações específicas, através de restrições adequadas em suas propriedades funcionais [AMA99].

Uma infra-estrutura de comunicação voltada ao desenvolvimento de aplicações distribuídas pode ser construída tirando proveito dos próprios recursos que o ambiente lhe oferece. Neste sentido podem ser utilizados protocolos de mais baixo nível (camada de transporte e de rede), tais como o *Transport Control Protocol* (TCP), o *User Datagram Protocol* (UDP) ou o *Multicast* IP.

Um aspecto importante relacionado ao desenvolvimento de aplicações distribuídas diz respeito à ordenação na comunicação, uma vez que, dependendo da aplicação, a utilização de um ou outro tipo de ordenação muitas vezes é necessário. Nem sempre se consegue atingir os requisitos de ordenação das aplicações através do uso exclusivo de protocolos de mais baixo nível; então são criados outros protocolos e serviços. Estes, por sua vez, podem ser utilizados na composição de sistemas de comunicação em grupo ou *frameworks* para comunicação, com a finalidade de atingir, além dos requisitos de ordenação, outras exigências de determinados tipos de aplicações.

A seguir, são explicados os tipos de ordenação na comunicação e, posteriormente, comenta-se a respeito dos protocolos de mais baixo nível, bem como relatam-se aspectos da comunicação em grupo e dos *frameworks*. A utilização de um ou outro destes protocolos e sistemas para o desenvolvimento de aplicações distribuídas traz certas vantagens e desvantagens.

2.1 Ordenação

Como já comentado anteriormente, as aplicações distribuídas apresentam bastante diversidade de requisitos funcionais. Com respeito à ordenação na comunicação, algumas podem não exigir qualquer ordenação entre as mensagens enviadas, ao passo que outras a exigem.

Para se compreender a necessidade de ordenação, considere-se a situação da figura 2.1, descrita em [TAN92]. Podem ser observados cinco nodos, cada um executando um processo, sendo que os processos P1 e P5 desejam, simultaneamente, difundir mensagens a alguns dos outros processos. Para tanto, estes processos devem enviar três mensagens (*unicast*[7]) separadas, pois assume-se que não se dispõe de *hardware* com suporte a *multicast*[8] ou *broadcast*[9].

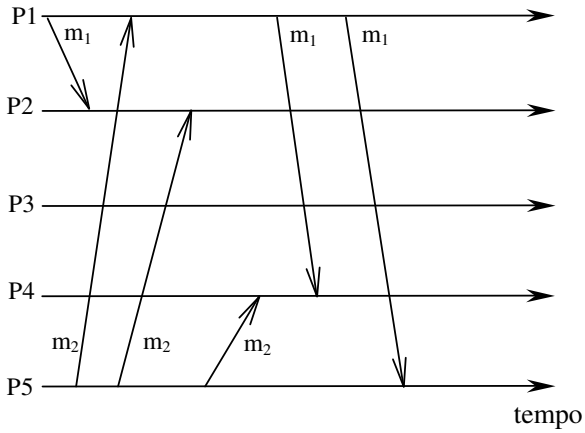

FIGURA 2.1 - Comunicação sem ordenação das mensagens

O processo P1 envia uma mensagem para os processos P2, P4 e P5, enquanto que o processo P5 envia para P1, P2 e P4. O problema aparece quando dois processos estão competindo para ganharem acesso a uma rede local, e a ordem em que as mensagens são enviadas é não-determinística. Na figura 2.1, observa-se que, imprevisivelmente, o processo P1 ganha a disputa e consegue enviar para processo P2. A seguir, entretanto, o processo P5 consegue ganhar três disputas sucessivas e conseguiu enviar mensagens para os processos P1, P2 e P4. Por fim, o processo P1 consegue novamente acessar a rede e envia as mensagens para os processos P4 e P5.

Considerando a situação do ponto de vista dos processos P2 e P4, o processo P2 primeiro recebe a mensagem enviada pelo processo P1, então, imediatamente após, recebe uma mensagem do processo P5; o processo P4 não recebe nada inicialmente, mas, em um segundo momento, recebe mensagens dos processos P5 e P1, nesta ordem. Desta forma, as duas mensagens chegam em uma ordem diferente, considerando o ponto de vista de P2. Se os processos P5 e P1 estivessem tentando atualizar o mesmo registro de uma base de dados, os processos P2 e P4 terminariam com valores diferentes. Faz-se necessário, por conseguinte, que o sistema possua uma semântica muito bem definida em relação à ordem em que as mensagens serão expedidas.

A ordenação das mensagens pode ser classificada da seguinte maneira [BAR94]:

- **Ordenação FIFO** (*First In First Out*): as mensagens enviadas por um mesmo processo (objeto) são entregues na mesma ordem de emissão pelos receptores,

[7] Comunicação do tipo ponto-a-ponto, ou seja, um emissor e um destinatário.
[8] Tipo de comunicação multiponto utilizado para transmitir uma mensagem para vários destinatários, que podem ser escolhidos através de uma identificação de grupo.
[9] Tipo de comunicação multiponto utilizado para transmitir uma mensagem para vários destinatários, mas estes não podem ser escolhidos (todos os nodos receberão a mensagem, inclusive o emissor).

enquanto mensagens enviadas por processos (objetos) distintos podem ser entregues por diferentes receptores em ordens aleatórias. A ordem FIFO produz uma ordenação parcial, cujos subconjuntos são as ordens de envio individuais dos participantes.

- **Ordenação Causal**: estende a ordenação do tipo FIFO para **causalidade potencial** no envio de mensagens por diferentes processos (objetos). Desta forma, em situações onde dois emissores E_1 e E_2 enviam, respectivamente, as mensagens m_1 e m_2, se a emissão de m_1 provocar potencialmente a emissão da mensagem m_2, a ordenação causal deve garantir que todos os receptores entregarão a mensagem m_1 antes da mensagem m_2; caso contrário, tais mensagens não serão entregues de forma idêntica por receptores distintos.
- **Ordenação Total**: onde todas as mensagens enviadas são entregues em uma mesma ordem por todos os receptores. Esta ordem de entrega não reflete, necessariamente, a ordem cronológica de emissão das mensagens.

2.2 TCP

O TCP é considerado até hoje o padrão de rede para comunicação *unicast* confiável, onde "confiável" significa comunicação por melhor-esforço (*best-effort*), isto é, pacotes perdidos são retransmitidos, mas não indefinidamente. Diversos protocolos e serviços são baseados no TCP/IP, tais como: TELNET e FTP [COM88], RPC [BIR84], NFS [SUN89] e HTTP [MAR95].

TCP fornece entrega orientada a fluxo (*stream*) com ordenação FIFO. A entrega ser orientada a fluxo significa que os limites das mensagens não são respeitados e uma única mensagem pode ser quebrada e entregue em vários pacotes no destinatário. Se os limites das mensagens precisarem ser respeitados, este problema pode ser resolvido através de um mecanismo de mais alto nível que garanta estes limites.

O TCP é baseado no IP, que não fornece confiabilidade, ou seja, as mensagens são transmitidas apenas uma vez e podem ser perdidas ou corrompidas. Para fornecer confiabilidade, o TCP/IP retransmite as mensagens perdidas, contudo utiliza um mecanismo bem simples para implementar a retransmissão. Os reconhecimentos, gerados pelo destinatário, contêm a informação referente à seqüência de pacotes recebidos: se está completa ou não. O emissor retransmite pacotes, iniciando pelo primeiro não-reconhecido, não importando se um ou mais dentre os pacotes intermediários já foram recebidos pelo destinatário. Um protocolo de janela deslizante, responsável pelo controle de fluxo e de congestionamento através da variação do tamanho da janela, permite a existência de mais de um pacote em transmissão em um dado momento [TAN94].

Considerando defeitos de *link* ou congestionamento de rede, o TCP abandona os pacotes não-reconhecidos após um determinado tempo (*timeout*) e aborta a conexão. Com respeito às conexões, o TCP/IP utiliza a interface de *socket* do sistema operacional em questão (*Unix, Linux, Windows*, etc.). Dependendo deste sistema operacional, há um limite com relação à quantidade de *sockets* abertos e seus descritores associados; atingido este limite, o sistema não permitirá a abertura de novas conexões TCP. Uma solução pode ser a abertura e o fechamento das conexões dinamicamente, entretanto abrir e fechar *sockets* geralmente é uma operação que degrada o desempenho do sistema, principalmente quando há muitas conexões ativas.

2.3 UDP

O UDP é um protocolo mais rápido do que o TCP, pelo fato de não verificar o reconhecimento das mensagens enviadas; por este mesmo motivo, não é confiável como o TCP. O protocolo é não-orientado a conexão e não provê muitas funções: não controla o fluxo, podendo os datagramas chegarem fora de seqüência ou até mesmo não chegarem ao destinatário [TAN94].

Como o UDP não oferece suporte para a recuperação de erros, perda ou duplicação de mensagens, entrega de mensagens fora de ordem ou perda de conectividade, estas questões devem ser ignoradas ou então tratadas pelas camadas superiores. Assim, a camada que utiliza o protocolo UDP é quem deve fornecer alguma forma que aumente a confiabilidade, pois o UDP é, em síntese, simplesmente um emissor e receptor de datagramas.

O UDP é, portanto, um protocolo de comunicação de baixo nível sem muitas garantias, mas é eficiente e sua interface simples é ideal para implementar protocolos com mais funcionalidades sobre ele. Além disso, uma importante característica relacionada ao UDP diz respeito às primitivas de envio e recebimento; a primitiva de envio (*send*) é não-bloqueante e a primitiva de recebimento (*receive*) pode também ser feita não-bloqueante. Esta característica de não-bloqueio é importante, pois permite a execução de outras tarefas durante a transmissão de uma mensagem.

2.4 *Multicast* IP

A comunicação *multicast* é constantemente empregada pelas aplicações distribuídas e pode ser implementada de diferentes maneiras:

- através do envio de *n* mensagens em *unicast*, sendo *n* o número de destinatários do *multicast*;
- utilizando *broadcast* e incluindo na mensagem a informação de quem são os destinatários;
- empregando *multicast IP* [DEE89], se a rede fornecer suporte.

O enfoque que utiliza *unicast* é uma emulação e extremamente ineficiente. O enfoque que utiliza *broadcast* é um pouco mais eficiente, mas a implementação do *multicast* é uma emulação da mesma forma, pois em um *multicast genuíno* apenas o emissor da mensagem e os destinatários devem fazer parte do algoritmo [GUE96]. O enfoque que utiliza *multicast* IP é uma forma eficiente de transmissão de dados para vários receptores, que é fornecida em algumas redes locais, como *Ethernet* e *Fiber Distributed Data Interface* (FDDI) [MON94].

O *multicast* IP foi projetado para transmitir dados de um processo para um conjunto de processos, que não necessariamente estão na mesma rede local. A particularidade, neste caso, relaciona-se ao modo como são endereçados os processos destinatários, já que os endereços destes não são conhecidos no momento de envio de mensagens. Para que seja possível o recebimento de mensagens em comum, os processos destinatários compartilham um endereço IP particular pertencente à classe D de endereços IP, cujos endereços variam de 224.0.0.0 a 239.255.255.255.

Os nós do sistema com funções de emissor e receptor de mensagens num grupo de *multicast* IP possuem uma forte relação entre si, definida pelo endereço de grupo que é conhecido por todos. Um emissor pode transmitir mensagens para um grupo de *multicast* IP da mesma forma que envia mensagens em *unicast*; a diferença está no uso de um dos endereços *multicast* para identificar o grupo para o qual deseja enviar a mensagem. Para receber mensagens destinadas a determinado grupo *multicast*, um nó receptor deve pertencer a este grupo. Por exemplo, supondo-se quatro processos: *A*, *B*, *C* e *D*. Se A deseja enviar uma mensagem via *multicast* IP aos processos *B*, *C* e *D*, deve enviá-la para um mesmo endereço de *multicast* ao qual esses processos estejam associados; este endereço de *multicast* identificará os processos *B*, *C* e *D* como sendo parte de um conjunto. Por exemplo: se *B*, *C* e *D* estão associados ao endereço de *multicast* 230.5.7.1, *A* deve enviar uma mensagem para este endereço IP se deseja que *B*, *C* e *D* recebam a mensagem.

O *multicast* IP é implementado no topo do protocolo IP e utiliza um *daemon* em cada rede local participante. Em decorrência disto, a interligação entre diversas redes locais através de *multicast* IP é dependente do funcionamento normal deste *daemon*. Em outras palavras, a falha deste *daemon* implica na impossibilidade de envio e recebimento de mensagens fora do escopo da rede local em que o *daemon* se localiza. Além disso, o *multicast* IP utiliza o UDP para o envio das mensagens e, portanto, apresenta as mesmas desvantagens relacionadas à confiabilidade que o UDP apresenta; sua grande vantagem é ter a mesma sobrecarga para o envio de uma mensagem que o apresentado pelo *unicast* [AMA99].

2.5 Comunicação em Grupo

O objetivo dos sistemas de comunicação em grupo é tornar algum serviço distribuído altamente disponível e/ou capaz de tolerar falhas parciais. Intuitivamente, isto é geralmente alcançado distribuindo cada servidor participante do serviço em nodos distintos da rede. O serviço se mantém ativo enquanto existirem cópias dos servidores, ou seja, as falhas em alguns servidores não prejudicam o funcionamento do serviço como um todo; tal serviço é dito ser **tolerante a falhas através de replicação** [BIR96].

A abstração de grupos é muito útil no tratamento de replicação: um servidor replicado pode ser implementado como um grupo de réplicas. Grupos constituem-se, então, em um modo conveniente de endereçar servidores replicados sem ter que designar cada réplica explicitamente. Este modelo vem sendo bastante aceito na comunidade de sistemas distribuídos como um modo adequado à obtenção de tolerância a falhas.

Uma vez que o modelo de grupos pode ser visto como uma extensão do modelo cliente-servidor, as plataformas resultantes são **orientadas a processos** ao invés de **orientadas a dados**, como em banco de dados; executado sobre sistemas operacionais como o *Unix*, por exemplo, um servidor replicado é geralmente gerenciado como um grupo de processos. Em decorrência disso, as pesquisas em abstrações de comunicação em grupo vêm tomando a direção da utilização do modelo orientado a objetos, ainda que de forma lenta. Como conseqüência, observa-se que diversas plataformas que fornecem grupos de objetos são construídas sobre ferramentas de grupos de processos e, desse modo, fornecem meramente uma interface orientada a objetos [GAR98].

Sistemas de comunicação em grupo tipicamente fornecem *multicast* confiável e serviços de *membership* (gerência de membros do grupo). A principal atividade do serviço de *membership* é manter uma lista dos processos correntemente conectados e ativos, entregando esta informação para a aplicação sempre que houver mudança na configuração. Um serviço de *membership* informa quem são os membros do grupo em determinado instante, o que é denominado de **visão**. Os serviços de *multicast* confiável entregam as mensagens para os membros da visão corrente [BIR96].

As políticas de implementação dos serviços em um grupo, bem como suas comunicações interprocessos internas, podem ser transparentes para o cliente. Assim, com o intuito de mostrar a importância dos sistemas de comunicação em grupo no contexto de sistemas operacionais distribuídos, principalmente no que se refere à troca de mensagens entre os processos, Tannenbaum [TAN92] classifica-os de acordo com sua estrutura interna.

São chamados de **grupos fechados** aqueles onde somente os membros podem enviar mensagens para o grupo, quem não pertence ao grupo pode enviar mensagens para membros individuais. Por outro lado, **grupos abertos** permitem que qualquer processo/objeto do sistema seja capaz de enviar mensagens para qualquer grupo. Em **grupos não-hierárquicos**, há igualdade entre os processos, nenhum é superior, e todas as decisões são tomadas coletivamente. Já quando os processos podem ser organizados hierarquicamente, são chamados de **grupos hierárquicos** onde, por exemplo, pode haver um membro (**coordenador**) com missão de gerenciar as tarefas dos demais. Grupos onde não é permitida a inclusão ou exclusão de membros são chamados de **grupos estáticos**. Por outro lado, em **grupos dinâmicos**, um processo pode, a qualquer momento, juntar-se ao grupo ou sair dele. Adicionalmente, há ambientes em que um processo pode ser membro de vários grupos simultaneamente, o que se chama de **sobreposição de grupos**.

A comunicação em grupo tem se mostrado uma abstração eficiente para a construção de aplicações distribuídas confiáveis. A experiência com sistemas de comunicação em grupo e aplicações distribuídas confiáveis tem mostrado que não há uma semântica de sistema "correta" para todo o tipo de aplicações [BIR96]: sistemas de comunicação em grupo diferentes são direcionados a tipos de aplicações diferenciados, onde há exigências próprias diversas quanto à **semântica** e à **qualidade de serviço**. Exemplos de sistemas de comunicação em grupo são: *Isis* [BIR91], *Consul* [MIS91], *Transis* [AMI92], *xAMp* [ROD92], *Relacs* [BAB94], *Horus* [REN94], *Newtop* [EZH95], *Phoenix* [MAL95], *Totem* [AMI95], dentre outros.

O sistema de comunicação em grupo *Horus* [REN94] trouxe um paradigma novo: **modularidade**. *Horus* e seu sucessor *Ensemble* [HAY96] são sistemas de comunicação em grupo flexíveis, compreendidos de camadas de protocolos independentes, que implementam níveis e semânticas de serviço diferenciados. Este enfoque permite ao construtor da aplicação ajustar o sistema de comunicação em grupo conforme necessitar, tratando as camadas de protocolos como **blocos de construção**.

2.6 *Frameworks*

Experiências significativas de reutilização de projeto envolvem a utilização de *frameworks*, pois objetos e classes são blocos de construção de *software* muito pequenos para alcançar níveis altos de reusabilidade [BUZ98]. Um ***framework*** é um projeto reutilizável expresso através de um conjunto de classes e pela forma como suas instâncias interagem; por definição, um *framework* é um projeto orientado a objetos [BIG87]. Ele não precisa ser implementado através de uma linguagem orientada a objetos, embora geralmente o seja. Ele é o esqueleto de uma aplicação que pode ser customizada pelo desenvolvedor. Uma definição de *framework*, bastante aceita pela comunidade acadêmica, foi proposta por Ralph E. Johnson: um *framework* é um conjunto de classes que abrangem um projeto abstrato, visando a solução de uma família de problemas correlacionados [JOH97].

Um *framework* define o comportamento de uma coleção de objetos, fornecendo um modo alternativo para a reutilização do projeto e do código do programa. Ele é um conjunto de blocos de construção que podem ser utilizados, estendidos, ou customizados pelos desenvolvedores, com a finalidade de propor soluções computacionais específicas.

Os *frameworks* orientados a objetos representam o topo da escala com respeito aos princípios fundamentais da programação orientada a objetos. Em programação orientada a objetos, uma nova classe herda características de uma classe mais geral; esta nova classe contém apenas o código que é diferente da **superclasse**. Os *frameworks* aplicam este princípio para um domínio de problema. Assim, um desenvolvedor acrescenta apenas o código que é diferente do *framework*.

Com *frameworks*, os desenvolvedores gastam menos tempo para construir suas aplicações, pois ultrapassam várias etapas em que necessitariam compreender muito bem todo o domínio de um problema, o que resulta em maior consistência, em melhor integração e em uma manutenção mais facilitada.

Ao contrário das abordagens tradicionais para a reutilização de *software*, que se limitam basicamente na construção de bibliotecas de classe, *frameworks* permitem reutilizar não apenas componentes isolados, mas toda a arquitetura de *software* projetada para um domínio específico. Aplicações baseadas em **bibliotecas de classes** possuem o controle (fluxo de eventos) localizado na própria aplicação. O programador da aplicação é encarregado de projetar/implementar o fluxo de controle da aplicação específica. Em contrapartida, aplicações baseadas em *frameworks* reutilizam o fluxo de eventos já embutido no próprio *framework*. Os programadores que utilizam um *framework* para construir a sua aplicação específica implementam apenas o código que será chamado pelo *framework* (*callbacks*). Desta forma, aplicações podem reutilizar o fluxo de eventos e a arquitetura de *software* que o *framework* fornece. Enquanto que os componentes de uma biblioteca de classes são utilizados individualmente, os componentes de um *framework* são reutilizados como um todo na resolução de uma instância específica de um certo conjunto de problemas relacionados [BUZ98].

Os *frameworks* também diferem dos **componentes**, que são instâncias auto-contidas de tipos de dados abstratos, que podem ser conectados juntos para formar aplicações completas. Os *frameworks* podem ser utilizados para o desenvolvimento de

componentes e vice-versa; geralmente, os *frameworks* são utilizados para simplificar o desenvolvimento de infra-estruturas de *softwares* e *middlewares*, ao passo que componentes são freqüentemente utilizados para simplificar o desenvolvimento de aplicações voltadas ao usuário final.

3 Projeto de Serviços de Comunicação Confiável

Em sistemas distribuídos, é normal tomar em consideração um nível lógico para as aplicações, em que todos os nodos são conectados por canais de comunicação (por suposição, o grafo que representa a rede é totalmente conectado). Na prática, isto significa que uma mensagem pode ser enviada de um determinado nodo para qualquer outro. Em outras palavras, se um nodo *p* deseja enviar mensagens para um nodo *q*, os nodos *p* e *q* não precisam estar diretamente conectados através de um canal de comunicação (*link*), mas é necessário que exista um caminho entre eles no grafo que representa a rede. Nestes sistemas, supõe-se com freqüência que uma mensagem enviada por um nodo chega intacta no receptor, bem como a existência de ordenação entre mensagens sucessivas [JAL94].

Associado a cada canal de comunicação que une dois processos quaisquer (por exemplo *p* e *q*) há duas primitivas de comunicação, chamadas *send* e *receive*. Se *p* invoca *send* com uma mensagem *m* como parâmetro para transmiti-la para *q*, diz-se que *p* envia m para *q*; ao retornar desta invocação, diz-se que *p* completou o envio de *m* para *q*. Por sua vez, quando o processo *q* retorna da execução do *receive* com a mensagem *m* como valor de retorno, diz-se que *q* recebeu *m* (de *p*). Também associado ao canal de comunicação entre *p* e *q*, há um *buffer* de saída em *p* e um *buffer* de entrada em *q*. Informalmente, quando *p* envia uma mensagem *m* para *q*, insere *m* em seu *buffer* de saída, o canal de comunicação transporta *m* até o *buffer* de entrada em *q*, e a mensagem *m* é então removida deste *buffer* e recebida por *q* [HAD94].

Linhas de comunicação reais, entretanto, às vezes perdem mensagens e introduzem erros. Desta forma, alguns mecanismos são necessários para garantir a suposição de que as mensagens cheguem intactas no destino ou, ainda, de que exista ordenação entre mensagens entregues sucessivamente. Estes mecanismos são fornecidos através de protocolos de comunicação, que são utilizados na implementação de serviços de comunicação confiável[10]. A disponibilidade de serviços de comunicação confiável traz ainda mais vantagens ao programador de aplicações distribuídas, no sentido de que este consegue concentrar-se preponderantemente na funcionalidade da aplicação em desenvolvimento. Uma camada de comunicação confiável situa-se entre a rede de comunicação e a aplicação distribuída (figura 3.1).

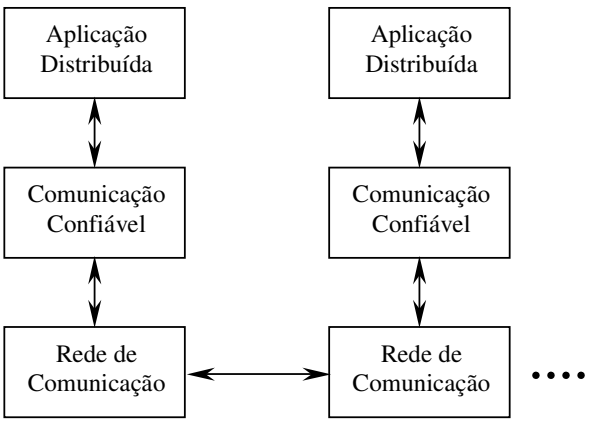

FIGURA 3.1 - Camada de comunicação confiável

[10] Importante salientar que protocolos e serviços são conceitos relacionados, porém distintos (seção 1.2).

A comunicação entre dois nodos, por exemplo *p* e *q*, é dita confiável se a mensagem *m* enviada por *p* é entregue pelo destinatário *q* em um tempo finito[11]. O ideal seria que esta propriedade fosse preservada mesmo que a rede se particione em dois ou mais grafos disjuntos. Isso, entretanto, pode exigir um tempo de espera muito longo, pois um nodo *p* que não consegue enviar uma mensagem para um nodo *q* localizado em outra partição da rede deve esperar que a rede se reconecte [JAL94]. Em decorrência disto, esta dissertação limita o enfoque para os casos onde não há particionamento da rede, ou seja, sempre haverá um caminho pelo qual uma mensagem enviada chegará até o destinatário.

A comunicação confiável faz parte dos requisitos arquiteturais de comunicação e, geralmente, é implementada baseada em protocolos de comunicação. Um protocolo de comunicação pode oferecer dois tipos diferentes de serviço de comunicação entre nodos [TAN94]: **orientado a conexão** e **sem-conexão**.

O serviço orientado a conexão é modelado como um sistema telefônico, onde alguém pega o telefone, disca o número, fala e depois desliga. De forma similar, para se utilizar um serviço de rede orientado a conexão, uma conexão é estabelecida, através desta os dados são então transferidos e, por fim, a conexão é encerrada. O aspecto essencial da conexão é que ela funciona como um tubo: o transmissor introduz objetos em extremo, e o receptor os retira do outro lado na mesma ordem.

Por outro lado, um serviço sem-conexão é modelado a partir do sistema de correios; cada mensagem (carta) leva consigo o endereço de destino completo e cada mensagem é roteada para o destino independentemente. Além disso, em serviços sem-conexão, é possível a perda de alguns dados, ou que alguns dados enviados em um evento posterior acabem chegando no destino antes dos dados enviados anteriormente (isto não ocorre em um sistema orientado a conexão). Um serviço sem-conexão não-confiável (significando sem confirmação) é freqüentemente chamado de **serviço de datagrama**, em analogia com o serviço de telegrama dos correios, que também não fornece confirmação de volta ao remetente.

Em outras situações, a conveniência de não ter que estabelecer uma conexão para transmitir uma mensagem curta é desejável, mas confiabilidade é essencial. Para estas aplicações, pode ser oferecido o **serviço de datagrama com confirmação**. Este serviço funciona como mandar uma carta registrada com aviso de recebimento; quando o aviso volta, o remetente tem certeza absoluta de que a carta foi entregue ao destinatário desejado.

Por fim, há o **serviço de pedido-resposta**. Neste serviço, o transmissor *p* envia um datagrama para *q* contendo um pedido e recebe de *q* uma mensagem contendo a resposta.

Um serviço de comunicação confiável desempenha um papel fundamental na obtenção de aplicações tolerantes a falhas, pois a comunicação confiável é considerada um dos blocos básicos de sistemas distribuídos tolerantes a falhas (figura 3.2) [JAL94].

[11] Tradução utilizada nesta dissertação para o termo em inglês *eventually*.

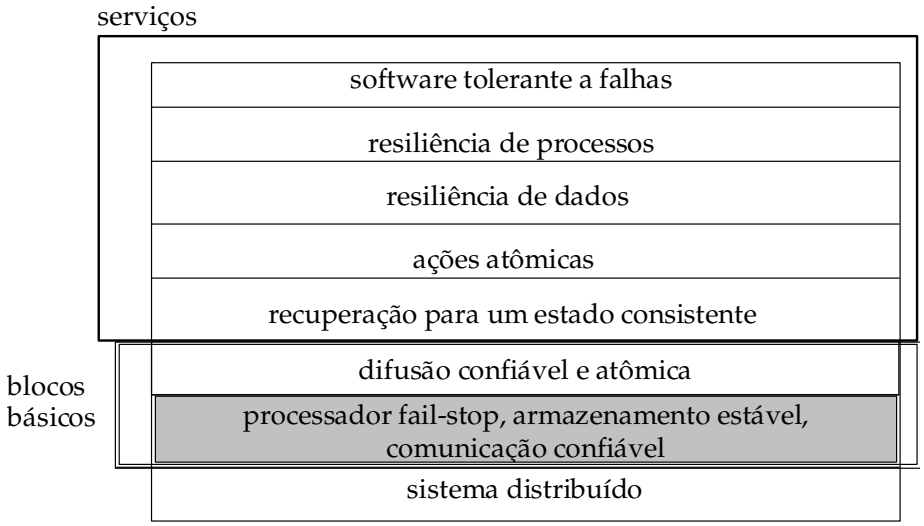

FIGURA 3.2 - Comunicação confiável em um sistema distribuído tolerante a falhas

Quando é questionado o projeto de um serviço de comunicação, a questão fundamental relaciona-se às aplicações para as quais o serviço é projetado. Dependendo da aplicação, as exigências serão bem diversas. Algumas aplicações podem operar com perda de mensagens ou retardos de tempo na entrega das mensagens, aceitando um serviço de comunicação não-confiável, enquanto que outras são mais exigentes, necessitando um serviço de comunicação confiável.

Pode-se afirmar que, devido às diferentes características exigidas pelas aplicações, não há um serviço de comunicação capaz de atender a todas as classes de aplicações; serviços de comunicação genéricos não atendem aos requisitos de certas aplicações, por isso há muitos serviços voltados para aplicações específicas (transferência de arquivos, tráfego de voz digitalizada, videoconferência, etc.).

Em comum, tem-se o fato de que se deve levar em conta algumas questões importantes durante a fase de projeto desses serviços, tais como: o **tipo da rede de comunicação**, a **topologia da rede de comunicação**, o **relacionamento dos eventos**, o **sincronismo do sistema**, o **modelo de falhas de nodos** e **de canais de comunicação**, a **detecção de defeitos** e o **método de obtenção de confiabilidade**. Essas questões são abordadas no restante do capítulo da seguinte maneira: o tipo da rede de comunicação é apresentado na seção 3.1; a topologia da rede de comunicação é mostrada na seção 3.2; o relacionamento dos eventos é descrito na seção 3.3; o sincronismo do sistema é discutido na seção 3.4; o modelo de falhas de nodos e de canais de comunicação aparece na seção 3.5; a detecção de defeitos é mostrada na seção 3.6; e o método de obtenção de confiabilidade é elucidado na seção 3.7.

3.1 Tipo da Rede de Comunicação

Uma rede de comunicação é uma coleção de dispositivos de computação interconectados que permitem a um grupo de pessoas compartilhar informações e recursos. As redes de comunicação têm-se tornado indispensáveis nos dias de hoje, interligando desde poucos computadores em uma sala de aula (compartilhando uma impressora e um servidor), a milhares de computadores espalhados pelo mundo todo, trocando informações. Uma rede é classificada de acordo com a distância entre os seus nodos (computador, servidor, periférico, etc.), apresentando as seguintes subdivisões: **redes confinadas**, **redes locais**, **redes metropolitanas** e **redes geograficamente distribuídas** [TAN94].

Redes Confinadas são pequenas, limitadas a áreas restritas (poucos metros); surgem quando, devido a situações particulares, não há condições de se interligar muitos computadores ou computadores muito distantes. **Redes Locais** de computadores (*Local Area Networks* - LAN) são sistemas cujas distâncias entre os módulos processadores se enquadram na faixa de alguns poucos metros a alguns poucos quilômetros; uma rede local é tipicamente encontrada em pequenos ambientes, como em um mesmo prédio ou em uma pequena empresa. **Redes Metropolitanas** *(Metropolitan Area Networks - MANs)* apresentam as mesmas características que as redes locais, porém as MANs cobrem distâncias maiores do que as LANs, bem como geralmente operam em velocidades maiores. **Redes Geograficamente Distribuídas** *(Wide Area Networks - WANs)* surgiram da necessidade de se compartilhar recursos especializados por uma comunidade maior de usuários dispersos geograficamente; devido ao alto custo dos meios de comunicação (circuitos para satélites e enlaces de microondas), a maioria destas redes é pública.

Serviços de comunicação voltados a atender aplicações que executam sobre uma rede local são bem mais simples de serem projetados do que serviços direcionados a aplicações sobre redes metropolitanas ou geograficamente distribuídas. É justamente no aspecto de como interligar as diversas sub-redes, mantendo a semântica de comunicação a que o serviço se propõe, que se encontra a maior complexidade de projeto destes serviços.

3.2 Topologia da Rede de Comunicação

Ao se abordar a questão da ligação dos computadores em redes, uma das primeiras preocupações que se deve ter é sobre a forma como os computadores são interligados, ou seja, como é a arquitetura e o *layout* da rede. Se for considerado um sistema distribuído (seção 1.1), a rede física é constituída de muitos nodos autônomos.

A arquitetura da rede nada mais é do que um conjunto de definições sobre a hierarquia e organização dos componentes nessa rede. Um bom projeto de arquitetura garante desempenho, manutenção de baixo custo, necessidades atendidas, segurança, entre outras propriedades, sendo fundamental para uma rede funcionar bem. O *layout*, por sua vez, descreve de que maneira os cabos e equipamentos estão dispostos.

Aspectos relacionados à arquitetura em conjunto com o *layout* da rede formam o conceito de **topologia de rede**, que é subdividida em duas categorias básicas [CAS97]:

- **topologia física**: descreve o *layout* atual do meio de transmissão da rede; e
- **topologia lógica**: descreve o caminho lógico pelo qual um sinal segue conforme passa pelo nodos da rede.

Em outras palavras, a topologia física descreve a forma com que a rede se parece. Já a topologia lógica define de que modo o dado passa entre os nodos.

Embora existam várias estratégias de topologia, suas variações sempre derivam de três topologias básicas, mais freqüentemente empregadas: **barramento**, **estrela** e **anel**. Redes pequenas com poucos dispositivos tendem a utilizar somente uma topologia, enquanto que redes maiores, envolvendo uma vasta área física, podem utilizar uma combinação de topologias.

Em uma topologia do tipo **barramento** (figura 3.3), cada nodo (computador, servidor, periférico, etc.) tem um endereço e liga-se diretamente ao meio de transmissão (cabo coaxial, cabo par-trançado ou cabo de fibra óptica), que é compartilhado. Os dados são enviados para este meio de transmissão e, dessa forma, circulam por todos os computadores. Esta topologia tem uma configuração multiponto, onde cada nodo conectado ao barramento tem acesso a (pode "ouvir") todas as informações transmitidas por qualquer um dos outros nodos através do meio comum (mensagens do tipo difusão). Este tipo de topologia é indicado para redes simples, já que um barramento único é limitado em termos de distância, gerência e quantidade de tráfego. Redes *Ethernet* constituem-se em exemplo de larga utilização deste tipo de topologia.

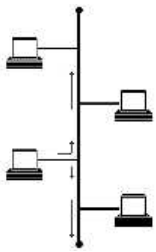

FIGURA 3.3 - Topologia do tipo barramento

Em uma topologia do tipo **estrela** (figura 3.4), cada nodo é interligado a um nodo central (mestre), que então permite a interligação dos computadores e através do qual todas as mensagens devem passar; este nodo central (freqüentemente um *hub*) geralmente não é passivo, funciona como centro de controle da rede, interligando os demais nodos escravos, amplificando e repetindo os sinais de dados. Com isso, as possibilidades da rede são ampliadas, em termos de distância, além de também se ter a possibilidade de gerência centralizada. As redes em estrela podem atuar por difusão ou não. Em redes por difusão, todas as informações são enviadas ao nodo central que tem como responsabilidade distribuí-las aos outros nodos da rede. A desvantagem da topologia está no caso de falha deste nodo central, onde toda a rede fica desconectada.

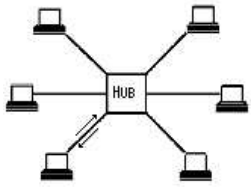

FIGURA 3.4 - Topologia do tipo estrela

Em uma topologia em anel (figura 3.5), as estações estão interligados entre si através de um canal fechado, ficando o último computador da rede também conectado ao primeiro. Redes em anel são capazes de transmitir e receber dados em qualquer direção. Em relação ao método de barramento, há a vantagem de que, dessa forma, os próprios computadores atuam como repetidores do sinal e a rede tem menos limitação em termos de distância.

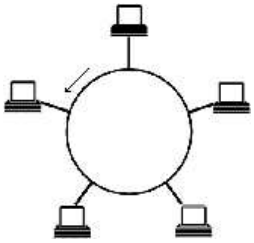

FIGURA 3.5 - Topologia do tipo anel

Dependendo da situação, uma ou outra topologia deve ser utilizada, com suas vantagens e desvantagens decorrentes, pois influenciam na questão o método de acesso utilizado e o tipo de cabos instalado, sendo possível utilizar mais de um método de acesso e mais de um tipo de cabo.

É importante ter-se clara a noção de que esses são padrões básicos de topologias e que, na prática, os diversos padrões são combinados, com a existência, por exemplo, de redes com a topologia em "barramento estrela", formada por *hubs* ligados em um barramento, com diversos computadores a eles ligados, em topologia estrela. Através da combinação das topologias mencionadas, pode-se compor redes das mais diversas. Por exemplo, pode-se utilizar um anel FDDI, de alta velocidade, interligando diversas redes barramento e redes estrela [CAS97].

Contudo, não é simples para uma empresa determinar a topologia de sua rede e se esta será baseada em par-a-par ou baseada em servidor. A complexidade na determinação é explicada porque não há fórmulas ou regras exatas e imutáveis; depende sempre das particularidades de cada ambiente e da análise de diferentes situações em face das vantagens e desvantagens de cada possibilidade a ser escolhida. Questões como as aplicações que serão executadas, o grau de segurança necessário, o número de estações, os requisitos de desempenho, a distância abrangida pela rede e muitas outras têm de ser levadas em conta, minuciosamente, para se desenvolver um projeto de rede.

Com relação ao projeto de serviços de comunicação, por sua vez, observa-se a busca por independência do tipo e da topologia da rede de comunicação. A rede de comunicação adotada é, na maioria das vezes, a ponto-a-ponto, pois, além de ser mais simples, é facilmente emulada pelas demais [HAD94].

3.3 Relacionamento dos Eventos

Em muitos esquemas e algoritmos em sistemas distribuídos, é importante ser capaz de especificar se um evento[12] aconteceu antes de um outro evento ou não. Como definido anteriormente (seção 1.1), um sistema distribuído consiste de uma coleção de sistemas de computação autônomos que são separados espacialmente e comunicam-se via troca de mensagens. Em sistemas distribuídos, não há um relógio global, cada nodo tem seu próprio relógio local independente. Este fato dificulta a definição do tempo relativo a diferentes eventos [JAL94].

Durante a execução de uma aplicação, ocorre uma seqüência de eventos, que é representada como e_1, e_2, ... Quando se diz que um evento e_1 ocorreu antes de um evento e_2, significa que o tempo de ocorrência do evento e_1 é anterior ao tempo de ocorrência do evento e_2. Além disso, significa que o tempo relativo a estes eventos foi mensurado pelo mesmo relógio. Claramente, se os tempos de eventos diferentes são mensurados por relógios distintos, desde que relógios distintos podem ter medidas de tempo desiguais, não é possível estabelecer se um evento ocorreu antes ou após um outro.

Com cada nodo na rede possuindo seu próprio relógio, as ações que tomam lugar neste nodo podem ser ordenadas através do seu relógio local. Em sistemas distribuídos, o problema ocorre quando se tenta definir uma relação de ordenação entre eventos de nodos diferentes. Um sistema distribuído consiste de muitos processos (objetos), cada um consistindo de uma seqüência de eventos. Já que um processo (objeto) é uma seqüência de eventos, é possível definir uma relação para os eventos de um processo particular, denominada *happened before* [LAM78]; a relação é denotada pelo símbolo $\rightarrow$.

Um evento *a*, realizado por um processo, é considerado *happened-before* a um outro evento *b*, realizado pelo mesmo processo, se o evento *a* ocorre antes do evento *b* na seqüência de eventos realizada pelo processo. A notação que representa este fato é a seguinte: $a \rightarrow b$.

O relacionamento *happened-before* pode ser estendido, em alguns casos, a eventos em processos distintos. Primeiramente, é preciso observar que em sistemas de troca de mensagens, o recebimento de uma mensagem não pode ocorrer antes do envio desta mensagem. Em outras palavras, pode-se afirmar que o envio de uma mensagem ocorre antes do seu recebimento. Esta ordenação de eventos realizada por processos (objetos) diferentes (o envio da mensagem é executado por um processo e o recebimento por outro) é o mecanismo básico para a ordenação de eventos em processos diferentes. No contexto de comunicação em aplicações distribuídas, os eventos de interesse são o envio de uma mensagem (*send*), o recebimento de uma mensagem

[12] A execução de uma instrução.

(*receive*) e a entrega de uma mensagem (*deliver*). O recebimento de uma mensagem é um evento separado da entrega desta mensagem, significando que uma mensagem pode ser recebida, mas a sua entrega pode levar um certo tempo, ou mesmo não ocorrer.

A relação *happened-before* ($\rightarrow$) sobre um conjunto de eventos em um sistema distribuído é a relação menor que satisfaz as três condições seguintes [LAM78]:

1. se a e b são eventos realizados pelo mesmo processo (objeto), e a é realizado antes de b, então $a \rightarrow b$;
2. se a é o envio de uma mensagem por um processo (objeto) e b é o recebimento da mesma mensagem por um outro processo, então $a \rightarrow b$;
3. se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$.

Dois eventos distintos são ditos concorrentes se nem $a \rightarrow b$, nem $b \rightarrow a$.

A relação $\rightarrow$ define uma relação parcial entre os eventos de um sistema distribuído; ela não define uma ordenação total. Isto é, há eventos (eventos concorrentes) no sistema que não são ordenados pela relação. Outra forma de visualizar a relação $\rightarrow$ é dizer que se $a \rightarrow b$, isto implica que o evento a pode causalmente afetar o evento b. Se os eventos são concorrentes, isto implica que nenhum pode afetar causalmente o outro [JAL94].

Entre as primitivas de comunicação *send*, *receive* e *deliver*, há um relacionamento que segue a relação $\rightarrow$. Pode-se concluir que o envio de uma mensagem é um evento que acontece antes do recebimento desta mensagem, e que o recebimento de uma mensagem é um evento que acontece antes da entrega desta mensagem (figura 3.6).

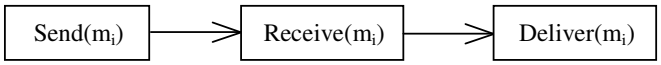

FIGURA 3.6 - Relacionamento entre as primitivas *send*, *receive* e *deliver*

3.4 Sincronismo do Sistema

Sincronismo é um atributo tanto de processos (objetos) quanto de comunicação. Um sistema é dito **síncrono** se satisfaz as seguintes propriedades [HAD93]:

- há um limite superior conhecido δ sobre o atraso de mensagens; este limite consiste no tempo em que leva o envio, transporte e recebimento de uma mensagem sobre um canal de comunicação;
- cada processo (objeto) p possui um relógio local C_p com uma taxa de ajuste conhecida $\rho \geq 0$ relativo ao tempo real. Isto é, para todo p e todo $t > t'$, $(1+\rho)^{-1} \leq \frac{C_p(t) - C_p(t')}{(t-t')} \leq (1+\rho)$, onde $C_p(t)$ é a leitura de C_p no tempo real t;
- são conhecidos os limites inferiores e superiores relativos ao tempo que um processo (objeto) leva para executar um passo.

Em sistemas síncronos, é possível mensurar quando expira o limite de tempo (*timeout*) relativo às mensagens, o que pode fornecer um mecanismo para detecção de

defeitos. Além disso, é possível implementar relógios aproximadamente sincronizados, isto é, relógios que, além de um limite na propriedade de ajuste, também satisfazem a seguinte condição: há um valor ε tal que para todo *t*, e quaisquer dois processos *p* e *q*, $\left|C_p(t) - C_q(t)\right| \leq \varepsilon$.

De fato, tais relógios podem ser implementados, mesmo na presença de defeitos. Para muitos problemas, relógios aproximadamente sincronizados podem também ser utilizados para simular relógios perfeitamente sincronizados (ε=0), o que simplifica o projeto de algoritmos distribuídos [HAD93].

Em um sistema **puramente assíncrono** não há limite sobre o tempo de entrega das mensagens, não há limite com relação à velocidade de execução dos processos e não há sincronização de relógios ou um relógio global. Neste modelo, um processo pode esperar por uma mensagem para sempre, dessa forma um processo com execução lenta e um processo com defeito não podem ser distinguidos. Apesar disso, em decorrência de sua semântica simples, as aplicações programadas com base no modelo assíncrono possuem maior portabilidade.

Síncrono e assíncrono são dois extremos de um espectro de possíveis modelos. Um enfoque intermediário é a adoção de um sistema **assíncrono com temporização** (*timed asynchronous*) [CRI96]. Este tipo de sistema é assíncrono, porque não possui ajuste nos relógios, nem garantias sobre a entrega das mensagens e sobre o escalonamento dos processos. O que o difere de um sistema puramente assíncrono é o fato de que, neste modelo, um processo não espera por uma mensagem indefinidamente. Todos os serviços fornecidos pelo sistema são temporizados: sua especificação descreve não apenas as transições de estado e os resultados esperados das invocações de operações, mas também um intervalo de tempo real dentro do qual tais transições de estado e resultados esperados devem ser observados.

Sistemas assíncronos com temporização permitem a clientes de serviços associarem controles de tempo a cada operação realizada. Na prática, quando este modelo é utilizado, a detecção de defeitos é feita através de controle sobre o tempo na troca de mensagens e o tratamento dos defeitos é realizado através do reenvio das mensagens.

3.5 Modelo de Falhas de Nodos e de Canais de Comunicação

Os termos falha *(fault*), erro *(error*) e defeito *(failure*) têm sido usados de diferentes maneiras e em diferentes contextos, não havendo ainda uma nomenclatura padrão. Assim sendo, é relevante entender o significado destes termos [JAN97] [VAS97] [NUN99] [JAL94].

Uma **falha** é uma condição física anômala. As falhas podem ocorrer na fase de projeto, através de equívocos na especificação do sistema ou em sua implementação; ser de origem física, como problemas de fabricação, fadiga, acidentes ou deterioração; ou ainda serem geradas por distúrbios externos, tais como condições ambientais nocivas, interferência eletromagnética, radiação iônica, entradas imprevistas ou mau uso do sistema. As falhas geradas na fase de projeto ou resultantes de fatores externos são

especificamente difíceis de modelar; delas também é difícil proteger-se, devido à imprevisibilidade de ocorrência e efeitos.

Um **erro** é a manifestação de uma falha no sistema, no qual o estado lógico de um elemento difere do valor previsto. Uma falha existente no sistema não necessariamente resulta em um erro. O erro ocorre apenas quando a falha é sensibilizada; isto é, para um dado estado particular do sistema e para um conjunto de entradas, resulta um próximo estado ou um conjunto de saídas incorreto; uma outra combinação de entradas a partir do mesmo estado poderia não resultar em erro. Assim, uma falha é dita **latente** quando ela ainda não foi sensibilizada pelo sistema.

O **defeito** corresponde à manifestação de incapacidade de algum componente (de *software* ou *hardware*) em realizar a função para a qual foi projetado. Os defeitos são gerados pela existência de falhas e conseqüentemente erros neste mesmo sistema, sendo percebidos pela aplicação. Deste modo, a percepção de defeitos é a ação de fato explorada pelos detectores, por isso o termo **detectores de defeitos** (seção 3.6).

Sistemas distribuídos tolerantes a falhas são necessários em várias situações na vida real [JAL94]: **sistemas de controle de tráfego aéreo**, **sistemas de monitoramento de pacientes**, **sistemas de direcionamento de mísseis**, **bancos**, **bolsas de valores**, entre outros. Claramente, nos tipos de aplicações descritas, um sistema altamente disponível e confiável faz-se necessário (mecanismos de tolerância a falhas em *hardware* e/ou *software* auxiliam no aumento da disponibilidade e confiabilidade). Entretanto, a variada gama de aplicações muitas vezes possui diferentes exigências na qualidade do serviço. Um sistema altamente disponível e confiável pode tornar-se economicamente inviável, se a aplicação não justificar os investimentos a partir de suas necessidades.

Dada a crescente dependência de empresas e aplicações ao uso de sistemas computacionais, observa-se que a busca por serviços de melhor qualidade continuará a crescer ao longo do tempo, o que tem motivado muitas pesquisas na área de tolerância a falhas e suas ferramentas de suporte, principalmente em sistemas distribuídos [NUN99].

Um processo (objeto) é dito **incorreto** em uma execução se o seu comportamento desvia-se daquele prescrito pelo algoritmo que está executando; de outra forma ele é dito **correto**. Um **modelo de defeitos** especifica de que formas um processo falho pode desviar-se de seu algoritmo. A seguir é exposta uma lista de modelos de defeitos que aparecem com freqüência na literatura [HAD93].

Crash (**Colapso**): defeito em que um processo (objeto) falho interrompe sua execução prematuramente e não executa mais nada daquele ponto em diante. Antes de parar, entretanto, ele comporta-se corretamente.

Omissão de Envio: defeito em que um processo (objeto) falho interrompe sua execução prematuramente, ou intermitentemente omite o envio das mensagens que supostamente teria enviado, ou ambas.

Omissão de Recebimento: defeito em que um processo (objeto) falho interrompe sua execução prematuramente, ou intermitentemente omite o recebimento das mensagens a ele enviadas, ou ambas.

Omissão Geral: defeito em que um processo (objeto) falho está sujeito a defeitos de omissão de envio ou de omissão de recebimento, ou ambas.

Arbitrário (às vezes denominado **Bizantino** ou **malicioso**): defeito em que um processo (objeto) falho age de modo totalmente imprevisível. Por exemplo, pode modificar seu estado arbitrariamente.

Arbitrário com autenticação de mensagem: defeito em que um processo (objeto) falho pode exibir comportamento arbitrário, mas um mecanismo para autenticação de mensagens através de assinaturas está disponível. Na presença de defeitos arbitrários, um processo falho pode reivindicar o recebimento de uma mensagem em particular de um processo correto, embora não a tenha recebido de fato. Um mecanismo de autenticação de mensagem permite a outros processos corretos validarem esta reivindicação.

Estes defeitos formam uma hierarquia, com as falhas de colapso sendo as mais simples e mais restritas e as falhas arbitrárias sendo as mais abrangentes e as mais difíceis de tratar. O modelo pode ser classificado em termos da "rigorosidade". Um modelo A é mais "rigoroso" do que um modelo B se um conjunto de comportamentos falhos permitido por B é um subconjunto do conjunto de comportamentos falhos permitidos por A. Desta forma, um algoritmo que consente defeitos do tipo A, também consente aqueles do tipo B. Defeitos arbitrários são os mais rigorosos, pois não impõem quaisquer restrições sobre o comportamento de um processo falho. Defeitos de colapso são os menos rigorosos entre os listados acima. Em geral, quanto menos restritiva a semântica de defeitos, ou seja, quanto mais robusto o sistema, mais cara e complexa torna-se a sua implementação [CRI91].

A classificação deste modelo de defeitos é ilustrada pela figura 3.7, onde uma flecha do tipo B para o tipo A indica que A é mais rigoroso do que B. Por exemplo, colapso é mais rigoroso do que omissão de envio.

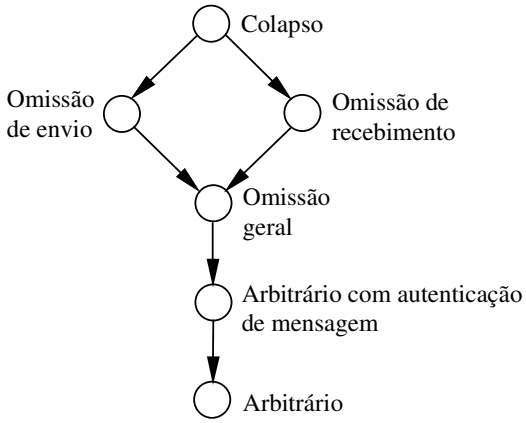

FIGURA 3.7 – Classificação do modelo de defeitos

Os modelos de defeitos acima descritos são aplicáveis a sistemas síncronos e sistemas assíncronos. Em contraste, o **defeito de temporização** é um modelo pertinente apenas aos sistemas síncronos. Observa-se que em tais sistemas há limites relativos a ajustes de relógio e ao tempo necessário à execução de um passo. Um processo (objeto)

sujeito a defeitos de temporização pode tornar-se defeituoso através de um ou mais dos modos seguintes [HAD93]:

- ele apresenta defeito de omissão geral;
- o ajuste de seu relógio local excede o limite especificado (um **defeito de relógio**); e/ou
- ele viola os limites sobre o tempo exigido para a execução de um passo (um **defeito de desempenho**).

Quando considera-se sistemas com relógios aproximadamente sincronizados e processos (objetos) sujeitos a defeitos de temporização, assume-se que apenas os relógios dos processos (objetos) corretos estão, quando muito, afastados um valor ε. Em sistemas síncronos há um limite superior δ sobre o atraso de mensagens. Se processos (objetos) estão sujeitos a defeitos de temporização, assume-se que este limite aplica-se apenas a mensagens sobre canais de comunicação entre processos corretos. Isto porque δ inclui o tempo para os passos de envio e recebimento, e com defeitos de desempenho estes passos podem levar qualquer quantidade de tempo.

Fazendo um adendo à classificação de defeitos anteriormente explicada, observa-se que os defeitos de temporização são mais rigorosos do que os de omissão geral, mas menos rigorosos do que os defeitos arbitrários com autenticação de mensagem. Nesta classificação, todos os tipos de defeitos que não são mais rigorosos do que os defeitos de temporização são denominados de **benignos**. Um processo (objeto) que apresenta um defeito benigno não modifica seu estado arbitrariamente, bem como não envia uma mensagem que não é prescrita pelo seu algoritmo, de acordo com seu estado atual [HAD93].

Os tipos de defeitos assumidos como passíveis de ocorrer, quando um serviço é projetado, são determinantes para o custo deste projeto e para o desempenho do sistema. Ao longo desta dissertação assume-se, como passíveis de ocorrer, defeitos em componentes físicos, especificamente os nodos e os canais de comunicação. Este enfoque não é incomum, pois o foco da maioria dos serviços para tolerância a falhas em sistemas distribuídos está no defeito de componentes físicos, principalmente enfocando defeitos em nodos e em canais de comunicação [JAL94].

Os canais de comunicação podem ser afetados pelos seguintes tipos de defeitos [HAD93]:

Colapso: defeito em que um canal de comunicação interrompe o transporte de mensagens. Antes de parar, entretanto, comporta-se corretamente.

Omissão: defeito em que um canal de comunicação omite, intermitentemente, o transporte de mensagens enviadas através dele.

Arbitrário (às vezes denominado **Bizantino** ou **malicioso**): defeito em que um canal de comunicação exibe um comportamento totalmente imprevisível. Por exemplo, pode gerar mensagens espúrias.

No caso de sistemas síncronos, há também o seguinte defeito:

Temporização: um canal de comunicação falho transporta mensagens mais rápido ou mais lento do que prescreve a sua especificação. Assim, em sistemas

síncronos com defeitos de temporização de processos e de canais de comunicação, assume-se que o limite superior δ, relativo ao atraso de mensagens, aplica-se apenas às mensagens trocadas entre processos (objetos) corretos sobre canais de comunicação corretos. Em tais sistemas, apenas o subconjunto consistindo de processos (objetos) corretos e canais de comunicação corretos é realmente síncrono.

Nesta dissertação, assume-se para os nodos e para os canais de comunicação a semântica de defeitos de colapso: um nodo correto está ativo e apresenta um comportamento normal de acordo com as especificações e um nodo falho (em colapso) não está em funcionamento e não exibe um comportamento incorreto para o meio externo. Com relação aos canais de comunicação, o comportamento é dito correto se, para qualquer mensagem *m* que for enviada um número infinito de vezes por um processo (objeto) correto *p*, há o equivalente recebimento de *m* por parte de um processo (objeto) correto *q* esse mesmo número infinito de vezes. Um canal de comunicação tem comportamento incorreto se não está de acordo com esta semântica.

3.6 Detecção de Defeitos

Desde que foi provada a impossibilidade de obtenção de consenso em um sistema assíncrono [FIS85], ainda que apenas um processo (objeto) apresente defeito, muita pesquisa foi feita nesta área. Uma forma de contornar esta impossibilidade surgiu com a criação de detectores de defeitos não-confiáveis [CHA96]. Nesta proposta, há detectores de defeitos, localizados um em cada nodo, que armazenam localmente a informação relacionada aos defeitos ou aos elementos sob suspeita. Em síntese, cada detector encapsula suposições de temporização sobre os eventos do sistema e transforma as violações ocorridas em uma forma de detecção dos defeitos. Observa-se que o mecanismo é não-confiável, ou seja, um nodo pode ser suspeito de estar defeituoso, quando na realidade não está.

Formalmente, um detector de defeitos é definido abstratamente através de propriedades de confiabilidade; os detectores foram classificados através de oito classes, com dois atributos ortogonais: *completeness* (completude) e *accuracy* (precisão) [CHA96]. A propriedade *completeness* descreve as exigências para que um detector de defeitos suspeite, em um tempo finito, de todos os processos (objetos) que estão realmente defeituosos. A propriedade *accuracy* descreve as restrições quanto aos enganos que um detector pode cometer. Mais especificamente, as oito classes mencionadas são definidas com base na combinação de diferentes nuances das propriedades de *completeness* (*strong* e *weak*) e de *accuracy* (*strong*, *weak*, *eventual strong* e *eventual weak*).

- ***Completeness***:
  - *strong*: todo processo (objeto) que apresenta defeito é tornado permanentemente suspeito por todo processo (objeto) correto, em um tempo finito;
  - *weak*: todo processo (objeto) que apresenta defeito é suspeito, de forma permanente, por algum processo (objeto) correto, em um tempo finito.
- ***Accuracy***:
  - *strong*: nenhum processo (objeto) é suspeito antes que ele apresente defeito;

- *weak*: algum processo (objeto) correto nunca torna-se suspeito antes que ele apresente defeito;
- *eventual strong*: há um tempo após o qual os processos (objetos) corretos não são suspeitos por qualquer processo (objeto) correto;
- *eventual weak*: há um tempo após o qual algum processo (objeto) correto nunca torna-se suspeito por qualquer processo (objeto) correto.

A combinação entre estas propriedades e a designação e simbologia utilizada para cada detector resultante são sintetizadas na tabela 3.1.

TABELA 3.1 - Oito classes de detectores de defeitos não-confiáveis

Completeness	**Accuracy**			
	Strong	Weak	Eventual Strong	Eventual Weak
Strong	*Perfect* P	*Strong* S	*Eventual Perfect* $\Diamond P$	*Eventual Strong* $\Diamond S$
Weak	Q	*Weak* W	$\Diamond Q$	*Eventual Weak* $\Diamond W$

Na prática, para ser feita a detecção de defeitos, cada processo tem acesso a um módulo detector de defeitos local, que monitora um grupo ou subgrupo de processos (objetos) no sistema, mantendo uma lista de processos (objetos) suspeitos. O detector é considerado não-confiável, pois pode suspeitar erroneamente de um ou mais processos (objetos) no sistema.

Observa-se que os algoritmos utilizados para a detecção de defeitos são baseados em mecanismos de controle de tempo; geralmente um defeito é detectado quando a ausência de resposta de um nodo remoto faz com que os protocolos de comunicação deixem de progredir. Tendo-se, por suposição, que cada processo (objeto) envia mensagens periodicamente, se não for recebida nenhuma comunicação de certo processo (objeto) em um determinado tempo, é razoável assumir que tal processo está com algum defeito. Os algoritmos de detecção de defeitos localizam-se normalmente na camada de transporte, que implementa a comunicação interprocessos. Para garantir a detecção de defeitos através de temporizadores na presença de tráfego baixo e unidirecional, alguns sistemas exigem que cada processo envie mensagens do tipo "*I am alive*" periodicamente [GUO98].

Os algoritmos de detecção de defeitos baseados em temporizadores podem ser divididos em duas categorias principais [VOG96], que são descritas resumidamente a seguir.

O primeiro esquema utiliza um mecanismo de *heartbeat*, onde cada processo envia mensagens do tipo "*I am alive*" através de múltiplas mensagens ponto-a-ponto ou de uma única mensagem via *multicast* IP. Cada processo registra o tempo de recebimento das mensagens. Se estiver faltando um certo número de *heartbeats* consecutivos, é levantada uma suspeita sobre o processo em questão. O tempo relativo a *heartbeats* consecutivos, bem como controles de tempo utilizados, é configurável pela aplicação que utiliza o esquema. Além disso, é possível enviar os *heartbeats* associados (*piggyback*) às mensagens de dados normais.

O segundo esquema utiliza um método de sondagem (*polling*), onde o detector de defeitos envia pedidos aos processos e coleta mensagens de resposta dos mesmos. Se nenhum dos ACKs correspondentes às solicitações for recebido após um certo número de reenvios, então o detector levanta uma suspeita de falha em relação ao(s) processo(s) em questão. Períodos de sondagens, temporizadores e limites de retransmissão são configuráveis pela aplicação.

A detecção de defeitos é uma área em constante pesquisa na comunidade científica de tolerância a falhas. No Grupo de Tolerância a Falhas da UFRGS, por exemplo, foi desenvolvida monografia que explora as particularidades dos diferentes tipos de detectores [EST2000], há uma dissertação em paralelo com esta que explora comparativamente diversos detectores de defeitos [EST2001] e uma proposta de tese de doutorado em andamento [NUN2000].

3.7 Métodos de Obtenção de Confiabilidade

Em redes de comunicação reais, observa-se que dados podem ser perdidos na troca de mensagens. Um protocolo de comunicação garante, através de técnicas como números de seqüência, retransmissão, etc., que os serviços desejados são fornecidos ao usuário da rede, ainda que esta rede não seja confiável. A comunicação confiável visa garantir que uma mensagem enviada por um processo (objeto) correto chegue ao destinatário e que, neste, seja entregue no máximo uma vez. Os principais métodos de obtenção de confiabilidade são [JAL94]: a utilização de redundância, a utilização de reconhecimentos positivos (ACKs) ou a utilização de reconhecimentos negativos (NACKs).

Métodos que utilizam ACKs são **orientados a emissor** e métodos que utilizam NACKs são **orientados a receptor**; no primeiro caso, o emissor obtém ACKs de todos os receptores periodicamente e, desta forma, pode liberar as mensagens armazenadas nos *buffers*; no segundo caso, os receptores enviam NACKs quando detectam mensagens perdidas e não há o envio de ACKs para o emissor [TAN94] [PIN94].

A seguir, são descritos alguns métodos para a obtenção de comunicação confiável. A seção 3.7.1 apresenta a proposta de **utilização de redundância**; em 3.7.2 e 3.7.3, são vistos, respectivamente, os enfoques **orientado a emissor** e **orientado a receptor**; a **combinação destes dois enfoques** está na seção 3.7.4; e a utilização de um enfoque com base num **modelo hierárquico** é elucidado na seção 3.7.5.

3.7.1 Enfoque Baseado em Redundância

Um sistema tolerante a falhas deve evitar o mau funcionamento (defeitos) mesmo na presença de falha de algum de seus componentes. A grande maioria das técnicas de tolerância a falhas utiliza a introdução de redundância em determinados componentes do sistema, cuja implementação pode ocorrer utilizando suporte tanto em *software*, quanto em *hardware*. O conceito de redundância implica no acréscimo de ações, de informações, de recursos ou de tempo além do que é necessário para a operação normal do sistema [JAL94].

Os tipos mais conhecidos de redundância utilizados na obtenção de níveis variados de tolerância a falhas são [PRA96]:

- **redundância de *hardware*** (também referenciada como **espacial**): quando envolve a replicação de componentes (ex.: redundância modular tripla);
- **redundância temporal**: quando diz respeito à repetição de ações (ex.: recomputações, retransmissões e transmissões múltiplas de mensagens);
- **redundância de informação**: quando se adicionam informações redundantes aos dados (ex.: códigos de detecção e correção de erros); e
- **redundância de *software***: quando usa algumas linhas de código, subrotinas ou mesmo programas inteiros de forma redundante à funcionalidade do programa (ex.: programação N-versões).

A utilização de redundância, pode se afirmar, é a forma mais básica de obtenção de comunicação confiável. Nesse contexto, a redundância mais aplicável é a redundância temporal, onde uma mensagem é transmitida ou retransmitida mais de uma vez para o(s) destinatário(s). Esta técnica exige um mecanismo que detecte o recebimento de mensagens duplicadas, que geralmente devem ser ignoradas. Um exemplo de utilização deste enfoque para a obtenção de um *multicast* confiável, ilustrado pela figura 3.8, é descrito a seguir:

1. o emissor *s* envia a mensagem *m* para os receptores r_1 e r_2;
2. r_1, ao receber *m*, a retransmite;
3. r_2, ao receber *m*, a retransmite;

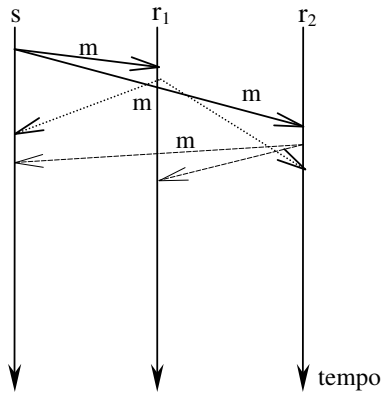

FIGURA 3.8 - Enfoque baseado em redundância

Este enfoque garante que a mensagem *m* chega a todos os destinatários corretos, desde que não ocorram mais do que *n-1* perdas, onde *n* representa o número de receptores, bem como o número de vezes que a mensagem é retransmitida. Uma otimização possível para este mecanismo seria não retransmitir *m* de volta ao emissor.

3.7.2 Enfoque Orientado a Emissor

No enfoque orientado a emissor, a responsabilidade de obtenção de confiabilidade recai sobre o emissor, que se mantém informado do estado dos receptores para os quais enviou mensagens; ou seja, mantém uma lista de receptores e, para cada mensagem enviada, a informação sobre se já foi reconhecida. A confiabilidade e a manutenção da

informação de estado são garantidas pelo retorno dos receptores, reconhecendo (ACK) as mensagens que foram recebidas corretamente, e pela utilização de temporizadores no emissor para o propósito de detecção de defeitos.

Um exemplo de funcionamento deste enfoque para a implementação de uma primitiva de *multicast* confiável é descrita a seguir [GUO98]:

1. sempre que o emissor enviar uma mensagem (através de *multicast*), ele inicializa um temporizador associado com esta mensagem;
2. periodicamente, um receptor envia uma mensagem ACK (através de *unicast*) para o emissor, identificando as mensagens que foram recebidas corretamente - o reconhecimento pode ser o de uma mensagem específica ou de uma janela de mensagens;
3. ao receber um ACK, o emissor atualiza a lista de ACKs associada às mensagens indicadas no reconhecimento;
4. sempre que o temporizador expirar antes da chegada do reconhecimento de todos os receptores, o emissor reenvia a mensagem (através de *multicast*);
5. sempre que uma mensagem for totalmente reconhecida, ou seja, todos os receptores enviaram um ACK, o emissor pode retirar a mensagem do seu *buffer* e cancelar o temporizador relativo a esta.

Exemplos de execuções dessa primitiva de comunicação *multicast* confiável, implementada através do enfoque orientado a emissor, são ilustrados a seguir. No primeiro caso (figura 3.9), após o emissor *s* transmitir a mensagem *m* (via *multicast*) para os receptores r_1 e r_2, ele inicializa um temporizador associado a *m*; após o recebimento do ACK (via *unicast*) de r_1 e r_2, o emissor cancela o temporizador.

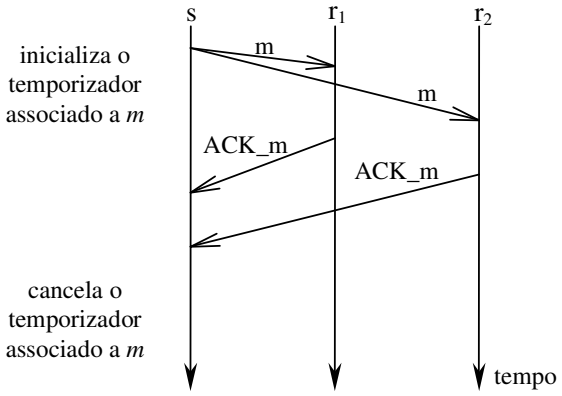

FIGURA 3.9 - Enfoque orientado a emissor sem perda de mensagens

No segundo caso (figura 3.10), o emissor *s* transmite a mensagem *m* (via *multicast*) para os receptores r_1 e r_2; *m* chega corretamente em r_1, mas é perdida no caminho para r_2. Se o temporizador associado a *m* expirar e o emissor *s* ainda não tiver recebido o ACK de r_2, então a mensagem *m* é reenviada.

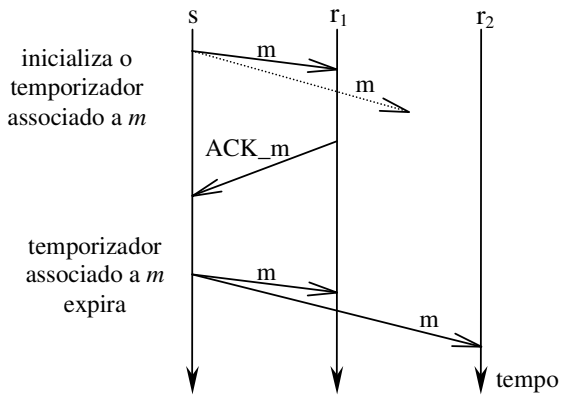

FIGURA 3.10 - Enfoque orientado a emissor com perda de mensagens

A principal limitação do enfoque orientado a emissor vem do fato de que emissor necessita conhecer o conjunto de receptores e necessita processar os reconhecimentos (ACKs) provindos deles. À medida que aumenta o número de receptores, o emissor é sobrecarregado por ter que armazenar grande quantidade de informação relativa ao estado dos receptores, além de sofrer de uma implosão de mensagens ACK. O problema da implosão de ACKs provoca o seguinte impacto [GUO98]:

- primeiro, o tremendo número de mensagens ACK ocasiona uma degradação do desempenho no emissor, bem como ocasiona atrasos na comunicação de dados;
- segundo, uma grande quantidade de mensagens ACK pode ocasionar um uso excessivo tanto do espaço dos *buffers*, como da largura de banda, provocando perdas adicionais de mensagens.

3.7.3 Enfoque Orientado a Receptor

Neste enfoque, a responsabilidade de obtenção de confiabilidade recai sobre o receptor, que o realiza através da utilização de reconhecimentos negativos (NACKs). Após um receptor detectar a perda de alguma mensagem, observando intervalos no número de seqüência das mensagens recebidas, ele solicita uma retransmissão através do envio de uma mensagem NACK.

Para proteger-se contra a perda da própria mensagem NACK, o receptor inicializa um temporizador associado à mensagem NACK quando a envia; caso o temporizador expire antes do recebimento da mensagem relacionada ao NACK, então a mensagem NACK é reenviada e o temporizador é reinicializado. O temporizador é cancelado quando do recebimento da mensagem relacionada ao NACK. Um exemplo de funcionamento deste enfoque para a implementação de uma primitiva de *multicast* confiável é descrita a seguir [GUO98]:

1. sempre que um receptor detecta a perda de uma mensagem, ele envia um reconhecimento negativo (NACK) e então inicializa um temporizador associado à mensagem;
2. sempre que o temporizador expirar antes do recebimento da mensagem requisitada, um receptor envia novamente a mensagem NACK e reinicializa o temporizador;
3. há dois enfoques para enviar o pedido de retransmissão (NACK): enviar a

mensagem NACK via *unicast* para o emissor, ou enviar a mensagem NACK para todo o conjunto de processos (englobando o emissor e todos os receptores). De forma combinada ao uso de um destes dois enfoques, podem ser usadas três variações:

- na primeira variação, ao receber uma mensagem NACK via *unicast*, o emissor reenvia a mensagem requisitada via *multicast*;
- na segunda variação, ao receber uma mensagem NACK via *multicast*, o emissor reenvia a mensagem requisitada via *multicast*. Sempre que um receptor receber uma mensagem NACK requisitando a mesma mensagem que ele requisitou ao enviar uma mensagem NACK, ele reinicializa o temporizador associado à mensagem perdida. Este mecanismo suprime mensagens NACK redundantes; e
- a terceira variação é semelhante à segunda, com exceção de que qualquer membro que possua a mensagem requisitada pode retransmiti-la, não somente o emissor.

A seguir são ilustrados três exemplos de execuções de uma primitiva de comunicação *multicast* confiável implementada através do enfoque orientado a receptor.

Na primeira variação (figura 3.11), o emissor *s* envia as mensagens m_1, m_2 e m_3 (via *multicast*) a dois receptores. As mensagens m_1 e m_3 chegam aos receptores corretamente, mas m_2 chega em r_1 e não chega em r_2. Ao detectar a perda de m_2, observando um intervalo no número de seqüência das mensagens recebidas, r_2 envia uma mensagem NACK (via *unicast*) para o emissor *s*, requisitando a retransmissão de m_2, e inicializa um temporizador associado à mensagem NACK de m_2.

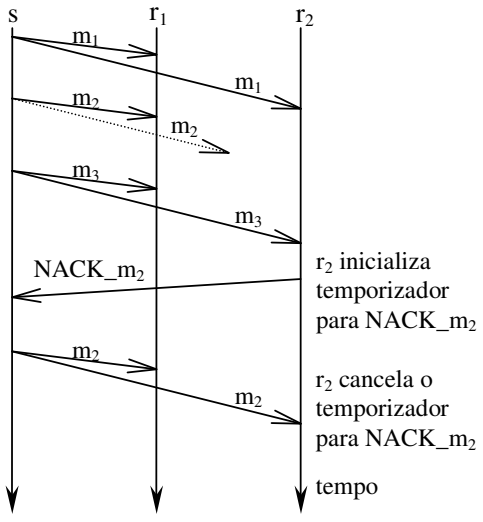

FIGURA 3.11 - Primeira variação do enfoque orientado a receptor

Quando o emissor *s* recebe a mensagem NACK de r_2, reenvia a mensagem m_2 (via *multicast*); r_1 ignora a mensagem m_2, pois detecta que é uma duplicata; e r_2 cancela o temporizador associado a m_2 após recebê-la.

Na segunda variação (figura 3.12), o emissor *s* envia as mensagens m_1, m_2 e m_3 (via *multicast*), mas m_2 não chega a nenhum dos receptores. O receptor r_1 é o primeiro a

detectar a perda da mensagem e então envia uma mensagem NACK (via *multicast*), requisitando o reenvio de m_2, e inicializa um temporizador associado ao NACK.

Após r_2 receber o NACK requisitando m_2, ele inicializa um temporizador associado a este NACK (como se fosse ele próprio o emissor da mensagem NACK). Quando o emissor *s* recebe o NACK de r_1, ele reenvia a mensagem m_2 (via *multicast*). Após o recebimento de m_2, r_1 e r_2 cancelam os temporizadores que estavam associados ao NACK de m_2.

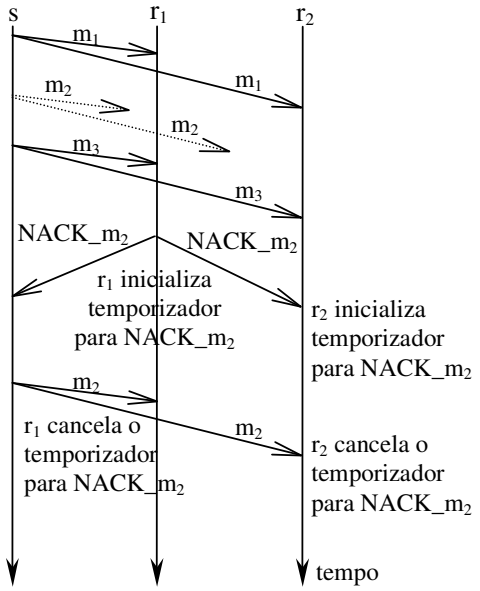

FIGURA 3.12 - Segunda variação do enfoque orientado a receptor

Na terceira variação (figura 3.13), o emissor *s* envia as mensagens m_1, m_2 e m_3 (via *multicast*), a mensagem m_2 chega em r_1 e não chega em r_2. Ao detectar a perda, r_2 envia uma mensagem NACK (via *multicast*) requisitando m_2 e inicializa um temporizador para este NACK. Ao receber o pedido de reenvio de m_2 (NACK de r_2), r_1 detecta que recebeu m_2 corretamente e faz o reenvio de m_2. Ao receber m_2, r_2 cancela o temporizador associado ao NACK correspondente. O emissor *s* recebeu o NACK para m_2 após ter recebido a própria mensagem: então ele ignora o NACK, pois nesta situação algum outro processo (objeto) já fez o reenvio de m_2.

Neste enfoque orientado a receptor, o emissor não sabe a quantidade de receptores; os receptores, por sua vez, não enviam reconhecimentos das mensagens recebidas. Em decorrência disso, não há no emissor um mecanismo seguro para a exclusão das mensagens armazenadas em seus *buffers*, exigindo um protocolo para a detecção de mensagens estáveis[13] (que podem ser excluídas dos *buffers*).

Além disso, este enfoque não é apropriado para o fornecimento de um serviço de comunicação totalmente confiável, porque não há um mecanismo que permita a qualquer processo detectar quando um receptor apresenta defeito [GUO98].

[13] Uma mensagem é declarada estável quando o processo (objeto) que a armazena toma conhecimento de que ela já foi recebida pelos processos (objetos) destinatários.

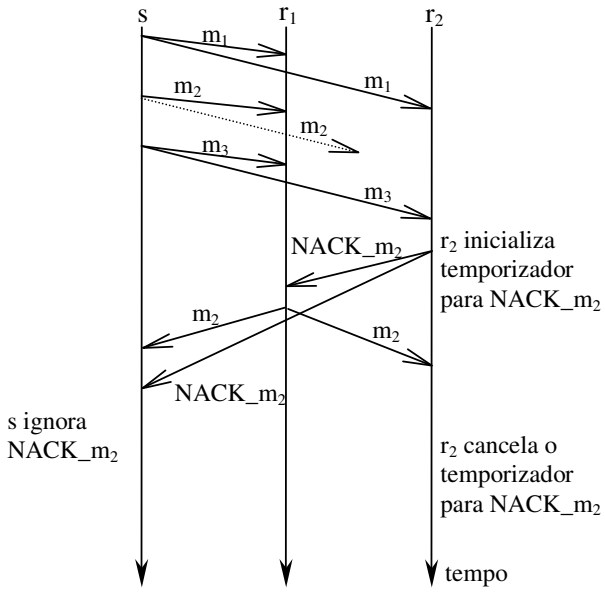

FIGURA 3.13 - Terceira variação do enfoque orientado a receptor

A vantagem deste enfoque provém do fato de que a informação de estado a ser armazenada é mínima, o que permite uma escalabilidade boa.

3.7.4 Enfoque Combinado: Orientado a Emissor e Orientado a Receptor

O enfoque puramente orientado a receptor exige um esquema de gerenciamento dos *buffers* para detectar quando uma mensagem foi recebida em todos os destinos e, portanto, pode ser excluída de cada *buffer* correspondente. Para combater esta limitação, na prática, muitos protocolos utilizam tanto ACKs como NACKs; este tipo de combinação aparece freqüentemente em protocolos baseados em um anel lógico [AMI95] [TAN94].

Os protocolos baseados em anel, e destinados ao envio confiável de mensagens, foram originalmente desenvolvidos para fornecer suporte a aplicações que exigiam atomicidade e ordenação total de transmissões em todos os receptores, combinando as vantagens de desempenho de NACKs com a confiabilidade de ACKs. A premissa básica deste mecanismo é ter apenas um *token site* (nodo no sistema que detém o *token*, em determinado instante) que seja responsável por enviar mensagens de ACK ao emissor. O emissor retransmite mensagens se não receber um ACK do *token site* em determinado período de tempo. O ACK também serve para registrar o *timestamp* (identificação de tempo) nas mensagens, fornecendo um mecanismo para a obtenção de ordenação total. O protocolo não permite que os receptores efetuem a entrega das mensagens para a aplicação antes que o *token site* tenha enviado o ACK.

Receptores enviam NACKs para o *token site*, visando a repetição seletiva de mensagens perdidas que haviam sido originalmente enviadas pelo emissor. O ACK enviado de volta para o emissor também serve como um mecanismo de passagem do

token. O *token* não é passado para o próximo membro do anel lógico de receptores até que o nó que está prestes a desempenhar essa função tenha recebido, corretamente, todos os pacotes que o nó anterior (que detinha o *token*) recebeu. Uma vez passado o *token*, é possível liberar os dados da memória no receptor; o emissor liberará dados da memória quando for recebido um reconhecimento.

Um exemplo de funcionamento deste enfoque, quando não ocorrem defeitos é ilustrado pela figura 3.14, onde: *s* é o emissor, r_1 e r_2 os receptores, sendo r_2 o *token site*. A mensagem *m* é transmitida (via *multicast*) por *s* para r_1 e r_2; *s* inicializa um temporizador associado a *m*. O *token site* envia a mensagem de ACK (via *multicast*), e *s* cancela o temporizador associado a *m*, quando a recebe.

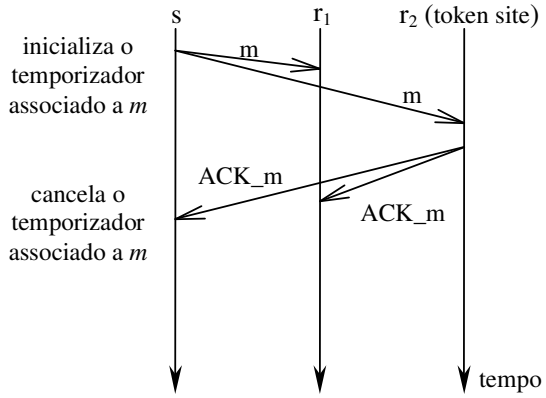

FIGURA 3.14 - Enfoque combinado sem defeitos

Um caso onde o receptor r_1 detecta a perda da mensagem *m* é ilustrado na figura 3.15. Neste exemplo, o emissor *s* transmite *m* (via *multicast*) para r_1 e r_2 (*token site*) e inicializa um temporizador associado a *m*. O *token site* envia a mensagem de ACK (via *multicast*), que é recebida tanto por *s*, quanto por r_1. Ao receber o ACK, *s* cancela o temporizador associado a *m*. O receptor r_1, por outro lado, detecta a perda da mensagem *m* ao receber o ACK, e pede retransmissão da mesma enviando uma mensagem de NACK para r_2 (*token site*) que a retransmite (via *unicast*).

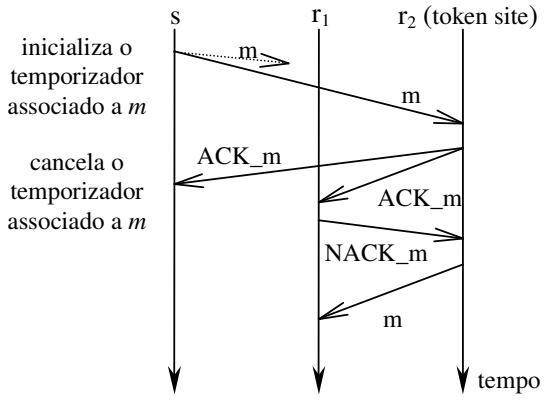

FIGURA 3.15 - Enfoque combinado com perda de mensagem

3.7.5 Enfoque Hierárquico

A utilização de uma estrutura hierárquica no enfoque orientado a emissor pode reduzir significativamente o problema decorrente da implosão de mensagens do tipo ACK. O enfoque hierárquico é caracterizado pela divisão do conjunto de receptores em grupos, distribuindo a responsabilidade de retransmissão sobre uma estrutura de árvore de reconhecimento. Dentro da estrutura formada, em cada ramo, há uma hierarquia entre nós-pais (líderes do grupo) e nós-filhos [YAV95].

Os nós-filhos enviam reconhecimentos hierárquicos (HACKs) aos respectivos pais e não ao nó emissor da mensagem, melhorando o desempenho. Cada pai, por sua vez, poderá também ser filho e a regra de envio de HACKs aplica-se recursivamente. Os pais colecionam os HACKs dos filhos no seu grupo e não liberam os dados da memória até que todos os filhos tenham feito o reconhecimento da transmissão respectiva. Claramente, uma árvore limitada a dois níveis, constituída de um emissor (pai) e de vários receptores (filhos nas folhas) corresponde ao esquema orientado a emissor.

Além disso, cada pai possui um temporizador que é ajustado quando ele envia uma mensagem; se acontecer de expirar o tempo de espera previsto para o recebimento de todos os HACKs, assume que a mensagem foi perdida e há um reenvio [AMA99].

Para exemplificar o mecanismo, considera-se a estrutura hierárquica ilustrada pela figura 3.16, onde *s* é o emissor da mensagem e r_1, r_2, r_3, r_4 e r_5 são os receptores. Nesta hierarquia, *s* só precisa aguardar as mensagens de HACK de r_1 e de r_2. O receptor r_1 aguarda a mensagem de HACK de r_3 e de r_4 e o receptor r_2 aguarda a mensagem de HACK de r_5.

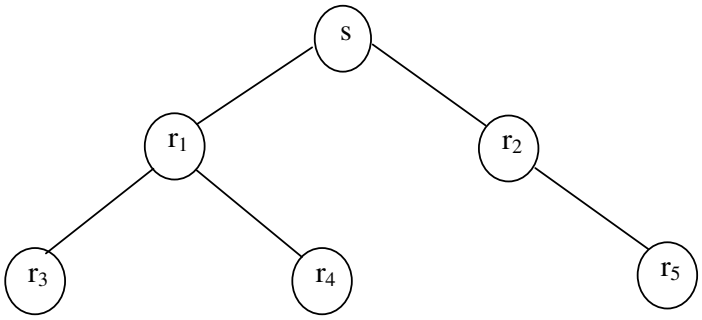

FIGURA 3.16 - Exemplo de uma estrutura hierárquica

Um exemplo de funcionamento deste enfoque, sem defeitos, utilizando a mesma estrutura hierárquica da figura 3.16, é ilustrado pela figura 3.17. Neste caso, o emissor *s* transmite a mensagem *m* (via *multicast*) para os receptores r_1, r_2, r_3, r_4 e r_5, e inicializa um temporizador associado a *m*. Ao receber a mensagem *m*, r_1 e r_2 enviam a mensagem de HACK para *s* e inicializam um temporizador associado a *m*.

Quando *s* recebe as mensagens de HACK de r_1 e de r_2, ele cancela o temporizador associado a *m*. Os receptores r_3, r_4 e r_5 enviam mensagens de HACK para seus respectivos pais na estrutura hierárquica, neste caso, os pais são r_1 e r_2. Além disso, r_3, r_4 e r_5 não precisam inicializar nenhum temporizador associado à mensagem *m*, pois

eles são nós-folha na estrutura hierárquica e, desta forma, não receberam nenhuma mensagem de HACK.

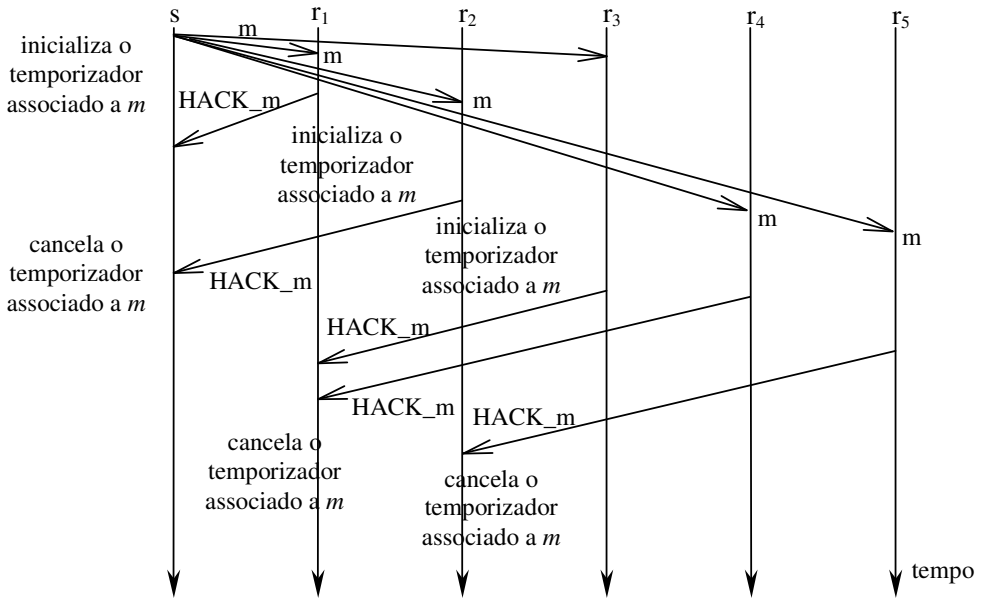

FIGURA 3.17 - Enfoque hierárquico sem defeitos

Os casos em que ocorrem perdas de mensagens são tratados de forma idêntica ao enfoque orientado a emissor (seção 3.7.2) e, portanto, não serão ilustrados nesta seção. A única diferença é que as retransmissões no enfoque hierárquico não são realizadas apenas pelo emissor, cada pai na estrutura hierárquica é quem retransmite aos filhos quando o tempo de espera por uma mensagem de HACK do filho correspondente expira.

É importante observar a necessidade de existir um protocolo para a eleição de um novo pai, sempre que algum apresentar uma semântica de defeito de colapso, sob pena dos filhos relativos não receberem as mensagens enviadas durante o colapso.

4 Protocolos de Comunicação

Aplicações distribuídas necessitam de primitivas de comunicação que forneçam diversas semânticas, desde o tipo **ponto-a-ponto não-confiável** (*unicast*) até o **multiponto confiável** (*reliable multicast/reliable broadcast*). Estas primitivas são fornecidas através de protocolos de comunicação, que, por sua vez, podem ser utilizados na implementação de serviços de comunicação. Protocolos mais simples constituem-se em blocos de construção básicos na composição de outros protocolos e serviços. Tendo este fator motivador, esta dissertação implementa quatro tipos básicos de protocolos de comunicação, que são utilizados na construção de serviços de envio de mensagens do tipo ponto-a-ponto e multiponto: ***unicast* não-confiável**, ***unicast* confiável**, ***multicast* não-confiável** e ***multicast* confiável**.

4.1 *Unicast* Não-Confiável

Este tipo de comunicação é utilizado para o envio de mensagens ponto-a-ponto entre nodos da rede onde a confiabilidade não é uma exigência, e onde a rapidez de comunicação é o fator mais relevante. Na figura 4.1, é apresentado o algoritmo utilizado para implementar este tipo de comunicação.

```
No emissor s
Unicast(r, msg):
       Send(s, msg) para r;
       Retorna true;

No receptor r
Receive(s, msg):
       Receive(s, msg);
       Deliver(s, msg);
```

FIGURA 4.1 - Algoritmo para comunicação *unicast* não-confiável

Neste algoritmo, s é o emissor da mensagem e r é o receptor. No nodo emissor, o envio da mensagem *msg* é feito através do método **`Unicast`**. Este método utiliza uma primitiva para o envio de mensagens fornecida pela camada de transporte (*UDP*). Anexada à mensagem vai a informação sobre quem é o emissor.

No nodo receptor, a mensagem é recebida através do método **`Receive`**, que utiliza uma primitiva para o recebimento de mensagens fornecida pela camada de transporte (*UDP*). O emissor da mensagem é identificado de acordo com o campo correspondente na mensagem recebida e, então, a mensagem está pronta para ser entregue à camada posterior, que pode ser a aplicação ou um outro protocolo.

O funcionamento deste algoritmo simples é ilustrado pela figura 4.2. O emissor s envia a mensagem m_1 para o receptor r. Ao receber m_1, r identifica o emissor e entrega a mensagem para a aplicação.

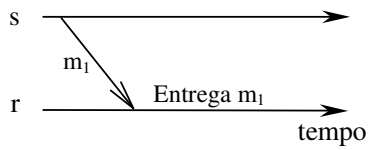

FIGURA 4.2 - Comunicação *unicast* não-confiável

4.2 *Unicast* Confiável

A inexistência de mecanismos que fornecem confiabilidade implica em implementá-los na própria aplicação, empregando precioso tempo de desenvolvimento e aumentando a complexidade da solução. Além disso, a maior desvantagem de se ter que implementar os mecanismos é que, geralmente, eles acabam se tornando específicos para cada aplicação. Mecanismos específicos não se adequam à reutilização em outras aplicações [AMA99].

Supondo que dois processos (objetos) *p* e *q* desejam se comunicar, o *unicast* confiável entre eles é um tipo de comunicação que atende às seguintes propriedades [HAD94]:

- **Validade**: se *p* envia *m* para *q*, e tanto *p* e *q* quanto o canal de comunicação entre eles são corretos, então *q* entrega *m* em um tempo finito.
- **Integridade Uniforme**: para qualquer mensagem *m*, *q* entrega *m* no máximo uma vez de *p*, e apenas se *p* enviou *m* para *q* anteriormente.

Para garantir integridade, as mensagens recebem um valor que as identifica univocamente. Além disso, em cada processo (objeto) deve existir uma variável indicando a última mensagem entregue, referente a cada emissor.

Adicionalmente, certas aplicações necessitam de algum tipo de ordenação na comunicação, sendo a forma mais básica a ordenação FIFO, implementada através de um protocolo que possui a garantia de que as mensagens enviadas são recebidas pelos destinatários na mesma ordem de envio.

A característica de confiabilidade neste *unicast* confiável é conseguida através da utilização de reconhecimentos positivos (ACKs), conforme elucidado na seção 3.7.2. A ordenação FIFO é obtida através da utilização de **identificadores únicos** em cada mensagem enviada. O identificador único é construído fazendo com que toda a mensagem *m* inclua os seguintes campos: um **identificador de emissor** e um **identificador de número de seqüência**. A seguir, descreve-se um algoritmo para a implementação do *unicast* confiável com ordenação FIFO (figura 4.3). Este algoritmo é baseado em um algoritmo descrito por Anish Karmarkar [KAR97].

Neste algoritmo, `s` é o emissor da mensagem e `r` é o receptor. Observando o emissor, a mensagem é enviada através do método **`ReliableUnicast`**. Cada emissor armazena números de seqüência que correspondem à última mensagem enviada para cada receptor, denotado por **`ucastSendSeqNo`**$_{\mathbf{r}}$, inicializado em -1. Este número de

seqüência é anexado a cada mensagem enviada, para que o receptor tome conhecimento e possa garantir a entrega de acordo com os requisitos necessários.

```
No emissor s
ReliableUnicast(r, msg):
       ucastSendSeqNo_r ← ucastSendSeqNo_r + 1;
       FAÇA reenvio de msg n vezes
          Send(s, ucastSendSeqNo_r, msg) para r;
          Espere até:
              1. TIMEOUT; ou
              2. ACK da mensagem enviada.
          Se for recebido ACK
              então retorna true;
          fim_Se
       Fim_FAÇA
       Retorna false;

No receptor r
Receive(s, seqNo, msg):
       Receive(s, msg);
       Send(s, ACK, seqNo) para s;
       Se seqNo = ucastRecvSeqNo_s + 1
         então
              Deliver(s, msg);
              ucastRecvSeqNo_s ← ucastRecvSeqNo_s + 1;
         senão
              Se seqNo > ucastRecvSeqNo_s + 1
                então
                     UcastBuf.insert(msg); //insere a mensagem no buffer
              fim_Se
       fim_Se
```

FIGURA 4.3 - Algoritmo para comunicação *unicast* confiável

Após enviar a mensagem, o emissor entra em um estado de bloqueio, esperando a ocorrência de um entre dois eventos: o reconhecimento positivo (ACK) da mensagem pelo receptor ou a ocorrência de uma expiração do controle de tempo (especificado com base na rede onde o algoritmo está sendo executado, número de nodos, entre outros fatores).

Quando expira o controle de tempo, há o reenvio da mensagem; este reenvio é feito um número predefinido de vezes, após o qual o método retorna *false*, significando que a operação não pode ser realizada com sucesso, o que ocorre quando em presença de algum defeito. O recebimento do ACK desbloqueia o emissor e significa que a operação foi completada com sucesso; desta forma, é retornado um valor *true*, significando sucesso na operação.

No lado do receptor, a mensagem é recebida, retirada do *buffer* que a armazena e processada no método **`Receive`**. O receptor então envia uma mensagem ACK para o emissor, confirmando o recebimento da mensagem. Cada receptor armazena o número de seqüência da última mensagem recebida de cada emissor, denotado por **$\mathbf{ucastRecvSeqNo_s}$**.

Quando uma mensagem é recebida, é verificado o seu número de seqüência. A mensagem só é entregue para a camada superior (aplicação ou outro protocolo) se o

número de seqüência que a identifica é o esperado pelo receptor. Este procedimento garante a ordenação FIFO das mensagens entregues, conforme a especificação a que se propõe o protocolo. Se o número de seqüência for menor do que o esperado, significa que a mensagem é uma duplicata e, desta forma, ela é descartada; se o número de seqüência for maior do que o esperado, significa que a mensagem ainda não está apta a ser entregue e, portanto, ela é armazenada novamente no *buffer*.

A seguir, serão ilustrados diversos exemplos de execução do algoritmo adotado para o envio confiável de mensagens do tipo *unicast*.

4.2.1 Exemplos de Execução do Algoritmo

Uma execução normal, sem a ocorrência de defeitos é ilustrada na figura 4.4. Neste caso, o emissor `s` envia a mensagem m_1 para o receptor `r`. Ao receber m_1, `r` envia uma mensagem de ACK para `s` e, como o número de seqüência é o esperado, entrega a mensagem para a aplicação. O emissor `s` é desbloqueado ao receber a mensagem de ACK, retornando *true* para indicar que a operação foi realizada com sucesso, ou seja, foi realizado o envio confiável da mensagem m_1 de `s` para `r`.

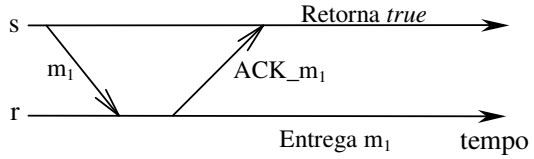

FIGURA 4.4 - *Unicast* confiável sem defeito

Na figura 4.5, é ilustrado o caso em que o emissor `s` envia a mensagem m_1 para o receptor `r`, mas m_1 é perdida, logo `s` não recebe a mensagem de ACK no tempo previsto. Neste caso, `s` reenvia a mensagem m_1 e novamente aguarda o ACK. Ao receber m_1, `r` envia uma mensagem de ACK para `s` e, como o número de seqüência é o esperado, entrega a mensagem para a aplicação. O emissor `s` é desbloqueado ao receber a mensagem de ACK, retornando *true* para significar que a operação foi realizada com sucesso, ou seja, foi realizado o envio confiável da mensagem m_1 de `s` para `r`.

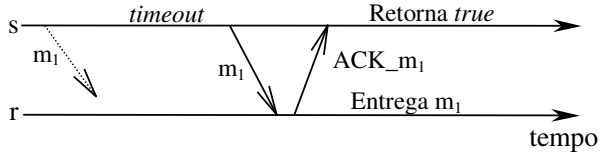

FIGURA 4.5 - *Unicast* confiável na ocorrência de perda de mensagem

Um caso quase idêntico ao anterior é ilustrado na figura 4.6. Neste caso, o ACK chega em `s` após expirar o controle de tempo, que já havia provocado o reenvio de m_1. Com isto, o receptor `r` recebe m_1 duas vezes; ao detectar que o número de seqüência da mensagem é menor do que o esperado, `r` verifica que é uma mensagem duplicada e não envia o ACK para `s` novamente e nem entrega m_1, já que o fez anteriormente.

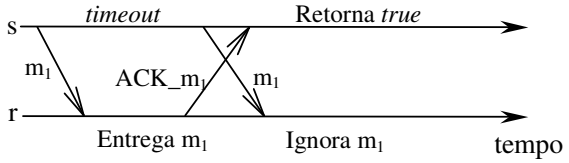

FIGURA 4.6 - *Unicast* confiável na ocorrência de duplicação de mensagem

Por fim, uma execução com a ocorrência de defeito no receptor é ilustrada na figura 4.7. Neste caso, o cenário é iniciado novamente com o emissor `s` enviando a mensagem m_1 para o receptor `r`. O controle de tempo no emissor `s` expira, e m_1 é reenviada. Se após um número máximo predefinido de reenvios (denotado por *n*) ainda não for recebido o ACK do receptor `r`, o emissor `s` suspeita de que `r` está falho e retorna *false* indicando que a operação não pode ser realizada, ou seja, não foi possível enviar a mensagem m_1 para *r*.

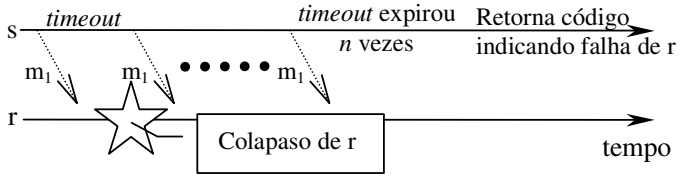

FIGURA 4.7 - *Unicast* confiável na ocorrência de defeito no receptor

4.3 *Multicast* Não-Confiável

Este tipo de comunicação é utilizado para difundir uma mensagem, ou seja, quando um determinado nodo deseja enviar uma mensagem para mais do que um destinatário. Além disso, este tipo de envio é utilizado quando não se necessita de confiabilidade, mas sim de rapidez na comunicação. Se há suporte de *hardware*, este tipo de comunicação pode tirar proveito e ser implementado através de *multicast* IP, senão utiliza-se o envio de vários *unicasts*. A seguir, são apresentados os algoritmos correspondentes a estas duas modalidades de implementação de *multicast* não-confiável.

No algoritmo para comunicação *multicast* não-confiável utilizando primitivas do tipo *unicast* (figura 4.8), `s` é o emissor da mensagem e `R` é o conjunto de receptores (*r* representa cada receptor). No nodo emissor, o envio da mensagem *msg* é feito através do método **`MulticastUc`**. Este método utiliza uma primitiva fornecida pela camada de transporte (*UDP*) para enviar a mensagem. Como a primitiva é do tipo *unicast*, é preciso enviar *n* mensagens, sendo que *n* representa a quantidade de receptores pertencentes a *R*; além disso, a informação sobre quem é o emissor é anexada a cada mensagem enviada.

```
No emissor s
MulticastUc(R, msg):
       FAÇA envio de msg n vezes (n é o n° de receptores)
              Send(s, msg) para r ∈ R;


No receptor r
       Receive(s, msg);
       Deliver(s, msg);
```

FIGURA 4.8 - Algoritmo para comunicação *multicast* não-confiável via múltiplos *unicasts*

Em cada nodo receptor, a mensagem é recebida através do método **`Receive`**, que utiliza uma primitiva para o recebimento de mensagens fornecida pela camada de transporte (*UDP*). O emissor da mensagem é identificado de acordo com o campo correspondente na mensagem recebida e, já que este protocolo não exige confiabilidade, a mensagem está pronta para ser entregue à camada posterior, que pode ser a aplicação ou um outro protocolo.

O funcionamento deste algoritmo simples é ilustrado através da figura 4.9. O emissor s envia a mensagem m_1 para os receptores r_1, r_2 e r_3, pertencentes a R. Ao receber m_1, cada receptor identifica o emissor e entrega a mensagem para a aplicação. Observa-se que, como o protocolo não exige confiabilidade e nem ordenação, não é preciso garantir que todos os receptores recebam a mensagem, nem ordenação entre mensagens sucessivas.

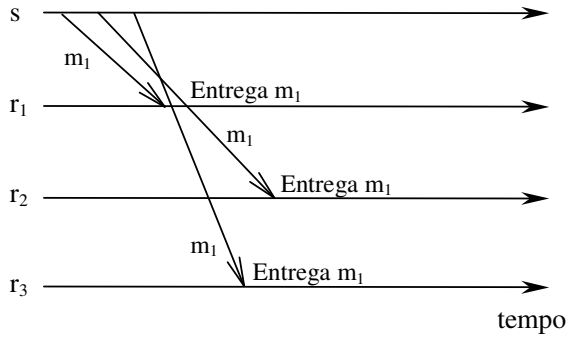

FIGURA 4.9 - *Multicast* não-confiável utilizando *unicasts*

A seguir, é descrito o algoritmo (figura 4.10) utilizado para o envio de mensagens na modalidade *multicast* não-confiável que tira proveito da existência de uma primitiva de *multicast* fornecida pela camada inferior.

```
No emissor s
Multicast(R, msg):
       Multicast(s, msg) para grupo R;


No receptor r
       Receive(s, msg);
       Deliver(s, msg);
```

FIGURA 4.10 - Algoritmo para comunicação *multicast* não-confiável com *multicast*

Neste algoritmo, `s` é o emissor da mensagem e `R` é o grupo de receptores. No nodo emissor, o envio da mensagem *msg* é feito através do método **`Multicast`**. Este método utiliza uma primitiva *multicast* fornecida pela camada inferior. Para que esta primitiva funcione, é necessário que cada receptor *r* pertença ao grupo identificado por *R*; os grupos são formados a partir de endereços IP na faixa que varia de 224.0.0.0 a 239.255.255.255 (seção 2.4). A informação sobre quem é o emissor é anexada a cada mensagem enviada para que cada receptor possa utilizá-la, se for necessário.

Em cada nodo receptor, a mensagem é recebida através do método **`Receive`**, que utiliza uma primitiva para o recebimento de mensagens fornecida pela camada de transporte (*UDP*). O emissor da mensagem é identificado de acordo com o campo correspondente na mensagem recebida e, já que este protocolo não exige confiabilidade, a mensagem está pronta para ser entregue à camada superior, que pode ser a aplicação ou um outro protocolo.

O funcionamento deste algoritmo para o envio na modalidade *multicast* não-confiável é ilustrado pela figura 4.11. O emissor `s` envia a mensagem m_1 para o conjunto de receptores (r_1, r_2 e r_3). Ao receber m_1, cada receptor identifica o emissor e entrega a mensagem para a aplicação. Observa-se que, como o protocolo não exige confiabilidade e nem ordenação, não é preciso garantir que todos os receptores recebam a mensagem, nem que mensagens sucessivas assegurem alguma propriedade de ordenação.

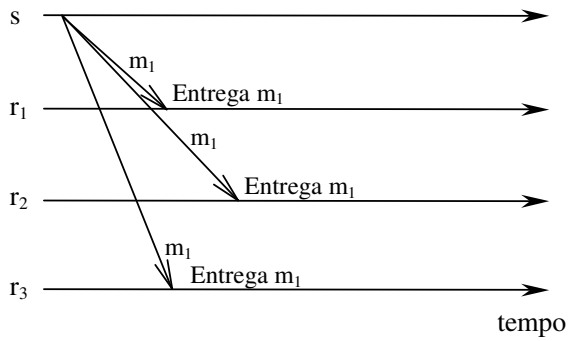

FIGURA 4.11 - *Multicast* não-confiável utilizando *multicast*

4.4 *Multicast* Confiável

Informalmente, um protocolo para *multicast* confiável garante que toda mensagem enviada por um nodo emissor é entregue aos nodos destinatários no máximo uma vez; além disso, durante a transmissão é comum se ter como suposição que o emissor e os receptores são corretos e que o canal de comunicação que os conecta não apresenta defeito durante a comunicação. Observa-se que esta última suposição restringe a aplicabilidade do protocolo.

De maneira mais geral, a definição de um protocolo para comunicação *multicast* confiável exige que as seguintes propriedades sejam satisfeitas [HAD94]:

- **Validade**: se um processo (objeto) correto envia uma mensagem *m*, então ele a entrega em um tempo finito (*eventually*);
- **Acordo**: se um processo (objeto) correto entrega uma mensagem *m*, então todos os processos (objetos) corretos entregam *m* em um tempo finito (*eventually*);
- **Integridade**: para qualquer mensagem *m*, cada processo (objeto) correto entrega *m* no máximo uma vez e somente se *m* foi transmitida anteriormente pelo emissor (*m*), que é um processo (objeto) correto.

Validade e **Acordo** garantem que uma mensagem transmitida (via *multicast*) por um processo (objeto) correto seja entregue por todos os processos (objetos) corretos. Observa-se que se o emissor da mensagem falhar, a especificação permite duas alternativas: a mensagem é entregue por todos os processos (objetos) corretos, ou não é entregue por nenhum deles (**atomicidade**). **Integridade** garante que cada nodo receptor só entrega uma vez para a camada superior a mensagem recebida.

Outra característica que pode ser interessante de estar presente em um protocolo de comunicação é a ordenação (seção 2.1). Em geral, cada mensagem possui um contexto, sem o qual pode ser mal interpretada. Um processo (objeto) não deveria entregar uma mensagem cujo contexto fosse desconhecido. Em algumas aplicações, o contexto de uma mensagem *m* é constituído pelas mensagens que foram transmitidas anteriormente pelo emissor de *m*. Por exemplo, em um sistema de reserva de passagens aéreas, o contexto de uma mensagem de cancelamento de uma reserva consiste da mensagem que estabeleceu aquela reserva anteriormente; a mensagem de cancelamento, desta forma, não deveria ser entregue em um nodo que ainda não tivesse recebido a mensagem de reserva.

De forma semelhante ao protocolo para *multicast* não-confiável, se há suporte de *hardware*, o *multicast* confiável pode tirar proveito e ser implementado através de *multicast* IP, senão utiliza-se o envio de vários *unicasts*. A seguir, são apresentados os algoritmos correspondentes a estas duas modalidades de implementação de *multicast* confiável. A implementação foi feita através da utilização do enfoque baseado em redundância (seção 3.7.1). A eleição deste enfoque tem como base a simplicidade de sua implementação, em decorrência do objetivo principal da dissertação voltar-se à definição de classes básicas para comunicação.

4.4.1 *Multicast* Confiável com Múltiplos *Unicasts*

No algoritmo para *multicast* confiável utilizando *unicast* como base (figura 4.12), `s` é o emissor da mensagem e `R` é o conjunto de receptores (*r* representa cada receptor). No nodo emissor, o envio da mensagem *msg* é feito através do método **`ReliableMulticastUc`**. Este método utiliza uma primitiva fornecida pela camada de transporte (*UDP*) para enviar a mensagem. Como a primitiva utilizada é do tipo *unicast*, é preciso enviar *n* mensagens, sendo que *n* representa a quantidade de receptores pertencentes a *R*.

Além da confiabilidade, deseja-se que o protocolo proposto também apresente ordenação FIFO e, conforme mencionado anteriormente, para fornecer esta propriedade é preciso identificar univocamente cada mensagem enviada. Isto é feito anexando a cada

mensagem enviada a informação sobre quem é o emissor, bem como um número de seqüência.

Em cada nodo receptor, a mensagem é recebida através do método **`Receive`**, que utiliza uma primitiva para o recebimento de mensagens, fornecida pela camada de transporte (*UDP*). O emissor da mensagem é identificado de acordo com o campo correspondente na mensagem recebida. Há três situações que podem ocorrer com respeito ao número de seqüência da mensagem: ele é igual ao esperado, ele é maior, ou ele é menor.

Se o número de seqüência da mensagem recebida é igual ao próximo número de seqüência, portanto corresponde ao esperado para aquele emissor, a mensagem então é reenviada para os demais receptores; se ele for maior do que o esperado, a mensagem é inserida em um *buffer*; e, por fim, se ele for menor, significa que a mensagem em questão já foi recebida, logo ela é descartada. Após reenviar a mensagem, cada receptor atualiza o número de seqüência relativo ao emissor da mensagem e entrega a mensagem à camada superior. O enfoque de confiabilidade utilizado, portanto, é o baseado em redundância (seção 3.7.1).

```
No emissor s
ReliableMulticastUc(R, msg):
       mcastSendSeqNo_r ← mcastSendSeqNo_r + 1;
       FAÇA envio de msg n vezes (n é o n° de receptores)
              Send(s, mcastSendSeqNo_r, msg) para r ∈ R;
       Retorna true;

No receptor r
       Receive(s, seqNo, msg):
       Se seqNo = mcastRecvSeqNo_s + 1
         então
              FAÇA envio de msg n vezes (n é o n° de receptores)
                     Send(s, mcastSendSeqNo_r, msg) para r ∈ R;
              mcastRecvSeqNo_s ← mcastRecvSeqNo_s + 1;
              Deliver(s, msg);
         senão
              Se seqNo > mcastRecvSeqNo_s + 1
                então
                     McastBuf.insert(msg);//insere a mensagem no buffer
              fim_Se
       fim_Se
```

FIGURA 4.12 - Algoritmo para *multicast* confiável utilizando *unicast*

Uma execução normal deste algoritmo, sem a ocorrência de defeitos, é ilustrada através da figura 4.13. O emissor s envia a mensagem m_1 para o conjunto de receptores (r_1, r_2 e r_3). Ao receber a mensagem m_1, cada receptor identifica o emissor e verifica se o número de seqüência contido na mensagem recebida corresponde ao que ele estava esperando. Após reenviar a mensagem aos demais receptores, a mensagem é entregue à camada superior.

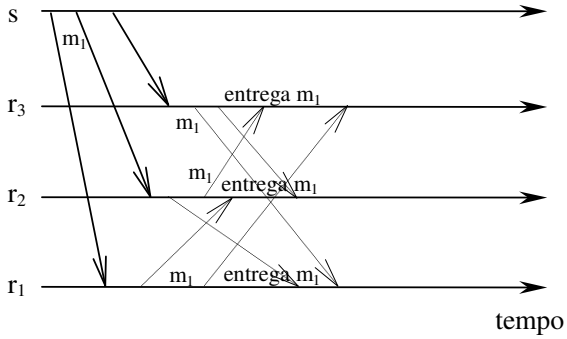

FIGURA 4.13 - *Multicast* confiável (*unicast*) sem defeito

Este protocolo utiliza o enfoque baseado em redundância (seção 3.7.1) para obter confiabilidade, ou seja, a retransmissão da mensagem por parte de cada receptor garante que todos os destinatários receberão a mensagem, ainda que o emissor apresente defeito enquanto estiver enviando a mensagem. Por exemplo, o emissor s deseja enviar a mensagem m_1 para o conjunto de receptores (r_1, r_2 e r_3), mas apresenta uma falha de colapso após enviar a mensagem para r_1 e r_2 e antes de enviar para r_3. Mesmo nesta situação, r_3 receberá a mensagem m_1, pois ela será retransmitida pelos demais receptores (figura 4.14).

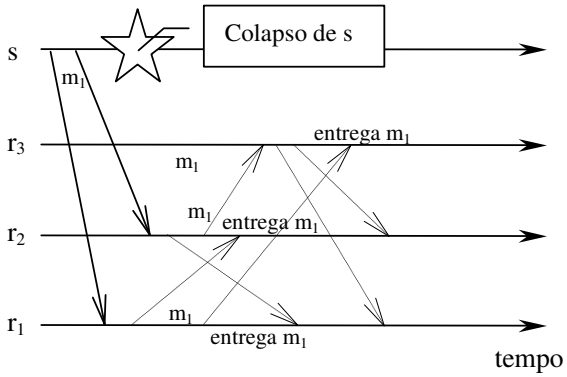

FIGURA 4.14 - *Multicast* confiável (*unicast*) com defeito do emissor

4.4.2 *Multicast* Confiável com Primitivas de *Multicast*

No algoritmo para *multicast* confiável baseado na disponibilidade de suporte de *hardware* para o *multicast* (figura 4.15), s é o emissor da mensagem e R é o conjunto de receptores (*r* representa cada receptor). No nodo emissor, o envio da mensagem *msg* é feito através do método **`ReliableMulticast`**.

Este método retira proveito do suporte de *hardware* para enviar as mensagens através de uma primitiva *multicast*. É preciso enviar apenas uma mensagem e esta será difundida para cada receptor *r* pertencente a *R*, ou seja, este caso difere do anterior porque a sobrecarga de envolvimento do emissor é menor.

```
No emissor s
ReliableMulticast(R, msg):
       mcastSendSeqNo_r ← mcastSendSeqNo_r + 1;
       Multicast(s, mcastSendSeqNo_r, msg) para R;
       Retorna true;

No receptor r
       Receive(s, seqNo, msg):
       Se seqNo = mcastRecvSeqNo_s + 1
         então
              Multicast(s, mcastSendSeqNo_r, msg) para R;
              mcastRecvSeqNo_s ← mcastRecvSeqNo_s + 1;
              Deliver(s, msg);
         senão
              Se seqNo > mcastRecvSeqNo_s + 1
                então
                     McastBuf.insert(msg);//insere a mensagem no buffer
              fim_Se
       fim_Se
```

FIGURA 4.15 - Algoritmo para *multicast* confiável utilizando suporte real de *multicast*

A confiabilidade é obtida também através do emprego do enfoque baseado em redundância. Além da confiabilidade, também se deseja que o protocolo proposto apresente ordenação FIFO: para garantir isto, a técnica utilizada é idêntica à apresentada pelo algoritmo que utiliza *unicast* (descrito anteriormente pela figura 4.12).

O papel representado por cada nodo receptor neste algoritmo também é idêntico ao representado pelo algoritmo que utiliza *unicast* como método de envio. O único aspecto que os diferencia é observado na retransmissão das mensagens. Neste caso, não é preciso enviar uma mensagem via *unicast* para cada receptor, pois há uma primitiva de *multicast* disponível.

Na figura 4.16, um exemplo de execução do algoritmo onde não há defeitos ilustra o mecanismo explicado.

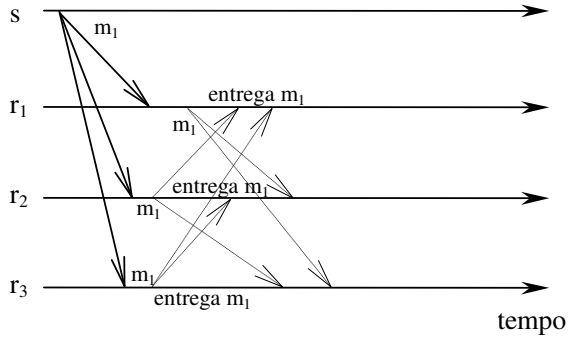

FIGURA 4.16 - *Multicast* confiável (*multicast*) sem defeito

Observa-se que, neste caso, o número de mensagens que trafegam na rede é bem menor do que no protocolo que utiliza múltiplos *unicasts*. A seguir, um exemplo apresentado na figura 4.17 ilustra um caso onde um dos receptores apresenta defeito. Observa-se que, mesmo assim, as propriedades para obtenção de confiabilidade e ordenação são garantidas, pois a mensagem é recebida por todos os receptores corretos.

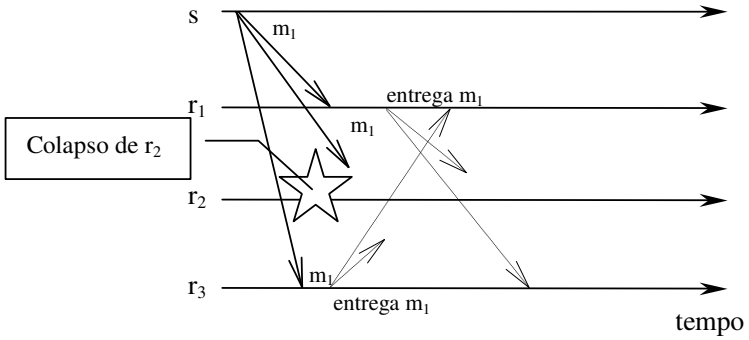

FIGURA 4.17 - *Multicast* confiável com defeito em um dos receptores

Neste caso, o receptor r_2 apresenta o defeito de colapso antes de receber a mensagem m_1, mas isto não prejudica o recebimento da mensagem pelos demais nodos. Desta forma, observa-se que, mesmo na ocorrência de defeito de colapso, o algoritmo garante a recepção da mensagem por parte dos nodos que estiverem corretos.

5 Classes para Comunicação

Já foi observado que a disponibilidade de serviços de comunicação facilita a tarefa dos programadores tanto na construção de aplicações distribuídas, como na construção de outros serviços utilizando estes como base. Também constata-se que serviços de comunicação básicos desejáveis são os que fornecem comunicação ponto-a-ponto e multiponto, com a possibilidade de escolha de envio confiável ou não-confiável. Neste sentido, este capítulo tem como objetivo a descrição das classes propostas para a implementação dos seguintes serviços de comunicação: *unicast* não-confiável, *multicast* não-confiável, *unicast* confiável e *multicast* confiável. Estas classes foram inicialmente modeladas através da **Linguagem de Modelagem Unificada** (UML) e após implementadas na linguagem orientada a objetos Java (pacote *jdk 1.2*).

A seguir, o diagrama das classes definidas é apresentado na seção 5.1 e nas seções subseqüentes do capítulo é feita uma descrição do conteúdo de cada uma destas classes. Nesta descrição, são mostrados trechos de código que exemplificam a utilização das classes propostas. Como o objetivo principal da dissertação está relacionado à comunicação, a ênfase maior está na descrição das classes para *unicast* e *multicast*, nas modalidades de envio não-confiável e confiável.

5.1 Modelagem em UML

No desenvolvimento de novos sistemas utilizando a orientação a objetos nas fases de análise de requisitos, análise de sistemas e de projeto, é necessária uma notação padronizada e realmente eficaz que abranja qualquer tipo de aplicação que se deseje. Neste contexto, no sentido de unificar uma série de padrões de modelagem orientada a objetos existentes é que surgiu a linguagem UML.

Quando a UML foi lançada, muitos desenvolvedores da área da orientação a objetos ficaram entusiasmados, já que a padronização proposta era o tipo de impulso que eles sempre esperaram. Existiam várias metodologias de modelagem orientada a objetos que, até o surgimento da UML, causavam uma guerra entre os integrantes da comunidade de desenvolvedores. A UML acabou com esta guerra, trazendo as melhores idéias de cada uma destas metodologias, e mostrando como deveria ser a migração das demais para a UML. A linguagem UML é, portanto, uma tentativa de padronizar a modelagem orientada a objetos de uma forma que qualquer sistema, seja qual for o tipo, possa ser modelado corretamente. Ela visa a criação de modelos consistentes, e ao mesmo tempo simples de serem atualizados e compreensíveis [ERI98].

A UML aborda o caráter estático e dinâmico do sistema a ser analisado, levando em consideração, já no período de modelagem, todas as futuras características do sistema em relação a utilização de *packages* (pacotes) próprios da linguagem a ser utilizada, bem como as diversas especificações do sistema a ser desenvolvido de acordo com as métricas finais do sistema.

As classes propostas por esta dissertação foram modeladas tendo por base o padrão proposto pela linguagem UML. Este padrão de modelagem foi escolhido tendo em vista que a UML está destinada a ser dominante, no sentido de ser a linguagem de modelagem padrão a ser utilizada nas empresas. Além disso, fatores de peso demonstram que a linguagem UML veio para ficar. Ela está totalmente baseada em conceitos e padrões extensivamente testados, os quais são provenientes das metodologias existentes anteriormente, além de apresentar boa documentação. A figura 5.1 ilustra o modelo das classes propostas por esta dissertação.

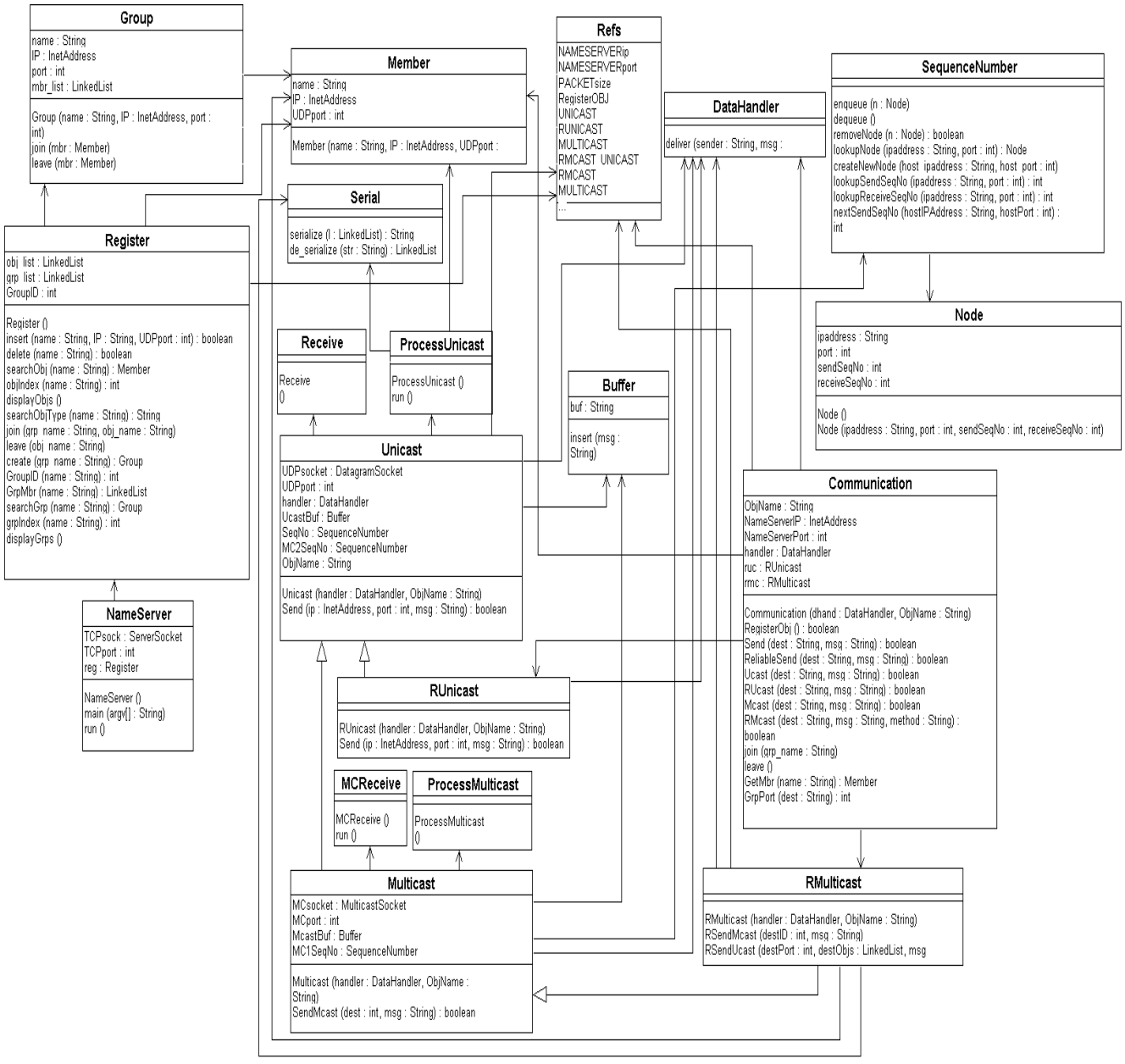

FIGURA 5.1 – Diagrama das classes propostas

5.2 Descrição das Classes Propostas

As próximas seções descrevem, individualmente, cada uma das classes presentes no diagrama ilustrado pela figura 5.1. Nesse diagrama, as classes **`DataHandler`**, **`Refs`**, **`SequenceNumber`**, **`Buffer`**, **`Serial`**, **`Unicast`**, **`RUnicast`**, **`Multicast`**, **`RMulticast`** e **`Refs`** são as principais, pois compõem e implementam os serviços de comunicação.

Observa-se que há herança entre **`RUnicast`** e **`Unicast`**, entre **`Multicast`** e **`Unicast`**, e entre **`RMulticast`** e **`Multicast`**. **`Receive`** e **`ProcessUnicast`** são classes que contém *threads* para dar apoio à classe **`Unicast`**; as classes **`MCReceive`** e **`ProcessMulticast`** possuem funcionalidade equivalente, fornecendo apoio à classe **`Multicast`**.

A classe **`Refs`** contém as constantes utilizadas pelas outras classes. **`Buffer`**, como o próprio nome indica, fornece a funcionalidade de um *buffer* e é utilizado pelas classes **`Unicast`** e **`Multicast`**. A classe **`Node`** é utilizada internamente pela classe **`SequenceNumber`**, que fornece às demais classes a implementação de números de identificação; estes números são úteis para identificar de forma única cada mensagem enviada. A classe **`Serial`**, utilizada pelas classes **`Unicast`** e **`RMulticast`**, fornece serialização de objetos. **`DataHandler`** fornece uma interface com a qual é possível implementar um mecanismo de *callback* para as mensagens.

As classes restantes **`Member`**, **`Group`**, **`Register`**, **`Communication`** e **`NameServer`** são utilizadas como apoio à implementação de um servidor de nomes, ou seja, com elas é possível nomear e armazenar referências a objetos, e a grupos, além de endereçar mensagens a estes nomes.

5.2.1 Interface DataHandler

A interface **`DataHandler`** (figura 5.2) é utilizada para a implementação de um mecanismo de *callback* entre a camada de comunicação e a camada superior (neste caso em particular, a aplicação).

```
1 public interface DataHandler {
2  public void deliver(String sender,String msg);
3 }
```

FIGURA 5.2 - Interface *DataHandler*

Para que o mecanismo de *callback* funcione, ou seja, para que a camada de comunicação possa chamar o método **`deliver`** (figura 5.3, linha 6) na aplicação, esta deve implementar a interface **`DataHandler`** e passar o seu endereço para a camada de comunicação, conforme aparece nesta aplicação exemplo.

```
 1 public class server implements DataHandler {
 2  public static void main(String argv[]) throws java.io.IOException {
 3   Communication server = new Communication(new server(),"server");
 4  }
 5
 6  public void deliver(String sender,String msg) {
 7   System.out.println("Delivered message: " + msg);
 8   System.out.println("Sender: " + sender);
 9  }
10 }
```

FIGURA 5.3 - Exemplo de uso de *DataHandler*

Neste exemplo, a aplicação **server** implementa a interface **DataHandler** (linha 1) e passa o endereço do objeto **server** para a camada de comunicação (linha 3): **new server()**. Com isto, quando uma mensagem está pronta para ser entregue, a camada de comunicação (neste caso representada por **Communication**) chama o método **deliver**; portanto, através da interface **DataHandler** foi possível implementar o mecanismo de *callback*.

5.2.2 Classe Refs

A classe **Refs** contém constantes que são utilizadas pelas demais classes (figura 5.4).

```
 1 public class Refs {
 2  public static final String NAMESERVERip = "227.228.229.230";
 3  public static final int    NAMESERVERport = 12345;
 4  public static final String GROUPSip = "230.230.230.230";
 5  public static final int    PACKETsize = 8192;
 6  public static final String RegisterOBJ = "Register_Object";
 7  public static final String SEARCH_OBJ = "Search_for_Object";
 8  public static final String SEARCH_GRP = "Search_for_Group";
 9  public static final String SEARCH_GRP_MBR = "Search_for_Group_Objects";
10  public static final String SEARCH_SEND = "Search_Dest_of_Send";
11  public static final String UNICAST = "Unreliable_Unicast_Message";
12  public static final String RUNICAST = "Reliable_Unicast_Message";
13  public static final String MULTICAST = "Unreliable_Multicast_Message";
14  public static final String RMCAST_MULTICAST = "multicast";
15  public static final String RMCAST_UNICAST = "unicast";
16  public static final String JOIN = "Join_to_Group";
17  public static final String LEAVE = "Leave_Group";
18  public static final int    RESEND_LIMIT = 5;
19  public static final int    TIMEOUT = 5000;
20  public static final String ACK = "Reliable_Unicast_Message_ACK";
21  public static final String OBJ_NOT_EXISTS = "Object_Name_Not_Exists";
22  public static final String GRP_NOT_EXISTS = "Group_Name_Not_Exists";
23  public static final String DELIM = "Delimitador";
24
25  static public InetAddress IP(String address) {
26    return(InetAddress.getByName(address));
27  }
28 }
```

FIGURA 5.4 - Classe Refs

A seguir, estão listadas todas as constantes e respectivas descrições do seu significado de utilização:

- **`NAMESERVERip`**: armazena o endereço de *multicast* IP do servidor de nomes*;*
- **`NAMESERVERport`**: armazena a porta em que o servidor de nomes está conectado*;*
- **`GROUPSip`**: armazena o endereço de *multicast* IP utilizado pelos grupos*;*
- **`PACKETsize`**: indica o tamanho do pacote utilizado na composição das mensagens*;*
- **`RegisterOBJ`**: utilizado na composição de mensagens de registro de objetos no servidor de nomes*;*
- **`SEARCH_OBJ`**: utilizado na composição de mensagens que visam a pesquisa de referência de objetos no servidor de nomes*;*
- **`SEARCH_GRP`**: utilizado na composição de mensagens que visam a pesquisa de referência de grupos no servidor de nomes*;*
- **`SEARCH_GRP_MBR`**: utilizado na composição de mensagens que visam a pesquisa de referência de lista de objetos que pertencem a determinado grupo no servidor de nomes*;*
- **`SEARCH_SEND`**: utilizado na composição de mensagens que visam a pesquisa no servidor de nomes para descobrir se determinado parâmetro refere-se a um objeto ou a um grupo*;*
- **`UNICAST`**: utilizado na composição de mensagens do tipo *unicast* não-confiável*;*
- **`RUNICAST`**: utilizado na composição de mensagens do tipo *unicast* confiável*;*
- **`MULTICAST`**: utilizado na composição de mensagens do tipo *multicast* não-confiável*;*
- **`RMCAST_MULTICAST`**: utilizado na composição de mensagens do tipo *multicast* confiável implementado através de *multicast* IP*;*
- **`RMCAST_UNICAST`**: utilizado na composição de mensagens do tipo *multicast* confiável implementado através de vários *unicasts;*
- **`JOIN`**: utilizado na composição de mensagens para executar a operação *join*, ou seja, inserir determinado objeto em um grupo*;*
- **`LEAVE`**: utilizado na composição de mensagens para executar a operação *leave*, ou seja, fazer com que um objeto deixe de pertencer a determinado grupo*;*
- **`RESEND_LIMIT`**: estabelece o limite de reenvio de mensagens, ou seja, um número de vezes máximo que uma mensagem será retransmitida*;*
- **`TIMEOUT`**: estabelece o valor, em milissegundos, adotado para controle de tempo, ou seja, um tempo limite máximo para uma determinada operação*;*
- **`ACK`**: utilizado na composição de mensagens de reconhecimento positivo (ACK)*;*
- **`OBJ_NOT_EXISTS`**: utilizado para indicar que um determinado objeto não existe*;*
- **`GRP_NOT_EXISTS`**: utilizado para indicar que um determinado grupo não existe*;*
- **`DELIM`**: utilizado na composição de mensagens do tipo *multicast* confiável implementado através de vários envios do tipo *unicast.*

Há ainda na classe **`Refs`** o método **`IP`**, que é utilizado para converter o parâmetro do tipo **`String`**, que lhe é passado, para o tipo **`InetAddress`**, que compõe o endereço IP de cada objeto.

5.2.3 Classe Buffer

A classe **Buffer** (figura 5.5) é utilizada para o armazenamento das mensagens recebidas em ambas as modalidades, *unicast* e *multicast*. Para realizar este armazenamento, utiliza a lista encadeada **buf** (linha 2) em conjunto com os métodos **insert** (linha 5) e **remove** (linha 9).

```
 1 public class Buffer {
 2  LinkedList buf;
 3  ...
 4
 5  public synchronized void insert(String msg) {
 6   ...
 7  }
 8
 9  public synchronized String remove() {
10   ...
11  }
12 }
```

FIGURA 5.5 - Classe Buffer

A palavra chave **synchronized** garante que a operação é realizada apenas por um objeto a cada vez. O método **insert** recebe como parâmetro a mensagem a ser armazenada (formato **String**) no *buffer*, enquanto o método **remove** retira uma mensagem armazenada (formato **String**) do *buffer*.

5.2.4 Classe Node

A classe **Node** (figura 5.6) é utilizada para armazenar as informações relativas às mensagens enviadas e recebidas por cada nodo do sistema distribuído. Ela armazena o endereço IP do nodo em **ipaddress** (linha 2), a porta utilizada para o recebimento de mensagens em **port** (linha 3), o número de seqüência da última mensagem enviada em **sendSeqNo** (linha 4) e o número de seqüência da última mensagem recebida em **receiveSeqNo** (linha 5).

```
 1 class Node {
 2  String ipaddress;
 3  int port;
 4  int sendSeqNo;
 5  int receiveSeqNo;
 6
 7  Node(String ipaddress,int port,int sendSeqNo,int receiveSeqNo) {
 8   ...
 9  }
10
11  Node() {
12   ...
13  }
14 }
```

FIGURA 5.6 - Classe Node

As informações armazenadas pela classe **Node** são utilizadas pela classe **SequenceNumber**, descrita a seguir.

5.2.5 Classe SequenceNumber

A classe **SequenceNumber** (figura 5.7) contém métodos para numerar seqüencialmente as mensagens enviadas e recebidas relativas a cada objeto da classe **Node**. O método **enqueue** (linha 3) enfileira um objeto da classe **Node** e o método **dequeue** (linha 7) desenfileira; estes métodos são utilizados internamente pela classe **SequenceNumber**.

Para remover uma referência **n** passada como parâmetro da lista que armazena os nodos, é utilizado o método **removeNode** (linha 11), que retorna um valor *booleano* informando se a operação foi ou não foi realizada com sucesso; este método é utilizado internamente pela classe **SequenceNumber**.

O método **lookupNode** (linha 15) retorna o nodo, que está na lista de nodos armazenados, correspondente ao endereço IP (**ipaddress**) e à porta (**port**) passados como parâmetros do método.

```
 1 class SequenceNumber {
 2
 3  synchronized void enqueue(Node n) {
 4   ...
 5  }
 6
 7  synchronized void dequeue() {
 8   ...
 9  }
10
11  synchronized boolean removeNode(Node n){
12   ...
13  }
14
15  synchronized Node lookupNode(String ipaddress,int port) {
16   ...
17  }
18
19  synchronized void createNewNode(String host_ipaddress,
20                                   int host_port){
21   ...
22  }
23
24  synchronized int lookupSendSeqNo(String ipaddress,int port) {
25   ...
26  }
27
28  synchronized int lookupReceiveSeqNo(String ipaddress,int port) {
29   ...
30  }
31
32  synchronized int nextSendSeqNo(String hostIPAddress,int hostPort) {
33   ...
34  }
35
36  synchronized int nextReceiveSeqNo(String hostIPAddress,int hostPort) {
37   ...
38  }
39 }
```

FIGURA 5.7 - Classe SequenceNumber

A criação de um novo nodo é feita através do método **createNewNode** (linha 19), que também o insere na lista de nodos. Este método é utilizado internamente pela classe **SequenceNumber**.

O método **lookupSendSeqNo** (linha 24) pesquisa pelo número de seqüência relativo à última mensagem enviada pelo **ipaddress/port** passado como parâmetro. Já o método **lookupReceiveSeqNo** (linha 28) pesquisa pelo número de seqüência relativo à última mensagem recebida de **ipaddress/port** passado como parâmetro.

O método **nextSendSeqNo** (linha 32) retorna o número de seqüência relativo à próxima mensagem a ser enviada pelo **ipaddress/port** passado como parâmetro. O método **nextReceiveSeqNo** (linha 36), por sua vez, retorna o número de seqüência relativo à última mensagem recebida de **ipaddress/port** passado como parâmetro.

5.2.6 Classe Member

A classe **Member** (figura 5.8) armazena três informações relativas a cada objeto no sistema: o nome do objeto - **name** (linha 2), o endereço IP - **IP** (linha 3) e a porta de comunicação associada - **UDPport** (linha 4); estas variáveis são inicializadas no construtor (linha 6).

Estas informações servem para identificar univocamente os objetos existentes durante as operações realizadas envolvendo objetos.

```
 1 public class Member implements java.io.Serializable {
 2  public String name;
 3  public String IP;
 4  public int UDPport;
 5
 6  public Member(String name,String IP,int UDPport) {
 7   this.name = name;
 8   this.IP = IP;
 9   this.UDPport = UDPport;
10  }
11 }
```

FIGURA 5.8 - Classe Member

5.2.7 Classe Group

A classe **Group** (figura 5.9), similar à classe **Member**, armazena informações relativas a cada objeto no sistema, só que neste caso os objetos são grupos. Ela armazena: o nome do grupo - **name** (linha 2), o endereço de *multicast* IP - **IP** (linha 3), a porta do grupo - **port** (linha 4) e a lista de membros pertencentes ao grupo - **mbr_list** (linha 5); estas variáveis são inicializadas no construtor (linha 7).

A inserção de um objeto em um grupo é tarefa do método **join** (linha 11), que insere um objeto **Member mbr**, passado como parâmetro, no grupo em questão. O método **leave** (linha 15) remove do grupo o objeto **Member mbr**, passado como

parâmetro. O método **index** (linha 19) retorna o índice relativo à **String** (que representa o nome do objeto) passada como parâmetro, na lista de objetos pertencentes ao grupo; este método é utilizado internamente pela classe.

```
 1 public class Group implements java.io.Serializable {
 2  String name;
 3  InetAddress IP;
 4  int port;
 5  LinkedList mbr_list;
 6
 7  public Group(String name,InetAddress IP,int port) {
 8   ...
 9  }
10
11  public synchronized void join(Member mbr) {
12   ...
13  }
14
15  public synchronized void leave(Member mbr) {
16   ...
17  }
18
19  public synchronized int index(String name) {
20   ...
21  }
22 }
```

FIGURA 5.9 - Classe Group

5.2.8 Classe Serial

A classe **Serial** (figura 5.10) é utilizada para a execução de operações que visam a serialização de objetos (**serializar** e **de-serializar**). A serialização de objetos é utilizada para transformar os objetos em tipos de dados que possam ser compreendidos e manipulados por outros métodos.

Neste caso específico da classe **Serial**, busca-se a serialização dos objetos de uma lista encadeada **LinkedList**, que representa a lista dos objetos pertencentes a um determinado grupo. Como todas as mensagens são do tipo **String**, é necessário serializar **LinkedList** para **String**, o que é feito pelo método **serialize** (linha 3). O inverso, de-serializar, é feito pelo método **de-serialize** (linha 7).

```
1 public class Serial {
2
3  public String serialize (LinkedList l) {
4   ...
5  }
6
7  public LinkedList de_serialize(String str) {
8   ...
9  }
```

FIGURA 5.10 - Classe Serial

5.2.9 Classe Unicast

A classe **Unicast** (figura 5.11) fornece um serviço de envio ponto-a-ponto não-confiável de mensagens. Para que as mensagens possam ser enviadas e recebidas, a classe contém um *socket* (**UDPsocket**), linha 2, e uma porta (**UDPport**), linha 3. Para transmitir a mensagem para as camadas superiores (outros protocolos ou aplicação), a classe faz uso de um mecanismo de *callback*, definido pelo objeto classe **handler** da interface **DataHandler** (linha 4); quando uma mensagem está pronta para ser entregue à camada superior, o método *deliver* desta interface é chamado. As mensagens recebidas são armazenadas em um *buffer*, definido pelo objeto **UcastBuf** da classe **Buffer** (linha 5).

Uma das restrições impostas pela linguagem escolhida para a implementação das classes é a impossibilidade do uso de herança múltipla (Java não oferece este recurso). Pelo fato de se prever que herança múltipla seria de interesse, foi necessária uma maneira de contornar as restrições impostas por Java, por isso na classe **Unicast** há a declaração de variáveis que, na verdade, seriam utilizadas por outras classes. Este é o caso de dois objetos da classe **SequenceNumber** (linha 6): **SeqNo** e **MC2SeqNo**. Estes objetos são utilizados, respectivamente, para numerar seqüencialmente mensagens do tipo *unicast confiável* (classe **RUnicast**) e do tipo *multicast confiável* (*método unicast* – classe **RMulticast**).

Foi estabelecida uma variável **String – ObjName** (linha 7), visando armazenar o nome do objeto que instanciou a classe. O construtor (linha 9), por sua vez, recebe como parâmetros o **handler** e o nome do objeto que instanciou a classe **ObjName**. Após, inicializa as variáveis da classe, e cria as *threads* **Receive** (linha 17) e **ProcessUnicast** (linha 23).

O método **Send** (linha 13) é utilizado para o envio de mensagens do tipo ponto-a-ponto não-confiável; ele recebe como parâmetros o endereço IP de destino (**ip**), a porta (**port**) e a mensagem a ser enviada (**msg**).

A *thread* **Receive** (linha 17) é utilizada para o recebimento de mensagens ponto-a-ponto. As mensagens são recebidas e inseridas no *buffer* **UcastBuf** (linha 19). A *thread* **ProcessUnicast** (linha 23), por sua vez, retira mensagens do *buffer* **UcastBuf** (linha 25) e processa as mensagens dos seguintes tipos: *unicast não-confiável* (linha 27), *unicast confiável* (linha 29) e *multicast confiável* (*método unicast*) (linha 31).

Os testes condicionais presentes na classe **Unicast** que não têm relacionamento com mensagens ponto-a-ponto não-confiável só estão ali como forma de contornar o problema da impossibilidade de herança múltipla de Java.

Observa-se que a classe **Unicast** serve tanto para protocolos que visam o envio não confiável de mensagens ponto-a-ponto, quanto como base para o desenvolvimento de classes que visam protocolos para o envio de mensagens ponto-a-ponto confiável e *multicast* (não-confiável ou confiável), implementado através de mensagens *unicast*.

```
 1 public class Unicast {
 2  DatagramSocket UDPsocket;
 3  int            UDPport;
 4  DataHandler    handler;
 5  Buffer         UcastBuf;
 6  SequenceNumber SeqNo, MC2SeqNo;
 7  String         ObjName;
 8
 9  public Unicast(DataHandler handler,String ObjName)
10   ...
11  }
12
13  public boolean Send(InetAddress ip,int port,String msg) {
14   ...
15  }
16
17  public class Receive extends Thread {
18   ...
19   UcastBuf.insert(msg);
20   ...
21  }
22
23  public class ProcessUnicast extends Thread {
24   ...
25   message = UcastBuf.remove();
26   ...
27   if (msg_type.equals(Refs.UNICAST))
28   else
29   if (msg_type.equals(Refs.RUNICAST))
30   else
31   if (msg_type.equals(Refs.RMCAST_UNICAST))
32   ...
33  }
34 }
```

FIGURA 5.11 - Classe *Unicast*

5.2.10 Classe RUnicast

A classe **RUnicast** (figura 5.12) implementa um serviço de envio de mensagens ponto-a-ponto confiável. Ela estende a classe **Unicast** (linha 1), ou seja, herda características desta, e sobrescreve o método **Send** (linha 7) para implementar neste um algoritmo de envio confiável de mensagens, baseado em reconhecimento positivo, utilizando um mecanismo de controle do tempo na troca de mensagens e retransmissões.

```
 1  public class RUnicast extends Unicast {
 2
 3  public RUnicast(DataHandler handler,String ObjName) {
 4   ...
 5  }
 6
 7  public boolean Send(InetAddress ip,int port,String msg) {
 8   ...
 9  }
10 }
```

FIGURA 5.12 - Classe *RUnicast*

5.2.11 Classe Multicast

A classe **Multicast** (figura 5.13) fornece um serviço de envio multiponto não-confiável de mensagens. Para que as mensagens possam ser enviadas e recebidas, a classe contém um *socket* (**MCsocket**), linha 2, e uma porta (**MCport**), linha 3. As mensagens recebidas são armazenadas em um *buffer*, definido pelo objeto **McastBuf** da classe **Buffer** (linha 4). O objeto **MC1SeqNo** da classe **SequenceNumber** (linha 5) é utilizado para numerar seqüencialmente mensagens do tipo *multicast confiável* (*método multicast*).

Para transmitir a mensagem às camadas superiores (outros protocolos ou aplicação), a classe faz uso de um mecanismo de *callback*, definido pelo objeto **handler** da interface **DataHandler** que é passado como parâmetro para o construtor (linha 7). Além deste, o método construtor recebe como parâmetro o nome do objeto que instanciou a classe **ObjName** e inicializa as variáveis da classe.

```
 1 public class Multicast extends Unicast {
 2  MulticastSocket MCsocket;
 3  int             MCport;
 4  Buffer          McastBuf;
 5  SequenceNumber  MC1SeqNo;
 6
 7  public Multicast(DataHandler handler,String ObjName) {
 8   ...
 9  }
10
11  public boolean SendMcast(int dest,String msg) {
12   ...
13  }
14
15  public void join(int groupID) {
16   new MCReceive();
17   new ProcessMulticast();
18  }
19
20  public class MCReceive extends Thread {
21   ...
22   McastBuf.insert(message);
23   ...
24  }
25
26  public class ProcessMulticast extends Thread {
27   ...
28   message = McastBuf.remove();
29   ...
30    if (msg_type.equals(Refs.MULTICAST)) {
31    ...
32    }//if Refs.MULTICAST message
33    else
34    if (msg_type.equals(Refs.RMCAST_MULTICAST)) {
35    ...
36    }
37  }
38 }
```

FIGURA 5.13 - Classe *Multicast*

O método **SendMcast** (linha 11) é utilizado para o envio de mensagens do tipo *multicast* não-confiável; ele recebe como parâmetros o destino (**dest**), que representa o identificador do grupo, e a mensagem a ser enviada (**msg**).

O método **join** (linha 15) é utilizado para inserir o objeto que o chama no grupo identificado pelo parâmetro passado (**groupID**). Neste método também são inicializadas as *threads* **MCReceive** e **ProcessMulticast**.

A *thread* **MCReceive** (linha 20) é utilizada para o recebimento de mensagens do tipo *multicast*. As mensagens *multicast* são recebidas e inseridas em **McastBuf** (linha 22). A *thread* **ProcessMulticast** (linha 26) retira mensagens de **McastBuf** (linha 28) e processa as mensagens dos seguintes tipos: *multicast* não-confiável (linha 30), *multicast* confiável (método *multicast*) (linha 34).

Novamente convém observar que trechos de código presentes na classe **Multicast**, que não se relacionam ao tipo de comunicação *multicast* não-confiável, só estão ali como forma de contornar as restrições de herança impostas pela linguagem de programação adotada.

5.2.12 Classe RMulticast

A classe **RMulticast** (figura 5.14) implementa um serviço de envio de mensagens do tipo *multicast* confiável. Ela estende a classe **Multicast** (linha 1), ou seja, herda características desta. O envio confiável de mensagens pode ser feito através de dois métodos: **RSendUcast** (linha 7) que implementa um *multicast* confiável através do envio de mensagens do tipo *unicast*; e **RSendMcast** (linha 11) que implementa um *multicast* confiável através do envio de mensagens do tipo *multicast* IP.

O método **RSendUcast** recebe como parâmetros o identificador do grupo em **destPort**, a lista de objetos **Member** (**destObjs**) que pertencem ao grupo destinatário da mensagem e a mensagem em **msg**. O método **RSendMcast** recebe como parâmetros o identificador do grupo destinatário da mensagem em **destID** e a mensagem em **msg**.

```
 1 public class RMulticast extends Multicast {
 2
 3  public RMulticast(DataHandler handler,String ObjName) {
 4   ...
 5  }
 6
 7  public boolean RSendUcast(int destPort,LinkedList destObjs,String msg){
 8   ...
 9  }
10
11  public boolean RSendMcast(int destID,String msg) {
12   ...
13  }
14 }
```

FIGURA 5.14 - Classe *RMulticast*

5.2.13 Classe Register

A classe **Register** (figura 5.15) armazena informações relativas aos objetos e aos grupos do sistema. Os objetos são armazenados em uma lista, **obj_list** (linha 2), os grupos são também armazenados em uma lista, **grp_list** (linha 3); cada grupo criado recebe um identificador único, **GroupID** (linha 4). O construtor da classe (linha 6) cria as listas que armazenarão os objetos e os grupos.

A inserção de um determinado objeto na lista de objetos é feita pelo método **insert** (linha 10); este método recebe como parâmetros o nome (**name**), o endereço IP (**IP**) e a porta (**UDPport**) e retorna um valor *booleano* indicando o resultado da operação. O método **delete** (linha 14) remove um objeto da lista de objetos; recebe como parâmetro o nome (**name**) do objeto a ser removido e retorna um valor *booleano* indicando o resultado da operação.

Pesquisar por um determinado objeto na lista de objetos é tarefa do método **searchObj** (linha 18); ele recebe como parâmetro o nome (**name**) do objeto a ser pesquisado e retorna um objeto **Member**, que pode ser o objeto procurado, se este for encontrado, ou um objeto *null*.

O método **objIndex** (linha 22) pesquisa pelo índice de um objeto na lista de objetos; recebe como parâmetro o nome (**name**) do objeto a ser pesquisado e retorna o seu índice; ele é utilizado internamente pela classe **Register**. Se, em determinado instante, for desejada a exibição dos objetos que constam na lista de objetos armazenados por **Register**, utiliza-se o método **displayObjs** (linha 26). O método **searchObjType** (linha 30) verifica se um determinado nome, passado como parâmetro (**name**), refere-se a um objeto ou a um grupo; o valor de retorno indica a qual deles se refere.

O método **join** (linha 34) insere um objeto em um grupo; o nome do grupo (**grp_name**) e o nome do objeto (**obj_name**) são passados como parâmetro. Para um objeto do grupo em questão, utiliza-se o método **leave** (linha 38); este retira o objeto passado como parâmetro (**obj_name**) do grupo ao qual pertence. A criação de um grupo é feita pelo método **create** (linha 42), que cria um grupo com o nome (**grp_name**) passado como parâmetro.

É possível obter algumas informações da classe **Register** como, por exemplo: o identificador do nome do grupo – através do método **groupID** (linha 46); a lista de objetos de um grupo– através do método **GrpMbr** (linha 50); a referência a um grupo – através do método **searchGrp** (linha 54); e o índice de um grupo dentro da lista de grupos – através do método **grpIndex** (linha 58).

O método **displayGrps** (linha 62) pode ser utilizado para o caso de se querer exibir a lista de grupos armazenados por **Register**.

```
 1 public class Register {
 2  public LinkedList obj_list;
 3  public LinkedList grp_list;
 4  static int GroupID = 3456;
 5
 6  public Register() {
 7   ...
 8  }
 9
10  public synchronized boolean insert(String name,String IP,int UDPport) {
11   ...
12  }
13
14  public synchronized boolean delete(String name) {
15   ...
16  }
17
18  public synchronized Member searchObj(String name) {
19   ...
20  }
21
22  public synchronized int objIndex(String name) {
23   ...
24  }
25
26  public synchronized void displayObjs() {
27   ...
28  }
29
30  public synchronized String searchObjType(String name) {
31   ...
32  }
33
34  public synchronized void join(String grp_name,String obj_name) {
35   ...
36  }
37
38  public synchronized void leave(String obj_name) {
39   ...
40  }
41
42  public synchronized Group create(String grp_name) {
43   ...
44  }
45
46  public synchronized int GroupID(String name) {
47   ...
48  }
49
50  public synchronized LinkedList GrpMbr(String name) {
51   ...
52  }
53
54  public synchronized Group searchGrp(String name) {
55   ...
56  }
57
58  public synchronized int grpIndex(String name) {
59   ...
60  }
61
62  public synchronized void displayGrps() {
63   ...
64  }
65 }
```

FIGURA 5.15 - Classe Register

5.2.14 Classe Communication

A classe **`Communication`** (figura 5.16) fornece diversos métodos que servem como interface entre a aplicação do usuário, o servidor de nomes e as demais classes presentes no pacote, que foram descritas nas seções anteriores. Em síntese, a finalidade desta classe, em conjunto com o **servidor de nomes** (descrito posteriormente), é tornar mais fácil a utilização do pacote de classes de comunicação proposto por parte dos programadores. Observa-se que, se o programador achar que as abstrações de programação aqui propostas degradam o desempenho da sua aplicação ou que não lhe fornecem o controle de programação que gostaria de possuir, não há problema em utilizar as classes básicas diretamente. Ou seja, a utilização das classes anteriormente descritas não depende da classe **`Communication`**, nem do **servidor de nomes**, que será descrito posteriormente. Exemplos de programas com e sem a utilização da classe **`Communication`** são descritos no capítulo 6.

Esta classe procura armazenar informações que serão utilizadas posteriormente, tais como: o nome do objeto que instanciou a classe **`Communication`**, armazenado em **`ObjName`** (linha 2); o endereço IP do servidor de nomes, que é armazenado em **`NameServerIP`** (linha 3); a porta em que o servidor de nomes está associado, e à espera por mensagens, armazenado em **`NameServerPort`** (linha 4); o endereço do objeto que instanciou a classe, armazenado em **`handler`** (linha 5); e os objetos utilizados para o envio de mensagens do tipo *unicast* não-confiável, *unicast* confiável, *multicast* não-confiável e *multicast* confiável (linhas 6 e 7).

O construtor da classe (linha 9) recebe como parâmetros o endereço do objeto que instanciou a classe (**`dhand`**) e o nome deste objeto (**`ObjName`**). No código pertencente ao construtor, há a instanciação dos objetos de algumas das demais classes do pacote de comunicação, bem como uma chamada ao método que faz o registro de **`ObjName`** no servidor de nomes. O método **`RegisterObj`** (linha 13) é quem registra um objeto, cujo nome é **`ObjName`**, no servidor de nomes.

Para enviar uma mensagem em modo não-confiável para o nome passado como parâmetro (**`dest`**), utiliza-se o método **`Send`** (linha 17). Como exposto anteriormente, o intuito da classe é o fornecimento de abstrações que facilitem o uso do pacote de classes para comunicação proposto.

Este fato pode ser comprovado através do próprio método **`Send`** pois, se na chamada do método o parâmetro refere-se a um objeto, então é feito um *unicast* não-confiável para este objeto; e se este parâmetro refere-se a um grupo, então é feito um *multicast* não-confiável para este grupo. O próprio método é quem decide se o envio da mensagem deve ser para um objeto ou para um grupo, ou seja, se deve ser feito um *unicast* ou um *multicast*.

O método **`ReliableSend`** (linha 21) envia uma mensagem em modo confiável para o nome passado como parâmetro (**`dest`**). Aqui neste método também foi implementada a mesma abstração de programação do método **`Send`**. Se o parâmetro passado ao método refere-se a um objeto, então é feito um *unicast* confiável para este objeto; se este parâmetro refere-se a um grupo, então é feito um *multicast* confiável para este grupo.

```
 1 public class Communication {
 2  String       ObjName;
 3  InetAddress  NameServerIP;
 4  int          NameServerPort;
 5  DataHandler  handler;
 6  RUnicast     ruc;
 7  RMulticast   rmc;
 8
 9  public Communication(DataHandler dhand,String ObjName) {
10   ...
11  }
12
13  public boolean RegisterObj() {
14   ...
15  }
16
17  public boolean Send(String dest,String msg) {
18   ...
19  }
20
21  public boolean ReliableSend(String dest,String msg) {
22   ...
23  }
24
25  public boolean Ucast(String dest,String msg) {
26   ...
27  }
28
29  public boolean RUcast(String dest,String msg) {
30   ...
31  }
32  public boolean Mcast(String dest,String msg) {
33   ...
34  }
35
36  public boolean RMcast(String dest,String msg,String method) {
37   ...
38  }
39
40  public void join(String grp_name) {
41   ...
42  }
43
44  public void leave() {
45   ...
46  }
47
48  public Member GetMbr(String name) {
49   ...
50  }
51
52  public int GrpPort(String dest) {
53   ...
54  }
55  public LinkedList GetObjList(String dest) {
56   ...
57  }
58 }
```

FIGURA 5.16 - Classe Communication

Caso o programador não deseje utilizar a abstração de programação anteriormente descrita, por achar que isto possa degradar um pouco o desempenho final da sua aplicação, existem ainda outras possibilidades.

Em decorrência disto, foram criados outros métodos para o envio de mensagens. O método **Ucast** (linha 25) envia uma mensagem do tipo *unicast* não-confiável para um nome passado como parâmetro **dest**. O método **RUcast** (linha 29) envia uma mensagem do tipo *unicast* confiável para um objeto cujo nome foi passado como parâmetro **dest**. O método **Mcast** (linha 32) envia uma mensagem do tipo *multicast* não-confiável para um grupo cujo nome foi passado como parâmetro **dest**.

O método **RMcast** (linha 36) envia uma mensagem do tipo *multicast* confiável para um grupo cujo nome foi passado como parâmetro **dest**, através do método especificado pelo parâmetro **method**. Convém observar que o método básico para o envio *multicast* confiável pode explorar uma dentre duas modalidades: através de vários *unicasts* ou através de vários *multicasts*.

O método **join** (linha 40) faz com que o objeto representado por **ObjName** seja inserido em **grp_name** passado como parâmetro. O método **leave** (linha 44) faz com que o objeto representado por **ObjName** seja removido do grupo ao qual pertence.

O método **GetMbr** (linha 48) retorna o objeto **Member** armazenado no servidor de nomes que representa o objeto **name** passado como parâmetro. O método **GrpPort** (linha 52) retorna a porta associada com o grupo **dest** passado como parâmetro. O método **GetObjList** (linha 55) retorna a lista de objetos pertencentes ao grupo **dest** passado como parâmetro.

5.2.15 Classe NameServer

A classe **NameServer** (FIGURA 5.17) fornece uma interface com a classe **Register**, através do objeto **reg** (linha 4). A classe associa nomes a objetos e pesquisa referências associadas a objetos e a grupos. O método **main** (linha 10) registra os objetos, e o método **run** (linha 16) pesquisa as referências associadas a grupos e a objetos.

```
 1 class NameServer extends Thread {
 2  ServerSocket TCPsock;
 3  int          TCPport;
 4  Register     reg;
 5
 6  public NameServer() {
 7   ...
 8  }
 9
10  public static void main(String argv[]) {
11   NameServer ns = new NameServer();
12   ns.start();
13   ...
14  }
15
16  public void run() {
17   ...
18  }
19 }
```

FIGURA 5.17 - Classe NameServer

6 Utilização das Classes

Neste capítulo, foram desenvolvidos diversos exemplos de utilização das classes descritas no capítulo 5. Inicialmente, são mostrados trechos de código onde o servidor de nomes não está presente, posteriormente são mostrados outros trechos de códigos que fazem uso dele.

Aproveita-se, também, para detalhar um pouco mais a implementação dos serviços de comunicação nas quatro modalidades propostas (*unicast* e *multicast*, em modo confiável e não-confiável). Estes serviços têm por base os algoritmos de comunicação descritos no capítulo 4.

Considerando que a dissertação não tem como ênfase a implementação de serviços de comunicação complexos, visando o tratamento de diversos modelos de defeitos, os serviços implementados são bem simples. O intuito é de que estes serviços, apesar de simples, possuam a funcionalidade suficiente para o modelo de defeitos a que se propõem e que sirvam como base para a construção de aplicações distribuídas, ou mesmo como base para a construção de outras classes que implementem outras funcionalidades e/ou serviços.

6.1 Envio em *Unicast* Não-Confiável

Esta modalidade de comunicação é utilizada quando se deseja enviar uma mensagem de um objeto para outro objeto, onde não é necessário que o emissor tenha a certeza de que o destinatário realmente recebeu a mensagem. Observa-se que os objetos podem estar na mesma máquina ou em máquinas distintas.

Por exemplo, a figura 6.1 ilustra o caso onde há um objeto na máquina A e outro objeto na máquina B. O objeto da máquina A, O_e, é o emissor da mensagem; o objeto da máquina B, O_r, é o receptor da mensagem.

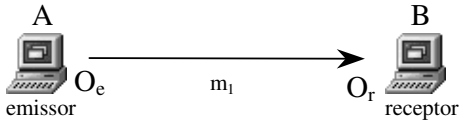

FIGURA 6.1 - Envio de *unicast* não-confiável de O_e para O_r

A seguir, descreve-se como pode ser implementado o código do emissor e do receptor para realizar o envio de uma mensagem na modalidade *unicast* não-confiável, ilustrado pela figura 6.1.

Inicialmente, será examinada uma implementação possível para o código do objeto emissor O_e (figura 6.2). As classes formam um pacote denominado **COM**, por isso

a existência da linha 1, em que as classes são importadas, possibilitando a instanciação de objetos a partir destas. A classe criada **`UnicastSender`** implementa a interface **`DataHandler`** (linha 4); isto é necessário para que O_e possa receber mensagens, que chegam automaticamente ao método **`deliver`** (linha 16), graças a um mecanismo de *callback* implementado internamente.

```
 1 import COM.*;
 2 import java.net.*;
 3
 4 public class UnicastSender implements DataHandler {
 5  public static void main(String argv[]) throws java.io.IOException {
 6   Unicast uc = new Unicast(new UnicastSender());
 7   String msg = "Hello my friend!";
 8   InetAddress ip = InetAddress.getByName(argv[0]);
 9   int port = (Integer.valueOf(argv[1])).intValue();
10
11   if (uc.Send(ip, port, msg) == false) {
12    System.out.println("Error sending unreliable unicast message...");
13   }
14  }
15
16  public void deliver(String sender,String msg) {
17   System.out.println("Delivered message: " + msg);
18   System.out.println("Sender: " + sender);
19  }
20 }
```

FIGURA 6.2 - Código Java da classe do objeto emissor O_e (*unicast* não-confiável)

Para que uma mensagem do tipo *unicast* não-confiável possa ser enviada, inicialmente é preciso instanciar um objeto da classe **`Unicast`** (linha 6), passando ao construtor desta a referência de endereçamento do objeto da classe **`UnicastSender`** (linha 6).

Neste ponto, é importante observar que o objeto emissor (O_e) é criado através do código **`new UnicastSender()`** (linha 6). O objeto **`uc`** criado na linha 6 precisa receber o endereço de O_e para que possa guardar uma referência de endereçamento deste objeto; esta referência permite que O_e receba mensagens das camadas inferiores, utilizando um mecanismo de *callback*.

A mensagem a ser enviada é composta e armazenada em **`msg`** (linha 7). Tanto o envio, quanto o recebimento de mensagens é feito através de *sockets*, o que exige o conhecimento do endereço IP e porta do destinatário. No código em análise, estas informações são passadas como argumentos de linha de comando e armazenadas pelas variáveis **`ip`** (linha 8) e **`port`** (linha 9).

Por fim, para enviar a mensagem, é chamado o método **`Send`** da classe **`Unicast`** (linha 11). Como é possível observar, este método recebe como parâmetros o endereço IP, a porta e a mensagem a ser enviada. O endereço IP e a porta identificam a localização do objeto destinatário, ou seja, em que máquina o objeto O_r se encontra e qual a porta que o referido objeto mantém sob escuta, a espera de mensagens.

Aguardando a chegada da mensagem está o objeto receptor O_r (figura 6.3), o qual tem seu código descrito a seguir. Observa-se que o código do receptor de uma

mensagem *unicast* não-confiável é bem similar ao código do emissor, porém bem mais simples. Da mesma forma, o pacote de classes é importado (linha 1). Na declaração da classe, é definido que ela implementa a interface **`DataHandler`** (linha 3), possibilitando a chamada automática do método **`deliver`** (linha 10), sempre que uma mensagem é endereçada ao objeto.

```
 1 import COM.*;
 2
 3 public class UnicastReceiver implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   Unicast uc = new Unicast(new UnicastReceiver());
 6
 7   while(true);
 8  }
 9
10  public void deliver(String sender,String msg) {
11   System.out.println("Delivered message: " + msg);
12   System.out.println("Sender: " + sender);
13  }
14 }
```

FIGURA 6.3 - Código Java da classe do objeto receptor O_r (*unicast* não-confiável)

É bom salientar que não basta declarar que a classe implementa a interface **`DataHandler`**. Para que as camadas inferiores possam chamar o método **`deliver`**, precisam conhecer o endereço do objeto em questão. Eis o porquê do construtor da classe **`Unicast`** ter recebido como parâmetro o endereço do objeto da classe **`UnicastReceiver`**, o qual fora recém-criado (linha 5).

Uma última observação, e de suma importância, diz respeito às linhas de execução no receptor. Internamente, a classe **`Unicast`** cria *threads* para enviar e tratar o recebimento das mensagens. A grande vantagem disto é a possibilidade de realizar várias tarefas "ao mesmo tempo". A desvantagem vem do fato de exigir uma atenção maior do programador, já que precisa gerenciar estas linhas de execução.

Em síntese, há duas linhas de execução principais. Uma delas é uma *thread* que fica a espera de mensagens e chama o método **`deliver`** automaticamente sempre que receber uma mensagem, através de um mecanismo de *callback*. A outra é a linha de execução criada pelo método **`main`** na classe **`UnicastReceiver`**. Este pode se subdividir em mais linhas de execução se o programador começar a criar outras *threads* por conta própria.

De nada adiantaria o código de um receptor onde a linha de execução do método **`main`** atingisse o final antes que o objeto pudesse receber alguma mensagem, já que isto provocaria a "morte" repentina do referido objeto. Observa-se, portanto, a necessidade de prover um mecanismo que trate este tipo de situação. Por exemplo, no código ilustrado pela figura 6.3, foi criado um *loop* infinito (linha 7), que não deixa o método **`main`** atingir o seu final, possibilitando o recebimento das mensagens pelo método **`deliver`**. É claro que esta não é uma solução elegante, mas funciona bem na implementação de um servidor, por exemplo.

O relacionamento entre as camadas durante o *unicast* não-confiável é ilustrado pela figura 6.4. Neste caso, para que o emissor possa enviar o *unicast*, em sua aplicação,

primeiramente há uma instanciação de um objeto da classe **`Unicast`**, após o envio é feito através do método **`Send`**. No lado do receptor, a aplicação instancia também um objeto da classe **`Unicast`**. Com isto, é criada uma *thread*, que permanece constantemente à espera de mensagens. Ao receber uma mensagem, esta *thread* a redireciona para a aplicação através de um mecanismo de *callback*, cuja funcionalidade é realizada pelo método **`deliver`**.

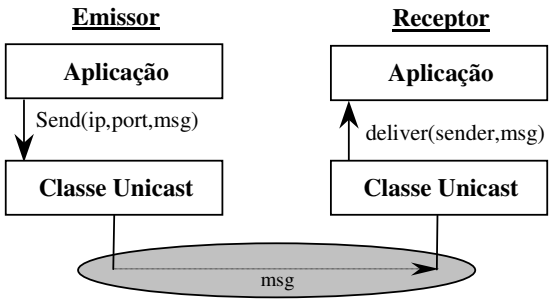

FIGURA 6.4 - Relacionamento entre camadas durante *unicast* não-confiável

A toda mensagem do tipo *unicast* não-confiável é anexado um cabeçalho contendo dois campos, um deles indicando que a mensagem é do tipo *unicast* não-confiável (**`Refs.UNICAST`**) e outro identificando o objeto emissor da mensagem (**`senderObject`**) (figura 6.5).

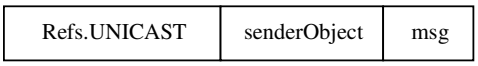

FIGURA 6.5 - Formato das mensagens do tipo *unicast* não-confiável

6.2 Envio em *Unicast* Confiável

Esta modalidade de comunicação é utilizada quando se deseja enviar uma mensagem de um objeto para outro objeto, onde é necessário que o emissor tenha a certeza de que o destinatário realmente recebeu a mensagem. Da mesma forma já considerada nas suposições referentes à modalidade de comunicação não-confiável, os objetos podem estar na mesma máquina ou em máquinas distintas.

Por exemplo, a figura 6.6 ilustra o caso onde há um objeto na máquina A e outro objeto na máquina B. O objeto da máquina A, O_e, é o emissor da mensagem; o objeto da máquina B, O_r, é o receptor da mensagem. O objeto emissor fica sabendo que o objeto receptor recebeu a mensagem na chegada de uma mensagem do tipo ACK, reconhecendo a mensagem. Se após um certo tempo o emissor não receber o reconhecimento positivo da mensagem, ele realiza um reenvio.

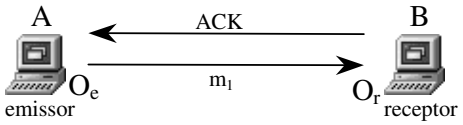

FIGURA 6.6 - Envio de *unicast* confiável de O_e para O_r

A seguir, descreve-se como pode ser implementado o código do emissor e do receptor para realizar um envio de uma mensagem na modalidade *unicast* confiável, como ilustrado esquematicamente pela figura 6.6.

Inicialmente, examina-se o código do objeto emissor O_e (figura 6.7).

```
 1 import COM.*;
 2 import java.net.*;
 3
 4 public class RUnicastSender implements DataHandler {
 5  public static void main(String argv[]) throws java.io.IOException {
 6   RUnicast ruc = new RUnicast(new RUnicastSender());
 7   String msg = "Hello my friend!";
 8   InetAddress ip = InetAddress.getByName(argv[0]);
 9   int port = (Integer.valueOf(argv[1])).intValue();
10
11   if (ruc.Send(ip, port, msg) == false) {
12    System.out.println("Error sending reliable unicast message...");
13   }
14  }
15
16  public void deliver(String sender,String msg) {
17   System.out.println("Delivered message: " + msg);
18   System.out.println("Sender: " + sender);
19  }
20 }
```

FIGURA 6.7 - Código Java da classe do objeto emissor O_e (*unicast* confiável)

Como já mencionado, as classes formam um pacote denominado **COM**; isso explica a existência da linha 1, em que as classes são importadas, possibilitando a instanciação de objetos a partir destas. A classe criada **RUnicastSender** implementa a interface **DataHandler** (linha 4); isto é necessário para que O_e possa receber mensagens, que chegam automaticamente ao método **deliver** (linha 16) - mecanismo de *callback*.

Para que uma mensagem do tipo *unicast* confiável possa ser enviada, inicialmente é preciso instanciar um objeto da classe **RUnicast** (linha 6), passando ao construtor desta a referência de endereçamento do objeto da classe **RUnicastSender** (linha 6).

É importante observar que o objeto emissor - O_e, que vem sendo comentado, é criado através do código **new RUnicastSender()** (linha 6). O objeto **ruc**, criado na linha 6, precisa receber o endereço de O_e para que possa guardar uma referência de endereçamento deste objeto; esta referência permite que O_e receba mensagens das camadas inferiores - mecanismo de *callback*.

A mensagem a ser enviada é composta e armazenada em **msg** (linha 7). Tanto o envio, quanto o recebimento de mensagens é feito através de *sockets*, o que exige o conhecimento do endereço IP e porta do destinatário. Estas informações, no código acima ilustrado, são passados como argumentos de linha de comando e armazenados pelas variáveis **ip** (linha 8) e **port** (linha 9).

Por fim, para enviar a mensagem, é chamado o método **Send** da classe **RUnicast** (linha 11). Observa-se que, como a classe **RUnicast** é filha da classe

Unicast, ela herda seus métodos; entretanto o método **Send** da classe **Unicast** não serve para a classe **RUnicast**, porque o interesse nela é envio confiável.

Desta forma, o método **Send** (não-confiável) foi sobrescrito, implementando o envio confiável. Ele recebe como parâmetros o endereço IP, a porta e a mensagem a ser enviada; endereço IP e porta identificam a localização do objeto destinatário (O_r).

A seguir, é descrito o código referente ao objeto receptor O_r, apresentado através da figura 6.8.

```
 1 import COM.*;
 2
 3 public class RUnicastReceiver implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   RUnicast ruc = new RUnicast(new RUnicastReceiver());
 6
 7   while(true);
 8  }
 9
10  public void deliver(String sender,String msg) {
11   System.out.println("Delivered message: " + msg);
12   System.out.println("Sender: " + sender);
13  }
14 }
```

FIGURA 6.8 - Código Java da classe do objeto receptor O_r (*unicast* confiável)

O código do receptor de uma mensagem *unicast* confiável é bem similar ao código do receptor de uma mensagem *unicast* não-confiável. Da mesma forma, o pacote de classes é importado (linha 1). Na declaração da classe é identificado que ela implementa a interface **DataHandler** (linha 3), possibilitando a chamada automática do método **deliver** (linha 10), sempre que uma mensagem for endereçada ao objeto.

Para que as camadas inferiores possam chamar o método **deliver**, não basta declarar que a classe implementa a interface **DataHandler**, pois é preciso conhecer o endereço do objeto em questão. Por isso, o construtor da classe **RUnicast** recebe como parâmetro o endereço do objeto da classe **RUnicastReceiver**, o qual foi recém-criado (linha 5).

Da mesma forma que explicado para o receptor de mensagens do tipo *unicast* não-confiável, foi criado um *loop* infinito (linha 7), que não deixa o método **main** atingir o seu final, possibilitando o recebimento das mensagens pelo método **deliver**.

O relacionamento entre as camadas durante o *unicast* confiável é ilustrado pela figura 6.9. Para o envio de mensagens, na aplicação do emissor há uma instanciação de um objeto da classe **RUnicast**, e a utilização do método **Send** sobrescrito da classe **Unicast**.

No lado do receptor, a aplicação instancia também um objeto da classe **RUnicast**, além de criar uma *thread*, que permanece constantemente à espera de mensagens. Ao receber uma mensagem, esta *thread* a redireciona para a aplicação

através de um mecanismo de *callback*, cuja funcionalidade é realizada pelo método **`deliver`**.

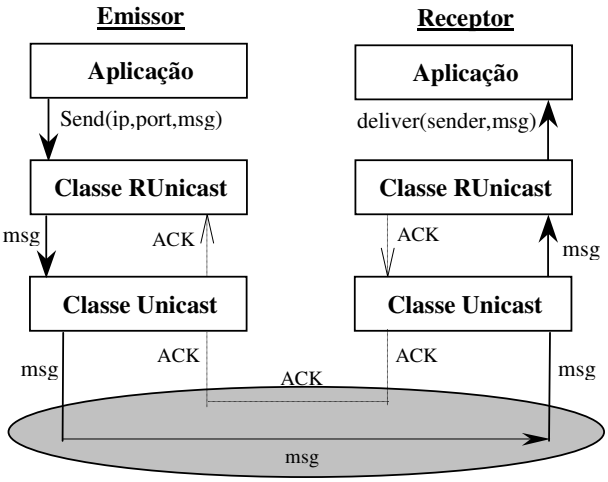

FIGURA 6.9 - Relacionamento entre camadas durante *unicast* confiável

A toda mensagem do tipo *unicast* confiável é anexado um cabeçalho contendo os seguintes campos: o primeiro (**`Refs.RUNICAST`**) indica o que a mensagem é do tipo *unicast* confiável; **`ACKport`** estabelece a porta para o reconhecimento positivo da mensagem; **`senderObject`**, **`senderIP`** e **`senderPort`**, identificam, respectivamente, o objeto emissor da mensagem, e o endereço IP e porta deste objeto; **`msgSeqNo`** identifica o número de seqüência da referida mensagem; e **`msg`** é a mensagem propriamente dita (figura 6.10).

Refs.RUNICAST	ACKport	senderObject	senderIP	senderPort	msgSeqNo	msg

FIGURA 6.10 - Formato das mensagens do tipo *unicast* confiável

6.3 Envio em *Multicast* Não-Confiável

Esta modalidade de comunicação é utilizada quando se deseja enviar uma mensagem de um objeto para um conjunto de objetos, onde não é necessário que o emissor tenha a certeza de que os destinatários realmente receberam a mensagem. Os objetos podem estar na mesma máquina ou em máquinas distintas (figura 6.11)

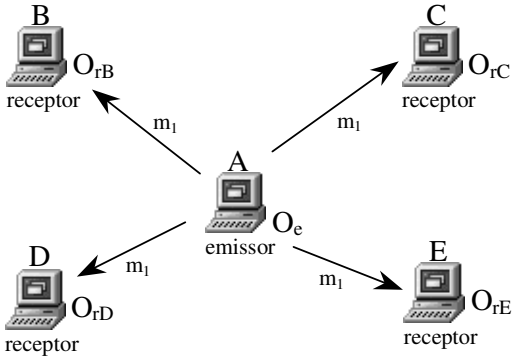

FIGURA 6.11 - Envio de *multicast* não-confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE}

A figura 6.11 ilustra o caso onde há objetos nas máquinas A, B, C, D e E. O objeto da máquina A, O_e, é o emissor da mensagem e os objetos das máquinas B, C, D e E (O_{rB}, O_{rC}, O_{rD}, O_{rE}) são os receptores da mensagem. A seguir, descreve-se como pode ser implementado o código do emissor e de cada receptor para realizar um envio de uma mensagem na modalidade *multicast* não-confiável, para o ambiente proposto na figura.

Inicialmente, examina-se o código do objeto emissor O_e, listado na figura 6.12.

```
 1 import COM.*;
 2 import java.net.*;
 3
 4 public class MulticastSender implements DataHandler {
 5  public static void main(String argv[]) throws java.io.IOException {
 6   Multicast mc = new Multicast(new MulticastSender());
 7   String msg = "Hello my friend!";
 8   int IDGroup = 7654;
 9
10   if (mc.SendMcast(IDGroup, msg) == false) {
11    System.out.println("Error sending unreliable multicast message...");
12   }
13  }
14
15  public void deliver(String sender,String msg) {
16   System.out.println("Delivered message: " + msg);
17   System.out.println("Sender: " + sender);
18  }
19 }
```

FIGURA 6.12 - Código Java da classe do objeto emissor O_e (*multicast* não-confiável)

O pacote denominado **`COM`** é importado (linha 1). A classe criada **`MulticastSender`** implementa a interface **`DataHandler`** (linha 4); isto é necessário para que O_e possa receber mensagens, que chegam automaticamente ao método **`deliver`** (linha 15), através de *callback*.

Para que uma mensagem do tipo *multicast* não-confiável possa ser enviada, inicialmente é preciso instanciar um objeto da classe **`Multicast`** (linha 6), passando ao construtor desta a referência de endereçamento do objeto da classe **`MulticastSender`** (linha 6). O construtor de **`Multicast`** recebe como parâmetro a referência de

endereçamento do objeto da classe **MulticastSender()**, o que lhe permite o recebimento de mensagens das camadas inferiores pelo método **deliver**.

A mensagem a ser enviada é composta e armazenada em **msg** (linha 7). O envio de mensagens é feito via *multicast* IP, que utiliza a classe D de endereços IP, e uma definição de porta identifica os grupos (seção 2.4). Neste caso, a definição da porta é armazenada em **IDGroup** (linha 8).

O método **SendMcast** (linha 10) é que fornece a primitiva de envio *multicast* não-confiável. Ele recebe como parâmetros a identificação do grupo e a mensagem a ser enviada.

A seguir, descreve-se o código referente aos objetos receptores O_r, listado na figura 6.13.

```
 1 import COM.*;
 2
 3 public class MulticastReceiver implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   Multicast mc = new Multicast(new MulticastReceiver());
 6   int IDGroup = 7654;
 7   mc.join(IDGroup);
 8   while(true);
 9  }
10
11  public void deliver(String sender,String msg) {
12   System.out.println("Delivered message: " + msg);
13   System.out.println("Sender: " + sender);
14  }
15 }
```

FIGURA 6.13 - Código Java da classe do objeto receptor O_r (*multicast* não-confiável)

O código do receptor de uma mensagem *multicast* não-confiável é similar ao código dos receptores para as outras classes de protocolos, que já foram mencionados. O pacote **COM** é importado (linha 1). A classe implementa a interface **DataHandler** (linha 3), possibilitando a chamada automática do método **deliver** (linha 11).

Para que as camadas inferiores possam chamar o método **deliver**, o construtor da classe **Multicast** recebe como parâmetro o endereço do objeto da classe **MulticastReceiver**, o qual fora recém criado (linha 5). Para receber as mensagens destinadas a um determinado grupo, o receptor deve fazer parte deste grupo e isto é feito através do método **join** (linha 7). O *loop* infinito (linha 8) não deixa que o método **main** atinja o seu final, possibilitando o recebimento das mensagens pelo método **deliver**.

O relacionamento entre as camadas durante o *multicast* não-confiável é ilustrado pela figura 6.14. Para o envio de mensagens, na aplicação do emissor, há uma instanciação de um objeto da classe **Multicast**, e a utilização do método **SendMcast**.

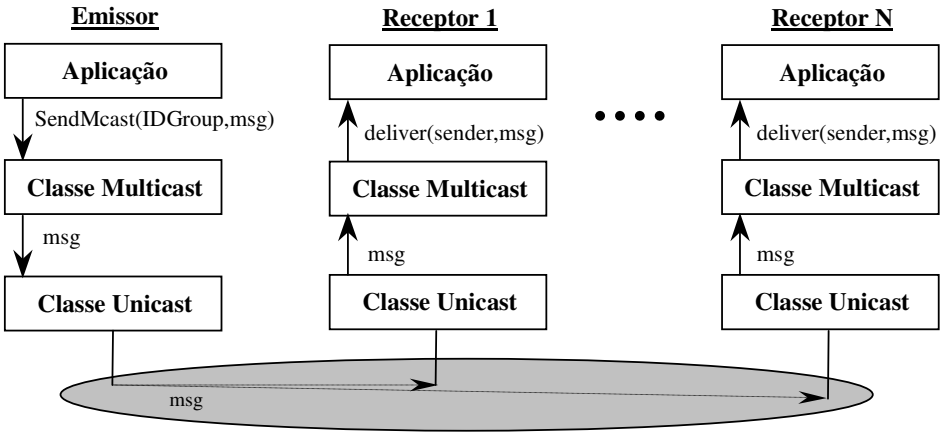

FIGURA 6.14 - Relacionamento entre camadas durante *multicast* não-confiável

No lado do receptor, a aplicação instancia também um objeto da classe **`Multicast`**, além de criar uma *thread*, que permanece constantemente à espera de mensagens. Ao receber uma mensagem, esta *thread* a redireciona para a aplicação através de um mecanismo de *callback*, cuja funcionalidade é realizada pelo método **`deliver`**.

A toda mensagem do tipo *multicast* não-confiável é anexado um cabeçalho contendo os campos: **`Refs.MULTICAST`** indica que a mensagem é do tipo *multicast* não-confiável, **`senderObject`** identifica o objeto emissor da mensagem, **`GroupPort`** é o identificador do grupo e **`msg`** é a mensagem propriamente dita (figura 6.15).

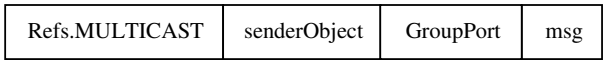

FIGURA 6.15 - Formato das mensagens do tipo *multicast* não-confiável

6.4 Envio em *Multicast* Confiável

Esta modalidade de comunicação é utilizada quando se deseja enviar uma mensagem de um objeto para um conjunto de objetos, onde é necessário garantir que todos os receptores corretos recebam a mensagem. Os objetos podem estar na mesma máquina ou em máquinas distintas. A implementação pode ser feita através de várias mensagens do tipo *unicast* (figura 6.16), ou pode utilizar *multicast* IP (figura 6.17).

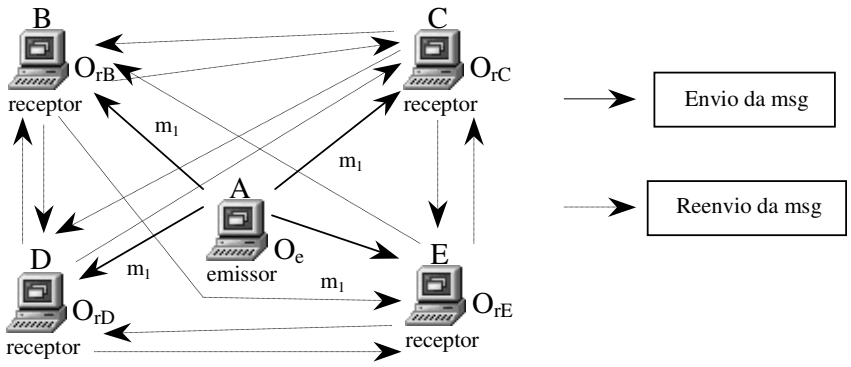

FIGURA 6.16 - Envio de *multicast* confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE} via *unicast*

Em ambos os casos, a técnica para garantir que todos os receptores corretos recebam a mensagem é a de utilização do enfoque baseado em redundância (conforme explicado na seção 3.7.1). Em síntese, este enfoque garante confiabilidade fazendo com que cada receptor correto retransmita a mensagem recebida para os demais destinatários, o que garante uma confiabilidade condicionada a receptores corretos, bem como canais de comunicação corretos.

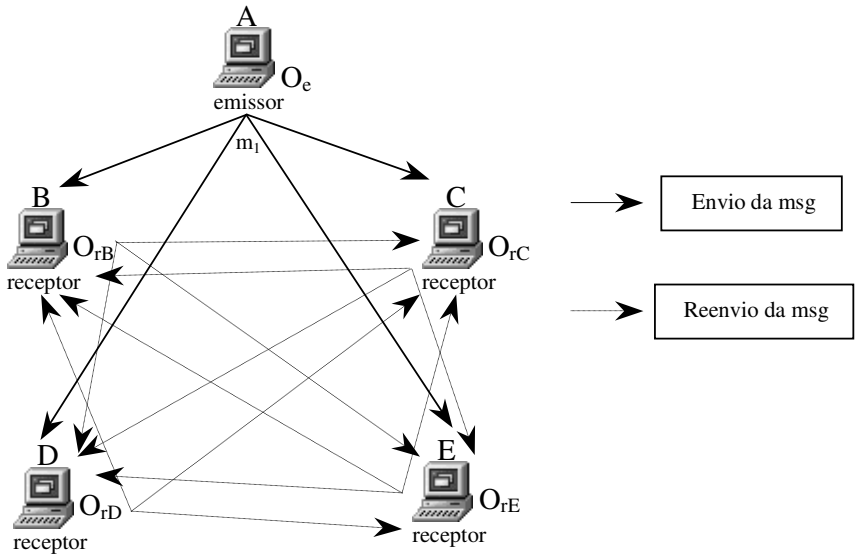

FIGURA 6.17 - Envio de *multicast* confiável de O_e para O_{rB}, O_{rC}, O_{rD}, O_{rE} via *multicast* IP

Neste caso, se um dos destinatários recebeu a mensagem, bastará que exista pelo menos um canal de comunicação correto, interligando cada destinatário correto, para garantir a entrega da mensagem; ou seja, se um destinatário não ficar isolado dos demais, ele receberá a mensagem.

A seguir, descreve-se como pode ser implementado o código do emissor e de cada receptor para realizar um envio de uma mensagem na modalidade *multicast* confiável.

Inicialmente, examina-se o código do objeto emissor O_e que utiliza vários *unicasts* para implementar o *multicast* confiável (figura 6.18).

```
 1 import COM.*;
 2 import java.net.*;
 3
 4 public class RMulticastSender implements DataHandler {
 5  public static void main(String argv[]) throws java.io.IOException {
 6   RMulticast rmc = new RMulticast(new RMulticastSender());
 7   String msg = "Hello my friend!";
 8   int IDGroup = 7654;
 9   LinkedList dests = new LinkedList();
10   ...
11   if (rmc.RSendUcast(IDGroup, dests, msg) == false) {
12    System.out.println("Error sending reliable multicast message...");
13   }
14  }
15
16  public void deliver(String sender,String msg) {
17   System.out.println("Delivered message: " + msg);
18   System.out.println("Sender: " + sender);
19  }
20 }
```

FIGURA 6.18 - Código Java da classe do objeto emissor O_e (*multicast* confiável/*unicast*)

O pacote denominado **`COM`** é importado (linha 1). A classe criada **`RMulticastSender`** implementa a interface **`DataHandler`** (linha 4), possibilitando a O_e receber mensagens, que chegam automaticamente ao método **`deliver`** (linha 16), através de *callback*.

Para que uma mensagem do tipo *multicast* confiável (implementado via vários *unicasts*) possa ser enviada, é preciso instanciar um objeto da classe **`RMulticast`** (linha 6), passando-lhe a referência de endereçamento do objeto da classe **`RMulticastSender`**. O construtor de **`RMulticast`** recebe como parâmetro a referência de endereçamento do objeto da classe **`RMulticastSender()`**; esta referência é utilizada pelo método **`deliver`**.

A mensagem a ser enviada é composta e armazenada em **`msg`** (linha 7). O envio de mensagens é feito através de vários *unicasts*; assim, além da porta que identifica o grupo **`IDGroup`** (linha 8), é criada uma lista **`dests`**, contendo os objetos destinatários da mensagem (linha 9).

O método **`RSendUcast`** (linha 11) é que fornece a primitiva de envio *multicast* confiável, recebendo como parâmetros a identificação do grupo, a lista de destinatários e a mensagem a ser enviada. Internamente, a mensagem é enviada para cada destinatário via *unicast* e para garantir a confiabilidade é utilizado o enfoque baseado em redundância, onde cada destinatário retransmite a mensagem recebida.

A seguir, examina-se como pode ser o código do objeto emissor O_e que utiliza *multicast* IP para implementar o *multicast* confiável (figura 6.19).

```
 1 import COM.*;
 2 import java.net.*;
 3
 4 public class RMulticastSender implements DataHandler {
 5  public static void main(String argv[]) throws java.io.IOException {
 6   RMulticast rmc = new RMulticast(new RMulticastSender());
 7   String msg = "Hello my friend!";
 8   int IDGroup = 7654;
 9
10   if (rmc.RSendMcast(IDGroup, msg) == false) {
11    System.out.println("Error sending reliable multicast message...");
12   }
13  }
14
15  public void deliver(String sender,String msg) {
16   System.out.println("Delivered message: " + msg);
17   System.out.println("Sender: " + sender);
18  }
19 }
```

FIGURA 6.19- Código Java da classe do objeto emissor O_e (*multicast* confiável/*multicast*)

O código para o emissor que utiliza *multicast* IP é quase idêntico ao código do emissor que utiliza vários *unicasts*. O pacote denominado **COM** é importado (linha 1). A classe criada **RMulticastSender** implementa a interface **DataHandler** (linha 4), possibilitando a O_e receber mensagens, que chegam automaticamente ao método **deliver** (linha 15), através de *callback*.

Para que uma mensagem do tipo *multicast* confiável (implementado via *multicast* IP) possa ser enviada, é preciso instanciar um objeto da classe **RMulticast** (linha 6), passando-lhe a referência de endereçamento do objeto da classe **RMulticastSender**. O construtor de **RMulticast** recebe como parâmetro a referência de endereçamento do objeto da classe **RMulticastSender()**; esta referência é utilizada pelo método **deliver**.

A mensagem a ser enviada é composta e armazenada em **msg** (linha 7). Ao se utilizar *multicast* IP, cada grupo deve ser identificado por um par (endereço IP/porta); o endereço IP é sempre o mesmo, o que varia é a porta utilizada para identificar cada grupo. Neste caso, a definição do grupo é **IDGroup** (linha 8).

O método **RSendMcast** (linha 10) fornece a primitiva de envio *multicast* confiável. Ele recebe como parâmetros a identificação do grupo destinatário e a mensagem a ser enviada. A mensagem é então enviada para os destinatários através de *multicast* IP e, para garantir confiabilidade, cada destinatário retransmite a mensagem (também via *multicast* IP) aos demais destinatários; ou seja, utiliza-se o enfoque baseado em redundância.

Agora descreve-se o código referente aos objetos receptores O_r, listado na figura 6.20.

```
 1 import COM.*;
 2
 3 public class RMulticastReceiver implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   RMulticast rmc = new RMulticast(new RMulticastReceiver());
 8   int IDGroup = 7654;
 7   rmc.join(IDGroup);
 8   while(true);
 9  }
10
11  public void deliver(String sender,String msg) {
12   System.out.println("Delivered message: " + msg);
13   System.out.println("Sender: " + sender);
14  }
15 }
```

FIGURA 6.20 - Código Java da classe do objeto receptor O_r (*multicast* confiável)

O código do receptor de uma mensagem *multicast* confiável é similar ao código dos receptores para as outras classes de protocolos, que já foram mencionados. O pacote **`COM`** é importado (linha 1). A classe implementa a interface **`DataHandler`** (linha 3), possibilitando a chamada automática do método **`deliver`** (linha 11).

Para que as camadas inferiores possam chamar o método **`deliver`**, é preciso conhecer o endereço do objeto em questão; por isso o construtor da classe **`RMulticast`** recebe como parâmetro o endereço do objeto da classe **`RMulticastReceiver`** (linha 5). Para receber as mensagens destinadas a um determinado grupo, o receptor deve fazer parte deste grupo e isto é feito através do método **`join`** (linha 7). O *loop* infinito (linha 8) não deixa que o método **`main`** atinja o seu final, possibilitando o recebimento das mensagens pelo método **`deliver`**.

O relacionamento entre as camadas durante o *multicast* confiável, implementado através de várias mensagens do tipo *unicast*, é ilustrado pela figura 6.21. Para o envio de mensagens, na aplicação do emissor, há uma instanciação de um objeto da classe **`RMulticast`** e a utilização do método **`RSendUcast`**. A mensagem é passada às camadas inferiores, através de um mecanismo de *downcall*, até ser transmitida pela camada de *socket*.

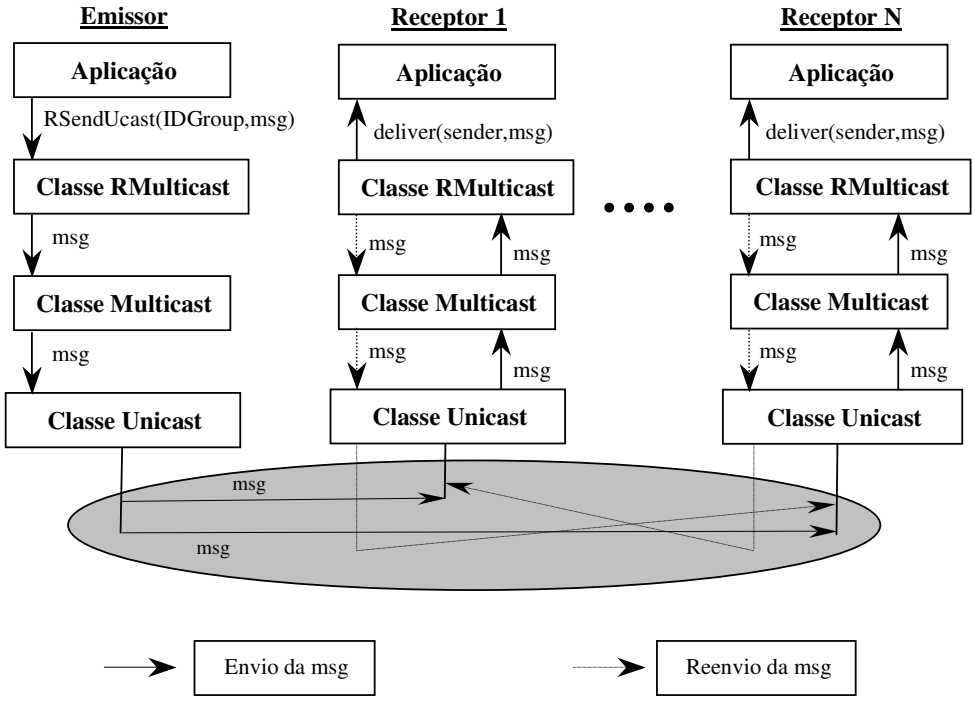

FIGURA 6.21 - Relacionamento entre camadas durante *multicast* confiável via *unicast*

Em cada receptor, a aplicação instancia também um objeto da classe **`RMulticast`**, além de criar uma *thread*, que permanece constantemente à espera de mensagens. Ao receber uma mensagem, esta *thread* a redireciona para a camada superior, neste caso, a classe **`Multicast`**. Isto se repete até que a mensagem alcance a aplicação, utilizando um mecanismo de *callback*.

A toda mensagem do tipo *multicast* confiável (cuja implementação foi feita através de várias mensagens do tipo *unicast*) é anexado um cabeçalho contendo os seguintes campos: **`Refs.RMCAST_UNICAST`** indica que a mensagem é do tipo *multicast* confiável (implementado por *unicast*); **`Refs.DELIM`** é utilizado como delimitador para o campo **`detsObjs`**, que contém a lista dos objetos destinatários da mensagem; **`senderObject`** identifica o objeto emissor da mensagem; **`senderIP`** é o endereço IP do emissor e **`senderPort`** é a porta; **`msgSeqNo`** é o número de seqüência que identifica univocamente cada mensagem transmitida por um endereço IP/porta; e **`msg`** é a mensagem propriamente dita (figura 6.22).

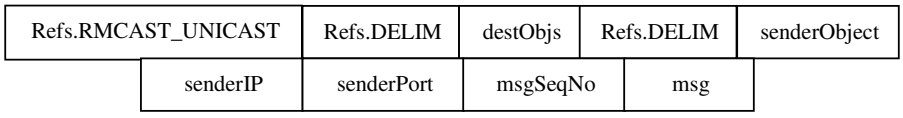

FIGURA 6.22 - Formato das mensagens do tipo *multicast* confiável (via *unicasts*)

O relacionamento entre as camadas durante o *multicast* confiável, que é implementado através de *multicast* IP, é ilustrado pela figura 6.23. Para o envio de

mensagens, na aplicação do emissor há uma instanciação de um objeto da classe **`RMulticast`**, e a utilização do método **`RSendMcast`**. A mensagem é passada às camadas inferiores através de um mecanismo de *downcall*, até ser transmitida pela camada de *socket*.

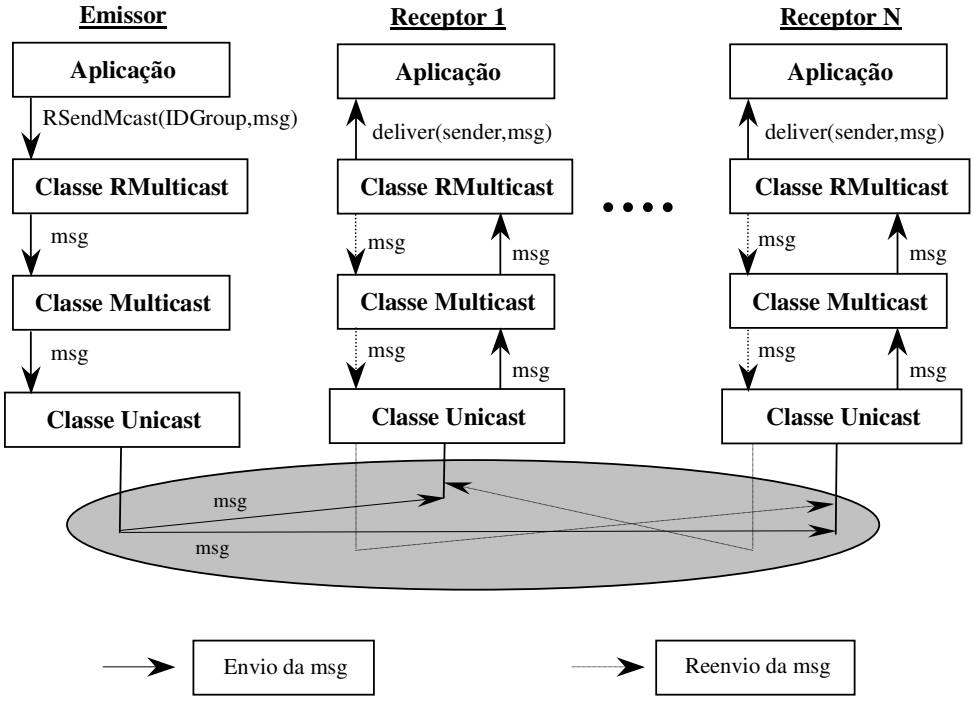

FIGURA 6.23 - Relacionamento entre camadas durante *multicast* confiável via *multicast*

Em cada receptor, a aplicação instancia também um objeto da classe **`RMulticast`**, além de ser criada uma *thread*, que permanece constantemente à espera de mensagens. Ao receber uma mensagem, esta *thread* a redireciona para a camada superior, neste caso a classe **`Multicast`**. Isto se repete até que a mensagem alcance a aplicação, utilizando um mecanismo de *callback*.

A toda mensagem do tipo *multicast* confiável (cuja implementação foi feita através de *multicast* IP) é anexado um cabeçalho contendo os seguintes campos: **`Refs.RMCAST_MULTICAST`** indica que a mensagem é do tipo *multicast* confiável (implementado por *multicast* IP); **`senderObject`** identifica o objeto emissor da mensagem; **`senderIP`** é o endereço IP do emissor e **`senderPort`** é a porta; **`destID`** é a identificação do grupo destinatário da mensagem; **`msgSeqNo`** é o número de seqüência que identifica univocamente cada mensagem transmitida por um endereço IP/porta; e **`msg`** é a mensagem propriamente dita (figura 6.24).

Refs.RMCAST_MULTICAST	senderObject	senderIP	senderPort	destID	msgSeqNo	msg

FIGURA 6.24 - Formato das mensagens do tipo *multicast* confiável (*multicast* IP)

6.5 Utilizando um Servidor de Nomes

Até agora, foram ilustradas classes onde o programador precisa se preocupar com detalhes adicionais como, por exemplo, manter o controle da referência dos objetos. Isto significa que para um objeto enviar uma mensagem, necessita ter o conhecimento do endereço IP e porta do objeto destinatário. Uma forma de facilitar a programação pode ser obtida com a utilização de um *servidor de nomes*. Com este, os objetos, bem como um conjunto de objetos (grupo), passam a ser referenciados através de um nome. Desta forma, para um objeto enviar uma mensagem, basta saber o nome do objeto destinatário. Esta vantagem, entretanto, degrada o desempenho das aplicações distribuídas, pois mais instruções precisam ser executadas, além de ser necessário o uso de mais memória para o armazenamento das estruturas referentes aos nomes.

Para a implementação do servidor de nomes, foi criada a classe **`NameServer`** (seção 5.2.15). Este servidor de nomes foi criado apenas como demonstração sobre como a programação pode se tornar mais facilitada com este tipo de abstração, onde os objetos e grupos possuem nomes. Desta forma, optou-se pelo enfoque centralizado, ou seja, o servidor de nomes não é replicado e não é tolerante a falhas.

O servidor de nomes pode ser inicializado em qualquer máquina e os objetos que desejam utilizá-lo ficam sabendo em que máquina o servidor de nomes se encontra de forma totalmente transparente, ou seja, o programador não precisa ter este controle. A única ressalva feita é a de que, entre a máquina onde o servidor de nomes se encontra e as demais máquinas onde os outros objetos se encontram, deve existir uma conexão de rede (um caminho para o tráfego das mensagens) e não devem existir roteadores que filtrem mensagens do tipo *multicast* IP.

A razão disso é que o servidor de nomes utiliza *multicast* IP para disseminar o seu endereço IP/porta. A seguir ilustra-se, passo-a-passo, como funciona o servidor de nomes. O funcionamento está esquematizado na figura 6.25.

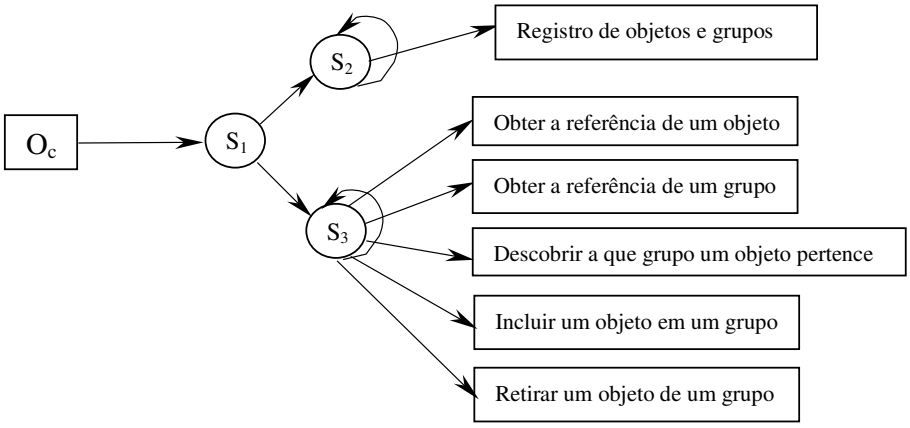

FIGURA 6.25 – Funcionamento interno do servidor de nomes

A questão inicial é como os demais objetos, como por exemplo O_c, ficam sabendo a referência do servidor de nomes; a resposta está na utilização de *multicast* IP.

Em S_1, o servidor de nomes é inicializado em uma das máquinas e conecta-se a um endereço IP/porta de *multicast* IP padrão, que é definido na classe **`Refs`** (seção 5.2.2). Assim, a sua referência é o endereço IP da máquina onde se encontra e a porta, que é obtida dinamicamente; ou seja, cada vez que um servidor de nomes é inicializado, uma porta diferente é obtida.

A seguir, o código do servidor de nomes é divido em duas linhas de execução (S_2 e S_3). Em S_2, é feito o registro dos nomes dos objetos e dos grupos. Em S_3, há a execução de uma *thread*, cuja funcionalidade implementa:

- registro de objetos e grupos, que são armazenados pela classe **`Register`** (seção 5.2.13);
- pesquisas para descobrir a referência relativa a um objeto ou a um grupo, bem como a que grupo um objeto pertence;
- inclusão de um objeto em um grupo; e
- remoção de um objeto de um determinado grupo.

Como forma de oferecer uma interface para que o programador utilize o servidor de nomes, foi criada a classe **`Communication`** (seção 5.2.14), que contém métodos para transparentemente:

- nomear um objeto e registrá-lo no servidor de nomes;
- enviar uma mensagem do tipo *unicast* não-confiável para um objeto referenciado pelo nome;
- enviar uma mensagem do tipo *unicast* confiável para um objeto referenciado pelo nome;
- enviar uma mensagem do tipo *multicast* não-confiável para um grupo (conjunto de objetos) referenciado pelo nome;
- enviar uma mensagem do tipo *multicast* confiável para um grupo (conjunto de objetos) referenciado pelo nome, através do método que utiliza várias mensagens do tipo *unicast*;
- enviar uma mensagem do tipo *multicast* confiável para um grupo (conjunto de objetos) referenciado pelo nome, através do método que utiliza *multicast* IP;
- enviar uma mensagem na modalidade não-confiável para um nome, se este nome referenciar um objeto, é feito um *unicast* e, se este nome referenciar um grupo, é feito um *multicast*;
- enviar uma mensagem na modalidade confiável para um nome, se este nome referenciar um objeto, é feito um *unicast* e, se este nome referenciar um grupo, é feito um *multicast*;
- inserir um objeto, referenciado pelo nome, em um determinado grupo;
- remover um objeto, referenciado pelo nome, de um determinado grupo;
- obter a referência relativa a um objeto ou a um grupo, que está armazenada no servidor de nomes.

A seguir, examina-se como pode ser o código de um objeto emissor O_e, que deseja enviar mensagens a outros objetos ou grupos, e que pretende fazê-lo utilizando o servidor de nomes. Para tanto, este objeto emissor utiliza a interface fornecida pela classe **`Communication`** (figura 6.26).

```
 1 import COM.*;
 2
 3 public class Sender implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   Communication Oe = new Communication(new Sender(),"Oe");
 6   String msg = "Hello my friend!";
 7
 8   Oe.Ucast("Or", msg);
 9   Oe.Mcast("dests", msg);
10   Oe.RMcast("dests", msg);
11
12   while(true);
13  }
14
15  public void deliver(String sender,String msg) {
16   System.out.println("Delivered message: " + msg);
17   System.out.println("Sender: " + sender);
18  }
19 }
```

FIGURA 6.26 - Código Java do objeto emissor O_e que utiliza o servidor de nomes

Neste código, observa-se como passar os parâmetros para o construtor da classe **`Communication`**, bem como enviar mensagens utilizando um objeto desta classe. As linhas 8, 9 e 10 ilustram, respectivamente, a sintaxe para o envio de uma mensagem do tipo *unicast* não-confiável, do tipo *multicast* não-confiável e do tipo *multicast* confiável. No caso, O_e referencia o objeto emissor da mensagem, O_r referencia o objeto receptor e **`dests`** referencia o grupo ao qual pertencem os objetos destinatários da mensagens do tipo *multicast* em modo confiável e não-confiável.

Observa-se que O_e pode tanto enviar como receber mensagens. As mensagens enviadas para O_e são percebidas na aplicação através do método **`deliver`** (linha 15). Desta forma, conclui-se que o código de um objeto receptor é quase idêntico ao código acima descrito, a única mudança relevante é o nome do objeto[14]. Além disso, a única alteração significativa no código de um receptor que pertence a um grupo é a inclusão de uma linha que executa a operação de inclusão em um grupo, como ilustra a figura 6.27, linha 7.

Apesar do código de exemplo ilustrar o caso onde o objeto O_r, pertencente ao grupo **`dests`**, envia uma mensagem do tipo *multicast* não-confiável para o grupo **`dests`**, não há esta restrição. O que se quer dizer com isto é que os grupos podem ser tanto fechados, quanto abertos. Além disso, os grupos são não-hierárquicos e dinâmicos, mas não foi implementado suporte para grupos sobrepostos (seção 2.5).

[14] Não é permitido objeto ou grupos com o mesmo nome no sistema, ou seja, cada objeto e grupo tem uma identificação baseada em nome, que é única.

```
 1 import COM.*;
 2
 3 public class Receiver implements DataHandler {
 4  public static void main(String argv[]) throws java.io.IOException {
 5   Communication Or = new Communication(new Receiver(),"Or");
 6   String msg = "Hello my friend!";
 7   Or.join("dests");
 8
 9   Oe.Ucast("Oe", msg);
10   Oe.Mcast("dests", msg);
11
12   while(true);
13  }
14
15  public void deliver(String sender,String msg) {
16   System.out.println("Delivered message: " + msg);
17   System.out.println("Sender: " + sender);
18  }
19 }
```

FIGURA 6.27 - Código Java da classe do objeto receptor O_r (*multicast* confiável)

O relacionamento entre as camadas durante um *multicast* confiável que utiliza a interface fornecida pela classe **`Communication`** é ilustrado pela figura 6.28. Para o envio de mensagens, na aplicação do emissor há uma instanciação de um objeto da classe **`Communication`**, e a utilização do método **`RMcast`**. O funcionamento de envio e de recebimento das mensagens, utilizando os mecanismos de *downcall* e *callback*, é similar a todos que foram ilustrados anteriormente.

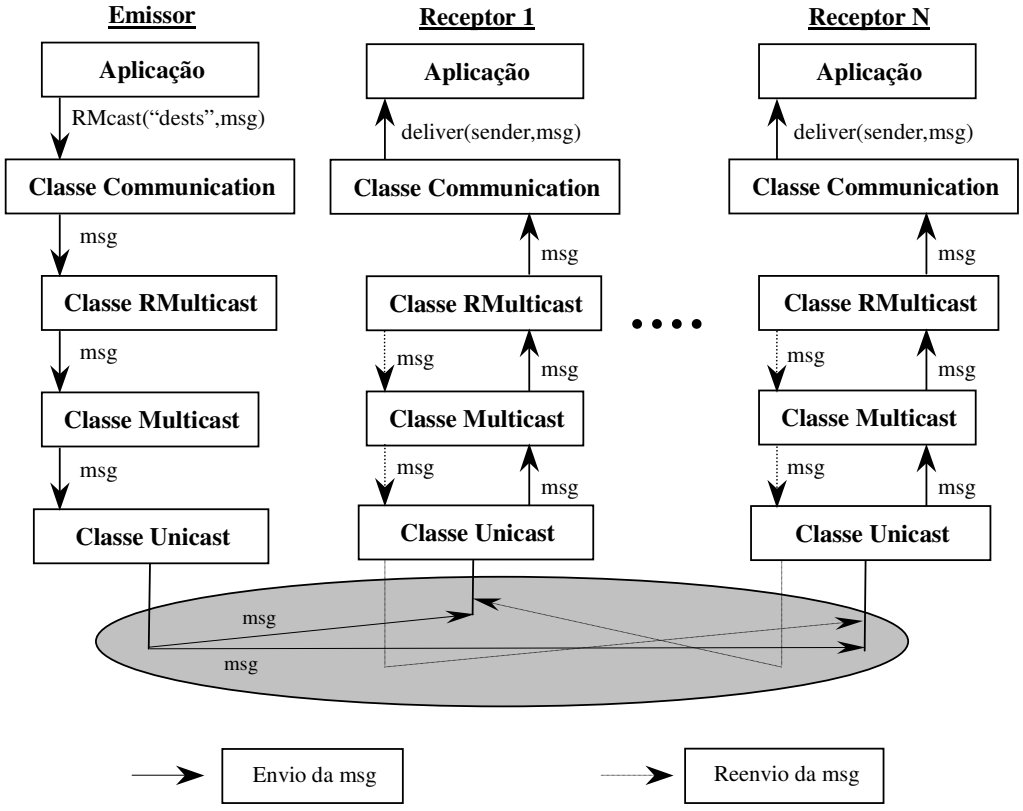

FIGURA 6.28 - Relacionamento entre camadas utilizando a classe Communication

Não foi ilustrado o relacionamento da classe **`Communication`** com o servidor de nomes. É importante observar que a classe **`Communication`** precisa obter a referência relativa ao grupo ou objeto destinatário da mensagem no servidor de nomes, pois a classe **`RMulticast`** não é capaz de compreender um nome como referência a um destinatário. Desta forma, antes que a mensagem chegue a **`Rmulticast`**, a classe **`Communication`** estabelece uma "conversa" com o servidor de nomes e obtém as informações necessárias para o envio da mensagem.

7 Conclusões

A área da computação evolui de forma muito rápida. Inicialmente, os sistemas de computadores eram altamente centralizados, hoje a interligação dos computadores através de uma rede é uma necessidade; as empresas precisam compartilhar periféricos e dados. Neste contexto, envolvendo serviços distribuídos e aplicações distribuídas, surgiram sistemas distribuídos.

A construção de sistemas distribuídos é motivada por aspectos como disponibilidade, distribuição, compartilhamento de recursos e tolerância a falhas. Estas características, entretanto, não são simples de serem obtidas. As aplicações distribuídas possuem requisitos funcionais diferentes das aplicações projetadas para sistemas centralizados. Estes requisitos as tornam mais difíceis de serem projetadas e implementadas do que as aplicações centralizadas. A complexidade de implementação é decorrente, principalmente, da dificuldade de tratamento e de gerência dos mecanismos de comunicação, exigindo uma equipe de programadores experientes.

Se há complexidade de implementação de aplicações distribuídas, é preciso que sejam disponibilizados mecanismos facilitadores. Neste sentido, muita pesquisa foi e continua sendo feita. A área de tolerância a falhas, por exemplo, tem mostrado que a implementação de diversas aplicações presentes em nosso dia-a-dia (por exemplo, sistemas de controle de tráfego aéreo, sistemas de monitoramento de pacientes, sistemas de direcionamento de mísseis, automação bancária, etc.) é complexa e possui diferentes exigências.

Observa-se que, na implementação de aplicações distribuídas, a comunicação é ponto chave. No caso de aplicações tolerantes a falhas, comunicação confiável constitui-se em um dos blocos básicos para a construção de outros serviços.

Paralelamente à evolução dos sistemas distribuídos e ao surgimento da área de tolerância a falhas, foi possível observar também a evolução das linguagens de programação, até o surgimento do paradigma da orientação a objetos. O sucesso da orientação a objetos se deve, provavelmente, à habilidade em modelar o domínio da aplicação ao invés da arquitetura da máquina em questão (o que é feito no enfoque imperativo) ou mapear conceitos matemáticos (conforme o enfoque funcional).

Pesquisas têm demonstrado que a orientação a objetos apresenta-se como um modelo promissor para o desenvolvimento de aplicações distribuídas tolerantes a falhas e modulares. A razão decorre de características inerentes ao próprio modelo de objetos, que facilita o controle da complexidade do sistema, pois promove uma melhor estruturação de seus componentes, e também permite que componentes já verificados/validados sejam reutilizados.

Esta dissertação foi motivada pela constatação da necessidade latente de mecanismos que facilitem a programação, em conjunto com a observação de que a comunicação do tipo ponto-a-ponto e do tipo multiponto (nas modalidades não-

confiável e confiável) se constitui um bloco de construção básico. Desta forma, foram implementados serviços fornecendo comunicação não-confiável e confiável, os quais podem ser utilizados na construção de aplicações distribuídas ou como blocos básicos na construção de outros serviços.

A linguagem Java foi escolhida para a implementação por ser, comparativamente às outras linguagens, mais vantajosa. Outro fator de peso na escolha por Java é que ela tem sido bastante utilizada no mercado de *software*. Esta característica particular ajuda a garantir atualizações e correções sobre a linguagem. Além disso, estão surgindo pesquisas no sentido de melhorar o desempenho de Java, já que este aspecto tem pesado negativamente sobre a linguagem. Estas pesquisas apontam para uma mudança de paradigma na área de linguagens de programação, particularmente com relação aos compiladores. Esforços estão sendo colocados no sentido de se conseguir uma compilação em tempo de execução. Assim, Java poderá obter um desempenho comparativamente melhor do que outras linguagens orientadas a objetos, como C++.

As classes desenvolvidas implementam *unicast* (não-confiável e confiável) e *multicast* (não-confiável e confiável). Além destes serviços de comunicação básicos, foram desenvolvidas classes que permitem referenciar os participantes da comunicação através de nomes. O pacote de classes compõe um *framework*, com as classes implementando uma interface que fornece um mecanismo de *callback* para o programador. Este tipo de mecanismo facilita muito a programação, pois é possível utilizá-lo para que objetos sejam avisados quando ocorrer um determinado evento como, por exemplo, a chegada de uma mensagem.

Foram implementados pequenos trechos de código utilizando e testando a funcionalidade de cada uma das classes de comunicação propostas. Observou-se que as classes de comunicação básicas, onde os destinatários são referenciados por endereço IP/porta, são mais adequadas à implementação de outros serviços, enquanto as classes que utilizam nomes como referência são mais adequadas ao desenvolvimento de aplicações. Estes testes demonstraram a facilidade de utilização das classes para a construção de aplicações ou para a composição de novos serviços. A validação dos protocolos foi feita através da injeção de falhas de *software* específicas, relacionadas à funcionalidade de cada protocolo.

Acredita-se que, com o que foi descrito na dissertação, seja possível ao programador adquirir o conhecimento básico necessário para a utilização do pacote de classes proposto, seja para a criação de aplicações finais ou para a criação de outras classes ou serviços.

7.1 Trabalhos Futuros

É de interesse que seja feita uma avaliação de desempenho das classes propostas, envolvendo fatores como:

- a influência do número de objetos existentes, se com o acréscimo de objetos no sistema, a comunicação fica mais lenta e se existe um limite neste número;
- o comportamento dos protocolos a medida que o número de mensagens aumenta; e
- um comparativo entre as classes, com relação ao número de objetos e ao número de mensagens.

A avaliação de desempenho pode ser direcionada de acordo com diferentes categorias ou domínios de aplicação.

Os defeitos nos nodos e nos canais de comunicação são contornados utilizando-se temporizadores, retransmissões de mensagens e números de seqüência anexos às mensagens. O desempenho dos protocolos propostos podem ser afetados pelos valores adotados para os temporizadores. Um valor alto para um temporizador pode ocasionar uma detecção de defeitos lenta; um valor baixo, pode resultar em uma detecção imprecisa. Portanto, outro trabalho futuro estaria envolvido na investigação dos valores apropriados para os temporizadores.

Um trabalho de interesse, também, seria a extensão dos protocolos propostos visando a interconectividade de redes locais, o que provavelmente envolveria a implementação de um protocolo para o roteamento de mensagens.

A implementação de outras classes de comunicação também é de interesse, utilizando enfoques de confiabilidade que não foram adotados nesta dissertação, tais como reconhecimentos positivos, reconhecimentos negativos, a combinação de ambos, ou mesmo um enfoque hierárquico. Após, poderia ser feito um comparativo destas implementações.

A ordenação de mensagens apresenta um caráter bastante simplificado nesta dissertação, foi implementada apenas a ordem com relação ao emissor (ordem FIFO). Outros tipos de ordenação, como causal e total, são de interesse para trabalhos futuros.

A implementação de protocolos que utilizem reconhecimentos positivos ou negativos, requer mecanismos para a gerência de *buffers* e, conseqüentemente, para a detecção de mensagens estáveis, ou seja, ter conhecimento quando mensagens podem ser removidas dos *buffers*.

A busca por eficiência levaria a trabalhos futuros visando o desenvolvimento de protocolos para o controle de fluxo e para o controle de congestionamento, bem como no sentido de um mecanismo que possibilite a fragmentação de mensagens.

Com relação à tolerância a falhas, poderia ser definido e implementado um detector de defeitos e a conseqüente integração do mesmo com as classes de comunicação. O servidor de nomes implementado tem um enfoque centralizado e sem tolerância a falhas. Neste caso, um trabalho futuro seria a implementação de um servidor de nomes tolerante a falhas. Outro trabalho futuro seria a implementação de um mecanismo para a gerência dos membros de um grupo (*membership*), bem como para o controle de visões, conforme a semântica dos sistemas de comunicação em grupo.

Anexo 1 Código Fonte

Unicast.java

```
package COM;

import java.net.*;
import java.io.*;
import java.util.StringTokenizer;
import java.util.LinkedList;

public class Unicast {
  DatagramSocket UDPsocket;
  int            UDPport;
  DataHandler    handler;
  Buffer         UcastBuf;
  SequenceNumber SeqNo,MC2SeqNo;
  String         ObjName;

   public Unicast(DataHandler handler,String ObjName) throws IOException {
    this.handler = handler;
    this.ObjName = ObjName;
    this.UDPsocket = new DatagramSocket();
    this.UDPport = UDPsocket.getLocalPort();
    this.UcastBuf = new Buffer();
    this.SeqNo = new SequenceNumber();
    this.MC2SeqNo = new SequenceNumber();

    new Receive();
    new ProcessUnicast();
  }

   public boolean Send(InetAddress ip,int port,String msg) {
    byte[] buffer;
    DatagramPacket packet;
    DatagramSocket socket;
    String message = Refs.UNICAST+" "+ObjName+" "+msg+" ";
    try {
      buffer = message.getBytes();
      packet = new DatagramPacket(buffer,buffer.length,ip,port);
      socket = new DatagramSocket();
      socket.send(packet);
      socket.close();
    }
    catch (java.io.IOException e) {
      System.err.println("Exception in Send method of Unicast class: " + e);
      return(false);
    }
    return(true);
  }


public class Receive extends Thread {
    Receive() {
      this.start();
    }

    public void run() {
      byte[] buf;
      DatagramPacket packet;
      String msg;

      while(true) {
        try {
          buf = new byte[Refs.PACKETsize];
```

```
        packet = new DatagramPacket(buf,buf.length);
        UDPsocket.receive(packet);
        msg = new String(buf);
        UcastBuf.insert(msg);
      }
      catch(IOException io) {
        System.out.println("Error in Receive Thread");
      }
    }
  }
}

public class ProcessUnicast extends Thread {

  ProcessUnicast() {
    this.start();
  }

  public void run() {
    while(true) {
      String message = null;
      String strIP;
      InetAddress senderIP = null;

      do { message = UcastBuf.remove();
      } while(message == null);

      StringTokenizer st = new StringTokenizer(message);
      String msg_type = st.nextToken();

      //if message is UNICAST
      if (msg_type.equals(Refs.UNICAST)) {
        String senderName = st.nextToken();
        String msg="";
        String tmp="";
        while(true) {
          tmp=st.nextToken();
          if (st.hasMoreTokens()) msg = msg+" "+tmp;
          else break;
        }
        handler.deliver(senderName,msg);
      }//if Refs.UNICAST message
      else
      //if message is RELIABLE UNICAST
      if (msg_type.equals(Refs.RUNICAST)) {
        int ACKport = (Integer.valueOf(st.nextToken())).intValue();
        String senderName = st.nextToken();
        strIP = st.nextToken();
        try { senderIP = InetAddress.getByName(strIP);  }
        catch(UnknownHostException uh) { }
        int senderPort =(Integer.valueOf
                         (st.nextToken())).intValue();
        int msgSeqNo = (Integer.valueOf(st.nextToken())).intValue();
        String msg="";
        String tmp="";
        while(true) {
          tmp=st.nextToken();
          if (st.hasMoreTokens()) msg = msg+" "+tmp;
          else break;
        }
        try {
          //send ACK to message sender
          DatagramSocket ACKsocket = new DatagramSocket();
          tmp = Refs.ACK+" ";
          byte[] ACKbuffer = tmp.getBytes();
          DatagramPacket ACKpacket = new DatagramPacket
                        (ACKbuffer,ACKbuffer.length,senderIP,ACKport);
          ACKsocket.send(ACKpacket);
```

```
    ACKsocket.close();
  } catch(IOException io) { }

   int lastSeqNo = SeqNo.lookupReceiveSeqNo(strIP,senderPort);
  //is it the next expected SeqNo?
  if (msgSeqNo == lastSeqNo + 1) {
    SeqNo.nextReceiveSeqNo(strIP,senderPort);
    handler.deliver(senderName,msg);
  }
  else
  //Is SeqNo > the next expected?
  if (msgSeqNo > lastSeqNo + 1) {
    UcastBuf.insert(message);
  }
}//if Refs.RUNICAST message
else
//if message is RELIABLE MULTICAST message (method unicast)
if (msg_type.equals(Refs.RMCAST_UNICAST)) {
  int delimSize = (Refs.DELIM).length();
  int typeSize = (Refs.RMCAST_UNICAST).length()+delimSize+1;
  String tmp = message.substring(typeSize,message.length());
  String destStr = tmp.substring(0,tmp.indexOf(Refs.DELIM));
  String t = tmp.substring(destStr.length(),tmp.length());
  StringTokenizer StrTok = new StringTokenizer(t);
  StrTok.nextToken();
  String senderName = StrTok.nextToken();
  strIP = StrTok.nextToken();
  try { senderIP = InetAddress.getByName(strIP);  }
  catch(UnknownHostException uh) { }
  int senderPort = (Integer.valueOf(StrTok.nextToken())).intValue();
  int msgSeqNo = (Integer.valueOf(StrTok.nextToken())).intValue();
   //get complete message
  String str_tmp = StrTok.nextToken();
  String msg = str_tmp;
  while(true) {
    str_tmp = StrTok.nextToken();
    if (StrTok.hasMoreTokens()) msg = msg+" "+str_tmp;
    else break;
  }//while
  int lastSeqNo = MC2SeqNo.lookupReceiveSeqNo(strIP,senderPort);
  if (msgSeqNo == lastSeqNo + 1) {
    //resends the same message to group
    String m = Refs.RMCAST_UNICAST+" "+Refs.DELIM+
                destStr+Refs.DELIM+" "+senderName+" "+strIP+" "+
                senderPort+" "+msgSeqNo+" "+msg+" ";
    byte[] buf = m.getBytes();
    DatagramPacket packet;
    Member mbr;
    DatagramSocket socket=null;
    try { socket = new DatagramSocket(); }
    catch(SocketException se) {
      System.out.println("Error in create socket
                                (RMCAST_UNICAST)");
    }
    Serial s = new Serial();
    LinkedList destObjs = s.de_serialize(destStr);
    for (int i=0; i<destObjs.size(); i++) {
      mbr = (Member)destObjs.get(i);
     try {
        InetAddress ip = InetAddress.getByName(mbr.IP);
        packet = new DatagramPacket(buf,buf.length,ip,mbr.UDPport);
        socket.send(packet);
      }
      catch(IOException e) {
         System.out.println("I/O error in send (RMCAST_UNICAST)");
      }
    }//for
    socket.close();
```

```
            MC2SeqNo.nextReceiveSeqNo(strIP,senderPort);
            handler.deliver(senderName,msg);
          }//if hasn't previously executed resend and deliver
          else
          //Is MC2SeqNo > the next expected?
          if (msgSeqNo > lastSeqNo + 1) {
            UcastBuf.insert(message);
            System.out.println("msgSeqNo: "+msgSeqNo+">lastSeqNo+1:"+
                               (lastSeqNo+1));
          }
        }//if RMCAST_UNICAST
      }//while
    }//run method of thread
  }//ProcessUnicast class

}//Unicast Class
```

RUnicast.java

```
package COM;

import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

public class RUnicast extends Unicast {

  public RUnicast(DataHandler handler,String ObjName) throws IOException {
    super(handler,ObjName);
  }

  public boolean Send(InetAddress ip,int port,String msg) {
    byte[] Sbuffer,Rbuffer;
    DatagramPacket Spacket=null,Rpacket;
    DatagramSocket socket;

    Rbuffer = new byte[Refs.PACKETsize];
    Rpacket = new DatagramPacket(Rbuffer,Rbuffer.length);

    try {
      //create a socket to send the message and
      //to receive ACK that sent message
      socket = new DatagramSocket();
      int ACKport = socket.getLocalPort();
      int msgSeqNo = SeqNo.nextSendSeqNo(ip.getHostAddress(),port);
      System.out.println(ObjName+" sends RUNICAST no. "+msgSeqNo+
                         " to "+ip.getHostAddress()+"/"+port);
      String m = Refs.RUNICAST+" "+ACKport+" "+ObjName+" "+
                 InetAddress.getLocalHost().getHostAddress()+" "+
                 UDPport+" "+msgSeqNo+" "+msg+" ";
      Sbuffer = m.getBytes();
      Spacket = new DatagramPacket(Sbuffer,Sbuffer.length,ip,port);

      // Send the message until RESEND_LIMIT or receive ACK
      for(int i=0; i<Refs.RESEND_LIMIT; i++) {
        socket.send(Spacket);
        socket.setSoTimeout(Refs.TIMEOUT);
        try {
          socket.receive(Rpacket);
          StringTokenizer st = new StringTokenizer(new String(Rbuffer));
          if (st.nextToken().equals(Refs.ACK)) {
            System.out.println("ACK Received");
            socket.close();
```

```
            return(true);
          }
        }
        catch(InterruptedIOException ie) {
          System.out.println("Timeout waiting for receiver ACK");
        }
      }//for
      socket.close();
    }
    catch (java.io.IOException e) {
      System.out.println("I/O Error in Send method of RUnicast class");
      return(false);
    }

  System.out.println("There is a failure at the receiver address");
    return(false);
  }
}//Reliable Unicast class
```

Multicast.java

```
package COM;

import java.net.*;
import java.io.*;
import java.util.StringTokenizer;

public class Multicast extends Unicast {
  public MulticastSocket MCsocket;
  public int             MCport;
  public Buffer          McastBuf;
  public SequenceNumber  MC1SeqNo;

  public Multicast(DataHandler handler,String ObjName) throws IOException {
    super(handler,ObjName);
    this.McastBuf = new Buffer();
    this.MC1SeqNo = new SequenceNumber();
  }//Multicast

  public Multicast(DataHandler handler) throws IOException {
    super(handler);
    this.McastBuf = new Buffer();
    this.MC1SeqNo = new SequenceNumber();
  }//Multicast

  public boolean SendMcast(int dest,String msg) {
    DatagramPacket packet;
    DatagramSocket socket;
    String m = Refs.MULTICAST+" "+ObjName+" "+dest+" "+msg+" ";
    byte[] buffer = m.getBytes();
    try {
      packet = new DatagramPacket(buffer,buffer.length,
                                  Refs.IP(Refs.GROUPSip),dest);
      socket = new DatagramSocket();
      socket.send(packet);
      socket.close();
    }
    catch(UnknownHostException uh) { }
    catch(java.io.IOException e) {
      System.out.println("I/O Error in SendMcast method of Multicast class");
      return(false);
    }
    return(true);
```

```
}//SendMcast

public void join(int groupID) {
  try {
    MCsocket = new MulticastSocket(groupID);
  }
  catch(IOException io) {
    System.out.println("I/O Error in join method of Multicast class");
  }

  try {
    MCsocket.joinGroup(Refs.IP(Refs.GROUPSip));
  }
  catch(UnknownHostException uh) {
    System.out.println("Unknown Host error in join method of Multicast");
  }
  catch(IOException io) {
    System.out.println("I/O Error in join method of Multicast class");
  }
  new MCReceive();
  new ProcessMulticast();

}//join

public class MCReceive extends Thread {
  MCReceive() {
    this.start();
  }

  public void run() {
    byte[] buf = null;
    DatagramPacket packet = null;
    InetAddress senderIP = null;
    String message;
    while(true) {
      if (MCsocket != null) {
      //wait for messages
        try {
          buf = new byte[Refs.PACKETsize];
          packet = new DatagramPacket(buf,buf.length);
          MCsocket.receive(packet);
          message = new String(buf);
          McastBuf.insert(message); //buffer received message
        }
        catch(IOException io) {
          System.out.println("I/O error in MCReceive thread");
        }
      }//if
    }//while
  }//run
}//MCReceive class


public class ProcessMulticast extends Thread {
  ProcessMulticast() {
    this.start();
  }
  public void run() {
    while(true) {
      String message = null;
      do {
        message = McastBuf.remove();
        try {
          if (message == null) Thread.sleep(200);
        }
        catch(InterruptedException ie) { }
      } while(message == null);
```

```
  StringTokenizer st = new StringTokenizer(message);
  String msg_type = st.nextToken();

  //if received MULTICAST message
  if (msg_type.equals(Refs.MULTICAST)) {
    String senderName = st.nextToken();
    int destID = (Integer.valueOf(st.nextToken())).intValue();
    //get complete message
    String tmp = st.nextToken();
    String msg = tmp;
    while(true) {
      tmp = st.nextToken();
      if (st.hasMoreTokens()) msg = msg+" "+tmp;
      else break;
    }
    handler.deliver(senderName,msg);
  }//if Refs.MULTICAST message
  else
  //if received RELIABLE MULTICAST message (method multicast)
  if (msg_type.equals(Refs.RMCAST_MULTICAST)) {
    String senderName = st.nextToken();
    String strIP = st.nextToken();
    InetAddress senderIP = null;
    try {
      senderIP = InetAddress.getByName(strIP);
    }
    catch(UnknownHostException uh) { }
    int senderPort = (Integer.valueOf(st.nextToken())).intValue();
    int destID = (Integer.valueOf(st.nextToken())).intValue();
    int msgSeqNo = (Integer.valueOf(st.nextToken())).intValue();
    //get complete message
    String tmp = st.nextToken();
    String msg = tmp;
    while(true) {
      tmp = st.nextToken();
      if (st.hasMoreTokens()) msg = msg+" "+tmp;
      else break;
    }
    //if hasn't previously executed resend and deliver
    int lastSeqNo = MC1SeqNo.lookupReceiveSeqNo(strIP,senderPort);
    if (msgSeqNo == lastSeqNo + 1) {
      //resends the same message to group
      //String m = msgSeqNo+" "+msg+" ";
      DatagramPacket packet;
      DatagramSocket socket;
      String m2 = Refs.RMCAST_MULTICAST+" "+senderName+" "+strIP+" "+
                  senderPort+" "+destID+" "+msgSeqNo+" "+msg+" ";
      byte[] buffer = m2.getBytes();
      try {
        packet = new DatagramPacket(buffer,buffer.length,
                                    Refs.IP(Refs.GROUPSip),destID);
        socket = new DatagramSocket();
        socket.send(packet);
        socket.close();
      }catch(UnknownHostException uh) { }
      catch(java.io.IOException e) {
       System.out.println("I/O Error in resend of ProcessMulticast");
      }
      MC1SeqNo.nextReceiveSeqNo(strIP,senderPort);
      handler.deliver(senderName,msg);
    }//if hasn't previously executed resend and deliver
    else
    //Is SeqNo > the next expected?
    if (msgSeqNo > lastSeqNo + 1) {
      McastBuf.insert(message);
    }
  }//if Refs.RMCAST_MULTICAST message
}//while block waiting for messages
```

```
    }//run method of thread
  }//ProcessMulticast class

}//Multicast Class
```

RMulticast.java

```
package COM;

import java.net.*;
import java.io.*;
import java.util.LinkedList;

public class RMulticast extends Multicast {
  public RMulticast(DataHandler handler,String ObjName) throws IOException {
    super(handler,ObjName);
  }//RMulticast

  public RMulticast(DataHandler handler) throws IOException {
    super(handler);
  }//RMulticast

  public boolean RSendUcast(int destPort,LinkedList destObjs,String msg) {
    int msgSeqNo = MC2SeqNo.nextSendSeqNo(Refs.GROUPSip,destPort);
    String m = null;
    Serial s = new Serial();
    try {
       m = Refs.RMCAST_UNICAST+" "+Refs.DELIM+s.serialize(destObjs)+
       Refs.DELIM+" "+ObjName+" "+InetAddress.getLocalHost().getHostAddress()+
       " "+UDPport+" "+msgSeqNo+" "+msg+" ";
    }
    catch(UnknownHostException uh) { }
    byte[] buf = m.getBytes();
    DatagramPacket packet;
    Member mbr;
    DatagramSocket socket=null;
    try {
      socket = new DatagramSocket();
    }
    catch(SocketException se) {
      System.out.println("Error in create socket (RMCAST_UNICAST)");
    }

    for (int i=0; i<destObjs.size(); i++) {
      mbr = (Member)destObjs.get(i);
      //System.out.println("Enviando mensagem para: "+mbr.name);
      try {
        InetAddress ip = InetAddress.getByName(mbr.IP);
        packet = new DatagramPacket(buf,buf.length,ip,mbr.UDPport);
        socket.send(packet);
      }
      catch(IOException e) {
        System.out.println("I/O error in send (RMCAST_UNICAST)");
        return(false);
      }
    }//for
    socket.close();
    return(true);
  }//RSendUcast

  public boolean RSendMcast(int destID,String msg) {
    DatagramPacket packet;
    DatagramSocket socket;
```

```
    int msgSeqNo = MC1SeqNo.nextSendSeqNo(Refs.GROUPSip,destID);
    String message = null;
    try {
      message = Refs.RMCAST_MULTICAST+" "+ObjName+" "+
                InetAddress.getLocalHost().getHostAddress()+" "+
                UDPport+" "+destID+" "+msgSeqNo+" "+msg+" ";
    }
    catch(UnknownHostException uh) { }
    byte[] buffer = message.getBytes();
    try {
      packet = new DatagramPacket(buffer,buffer.length,
               Refs.IP(Refs.GROUPSip),destID);
      socket = new DatagramSocket();
      socket.send(packet);
      socket.close();
    }
    catch(UnknownHostException uh) { }
    catch(java.io.IOException e) {
      System.out.println("I/O Error in Send method of Multicast class");
      return(false);
    }
    return(true);
  }//RSendMcast

}//RMulticast Class
```

DataHandler.java

```
package COM;

public interface DataHandler {
  public void deliver(String sender,String msg);
}
```

Buffer.java

```
package COM;

import java.util.LinkedList;

public class Buffer {
  LinkedList buf;

  public Buffer() {
    this.buf = new LinkedList();
  }

  public synchronized void insert(String msg) {
    buf.add(msg);
  }

  public synchronized String remove() {
    String str = null;
    if (buf.size() > 0) {
      str = (String) buf.get(0);
      buf.remove(0);
    }
    return(str);
  }

}//Buffer Class
```

SequenceNumber.java

```
package COM;

class SequenceNumber {
    Node head;
    Node tail;
    int size;

    SequenceNumber() {
       super();
       head = null;
       tail = null;
       size = 0;
    }

    synchronized void enqueue(Node n){
       if(head == null){  //if the list is empty
         n.next = null;
         n.previous = null;
         head = n;
         tail = n;
       }
       else {  //if the list is not empty
         n.next = null;
         n.previous = tail;
         tail.next = n;
         tail = n;
       }
       size++;
    }//enqueue

    synchronized void dequeue() {
       if(head != null){
         head = head.next;
         size--;
         if(head != null){ //this was not the only element in the queue
           head.previous = null;
         }
         else {
           tail = head;  //there was only one element
         }
       }
    }//dequeue

    synchronized boolean removeNode(Node n){
       for(Node i=head; i!=null; i=i.next){
         if(i==n){
           //remove this node from the list
           if(i.previous != null){
             //not the head
             if(i.next != null){
               //not a tail
              (i.previous).next = i.next;
              (i.next).previous = i.previous;
             }
             else{
              //not a head but a tail
              (i.previous).next = i.next;
              tail = i.previous;
             }
           }
           else{
             //is a head
             if(i.next != null){
               //not a tail
               (i.next).previous = i.previous;
               head = i.next;
```

```
        }
        else{
          //is a head and a tail
          head = i.next;
          tail = i.previous;
        }
      }
      i.previous = null;
      i.next = null;
      return true;
    }
   }
    return false;  //did not find the node in the linked list
}//removeNode

synchronized Node lookupNode(String ipaddress,int port) {
  for(Node n=head; n!=null; n=n.next) {
   if(n.ipaddress.equals(ipaddress) && (n.port == port)) {
     return(n);
   }
  }
  return null; //node not found in the list
}//lookupNode

synchronized void createNewNode(String host_ipaddress,int host_port) {
  Node newNode = new Node(host_ipaddress,host_port,-1,-1);
  enqueue(newNode);
}//createNewNode

synchronized int lookupSendSeqNo(String ipaddress,int port) {
  for(Node n=head; n!=null; n=n.next) {
   if(n.ipaddress.equals(ipaddress) && (n.port == port)) {
       return(n.sendSeqNo);
   }
  }
  return(-1);
}//lookupSendSeqNo

synchronized int lookupReceiveSeqNo(String ipaddress,int port) {
  for(Node n=head; n!=null; n=n.next) {
   if(n.ipaddress.equals(ipaddress) && (n.port == port)) {
     return(n.receiveSeqNo);
   }
  }
  return(-1);
}//lookupReceiveSeqNo

synchronized int nextSendSeqNo(String hostIPAddress,int hostPort) {
  for(Node n=head; n!=null; n=n.next){
   if(n.ipaddress.equals(hostIPAddress) && (n.port == hostPort)) {
     n.sendSeqNo++;
     return(n.sendSeqNo);
   }
  }
  Node newNode = new Node(hostIPAddress,hostPort,0,-1);
  enqueue(newNode);
  return(0);  //since this is a new Node therefore the seqNo is 0
}//nextSendSequenceNumber

synchronized int nextReceiveSeqNo(String hostIPAddress,int hostPort) {
  for(Node n=head; n!=null; n=n.next) {
   if(n.ipaddress.equals(hostIPAddress) && (n.port == hostPort)) {
     n.receiveSeqNo++;
     return(n.receiveSeqNo);
   }
  }
  Node newNode = new Node(hostIPAddress,hostPort,-1,0);
  enqueue(newNode);
```

```
      return(0);  //since this is a new Node therefore the seqNo is 0
    }//nextReceiveSequenceNumber

}//SequenceNumber class

class Node {
  String ipaddress;  //key = ipaddress and port
  int port;
  int sendSeqNo;
  int receiveSeqNo;

  Node next;  //its a doubly linked list
  Node previous;

  Node(String ipaddress,int port,int sendSeqNo,int receiveSeqNo) {
    this.ipaddress = ipaddress;
    this.port = port;
    this.sendSeqNo = sendSeqNo;
    this.receiveSeqNo = receiveSeqNo;
  }//Node constructor

  Node() {
    super();
    sendSeqNo = -1;
    receiveSeqNo = -1;
  }//Node constructor

}//Node class
```

Refs.java

```
package COM;

import java.net.InetAddress;

public class Refs {
  public static final String NAMESERVERip = "227.228.229.230";
  public static final int    NAMESERVERport = 12345;
  public static final String GROUPSip = "230.230.230.230";
  public static final int    PACKETsize = 8192;
  public static final String RegisterOBJ = "Register_Object";
  public static final String SEARCH_OBJ = "Search_for_Object";
  public static final String SEARCH_GRP = "Search_for_Group";
  public static final String SEARCH_GRP_MBR = "Search_for_Group_Objects";
  public static final String SEARCH_SEND = "Search_Dest_of_Send";
  public static final String UNICAST = "Unreliable_Unicast_Message";
  public static final String RUNICAST = "Reliable_Unicast_Message";
  public static final String MULTICAST = "Unreliable_Multicast_Message";
  public static final String RMCAST_MULTICAST = "multicast";
  public static final String RMCAST_UNICAST = "unicast";
  public static final String JOIN = "Join_to_Group";
  public static final String LEAVE = "Leave_Group";
  public static final int    RESEND_LIMIT = 5;
  public static final int    TIMEOUT = 5000;
  public static final String ACK = "Reliable_Unicast_Message_ACK";
  public static final String OBJ_NOT_EXISTS = "Object_Name_Not_Exists";
  public static final String GRP_NOT_EXISTS = "Group_Name_Not_Exists";
  public static final String DELIM = "Delimitador";

  static public InetAddress IP(String address) throws
                                          java.net.UnknownHostException {
    return(InetAddress.getByName(address));
  }
}//Refs Class
```

Member.java

```
package COM;

public class Member implements java.io.Serializable {
  public String name=null;
  public String IP=null;
  public int UDPport;

  public Member(String name,String IP,int UDPport) {
    this.name = name;
    this.IP = IP;
    this.UDPport = UDPport;
  }

}//Member Class
```

Group.java

```
package COM;

import java.net.InetAddress;
import java.util.LinkedList;

public class Group implements java.io.Serializable {
  String name;
  InetAddress IP;
  int port;
  LinkedList mbr_list;

  public Group(String name,InetAddress IP,int port) {
    this.name = name;
    this.IP = IP;
    this.port = port;
    this.mbr_list = new LinkedList();
  }

  public synchronized void join(Member mbr) {
    mbr_list.add(mbr);
  }

  public synchronized void leave(Member mbr) {
    int r = index(mbr.name);
    if (r!=-1) mbr_list.remove(r);
  }

  public synchronized int index(String name) {
    Member mbr;
    for(int c=0; c<mbr_list.size(); c++) {
      mbr = (Member)mbr_list.get(c);
      if ((mbr.name).equals(name)) return c;
    }
    return -1;
  }

}//Group Class
```

Register.java

```
package COM;

import java.util.*;
import java.net.*;

public class Register {
   public  LinkedList obj_list;
   public  LinkedList grp_list;
   static  int GroupID = 3456;

   public Register() {
     obj_list = new LinkedList();
     grp_list = new LinkedList();
   }

   public synchronized boolean insert(String name,String IP,int UDPport) {
     if ((searchObj(name).name).equals(Refs.OBJ_NOT_EXISTS)) {
       Member mbr = new Member(name,IP,UDPport);
       obj_list.add(mbr);
       return(true);
     }
     else return(false);
   }

   public synchronized boolean delete(String name) {
     int r = objIndex(name);
     if (r!=-1) {
       obj_list.remove(r);
       return(true);
     }
     else return(false);
   }

   public synchronized Member searchObj(String name) {
     int r = objIndex(name);
     if (r!=-1) return((Member)obj_list.get(r));
     else {
      Member m = null;
      try {
       m = new Member(Refs.OBJ_NOT_EXISTS,
              InetAddress.getLocalHost().getHostAddress(),7777);
      }
      catch(UnknownHostException uh) { }
      return(m);
     }
   }

   public synchronized int objIndex(String name) {
     Member mbr;
     for(int c=0; c<obj_list.size(); c++) {
       mbr = (Member)obj_list.get(c);
       if ((mbr.name).equals(name)) return(c);
     }
     return(-1);
   }

   public synchronized void displayObjs() {
     Member mbr;
     System.out.println("Registered Objects [Name,IP Address,UDP port]:");
     for(int c=0; c<obj_list.size(); c++) {
       mbr = (Member)obj_list.get(c);
       System.out.println("["+mbr.name+","+mbr.IP+","+mbr.UDPport+"]");
     }
   }
```

```
public synchronized String searchObjType(String name) {
  int r = objIndex(name);
  if (r!=-1) return("object");
  else {
    r = grpIndex(name);
    if (r!=-1) return("group");
    else return(null);
  }
}

public synchronized void join(String grp_name,String obj_name) {
  Member mbr = searchObj(obj_name);
  Group  grp = searchGrp(grp_name);
  if ((grp.name).equals(Refs.GRP_NOT_EXISTS)) grp = create(grp_name);
  grp.join(mbr);
}

public synchronized void leave(String obj_name) {
  Group grp;
  Member mbr;
  for(int g=0; g<grp_list.size(); g++) {
    grp = (Group)grp_list.get(g);
    for(int m=0; m<grp.mbr_list.size(); m++) {
      mbr = (Member)grp.mbr_list.get(m);
      if ((mbr.name).equals(obj_name)) grp.leave(mbr);
    }
    if (grp.mbr_list.size()==0) grp_list.remove(g);
  }//for
  delete(obj_name);
}

public synchronized Group create(String grp_name) {
  try {
    GroupID = GroupID + 3;
    Group grp = new Group(grp_name,Refs.IP(Refs.GROUPSip),GroupID);
    grp_list.add(grp);
    return(grp);
  }
  catch(UnknownHostException uh) { return(null); }
}

public synchronized int GroupID(String name) {
  Group g = searchGrp(name);
  return(g.port);
}

public synchronized LinkedList GrpMbr(String name) {
  Group g = searchGrp(name);
  return(g.mbr_list);
}

public synchronized Group searchGrp(String name) {
  int r = grpIndex(name);
  if (r!=-1) {
    Group g = (Group) grp_list.get(r);
    return(g);
  }
  else {
    Group g = null;
    try {
      g = new Group(Refs.GRP_NOT_EXISTS,Refs.IP(Refs.GROUPSip),-1);
    }
    catch(UnknownHostException uh) { }
    return(g);
  }
}
```

```
   public synchronized int grpIndex(String name) {
     Group grp;
     for(int c=0; c<grp_list.size(); c++) {
       grp = (Group)grp_list.get(c);
       if ((grp.name).equals(name)) return c;
     }
     return -1;
   }

   public synchronized void displayGrps() {
     Group grp;
     Member mbr;
     System.out.print("Registered Groups [Group_Name,Group_ID]:");
     for(int g=0; g<grp_list.size(); g++) {
       grp = (Group)grp_list.get(g);
       System.out.println("");
       System.out.print("["+grp.name+","+grp.port+"]:");
       for(int m=0; m<grp.mbr_list.size(); m++) {
         mbr = (Member)grp.mbr_list.get(m);
         System.out.print(" " + mbr.name);
       }
     }//for
     System.out.println(" ");
     System.out.println("-------------------------------------------------");
   }

}//Register Class
```

Serial.java

```
package COM;

import java.util.LinkedList;
import java.io.*;

public class Serial {
  public String serialize (LinkedList l) {
     try {
         ByteArrayOutputStream bout = new ByteArrayOutputStream();
         ObjectOutputStream out = new ObjectOutputStream
                                     (new BufferedOutputStream(bout));
         out.writeObject(l);
         out.flush();
         out.close();

         byte ba[] = bout.toByteArray();
         String str = new String(ba);
         return(str);
      } catch (Exception e) { return(null); }
   }

  public LinkedList de_serialize(String str) {
      try {
         byte ba[] = str.getBytes();
         ByteArrayInputStream bin = new ByteArrayInputStream(ba);
         ObjectInputStream in = new ObjectInputStream
                                     (new BufferedInputStream(bin));
         LinkedList l2 = (LinkedList)in.readObject();
         in.close();
         return(l2);
      } catch (Exception e) { return(null); }
   }
}
```

Bibliografia

[AMA99] AMARAL, J. B. **Protocolos para o envio confiável de mensagens em *multicast***. Porto Alegre: PPGC-UFRGS, 1999. (TI-779).

[AMI92] AMIR, Y. et al. Transis: a communication sub-system for high availability. In: ANNUAL INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT COMPUTING, 22., 1992. **Proceedings...** [S.l.:s.n.], 1992. p.76-84.

[AMI95] AMIR, Y. et al. The Totem single-ring ordering and membership protocol. **ACM Transactions on Computer Systems**, New York, v.13, n.4, Nov. 1995.

[BAB94] BABAOGLU, O. et al. **Relacs**: a communication infrastructure for constructing reliable applications in large-scale distributed systems. Bologna: Laboratory of Computer Science, University of Bologna, 1994. (Technical Report UBLCS 94-15).

[BAN2001] BAN, B. **Design and implementation of a reliable group communication toolkit for Java**. Cornell University. Disponível em: <http://www.cs.cornell. edu/Info/Projects/JavaGroupsNew/papers> Acesso em: 6 mar. 2001.

[BAR94] BARCELOS, P. P. A. **Estudo e análise de protocolos de difusão confiável**. Porto Alegre: PPGC-UFRGS, 1994. (TI-370).

[BIG87] BIGGERSTAFF, T. J.; RICHTER, C. Reusability framework, assessment, and directions. **IEEE Software**, New York, v.4, n.1, p.41-49, Mar. 1987.

[BIR84] BIRREL, A. D.; NELSON, B. J. Implementing Remote Procedure Calls. **ACM Transactions on Computer Systems**, New York, v.2, n.1, p.39-59, 1984.

[BIR91] BIRMAN, K. P.; SCHIPER, A.; STEPHENSON, P. Lightweight causal and atomic group multicast. **ACM Transactions on Computer Systems**, New York, v.9, n.3, p.272-314, 1991.

[BIR96] BIRMAN, K. P. **Building secure and reliable network applications**. [S.l]: Manning Publishing Company and Prentice Hall, 1996.

[BOB88] BOBROW, D. et al. Common Lisp object system specification. **ACM SIGPLAN Notices**, New York, v.23, n.9, Sept. 1988.

[BUZ98] BUZATO, L. E.; RUBIRA, C. M. F. Construção de sistemas orientados a objetos confiáveis. In: ESCOLA DE COMPUTAÇÃO, 11., 1998, Rio de Janeiro. **Anais...** Rio de Janeiro: DCC\IM; COPPE Sistemas; NCE\UFRJ, 1998.

[CAS97] CASAD, J.; NEWLAND, D. **MCSE Training Guide**: networking essentials. Indianapolis: New Riders Publishing, 1997.

[CHA96] CHANDRA, T. D.; TOUEG, S. Unreliable failure detectors for reliable distributed systems. **Journal of the ACM**, New York, v.34, n.1, p.225-267, Mar. 1996.

[COM88] COMER, D. E. **Internetworking with TCP/IP**: principles, protocols, architecture. Stevenage: Prentice Hall, 1988.

[CRI91] CRISTIAN, F. Understanding fault tolerant distributed systems. **Communications of the ACM**, New York, v.34, n.2, p.57-78, Feb. 1991.

[CRI96] CRISTIAN, F. Synchronous and asynchronous group communication. **Communications of the ACM**, New York, v.39, n.4, p.88-97, Apr. 1996.

[DEE89] DEERING, S. **Host extension for IP multicasting**. [S.l.]:IETF, 1989. (Technical Report RFC-1112).

[ERI98] ERIKSSON, H.; PENKER, M. **UML Toolkit**. [S.l]: Wiley, 1998.

[EST2000] ESTEFANEL, L. A. **Detectores de defeitos não-confiáveis**. Porto Alegre: PPGC-UFRGS, 2000. (TI-808).

[EST2001] ESTEFANEL, L. A. **Avaliação dos detectores de defeitos e sua influência nas operações de consenso**. Porto Alegre: PPGC-UFRGS, 2001. Dissertação de Mestrado

[EZH95] EZHILCHELVAN, P. D.; MACEDO, A.; SHRIVASTAVA, S. K. Newtop: a fault tolerant group communication protocol. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED COMPUTING SYSTEMS, 15., 1995. **Proceedings...** [S.l:s.n], 1995.

[FIS85] FISCHER, M. J.; LYNCH, N. A.; PATERSON, M. S. Impossibility of distributed consensus with one faulty process. **Journal of the ACM**, New York, v.32, n.2, p.374-382, Apr. 1985.

[GAR98] GARBINATO, B. **Protocol objects and patterns for structuring reliable distributed systems**. [S.l.]: École Polytechnique Fédérale de Lausanne, 1998. Tese de Doutorado.

[GOL83] GOLDBERG, A.; ROBSON, D. **Smalltalk-80**: the language and its implementation. [S.l]:Addison-Wesley, 1983.

[GOS96] GOSLING, J.; JOY, B.; STEELE, G. **The Java language specification**. New York: Addison-Wesley, 1996.

[GUO98] GUO, K. H. **Scalable message stability detection protocols**. [S.l.]: Cornell University, 1998. Tese de Doutorado.

[HAD93] HADZILLACOS, V.; TOUEG, S. Fault Tolerant Broadcasts and Related Problems. In: MULLENDER, Sape (Ed.). **Distributed Systems**. 2nd ed. New York: ACM Press, 1993.

[HAD94] HADZILACOS, V.; TOUEG, S. **A modular approach to fault-tolerant broadcasts and related problems**. [S.l.]: Cornell University, Computer Science Department, May 1994. (Technical Report 94-1425).

[HAY96] HAYDEN, M., RENESSE, R.v. **Optimizing layered communication protocols**. [S.l.]: Cornell University, Computer Science Department, 1996. (Technical Report TR96-1613).

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

[JAN97] JANSCH-PÔRTO, I. E. S.; WEBER, T. S. Recuperação em sistemas distribuídos. In: JORNADA DE ATUALIZAÇÃO EM INFORMÁTICA, 16., 1997. **Proceedings...** Brasília: [s.n], 1997.

[JOH97] JOHNSON, R. E. Frameworks = (components + patterns). **Communications of ACM**, New York, v.40, p.39-42, Oct. 1997.

[KAR97] KARMARKAR, A. S. **A framework for communication support in object oriented distributed systems**. [S.l.]: Texas A&M University, 1997. Tese de Doutorado.

[LAM78] LAMPORT, L. Time, clocks and the ordering of events in a distributed system. **Communications of the ACM**, New York, v.21, p.558-565, July 1978.

[MAL95] MALLOTH, C. P. et al. Phoenix: a toolkit for building fault-tolerant distributed applications in large scale. In: WORKSHOP ON PARALLEL AND DISTRIBUTED PLATFORMS IN INDUSTRIAL PRODUCTS, 1995. **Proceedings...** [S.l:s.n], 1995.

[MAR95] MARSHALL, V. A. **WWW – The World Wide Web**. Berlin: Springer-Verlag, 1995. (Lecture Notes in Computer Science, v.1012).

[MEY88] MEYER, B. **Object-oriented software construction**. [S.l]: Prentice-Hall, 1988.

[MIS91] MISHRA, S.; PETERSON, L. L.; SCHLICHTING, R. L. **Consul**: a communication substrate for fault-tolerant distributed programs. [S.l.]: Dept. of Computer Science, University of Arizona, 1991. (Technical Report 91-32).

[MON94] MONTGOMERY, T. **Design, implementation, and verification of the Reliable Multicast Protocol**. [S.l.]: University of West Virginia, 1994. Masters Thesis.

[NAU96] NAUGHTON, P. **Dominando o Java**. São Paulo: Makron Books, 1996.

[NUN99] NUNES, R. C. **Suporte a partições de rede nos serviços de comunicação de grupo**. Porto Alegre: PPGC-UFRGS, 1999. (EQ-42).

[NUN2000] NUNES, R. C. Self-tuned failure detectors. In: SIMPÓSIO BRASILEIRO DE COMPUTAÇÃO TOLERANTE A FALHAS, 9., 2000. [S.l:s.n], 2000.

[OMG95] OMG. **CORBA**: architecture and specification. New York: OMG, 1995.

[OPE94] OPEN SOFTWARE FOUNDATION. **Introduction to OSF DCE**. New Jersey: Prentice Hall, 1994.

[PIN94] PINGALI, S.; TOWSLEY, D.; KUROSE, J. F. A comparison of sender-initiated and receiver-initiated reliable multicast protocols. **Performance Evaluation Review**, [S.l.], v.22, p.221-230, May 1994.

[PRA96] PRADHAN, D. K. **Fault-tolerant computer system design**. Upper Saddle River: Prentice Hall, 1996.

[REI2000] REINHOLTZ, K. Java will be faster than C++. **Communications of the ACM**, New York, v.35, n.2, p.25-28, Feb. 2000.

[REN94] RENESSE, R. V.; HICKEY, T. M.; BIRMAN, K. P. **Design and Performance of Horus**: a lightweight group communication system. [S.l.]: Dept. of Computer Science, Cornell University, 1994. (Technical Report 94-1442).

[ROD92] RODRIGUES, L.; VERISSIMO, P. **xAMp, a protocol suite for group communication**. [S.l]: INESC, 1992. (Technical Report 43-92).

[STR92] STROUSTRUP, B. **The C++ programming language**. 2nd ed. [S.l]:Addison-Wesley, 1992.

[SUN89] SUN MICROSYSTEMS INC. **NFS**: network file system protocol Specification. [S.l.]: SRI Network Information Center, 1989. (RFC 1094).

[TAN92] TANNENBAUM, A. S. **Modern Operating Systems**. New Jersey: Prentice-Hall, 1992. 728p.

[TAN94] TANNENBAUM, A. S. **Redes de Computadores**. 2. ed. Rio de Janeiro: Campus, 1994.

[VAS97] VASCONCELOS, S. R. A.; BRASILEIRO, F. V. Serviços para tolerância a faltas no ambiente operacional Seljuk-Amoeba. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 7., 1997. **Anais...** Campina Grande:[s.n], 1997. p.237-251.

[VOG96] VOGELS, W. World wide failures. In: ACM SIGOPS EUROPEAN WORKSHOP, 7., 1996. **Proceedings...** Connemara: [s.n], 1996.

[YAV95] YAVATKAR, R.; GRIFFIOEN, J.; SUDAN, M. A reliable dissemination protocol for interactive collaborative applications. ACM MULTIMEDIA, 1995. **Proceedings...** [S.l:s.n], 1995.