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

FlexGroup
Um Ambiente Flexível
para Comunicação em Grupo

por

RODRIGO DIAS RIVERA

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

Prof. Dr. Cláudio F. Resin Geyer
Orientador

Porto Alegre, agosto de 1999

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Rivera, Rodrigo Dias

FlexGroup – Um ambiente flexível para comunicação em grupo / por Rodrigo Dias Rivera. – Porto Alegre: PPGC da UFRGS, 1999.

77p.: il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2000. Orientador: Geyer, Cláudio F. Resin.

1. Sistemas distribuídos. 2. Tolerância a falhas. I. Geyer, Cláudio F. Resin. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL

Reitora: Profa. Wrana Panizzi

Pró-Reitor de Pós-Graduação: Prof. Franz Rainer Semmelmann

Diretor do Insituto de Informática: Prof. Philippe Olivier Alexander Navaux

Coordenadora do PPGC: Profa. Carla Maria Dal Sasso Freitas

Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Agradecimentos

Gostaria de agradecer em primeiro lugar aos meus pais Roberto e Marina por todo o apoio que me deram durante estes três longos anos e durante toda a minha vida. Agradeço a eles também por terem agüentado o meu mau humor durante tantos dias e noites escrevendo esta dissertação e por terem sempre me incentivado a ir em frente. E, principalmente, por ter me dado o exemplo de como agir como profissional e como ser humano.

Agradeço o meu orientador Cláudio Geyer pela sua orientação, conselhos, incentivo e paciência durante toda a elaboração deste trabalho.

Agradeço também a toda a minha família, que sempre acreditou em mim, especialmente ao meu irmão Vitor, que mesmo durante as inúmeras vezes que brigamos, sempre estava do meu lado.

Agradeço aos meus amigos que foram durante muitas vezes a única razão de alegria na minha vida e que sempre me deram força para prosseguir em frente.

Aos colegas de projeto e de mestrado, em especial o Vinícius Mantay e o Alessandro "Caverna" que sempre foram parceiros para pesquisas científicas, pesquisas nem tão científicas, jogos de truco e noitadas em geral.

Agradeço aos professores e funcionários do Instituto de Informática e da UFRGS, que mesmo submetidos a uma série de problemas econômicos e políticos que afligem o ensino superior público e gratuito, continuam mantendo esta instituição como um centro de excelência no ensino e na pesquisa.

Finalmente, agradeço ao Megaflops Futebol Clube e ao Grêmio Footbal Porto Alegrense por terem me dado tantas alegrias durante estes últimos anos.

Sumário

Lista de Figuras **6**
Lista de Tabelas **7**
Lista de Quadros **8**
Resumo **9**
Abstract **10**
1 Introdução **11**
2 Comunicação em Grupo **13**
2.1 Difusão Confiável **13**
2.1.1 Ordenação 14
2.2 Características da Comunicação em Grupo **16**
2.2.1 Grupos Fechados e Grupos Abertos 17
2.2.2 Grupos Hierárquicos e Grupos de Pares 17
2.2.3 Grupos Sobrepostos 18
2.2.4 Composição do Grupo 19
2.2.5 Atomicidade 20
2.2.6. Sincronismo Virtual 21
2.3 Protocolos de Comunicação em Grupo **22**
2.3.1 xAMP 22
2.3.2 HORUS - ISIS 25
3 Descrição do Ambiente FlexGroup **27**
3.1 Introdução **27**
3.2 Estrutura do FlexGroup **28**
3.3 Biblioteca **29**
3.4 Servidor de Comunicação em Grupo **30**
3.4.1 Núcleo do Servidor 32
3.4.2 Subprotocolos 33
3.5 Funcionamento do Sistema **34**
3.5.1 Camada de Comunicação 34
3.5.2 Difusão Confiável entre os Servidores 35

3.5.3 Inicialização do Servidor 38
3.5.4 Encerramento de um Servidor 39
3.5.5 Criação de um Grupo 40
3.5.6 Destruição de um grupo 41
3.5.7 Entrada de um processo no grupo 42
3.5.8 Saída de um processo do grupo 43
3.5.9 Envio de mensagem 44
3.5.10 Recepção de uma mensagem 46
4 Implementação 47
4.1 Biblioteca 47
4.2 Núcleo do Servidor 49
4.2.1 Mensagens dos Servidores 53
4.2.2 Mensagens dos Subprotocolos 54
4.2.3 Mensagens das Aplicações 55
4.3 Subprotocolos 56
4.3.1 Subprotocolo de Difusão Confiável 60
4.3.2 Subprotocolo de Difusão Causal 63
4.3.3 Subprotocolo de Difusão Atômica 65
4.4 Utilização do FlexGroup 67
4.5 Criação de um Subprotocolo 69
4.6 Análise dos Resultados 70
4.6.1 Tolerância a Falhas 70
4.6.2 Desempenho 71
4.6.3 Custo e Complexidade 71
4.6.4 Escalabilidade e Flexibilidade 72
5 Conclusão 73
Bibliografia 75

Lista de Figuras

FIGURA 2.1 - Relação entre as primitivas de difusão 16
FIGURA 2.2 - Exemplo de grupo fechado e grupo aberto 17
FIGURA 2.3 - Exemplo de sobreposição de grupos 18
FIGURA 2.4 - Entrada atômica e não-atômica em um grupo. 20
FIGURA 2.5 - Pilhas de protocolos no Horus 26
FIGURA 3.1 - Estrutura do Ambiente de Comunicação em Grupo 28
FIGURA 3.2 - Estrutura do Servidor de Comunicação em Grupo 32
FIGURA 3.3 - Difusão confiável entre os servidores de comunicação em grupo 37
FIGURA 3.4 - Detecção de falhas entre os servidores de comunicação em grupo 38
FIGURA 3.5 - Algoritmo de entrada de um novo servidor 39
FIGURA 3.6 - Envio de uma mensagem por uma aplicação 44
FIGURA 3.7 - Recepção de mensagem pelo Servidor de Comunicação 45
FIGURA 4.1 - Informações utilizados pelos componentes do FlexGroup 50
FIGURA 4.2 - Fluxo de Mensagens do Núcleo do Servidor FlexGroup 53
FIGURA 4.3 - Fluxo de mensagens de um subprotocolo 58
FIGURA 4.4 - Exemplo de comunicação, utilizando o subprotocolo de difusão causal 65

Lista de Tabelas

TABELA 4.1 – Métodos da classe *Group* 48

Lista de Quadros

QUADRO 4.1 – Descrição das classes utilizadas na implementação dos subprotocolos 58
QUADRO 4.2 – Recepção de uma mensagem pelo subprotocolo de difusão confiável..... 62

Resumo

Mecanismos de comunicação entre processos são fundamentais no desenvolvimento de sistemas distribuídos, já que constituem o único meio de compartilhar dados entre processos que não dispõem de memória comum. Um dos principais mecanismos de comunicação utilizados é a troca de mensagens entre os processos componentes do sistema. Existem muitas aplicações que são compostas por um conjunto de processos que cooperam para realizar uma determinada tarefa e que são mais facilmente construídas se o sistema operacional oferecer a possibilidade de se enviar uma mensagem a diversos destinos. Neste caso são necessários mecanismos que permitam a difusão confiável de uma mensagem para um grupo de processos em uma única operação. Tendo em vista esta necessidade, diversos protocolos têm sido apresentados na literatura para permitir a comunicação entre um grupo de processos com diferentes graus de complexidade e de desempenho.

Este trabalho apresenta um ambiente para desenvolvimento e utilização de protocolos de comunicação em grupo, denominado FlexGroup. O ambiente divide os protocolos em suas características fundamentais, permitindo que estas características possam ser desenvolvidas separadamente como subprotocolos. Os subprotocolo são interligados através de uma interface comum e gerenciados pelo núcleo do ambiente. A comunicação entre as diversas máquinas da rede é gerenciada pelo FlexGroup, permitindo que o desenvolvedor de um novo subprotocolo possa somente se focar nas características específicas do seu protocolo. Esta modularidade permite, ainda, que apenas as partes de interesse de um novo protocolo precisem ser implementadas, além de também viabilizar a criação de um protocolo baseado nos já existentes no ambiente.

Além disso, o ambiente permite que as aplicações de comunicação em grupo possam definir, através de uma biblioteca, o conjunto de subprotocolos que desejam utilizar, em tempo de execução, sem necessidade de conhecer a implementação interna dos subprotocolos.. Da mesma forma, alguém que se proponha a realizar comparações com os protocolos existentes, pode utilizar os diversos subprotocolos e as aplicações existentes, bastando alterar os protocolos utilizados em tempo de execução e avaliando somente as características que deseje analisar.

Palavras-chaves: Sistemas Distribuídos, Tolerância a Falhas, Comunicação entre Processos, Difusão Confiável, Comunicação em Grupo, Protocolos.

TITLE: "FLEXGROUP – A FLEXIBLE ENVIRONMENT FOR GROUP COMMUNICATION"

Abstract

Communication mechanisms among processes are fundamental in the development of distributed systems, since they are the only way of sharing data among processes who do not have common memory. One of the most important inter-process comunication mechanisms is the exchange of messages among processes that make up the system. Many applications are composed by a set of processes that cooperate to accomplish a certain task and this kind of applications are more easily constructed if the operating system provides the possibility of sending a message for several destinations. In that case, it is necessary that the system ensures reliable broadcast of a message for a group of processes in one single operation. To address this necessity, many protocols have been presented in the literature to allow communication of a group of processes with different complexity and performance degrees.

This essay presents an environment for development and use of group communication protocols, named FlexGroup. The environment divides the protocols in its basic components, allowing them to be developed separately as subprotocols, later interlinked through a common interface, managed by the kernel of the system. The communication among the machines on the network is managed by the FlexGroup, thus allowing that the developer of a new protocol can only focus on the specific characteristic of his protocol. Moreover, this kind of modularity allows that only the parts of interest of the new protocol need to be implemented, and also offers the possibility of developing a new protocol based on the ones that already exist in the environment.

Besides, the FlexGroup system allows that the group communication applications can define, through a library, the subset of subprotocols that it wants to use in runtime, without the necessity of the applications knowing the internal implementation of the subprotocols. In the same way, someone who wants to make comparisons with existent protocols can use the protocols and the applications availabe in the environment, just changing the protocols used in runtime and evaluating only the features of concern.

Keywords: Distributed Systems, Fault Tolerance, Inter-Process Communications, Reliable Broadcast, Group Communication, Protocol

1 Introdução

Uma das principais necessidades em sistemas distribuídos é a comunicação entre os diversos processos que compõem o sistema. Uma das maneiras mais utilizadas para realizar esta comunicação entre processos é a troca de mensagens entre dois processos. A troca de mensagens pode assumir diversas formas quanto ao serviço oferecido e a sua implementação, como por exemplo, envio e recepção síncrono/assíncrono, armazenamento das mensagens, confiabilidade, e outras. Além disso a troca de mensagens poder ser utilizada como mecanismo básico de comunicação entre processos para construir outros serviços mais elaborados, como por exemplo, memória compartilhada distribuída ou chamada remota de procedimentos.

Um dos serviços muitas vezes construído sobre a troca de mensagens entre dois processos é o envio de mensagens a múltiplos destinos. Muitas aplicações são mais facilmente construídas se o sistema operacional oferece a possibilidade de se enviar uma mensagem a diversos destinos em uma única operação [TAN95]. O envio de mensagens a múltiplos destino pode ser dividido em dois casos, difusão e multicast.

O termo difusão é empregado para representar o envio de mensagens de um processo para todos os outros processos do sistema. Como nem todas as redes possuem a capacidade de realizar difusão, esta operação muitas vezes tem que ser implementada baseando-se em operação de envio ponto-a-ponto. Desta forma, mesmo o envio de mensagens sendo confiável, o sistema fica sujeito a falhas durante a difusão, pois o processo emissor da mensagem pode falhar durante o envio das mensagens [JAL94].

Desta forma, para permitir a utilização deste paradigma é necessária a utilização de um protocolo de difusão confiável. A difusão confiável deve garantir que as mensagens difundidas no sistema sejam entregues a todos os processos. Diversas técnicas têm sido descritas na literatura para resolver este problema de difusão confiável, utilizando soluções baseadas em hardware, camada de comunicação ou por algoritmos implementados no sistema operacional ou em pacotes de comunicação.

O termo multicast é empregado para descrever o envio seletivo de uma mensagem para um subconjunto dos processos do sistema. Da mesma forma que a difusão, caso o multicast seja baseado no envio de mensagens entre os processos envolvidos, ele está sujeito a erros, necessitando de mecanismos que garantam o envio confiável das mensagens para todos os processos destinatários da mensagem. Além destes problemas, o multicast também atender a outros requisitos, como por exemplo, de que maneira serão endereçados os processos destinatários da mensagem.

Uma classe de protocolos de difusão confiável são os algoritmos para comunicação em grupo. A comunicação em grupo pode ser vista como uma difusão para todos os membros do grupo, apresentando, desta forma, as mesmas dificuldades da difusão confiável e os do multicast. Entretanto, existem ainda algumas outras possibilidades que devem ser consideradas quando se deseja construir um protocolo de comunicação em grupo, como, por exemplo, a entrada e saída de processos no grupo e a manutenção de uma visão consistente dos membros do grupo.

Tendo em vista estas necessidades, vários protocolos foram apresentados na literatura. Estes protocolos apresentam características bastante diversas em relação aos serviços que oferecem, como por exemplo, classe de falhas tratadas, ordenação das mensagens e escalabilidade. Dessa forma, cada protocolo possui uma classe de aplicações à qual atende plenamente e outras às quais não atende tão satisfatoriamente. Uma das alternativas adotadas foi a criação de uma hierarquia de protocolos[REN96].

Porém, mesmo utilizando uma hierarquia de protocolos, o número de protocolos disponíveis é limitado, não permitindo assim um maior grau de flexibilidade e otimização para as aplicações.

O presente trabalho propõe um ambiente de desenvolvimento de protocolos de comunicação em grupo que disponibilize facilidades tanto para o desenvolvimento de novos protocolos de comunicação em grupo, quanto para o desenvolvimento de aplicações que utilizem este paradigma de comunicação. Esta flexibilização é baseada na modularização dos protocolos de comunicação em grupo, que são divididos em seus subcomponentes, que podem ser implementados de maneira independente e utilizados de acordo com as características das aplicações que utilizam o ambiente.

Este ambiente permite que uma aplicação possa, dinamicamente, determinar o tipo de protocolo mais adequado a suas necessidades, sem que seja preciso ter conhecimento sobre a implementação interna do protocolo, aumentando o grau de flexibilidade e eficiência das aplicações que utilizem o sistema. Cada grupo de processos pode adotar um protocolo diferente para a sua comunicação, sendo que diversos grupos e consequentemente diversos protocolos podem existir no sistema em um mesmo momento.

O ambiente é estruturado de forma modular, com diversos componentes que se comunicam de forma independente e com uma interface bem definida para interação entre estes componentes. Dessa forma, alguém que se proponha a desenvolver um novo protocolo pode facilmente reutilizar as diversas propriedades já implementadas por outros protocolos desenvolvidos no sistema e preocupar-se somente com a implementação das características que deseja desenvolver.

Este trabalho está estruturado da seguinte maneira: O capítulo 2 descreve as principais características da comunicação em grupo e analisa dois protocolos já existentes, o xAMP e o HORUS.

O capítulo 3 apresenta o ambiente proposto, o FlexGroup, descrevendo as suas características e seus principais componentes. Neste capítulo são descritos também os principais mecanismos utilizados pelo ambiente para oferecer os serviços propostos.

A implementação do ambiente e seu funcionamento são descritos no capítulo 4, juntamente com a descrição de como utilizá-lo para construir aplicações de comunicação em grupo e de como construir novos protocolos. Este capítulo também apresenta alguns resultados obtidos com a construção de um protótipo do ambiente e com a utilização deste protótipo.

Por fim, o capítulo 5 apresenta algumas conclusões decorrentes da realização deste trabalho, bem como as perspectivas futuras para o FlexGroup e para esta área de pesquisa.

2 Comunicação em Grupo

O mecanismo básico de comunicação entre processos em um sistema distribuído é a troca de mensagens entre dois processos (mensagens ponto-a-ponto) [TAN95]. Apesar da comunicação ponto-a-ponto ser suficiente para muitas aplicações, existem muitas outras onde um nodo precisa enviar uma mensagem para muitos outros nodos do sistema. Neste tipo de aplicação, uma forma de comunicação multicast se faz necessária.

No problema de difusão confiável, um processador dissemina um valor para todos os outros processadores de um sistema distribuído, onde tanto os processadores como os componentes de comunicação estão sujeitos a falhas. Como a primitiva básica suportada pela maioria das redes é ponto-a-ponto, as primitivas de difusão são em sua maioria construídas sobre este tipo de primitiva, tornando a implementação de protocolos de difusão suscetíveis a falhas de nodos e de comunicação. É possível que o nodo origem falhe enquanto está fazendo a difusão de uma mensagem, criando a possibilidade de somente alguns dos nodos receberem a mensagem. Apesar disso ser aceitável para algumas aplicações, isto não pode ser aceito quanto se está construindo um sistema que exija a atomicidade na entrega das mensagens [JAL94].

2.1 Difusão Confiável

Em um sistema onde os processos se comunicam por difusão (*broadcast*), se houver uma falha durante a difusão, pode ocorrer que somente um subconjunto dos processos realmente receba a mensagem que foi difundida. Este tipo de inconsistência pode comprometer a integridade do sistema, e consequentemente difusão **não-confiável** não é uma ferramenta apropriada para construir aplicações tolerantes a falhas [HAD93].

A difusão confiável de uma mensagem tem o seguinte funcionamento:

1. O processo origem **difunde** a mensagem;
2. Os processos destinos recebem a mensagem e a colocam na sua fila de mensagens;
3. Se a mensagem satisfizer os requisitos do protocolo de difusão confiável, a mensagem é **entregue** à aplicação.

Informalmente a difusão confiável requer que todos os processos entreguem o mesmo conjunto de mensagens e que este conjunto inclua todas as mensagens difundidas por processos corretos, e que mensagens espúrias não sejam entregues. Formalmente, difusão confiável é definida em termos de duas primitivas: **difundir**(*m*) e **entregar**(*m*), onde *m* é uma mensagem.

Como cada processo pode difundir várias mensagens, é importante poder determinar a identidade do processo originador da mensagem, e distinguir as diferentes mensagens difundidas por um mesmo processo. Assume-se que cada mensagem carregue consigo o identificador do processo que a enviou (origem) e um número de seqüência dentro das mensagens enviadas por cada processo (*seq*). Desta maneira, cada mensagem é identificada univocamente por este par (origem, *seq*) [HAD93].

Processos são ditos corretos se não sofrem nenhum tipo de falha durante a sua execução. Processos que sofrem alguma falha durante a sua execução são denominados processos faltosos.

Para que um protocolo de difusão seja **confiável** é necessário que ele garanta ao menos três propriedades: acordo, validade e integridade [VER89]. :

- **Validade**: Se um processo correto difunde uma mensagem *m*, então todos os processos corretos entregarão *m*.
- **Acordo**: Se um processo correto entrega uma mensagem *m*, então todos os processos corretos entregarão *m*.
- **Integridade**: Para cada mensagem *m*, cada processo correto entrega *m* somente uma vez e somente se *m* foi previamente difundida por origem(*m*).

Se um processo falha durante a difusão de uma mensagem, um protocolo de difusão confiável permite dois comportamentos: ou a mensagem é entregue a todos os processos corretos, ou não é entregue a nenhum processo.

2.1.1 Ordenação

Apesar destas propriedades serem suficientes para algumas aplicações, difusão confiável não impõe nenhuma restrição na **ordem** em que as mensagens são entregues. Pode ser definida uma hierarquia de protocolos de difusão mais fortes, diferindo nas garantias que eles provêem na ordem de entrega das mensagens.

Em geral, cada mensagem difundida tem um contexto associado, sem o qual a mensagem pode ser mal interpretada. Uma mensagem destas não pode ser entregue a um processo que não conheça o seu contexto. Em algumas aplicações, o contexto de uma mensagem *m* consiste nas mensagens previamente difundidas pelo processo origem de *m*. Estas aplicações requerem a semântica de uma **difusão FIFO**, que é uma difusão **confiável** que satisfaz o seguinte requisito na entrega de mensagens:

- **Ordem FIFO**: Se um processo difunde uma mensagem *m* antes de difundir uma mensagem *m'*, então nenhum processo correto entrega *m'*, a não ser que tenha previamente entregue *m* [HAD93].

Esta ordenação FIFO é muitas vezes garantida pela camada de comunicação das redes utilizadas, que garantem o envio ordenado de mensagens entre dois processos.

A ordenação FIFO é adequada quando o contexto de uma mensagem *m* consiste somente nas mensagens que o processo origem de *m* difundiu antes de difundir *m*. Entretanto, uma mensagem *m* pode também depender de mensagens que o processo origem entregou antes de difundir *m*. Neste caso, a ordenação de entrega de mensagens garantida pela difusão FIFO não é suficiente [JAL94].

A difusão **causal** estende o conceito de contexto de uma mensagem, garantindo que uma mensagem não é entregue até que todas as mensagens das quais depende tenham sido entregues.

Esta noção de dependência pode ser formalizada como se segue [HAD93]. A execução de uma primitiva de difusão ou de entrega por um processo é chamada de evento. Diz-se que um evento *e* **precede causalmente** um evento *f*, denotado $e \rightarrow f$, se e somente se: um processo executa tanto *e* quanto *f*, nesta ordem, ou *e* é a difusão de uma

mensagem por um processo *i* e *f* é a entrega desta mensagem por um processo *j* ou existe um evento *h,* tal que $e \rightarrow h$ e $h \rightarrow f$.

- **Ordenação Causal**: Se a difusão de uma mensagem *m* precede causalmente a difusão de uma mensagem *m'*, então nenhum processo correto entrega *m'* a não ser que tenha entregue *m* [HAD93].

A definição anterior garante que, mesmo na ocorrência de falhas, uma mensagem só é entregue a um processo quando todas as mensagens que a precedem causalmente tiverem sido entregues a este processo.

A difusão causal não impõe nenhuma ordenação de entrega nas mensagens que não são causalmente relacionadas. Então, dois processos corretos podem entregar mensagens não relacionadas causalmente em ordens diferentes, criando problemas para algumas aplicações. Por exemplo, se em um banco de dados bancário replicado em diferentes nodos é difundida uma mensagem de depósito na conta e uma outra mensagem aplicando uma taxa de juros sobre o saldo da conta. Como estas duas mensagens não são causalmente relacionadas, a difusão causal permite que duas cópias da conta entreguem estas mensagens em ordens diferentes, resultando em uma inconsistência no banco de dados.

Para prevenir estes problemas, a difusão **atômica** requer que todos os processos corretos entreguem todas as mensagens na mesma ordem. Esta ordenação total da entrega de mensagens assegura que todos os processos corretos tem a mesma visão do sistema, então eles podem atuar consistentemente sem nenhuma comunicação adicional [HAD93]. Formalmente, uma Difusão Atômica é uma difusão confiável que satisfaz a seguinte condição:

- **Ordenação Total**: Se dois processos corretos *p* e *q* entregam as mensagens *m* e *m'*, então *p* entrega *m* antes de *m'* se e somente se *q* entrega *m* antes de *m'*.

Os requisitos de acordo e ordenação total da difusão atômica implicam que os processos corretos entregam a mesma seqüência de mensagens.

A Difusão Atômica não exige que as mensagens sejam entregues em ordem causal. Pode-se, porém, definir uma Difusão Atômica Causal, onde a entrega das mensagens respeite tanto a ordenação total quanto a ordenação causal. Da mesma forma, pode-se estender a ordenação FIFO para uma ordenação Atômica FIFO. A figura 2.1 [HAD94] ilustra a relação entre as diversas classes de protocolos de difusão.

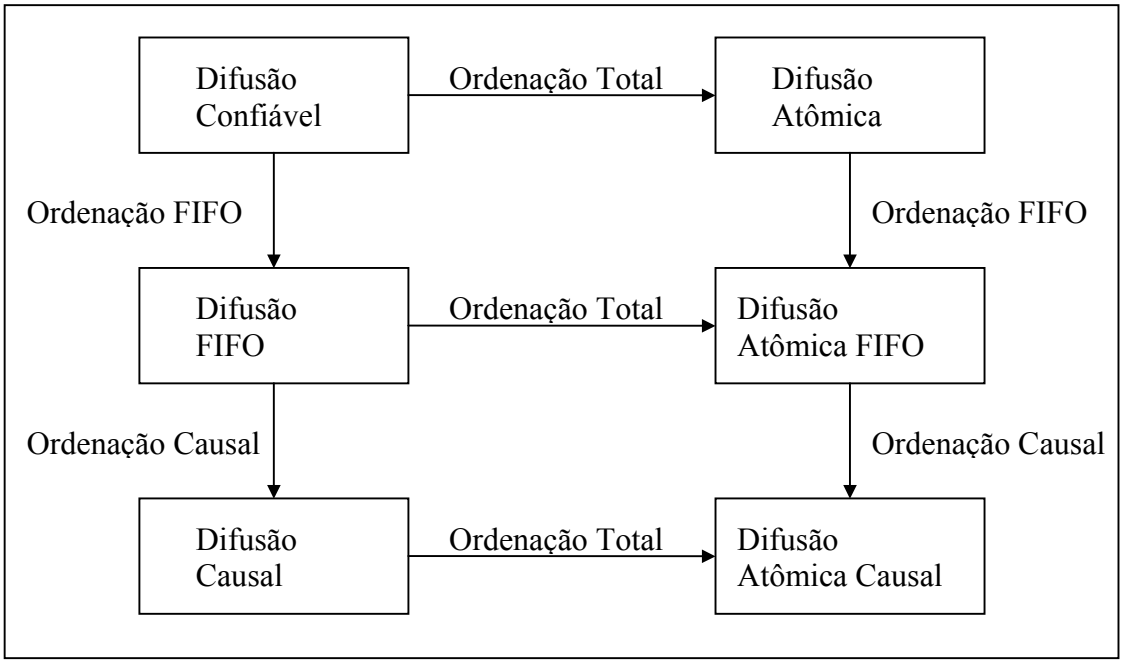

FIGURA 2.1 - Relação entre as primitivas de difusão

2.2 Características da Comunicação em Grupo

Na seção anterior foram apresentados os principais aspectos relativos à difusão de mensagens para todos os nodos de um sistema distribuído. Existem casos, porém, onde um processo deseja se comunicar apenas com um conjunto conhecido de processos. Este tipo de comunicação pode ser modelado através do uso da comunicação em grupo.

Um grupo é um conjunto de processos que trabalham cooperativamente de alguma maneira definida pelo usuário ou pelo sistema. A principal propriedade que um grupo apresenta é que quando uma mensagem é enviada ao grupo todos os seus membros a recebem. As principais vantagens oferecidas pela utilização de grupos é que eles provêem um nível mais alto de abstração para a programação e escondem das aplicações de usuário os estados internos do grupo (por exemplo, alterações na composição do grupo).

Grupos podem ser dinâmicos. Novos grupos podem ser criados e grupos existentes podem ser destruídos. Um processo pode entrar ou sair do grupo quando quiser durante seu processamento, da mesma forma um processo pode ser membro de vários grupos em um mesmo instante, se for permitida a sobreposição de grupos. Devido a estas características, novos mecanismos são necessários para gerenciar os grupos.

O objetivo de introduzir-se grupos é permitir que os processos trabalhem com conjuntos de processos como uma abstração. Desta forma, um processo pode enviar uma mensagem para um grupo de servidores sem se preocupar em saber quantos são ou onde eles estão, o que pode variar de uma chamada para outra [TAN92].

A comunicação em grupo apresenta todas as características da difusão confiável, como, por exemplo, atomicidade e ordenação. Além destas características, a utilização de grupos apresenta algumas características peculiares quanto à maneira como estes grupos são construídos.

2.2.1 Grupos Fechados e Grupos Abertos

Sistemas que suportam comunicação em grupo podem ser divididos em duas categorias, dependendo de quem pode enviar mensagens para o grupo. Alguns sistemas suportam apenas grupos **fechados**, nos quais somente os membros do grupo podem enviar para o grupo. Processos externos não podem enviar mensagens para o grupo como um todo, entretanto eles podem enviar mensagens para algum membro do grupo individualmente. Em contraste, outros sistemas permitem grupos **abertos**, onde qualquer processo no sistema pode enviar mensagens para qualquer grupo. A diferença entre grupos fechados e abertos é ilustrada na figura 2.2 [TAN95].

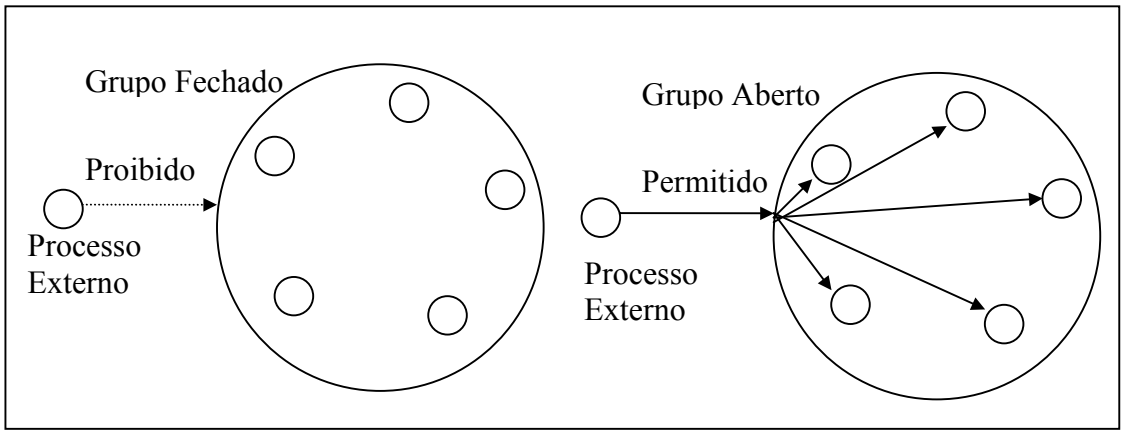

FIGURA 2.2 - Exemplo de grupo fechado e grupo aberto

A decisão do sistema de suportar grupos fechados ou grupos abertos normalmente depende das aplicações que utilizarão o serviço de comunicação em grupo. Grupos fechados são utilizados tipicamente para processamento paralelo. Por exemplo, um conjunto de processos que cooperam para resolver um conjunto de equações matemáticas formam um grupo fechado. Eles têm cada um sua tarefa e não interagem com outros processos do sistema.

De outra forma, quando a utilização de grupos servir para suportar a replicação de servidores, por exemplo, é importante que processos clientes, que não são membros do grupo, possam enviar mensagens para o grupo. Além disso, os membros do grupo podem também ter que utilizar comunicação em grupo entre si, para resolver uma requisição qualquer [SCH95]. A distinção entre grupos fechados e abertos é feita muitas vezes somente por motivos de implementação.

2.2.2 Grupos Hierárquicos e Grupos de Pares

A distinção de grupos fechados e abertos se relaciona com quem pode se comunicar com o grupo. Outra distinção importante tem a ver com a estrutura interna do grupo. Em alguns grupos, todos os processos são iguais (processos pares) e as decisões são tomadas coletivamente. Em outros grupos, existe algum tipo de hierarquia. Por exemplo, um processo é o coordenador e os outros são os trabalhadores. Neste modelo, quando uma requisição é gerada, por um cliente externo ou por um dos trabalhadores, ela é enviada ao coordenador. O coordenador então decide qual trabalhador é o mais

indicado para executar a tarefa e repassa-a para ele [TAN92]. Hierarquias mais complexas, como uma árvore de processos também podem ser implementadas.

Cada uma dessas organizações tem suas vantagens e desvantagens. O grupo de pares é simétrico e não possui um ponto único de falha. Se um dos processos falha, o grupo simplesmente torna-se menor e continua o processamento. A desvantagem é que os algoritmos de tomada de decisão são bem mais complicados e implicam em atrasos e *overheads* adicionais ao sistema.

Os grupos hierárquicos têm propriedades opostas. A perda do coordenador pode fazer com que o grupo inteiro falhe ou tenha que ser reconfigurado. Porém enquanto o coordenador funciona, ele toma decisões sem consultar nenhum outro processo, introduzindo muito pouco overhead no sistema. Este tipo de grupo pode ser utilizado por aplicações paralelas, onde um processo coordenador toma decisões sobre a distribuição das tarefas e os processos trabalhadores realizam o processamento.

2.2.3 Grupos Sobrepostos

Um mesmo processo pode ser membro de diversos grupos no mesmo momento. Este fato pode levar a outros tipos de inconsistência, não encontradas nos protocolos de difusão confiável, onde se considerava um conjunto único, consistindo de todos os processos do sistema. A figura 2.3 ilustra um destes problemas [TAN95]. Os processos A, B e C são membros do grupo 1, enquanto B, C e D são membros do grupo 2.

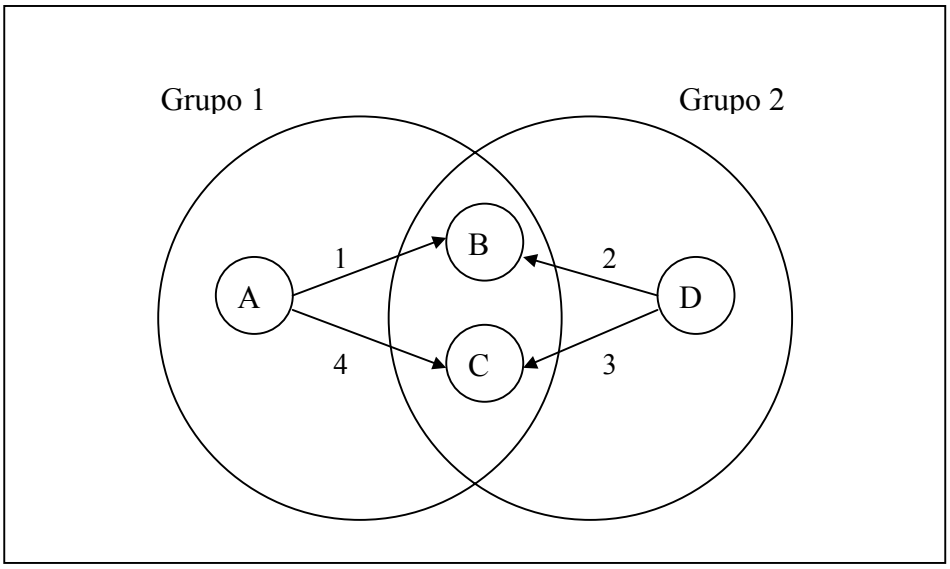

FIGURA 2.3 - Exemplo de sobreposição de grupos

Considera-se que os processos A e D enviam simultaneamente mensagens para os seus respectivos grupos, e que os grupos realizam ordenação total das mensagens dentro do grupo. A ordem de recebimento das mensagens é indicada na figura pelos números de 1 a 4. Neste caso, tem-se uma situação onde dois processos, B e C, receberam mensagens em ordens diferentes. B recebe primeiro a mensagem de A seguida da mensagem de D, enquanto C recebe as mensagens na ordem inversa. Este tipo de comportamento pode ser aceitável para algumas aplicações, porém em certos casos, isto poderia levar o sistema a um estado inconsistente.

Deve-se ressaltar que mesmo existindo uma ordenação total dentro de cada grupo, não existe necessariamente nenhuma coordenação entre múltiplos grupos [SCH95]. Alguns sistemas suportam ordenação entre múltiplos grupos, enquanto outros não tratam este tipo de situação. Deve-se levar em conta que o custo de se implementar um serviço que garanta este tipo de propriedade é alto, não sendo razoável utilizá-lo quando não for estritamente necessário.

2.2.4 Composição do Grupo

O gerenciamento da composição do grupo é um dos principais componentes de um sistema que se proponha a implementar um serviço de comunicação em grupo. De maneira geral, um protocolo de composição do grupo gerencia a formação e manutenção de um conjunto de processos, chamado de grupo. Como novos processos podem querer entrar em grupo já existente, e outros podem querer deixar o grupo ou ter que ser removidos depois de falharem, a composição do grupo muda dinamicamente. Um protocolo de composição precisa administrar estas mudanças de uma maneira coerente. Cada processo possui uma visão local da composição atual do grupo, e os processos mantêm alguma forma de concordância nestas visões locais [ANC95].

Em sistemas assíncronos, dois tipos de serviço de composição do grupo têm sido apresentados: partição-primária e particionáveis [CHA95].

Um serviço de composição com partição-primária mantém uma única visão válida do grupo, ou seja, os processos concordam nas suas visões locais do grupo. Este tipo de serviço é indicado para sistemas sem particionamento de rede, ou que só permitem que a composição do grupo continue funcionando em uma única partição da rede, a partição-primária [ANC95].

Por outro lado, um serviço de composição do grupo particionável permite que múltiplas visões do grupo coexistam e evoluam concorrentemente. Podem existir vários subconjuntos disjuntos de processos, de tal forma que processos em cada subconjunto concordem que eles são os membros atuais do grupo. Em outras palavras, este tipo de serviço permite divisão do grupo, por exemplo, quando ocorre particionamento da rede, e fusão do grupo, por exemplo, quando a comunicação entre as partições é restaurada [ANC95].

A definição formal dos requisitos que um protocolo de composição dos grupos precisa atender ainda é um problema em aberto. Em [ANC95] é proposta uma especificação formal para um serviço de grupo com partição-primária, porém em [CHA95] é provado, que em um sistema assíncrono, mesmo com apenas falhas de crash, o problema de obter um consenso sobre a composição da partição ativa do grupo não pode ser resolvido. A maioria dos protocolos procura escapar desta impossibilidade utilizando-se de mecanismos de “suspeita” [AMI95], [MAC95a], [BIR96]. Um processo suspeito de ter falhado pode ser excluído do grupo pelos outros processos, mesmo que não tenha falhado na realidade (seja apenas um processo mais lento, por exemplo). Porém com algoritmos deste tipo, é impossível ter uma especificação que garanta a continuidade do protocolo. Os processos podem suspeitar uns dos outros e o protocolo não conseguiria prosseguir ou então todos os processos seriam retirados do grupo [CHA95].

As propriedades dos protocolos particionáveis foram descritas em [DOL92] e não se enquadram na impossibilidade descrita acima por permitirem que processos em partições diferentes concordem em visões distintas do grupo. Entretanto, eles apresentam um outro problema. A definição dos requisitos precisa ser forte o suficiente para impedir a criação de protocolos de composição que possam dividir o grupo em

subconjuntos caprichosamente, sem a ocorrência real de falhas. Porém esta especificação deve ser fraca o suficiente para poder ser resolvida [CHA95]. Além disso, a maioria destes protocolos não se preocupa com a semântica da entrega de mensagens quando ocorre uma recomposição do grupo após um particionamento, deixando esta tarefa a cargo das aplicações [DOL92].

2.2.5 Atomicidade

No gerenciamento de grupos de processos, existem dois tipos de atomicidade: atomicidade de composição e atomicidade de falhas. A primeira provê a ilusão de uma composição do grupo que muda instantaneamente quando membros do grupo entram, saem ou falham. A segunda assegura que difusões interrompidas por uma falha serão transparentemente terminadas.

É mais difícil construir aplicações com grupos de processos nos quais a expansão do endereço de um grupo para uma lista dos membros não é atômica, ou seja, não há garantias sobre exatamente quais processos receberão uma difusão. A atomicidade de composição garante que uma mensagem difundida será entregue para todos os membros da composição, definidos no momento lógico em que a entrega da mensagem ocorrer [BIR91a]. A figura 2.4 ilustra esta propriedade.

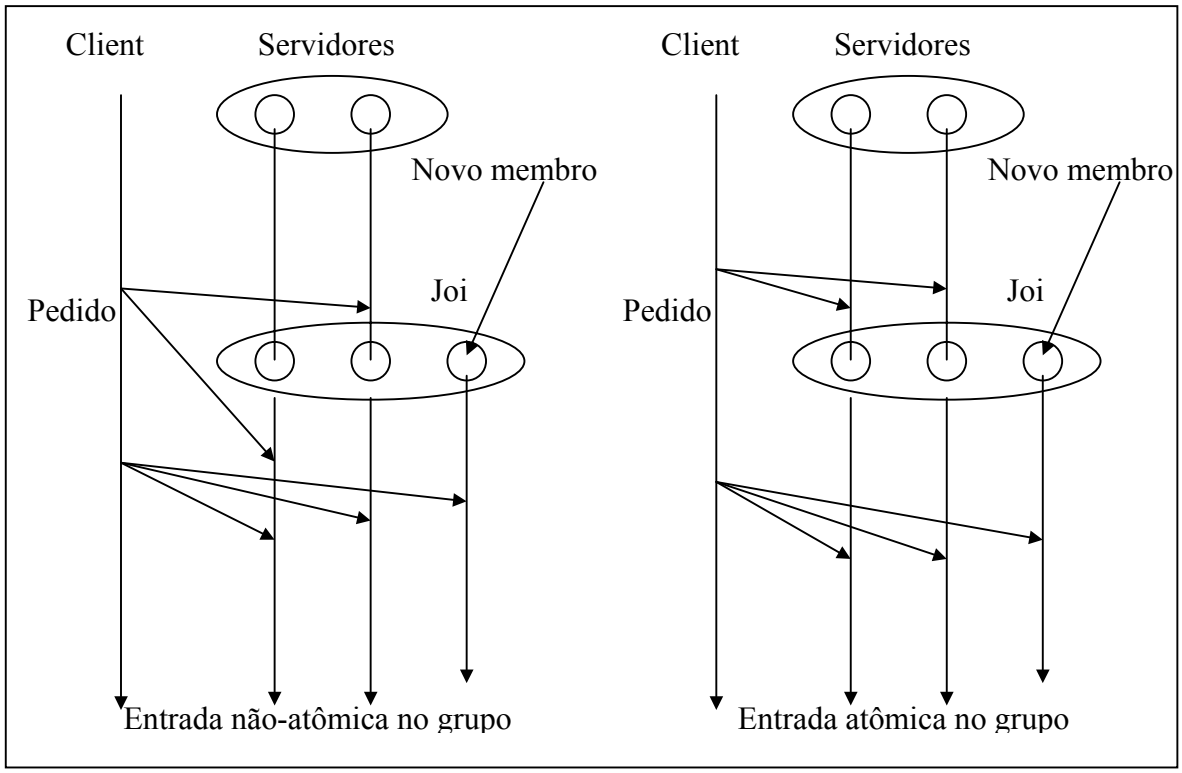

FIGURA 2.4 - Entrada atômica e não-atômica em um grupo.

Considerações similares podem ser feitas para a atomicidade de falhas. Algoritmos que utilizam comunicação em grupo são bastante simplificados pela capacidade de difundir uma mensagem sem precisar se preocupar com que algum evento inesperado, como uma falha, cause uma entrega parcial da mensagem [TAN95]. Quando um membro do grupo falha, a atomicidade de falhas garante que os outros

processos irão receber a notificação da falha somente após terem recebidos todas as mensagens do processo faltoso que foram enviadas antes da falha e que possam ter sido entregues por algum processo [BIR91a]. Esta propriedade elimina seqüências de falhas anômalas, como a entrega de uma mensagem de um processo após os membros do grupo terem sido informados de sua saída e terem eliminado as informações a seu respeito.

A atomicidade na composição dos grupos é útil também por outra razão: ela dá aos membros do grupo conhecimento implícito sobre o estado dos outros membros. Isto permite, por exemplo, que cada membro do grupo use a mesma função determinística para escolher o *primary-site* em um algoritmo de replicação de dados, ou para subdividir o trabalho em uma computação paralela. Se houver atomicidade na composição, estas funções podem operar somente em dados locais (a lista *sincronizada* dos membros do grupo), porém elas terão uma consistência global ao grupo. Na seção seguinte, será abordado o conceito de **sincronismo virtual**, uma propriedade que pode ser considerada uma extensão das propriedades de atomicidade aqui descritas.

2.2.6. Sincronismo Virtual

Em sistemas de comunicação em grupo, a entrega e o conseqüente processamento de mensagens difundidas no grupo podem alterar o estado ou os dados mantidos por vários processos. Se as mensagens são recebidas em ordens diferentes por membros do grupo, os dados destes processos podem se tornar inconsistentes. Se processos falham, saem do grupo ou entram no grupo dinamicamente, processos diferentes podem ter visões diferentes da composição do grupo, o que pode resultar em comportamentos inconsistentes [ROD95].

O modelo de sincronismo virtual, introduzido pelo sistema ISIS, ordena mudanças de composição do grupo em relação às mensagens difundidas no grupo. Este modelo assegura que, se dois processos continuam a fazer parte de um mesmo grupo, entre a duas visões consecutivas, eles entregam o mesmo conjunto de mensagens durante este intervalo. O sincronismo virtual não estabelece nenhuma restrição ao comportamento de processos falhos ou isolados. Processos falhos, caso sejam recuperados, são tratados como processos novos.

Formalmente, o conceito de um serviço de comunicação em grupo virtualmente síncrono pode ser definido da seguinte maneira [ROD95]:

Difusão Virtualmente Síncrona: Considera-se um conjunto de processos G, uma visão $Vi(G)$, e uma mensagem m difundida para os membros do grupo $Vi(G)$. Se existe um processo p qualquer, pertencente a $Vi(G)$ que tenha entregue m na visão $Vi(G)$ e que tenha instalado uma visão $Vi+1(G)$, então cada processo q, pertencente a $Vi(G)$, que tenha instalado $Vi+1(G)$, entregou m antes de instalar $Vi+1(G)$. A difusão da mensagem m é chamada difusão virtualmente síncrona. O sistema é virtualmente síncrono se, e somente se, todas as difusões são difusões virtualmente síncronas.

Esta definição impõe uma ordenação total entre mudanças de visão e difusões, porém não impõe nenhuma ordenação entre as mensagens entregues na mesma visão. A implementação do sincronismo virtual requer o uso de um *detetor de falhas* [CHA91] ou alguma forma de monitoramento dos processos, mais a execução de alguma forma de protocolo de esvaziamento (*flush*) para assegurar que todas as mensagens entregues a alguns processos em uma visão são entregues a todos os processo desta visão, antes que uma nova visão seja instalada. Para garantir a terminação do protocolo de esvaziamento, o tráfego pode ser temporariamente interrompido durante a execução do protocolo. Isto pode levar a uma queda de desempenho do sistema durante a execução do protocolo de

instalação de uma nova visão, porém simplifica bastante o projeto de aplicações. Existem, ainda, protocolos que permitem a continuidade do fluxo de mensagens durante mudanças de visão, porém estes protocolos precisam colocar informações adicionais nas mensagens para identificá-las como pertencentes à nova visão e a sua entrega terá que ser retardada até que todos os processos tenham instalado a nova visão.

Em sistemas onde a ocorrência de falhas seja relativamente baixa, o primeiro mecanismo apresenta melhor desempenho, pois não acrescenta nenhum overhead no envio normal de mensagens. Porém se o número de falhas no sistema for alto, ou uma grande disponibilidade for exigida, um mecanismo que não impeça a difusão de novas mensagens é mais indicado.

2.3 Protocolos de Comunicação em Grupo

Conforme foi visto nas seções anteriores, protocolos de comunicação em grupo precisam atender a uma variada gama de propriedades e realizar uma série de escolhas sobre os tipos de serviços que serão oferecidos pelo sistema. Devido a essa diversidade de opções, numerosos protocolos tem sido apresentados na literatura. Cada protocolo possui uma classe de aplicações à qual atende plenamente e outras às quais não atende tão satisfatoriamente. Dentro desta ótica, começaram a surgir hierarquias de protocolos, como por exemplo os protocolos xAMP[ROD92] e Horus[REN96]. A seguir, são apresentadas as principais características desses dois protocolos.

2.3.1 xAMP

O protocolo xAMP (eXtended Atomic Multicast Protocol) é um serviço de comunicação em grupo desenvolvido na Universidade de Lisboa [ROD92, VOG92], com o intuito de suportar o desenvolvimento de aplicações distribuídas com diferentes requisitos de dependabilidade, funcionalidade e desempenho. O xAMP define diversas qualidades de serviços a serem oferecidos (QOS - Quality of Service). A seleção destes serviços é feita pelos requisitos que os usuários necessitam para as diversas classes de aplicações distribuídas. O serviço de grupos é estruturado sobre uma primitiva básica de comunicação com resposta e a partir desta primitiva são estruturados os outros serviços, como, por exemplo, multicast causal e multicast atômico.

2.3.1.1 Funcionamento do Protocolo

O xAMP utiliza um modelo abstrato de serviço de rede que possui um multicast não-confiável. O protocolo de comunicação em grupo utiliza um procedimento de transmissão com resposta, utilizando mensagens de reconhecimento para confirmar a recepção das mensagens e para detectar erros de omissão. Trata-se da primitiva básica do xAMP, na qual todas as outras se baseiam.

- **Difusão Confiável**

A chamada do procedimento é feita da seguinte maneira:
tr-w-resp([*m*], *ord*, *send*, *Mr*, *Pr*, *nr*), onde:

- *m* é a mensagem a ser enviada.
- *ord* é uma variável booleana que especifica se é necessária ordenação na entrega das mensagens.

- *send* é uma variável booleana que permite que a primeira transmissão seja ignorada.
- *Mr* é uma "bolsa de respostas".
- *Pr* é o conjunto de processadores dos quais a resposta (confirmação) é esperada.
- *nr* é o número de repostas esperadas. Normalmente *nr* é igual ao número de processadores em *Pr* e *Pr* é igual ao conjunto de processadores destinatários da mensagem.

O procedimento tr-w-resp consiste de um laço onde as mensagens são enviadas através da rede e as respostas são esperadas. O procedimento espera durante um tempo pré-determinado pelas respostas, que são inseridas em uma bolsa de respostas (*Mr*), e termina quando o número desejado de respostas é coletado. Se algumas respostas foram perdidas a bolsa é reinicializada e a mensagem é retransmitida. O laço termina quando todas as respostas são recebidas ou quando um determinado número de tentativas é alcançado.

Para preservar a ordem das mensagens na rede, o procedimento retransmite a mensagem até que ela seja confirmada por todos os receptores na mesma transmissão. Quando não é necessário preservar a ordem, o procedimento pode ser otimizado para manter as respostas na bolsa de uma retransmissão para outra. Isso faz com que a bolsa de respostas encha mais rápido, em casos onde nem todos os processos confirmam o recebimento da mensagem na primeira transmissão.

O procedimento tr-w-resp é utilizado no xAMP para proporcionar a entrega confiável de mensagens. Ativado pelo emissor da mensagem, este precisa permanecer correto durante a execução do protocolo, senão o número de receptores da mensagem não pode ser determinado. Uma primitiva de comunicação muito eficiente é oferecida desta forma pelo xAMP, com o nome de *bestEffort*. Do ponto de vista do emissor, *bestEffort* é somente uma chamada a *tr-w-resp*. A escolha apropriada de *Pr* e *nr* permite um retorno rápido em caso de omissões, quando nem todos os destinatários precisam receber a mensagem. Por exemplo, quando *nr* = 0, o procedimento termina imediatamente após o envio da mensagem sem esperar por respostas. Isto corresponde a um multicast não-confiável.

A qualidade de serviço *bestEffort* não é capaz de garantir a entrega em caso de falha do emissor. Para assegurar a entrega de mensagens na presença de falhas do emissor, chamado de *atLeast* QOS, cada receptor é responsável pela terminação do algoritmo. Em conseqüência o procedimento *tr-w-resp* é invocado tanto pelo emissor quando pelos receptores. Entretanto, para evitar retransmissões desnecessárias de mensagens, os receptores não executam a primeira etapa do procedimento t*r-w-resp*, utilizando o parâmetro booleano *send* para indicar esta situação. No caso de não ocorrência de falhas, a mensagem será reconhecida por todos os receptores desejados, estes reconhecimentos serão vistos por todos os participantes e não será necessária nenhuma retransmissão.

No serviço *atLeast*, do mesmo modo que na primitiva *bestEffort*, algumas variantes podem ser obtidas pela escolha apropriada de *Pr* e *nr*. Por exemplo, se for escolhido *nr* de maneira que *nr* seja menor que o número de processos conectados, a primitiva vai assegurar que ao menos *nr* dos processos receberão a mensagem. Isto pode ser satisfatório para implementar protocolos baseados em quórum. Quando *Pr* é igual a todos os processos conectados e *nr* é igua ao número de processos em Pr, a primitiva é também chamada de multicast confiável, que é a base das primitivas de multicast causal e atômico.

- **Difusão Causal**

No xAMP a ordenação causal é obtida com o uso de *históricos de causalidade*, ou seja, mantendo um registro das mensagens enviadas e recebidas e trocando estas informações junto com as mensagens de dados. Um histórico causal é uma lista de pares causais ($id(m)$, $D(m)$), onde $id(m)$ é o identificador da mensagem e $D(m)$ é o conjunto dos receptores da mensagem. Uma mensagem enviada através da primitiva de multicast causal sempre carrega o histórico causal do seu emissor. Este histórico causal é modificado todas vez que uma mensagem é enviada ou entregue. Quando uma mensagem é enviada, o seu par causal é adicionado ao histórico causal do emissor e quando é entregue, o par da mensagem e o histórico causal associado à mensagem são adicionados ao histórico causal do receptor.

Para poder ser entregue, uma mensagem precisa se tornar *estável*. Uma mensagem m é estável, em um dado processador k, se todas as mensagens no histórico causal de m, que eram destinadas a k já foram entregues. Para prevenir o crescimento infinito dos históricos causais, periodicamente são removidos identificadores estáveis. O tempo entre uma remoção e outra é dado por Δ, que eqüivale ao tempo máximo necessário para o envio de uma mensagem e sua posterior entrega.

- **Difusão Atômica**

A qualidade de serviço de multicast atômico garante a ordenação total das mensagens em todos os receptores. Para preservar a ordenação, um mecanismo precisa garantir que as mensagens sejam entregues aos usuários respeitando a ordem em que foram transmitidas, e que se uma mensagem for retransmitida, apenas uma destas retransmissões será usada para estabelecer esta ordenação.

Cada receptor mantém uma fila de recepção, onde as mensagens são inseridas pela ordem em que atravessam a rede. Como um receptor não pode saber se uma mensagem que acabou de receber já foi recebida pelos outros membros do grupo, esta mensagem não pode ser entregue imediatamente. Assim, esta mensagem recebe um carimbo de *não-aceita* e é mantida na fila até que esteja assegurado de que foi inserida na mesma posição em todas as filas dos outros receptores. Se uma retransmissão é recebida, a mensagem é enviada para o fim da fila.

O emissor invoca o procedimento *tr-w-resp* ativando o indicador *ord*, requisitando assim a retransmissão da mensagem até que todos os receptores reconheçam a mesma tentativa. Quando for detectada uma transmissão de sucesso, o emissor envia uma mensagem de aceitação, fazendo a confirmação da mensagem difundida. Quando a mensagem de aceitação é recebida, os receptores marcam a mensagem associada como aceita e a entregam assim que ela alcançar o topo da fila.

Se um receptor não pode processar a mensagem, por exemplo, por falta de espaço em buffers, ele notifica o emissor através de uma mensagem de não-ok. Com isto o emissor envia uma mensagem de rejeição da mensagem e todos os outros processos receptores devem descartar a mensagem rejeitada.

O serviço atômico consiste de um protocolo de aceitação de duas fases que se parece com um protocolo de commit, onde o emissor coordena o protocolo. Na fase de disseminação, a mensagem de dados é enviada para todos os receptores, que devem responder se serão capazes de processar a mensagem. Na segunda fase (fase de decisão), o emissor envia uma mensagem de aceitação ou de rejeição da mensagem disseminada. Para aumentar o desempenho, a mensagem é enviada utilizando um esquema de reconhecimento negativo: se um receptor não recebeu uma mensagem de

decisão, devido a uma falha de omissão, ele irá detectar este problema através de um timeout e enviará um frame de requisição de decisão (Request-Decision). Usando este esquema, uma segunda rodada de reconhecimento é evitada, aumentando o desempenho do protocolo.

Como em ambas as fases, o emissor é quem coordena o protocolo, é necessário um mecanismo para mascarar as falhas do emissor. No serviço atômico, a execução do protocolo é continuada, no evento de uma falha do emissor, por um protocolo de terminação. Este protocolo de terminação é disparado por uma função de monitoramento atômica. Entretanto, não existe nenhuma atividade de monitoramento permanente. O monitor somente existe quando necessário, ou seja, quando um processo do grupo desconfia de outros processos e dispara a função de monitoramento. O monitor substitui o emissor falho, mas nunca retransmite uma mensagem em seu nome, ele somente coleta informações sobre o estado da transmissão e dissemina a decisão a ser tomada.

Desta forma, o xAMP oferece várias possibilidades de utilização dos seus protocolos, todos eles baseados em um primitiva básica e depois estendidos para assegurar outras propriedades como difusão causal e atômica. Estas possibilidades estão, no entanto, limitadas aos protocolos implementados no xAMP e exigem sempre o uso da primitiva de tr-w-resp, não permitindo a criação de novos protocolos, nem uma combinação dos já existentes. Muitas aplicações têm seus requisitos atendidos por estas primitivas de comunicação do xAMP, porém existem outros que não são atendidos.

2.3.2 HORUS - ISIS

O projeto Horus foi desenvolvido na Universidade de Cornell, e é uma continuação do ambiente ISIS, desenvolvido nesta mesma universidade no final da década de 80. O Horus utiliza uma arquitetura onde protocolos são construídos através da utilização de vários pequenos microprotocolos, que suportam uma interface comum [REN96], [BIR96].

O núcleo do Horus é uma abstração de grupo dimensionada para ser flexível o suficiente para vários tipos de grupos de processos, e ainda proporcionar um ambiente de implementação de protocolos. O sistema é composto de vários blocos, que oferecem um serviço bem definido, e que podem ser encaixados de diversas maneiras para formar uma pilha, que implementa um protocolo completo. Entretanto nem todas estas combinações fazem sentido. Existem cerca de 40 blocos disponíveis [REN96], sendo que dez blocos somente podem ser utilizados como base ou topo da pilha. Eles provêem ou uma interface de baixo nível para o sistema operacional, ou então uma interface de alto nível para o usuário.

Tipicamente cada bloco é um módulo de software com uma série de pontos de entrada para chamadas do nível superior e para o nível inferior. Existe um conjunto mínimo de chamadas que cada nível deve suportar. Cada bloco registra-se no runtime do Horus com um nome ASCII e o usuário especifica uma lista deste nomes que devem ser instanciados, sendo permitido que diversas pilhas independentes existam no sistema, cada uma com suas próprias threads e espaço de memória, conforme pode-se ver na figura 2.5 [REN96]. A aplicação escolhe uma das pilhas existentes que deseje utilizar.

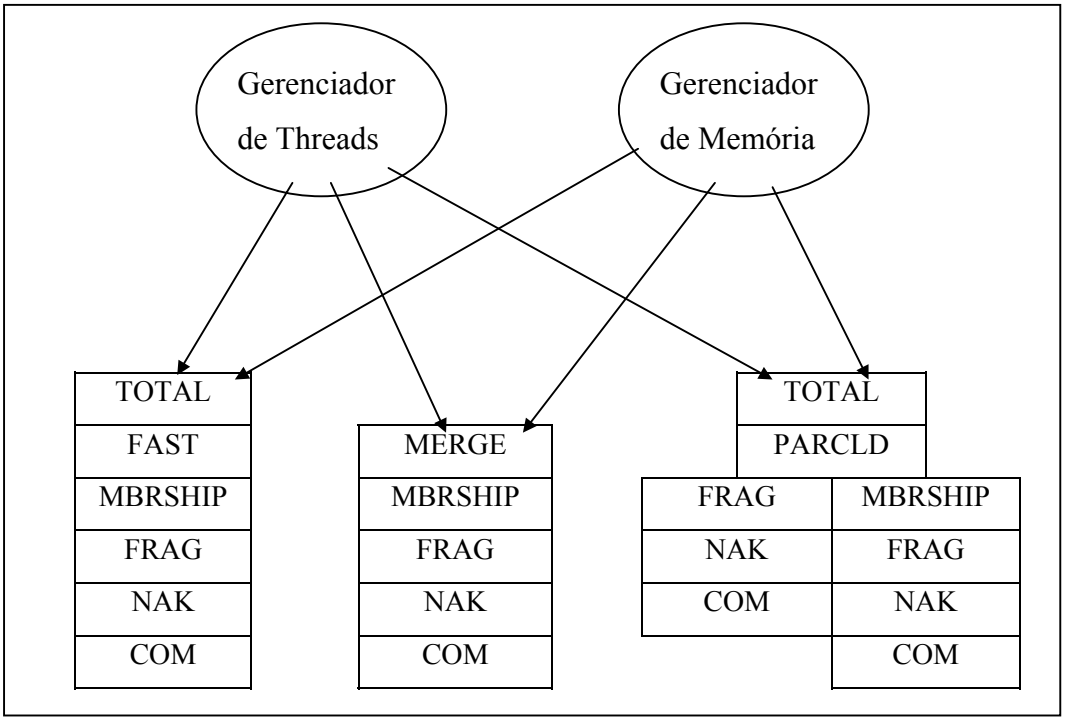

FIGURA 2.5 - Pilhas de protocolos no Horus

Esta arquitetura multi-threaded do Horus é favorável tanto do ponto de vista de desempenho, através do aumento da concorrência, quanto do ponto de vista de simplicidade, devido à modularidade do código [REN96].

A seguir são listados alguns dos níveis mais importantes presentes no Horus:

- COM (Comunicação básica) - não executa nenhum protocolo, somente encapsula a interface com a camada de troca de mensagens da rede (IP, UDP, ATM, MACH);
- NAK (Comunicação FIFO) - oferece multicast FIFO confiável sobre os mecanismos de comunicação não-confiável;
- FRAG (Fragmentação) - implementa a fragmentação e defragmentação de mensagens muito grandes para o tamanho do pacote aceito pela rede;
- MBRSHIP (Composição) - fornece o serviço de gerenciamento da composição dos grupos;
- FC (Controle de Fluxo) - implementa o controle de fluxo, utilizando um mecanismo baseado em janelas e no conceito de mensagens estáveis;
- CAUSAL (Ordenação Causal) - garante a ordenação causal entre as mensagens enviadas para um grupo, utilizando vetores-de-relógio;
- TOTAL (Ordenação Total) - garante a ordenação total de mensagens enviadas para um grupo.
- LWG (Grupos leves) - implementa um serviço de grupos leves (*lightweight*), onde diversos grupos lógicos são multiplexados em apenas um grupo em nível de sistema. Funciona similarmente a um mecanismo de se ter várias threads dentro de um único processo.

3 Descrição do Ambiente FlexGroup

3.1 Introdução

Com o objetivo de se avaliar e comparar o funcionamento de diversos protocolos de comunicação em grupo foi realizado um estudo sobre esses protocolos [RIV96], enfocando as características desejáveis na comunicação em grupo e o funcionamento de alguns dos principais protocolos de comunicação em grupo existentes na literatura. Neste estudo foi detectado que os serviços de comunicação em grupo existentes atendiam somente algumas aplicações, não sendo apropriados para outras. Surgiu então a idéia de se projetar um ambiente de comunicação em grupo que pudesse atender o maior número possível de aplicações e que fosse de fácil utilização e extensão.

Neste capítulo será apresentado o projeto de um ambiente de desenvolvimento de protocolos de comunicação em grupo, denominado FlexGroup. Serão apresentadas as principais características da arquitetura adotada, bem como as funcionalidades oferecidas pelo ambiente. As principais decisões do projeto buscaram agregar facilidades tanto para o desenvolvimento de novos protocolos de comunicação em grupo, quanto para o desenvolvimento de aplicações que utilizem este paradigma de comunicação.

Foi decidido desenvolver um protótipo, contemplando as características descritas na arquitetura proposta, que pudesse ser utilizado para validar esta arquitetura e permitir uma avaliação da utilização do ambiente para programação de aplicações distribuídas e para o desenvolvimento de novos protocolos de comunicação em grupo.

O ambiente permite que a aplicação possa, dinamicamente, determinar o tipo de protocolo que deseja utilizar, sem necessidade de ter conhecimento sobre a implementação interna do protocolo, tendo uma interface comum bem definida para acesso aos diversos serviços que podem ser utilizados.

Da mesma forma, alguém que se proponha a desenvolver um novo protocolo, ou realizar comparações com os protocolos existentes, pode reutilizar as diversas propriedades já implementadas por outros protocolos desenvolvidos no sistema. Para poder atingir esta meta, foi necessário definir os principais componentes de um protocolo, e definir interfaces para integração dos diversos módulos que implementam estes componentes, além de um mecanismo para gerenciar a integração entre estes módulos e produzir uma interface única entre o ambiente de comunicação em grupo e as aplicações que o utilizam.

O ambiente foi projetado utilizando orientação a objetos, visando permitir uma melhor reutilização de componentes de software através de mecanismos como herança, métodos virtuais e sobrescrita de métodos derivados. Tanto a biblioteca de usuário como os próprios protocolos de comunicação em grupo foram desenvolvidos desta maneira.

Visando permitir uma maior gama de utilização, procurou-se colocar o menor número de exigências para a utilização do sistema, porém existem algumas características necessárias para o funcionamento correto do sistema. Era necessária a utilização de alguma ferramenta que permitisse a comunicação entre processos localizados em máquinas distintas, devido à complexidade de se utilizar algum mecanismo de comunicação de mais baixo nível, como sockets, por exemplo. Foi escolhido o sistema HetNOS [BAR93], devido a sua disponibilidade na UFRGS e ao conhecimento prévio deste ambiente.

Para facilitar o desenvolvimento do protótipo do ambiente foram utilizadas algumas características do HetNOS, como a utilização de strings para identificação dos processos e para o envio das mensagens. O HetNOS também garante que as mensagens enviadas de um processo para outro são confiáveis, porém não garante que a difusão seja confiável. Esta característica foi utilizada para facilitar a implementação da difusão confiável.

3.2 Estrutura do FlexGroup

A arquitetura proposta para o ambiente FlexGroup é estruturada em camadas, com cada camada interagindo somente com a camada inferior. Ele é composto de duas partes principais: um Servidor de Comunicação em grupo, que se comunica com a camada de comunicação e com o sistema operacional e uma biblioteca que é ligada ao programa cliente do serviço de comunicação em grupo e que faz a comunicação com o Servidor. A figura 3.1 resume esta estrutura:

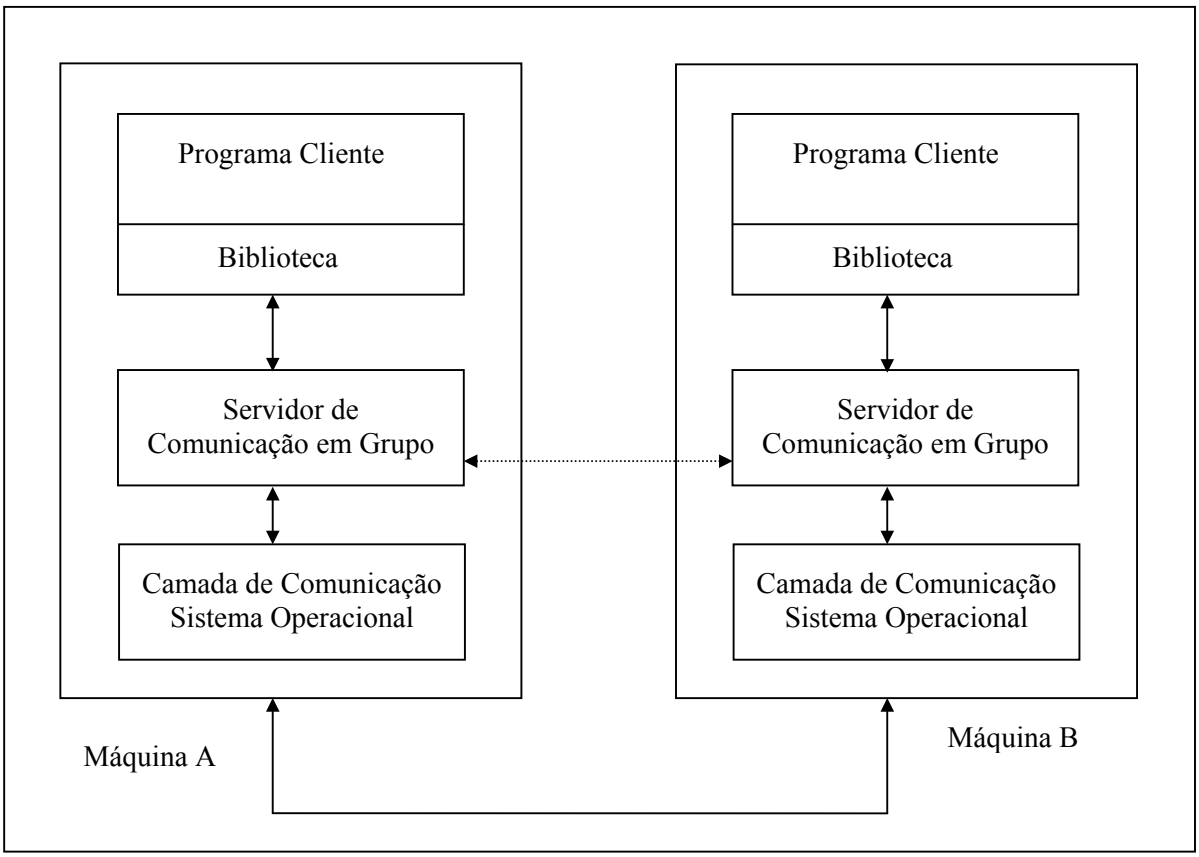

FIGURA 3.1 - Estrutura do Ambiente de Comunicação em Grupo

As aplicações são ligadas a uma biblioteca orientada a objetos, que encapsula a visão do grupo para a aplicação. Esta biblioteca comunica-se com o Servidor de Comunicação, que é quem realmente realiza as operações de comunicação e gerência dos grupos. O Servidor de Comunicação em Grupo se comunica com os outros servidores nas outras máquinas através de uma camada de comunicação colocada sobre o sistema operacional existente. Conforme já foi descrito anteriormente, a camada de comunicação inicialmente escolhida foi o HetNOS [BAR93], Sistema Operacional de

Rede, desenvolvido na própria UFRGS e que oferece primitivas para o envio e recepção de mensagens, de maneira similar ao PVM [BEG93].

Esta estrutura deve estar replicada em todas as máquinas que desejarem utilizar o ambiente FlexGroup. Os Servidores de Comunicação em cada máquina são os responsáveis pelo envio e recepção das mensagens dos processos pertencentes aos grupos e também pela troca de mensagens dos próprios subprotocolos, além de se comunicarem entre eles para atualizar a visão dos grupos. O Servidor de Comunicação engloba um núcleo, responsável por estas atividades, e os subprotocolos, que são os responsáveis pelos algoritmos de comunicação em grupo, conforme será descrito na seção 3.3.

Os Servidores de Comunicação do FlexGroup são também um grupo, sendo necessário que eles tenham uma visão única das máquinas que participam deste grupo, e que as mensagens enviadas por um servidor para os outros apresentem o comportamento de difusão confiável descrito no capítulo anterior. Além disso, deve-se garantir que falhas em algum destes processos não impeçam o funcionamento do ambiente. Porém este tratamento de difusão confiável e de tolerância a falhas não é estendido às aplicações em grupo e as mensagens enviadas por ela, sendo necessária a utilização de protocolos que realizem estas tarefas.

3.3 Biblioteca

A biblioteca do FlexGroup (flexlib) é orientada a objetos e disponibiliza para as aplicações de comunicação em grupo uma classe *Group* que encapsula a visão de um grupo para a aplicação. Para poder utilizar a biblioteca, o desenvolvedor da aplicação deve incluir na aplicação o arquivo **flexgrp.h**, que contém a definição das constantes, classes e funções necessárias para utilizar o FlexGroup.

Para cada grupo em que o processo deseje fazer parte, ele deve criar um objeto da classe *Group*. Os grupos são identificados por um nome simbólico (string), visando facilitar a identificação dos grupos e a programação das aplicações Todas as operações de comunicação em grupo são invocadas a partir destes objetos, e a comunicação com o servidor é toda encapsulada por esta classe, bem como a visão do grupo para a aplicação. Esta classe possui diversos métodos públicos que são as interfaces pelas quais as aplicações acessam os serviços do ambiente. Os métodos públicos existentes são os seguintes:

- GroupExists (GroupName) – Verifica se um grupo com o nome informado já existe no ambiente. É realizada uma consulta ao Servidor local que verifica a sua existência localmente e responde para a aplicação.
- CreateGroup (GroupName, Protocols, Owner) – Cria um grupo identificado por GroupName. Protocols é um string com os nomes simbólicos dos protocolos a serem utilizados, separados por espaços, na ordem em que serão colocados na pilha de subprotocolos. Estes nomes estão definidos no arquivo de cabeçalho flexgrp.h que deve ser incluído na aplicação. Owner identifica o processo responsável pelo grupo, somente este processo pode encerrar o grupo a não ser que ele deixe de fazer parte do grupo, passando o grupo a não ter mais responsável. Se este parâmetro for nulo, qualquer processo pode encerrar o grupo.
- DestroyGroup – Destrói o grupo, desalocando todas as estruturas reservadas para ele. Somente o responsável pelo grupo pode chamar este método, caso tenha sido definido um responsável na sua criação; se outro processo tentar

encerrar o grupo, é retornado um erro. O servidor não executa esta operação imediatamente, pois podem existir mensagens pendentes. Portanto, a criação de um grupo com o mesmo nome logo após a sua destruição pode retornar um erro, pois o servidor ainda não o tinha excluído totalmente. Deve-se chamar o método GroupExists para verificar a sua existência.

- JoinGroup (GroupName, ProcessName)- Insere o processo identificado por ProcessName no grupo identificado por GroupName e busca as informações relativas a este grupo no servidor. Se o grupo tiver sido criado por este processo, o parâmetro GroupName não precisa ser informado. Após esta operação, a aplicação pode enviar e receber mensagens do grupo indicado.
- LeaveGroup (ProcessName) - Retira um processo do grupo. Não afeta o objeto grupo que continua existindo, nem o grupo, que continua operando normalmente, porém não permite mais o envio ou recepção de mensagens para este grupo por parte da aplicação que solicitou a saída.
- Send (msg) - Envia uma mensagem para o grupo. A mensagem é enviada pela biblioteca para o Servidor que a encaminhará para os seus destinatários, passando pelos subprotocolos.
- Receive (&msg)- Recebe uma mensagem do grupo e a armazena em msg. A mensagem é recebida do Servidor de Comunicação depois de ter passado por todos os subprotocolos definidos para o grupo. Não existe fila de mensagens na aplicação, pois as mensagens ficam armazenadas no Servidor, somente sendo enviadas à aplicação quando requisitadas pela chamada do método receive.
- GetGroupInfo (GroupName, &Info) - Recupera as informações a respeito de um grupo. As informações retornadas são os subprotocolos que o grupo utiliza e a visão corrente da composição do grupo para o Servidor de Comunicação local.

Além destes métodos públicos, a classe possui métodos privados que encapsulam a comunicação com o servidor de comunicação em grupo. No construtor do objeto é feita a conexão com o HetNOS, caso ela ainda não exista. Em seguida, utilizando já as primitivas de comunicação HetNOS, é feita a conexão com o Servidor de Comunicação. No destrutor do objeto *Group* são desalocadas as estruturas internas do objeto e, se for o último objeto *Group* da aplicação, é realizada a desconexão da aplicação do HetNOS. Todos os métodos públicos descritos acima utilizam primitivas de comunicação HetNOS, garantindo a comunicação confiável entre a biblioteca e o servidor.

Todos os objetos da classe *Group* que venham a ser instanciados compartilham o mesmo canal de comunicação HetNOS com o Servidor de Comunicação em Grupo, porém como fazem parte de um único processo, as operações são serializadas, não existindo problemas de interferência entre dois ou mais grupos. As aplicações podem também utilizar as primitivas de gerência de processos e de comunicação ponto-a-ponto do HetNOS, em conjunto com a utilização do serviço de comunicação em grupo.

3.4 Servidor de Comunicação em Grupo

O Servidor de Comunicação em Grupo é o responsável pela implementação dos protocolos de comunicação em grupo. Ele deve ser executado em todas as máquinas onde existam aplicações que desejem utilizar os serviços de comunicação em grupo. Como o servidor utiliza os serviços de comunicação do HetNOS, é necessário que o HetNOS esteja sendo executado também na máquina onde o servidor for executado. O Servidor de Comunicação em Grupo funciona como uma aplicação HetNOS normal,

não estando integrada ao sistema e podendo ser portado para utilizar outras bibliotecas de comunicação como PVM ou MPI. Maiores informações sobre como utilizar o HetNOS podem ser encontradas em [BAR93].

Para disparar o servidor, é necessário indicar o nome de uma máquina onde um outro servidor já tenha iniciado a sua execução. Os servidores nas diversas máquinas comportam-se como um grupo. O novo servidor utiliza este servidor remoto, informado na sua chamada, para obter a lista dos servidores e conseguir comunicar a todos da sua entrada no grupo de servidores. Caso não seja indicada nenhuma máquina ao disparar o novo servidor, ele assume que é o primeiro a ser disparado e cria um grupo de servidores que contém somente a si próprio.

O Servidor de Comunicação em Grupo possui dois componentes: o núcleo e os subprotocolos. O núcleo é responsável pela comunicação com as aplicações e com as outras máquinas e por repassar as mensagens e informações sobre mudanças nos grupos aos subprotocolos. Cada subprotocolo implementa uma característica da comunicação em grupo e opera independentemente dos outros subprotocolos que estejam sendo utilizados. Estes subprotocolos são organizados na forma de uma pilha, onde cada subprotocolo no envio de mensagens agrega as suas informações à mensagem e passa para o nível de baixo, e ao receber uma mensagem, retira as suas informações e passa para o nível de cima. Esta passagem para o nível de cima não é automática, a mensagem pode ficar parada em um nível até que uma condição seja satisfeita. Existe uma pilha independente para cada grupo, porém existe somente uma instância de cada subprotocolo. Desta forma, um subprotocolo pode estar acima de outro subprotocolo em um grupo e abaixo em outro. Por exemplo, em um grupo, o subprotocolo de detecção de falhas pode ser realizado antes do de ordenação (detecção de falhas está abaixo da ordenação), enquanto em outro a ordenação ocorre primeiro (detecção fica acima da ordenação). O resultado destas composições é determinado pelos protocolos utilizados e pela ordem na qual forem colocados. Esta ordenação da execução dos subprotocolos é definida na criação do grupo, através da ordem dos subprotocolos informados. A figura 3.2 exemplifica como grupos diversos podem utilizar os mesmos subprotocolos em ordens diferentes:

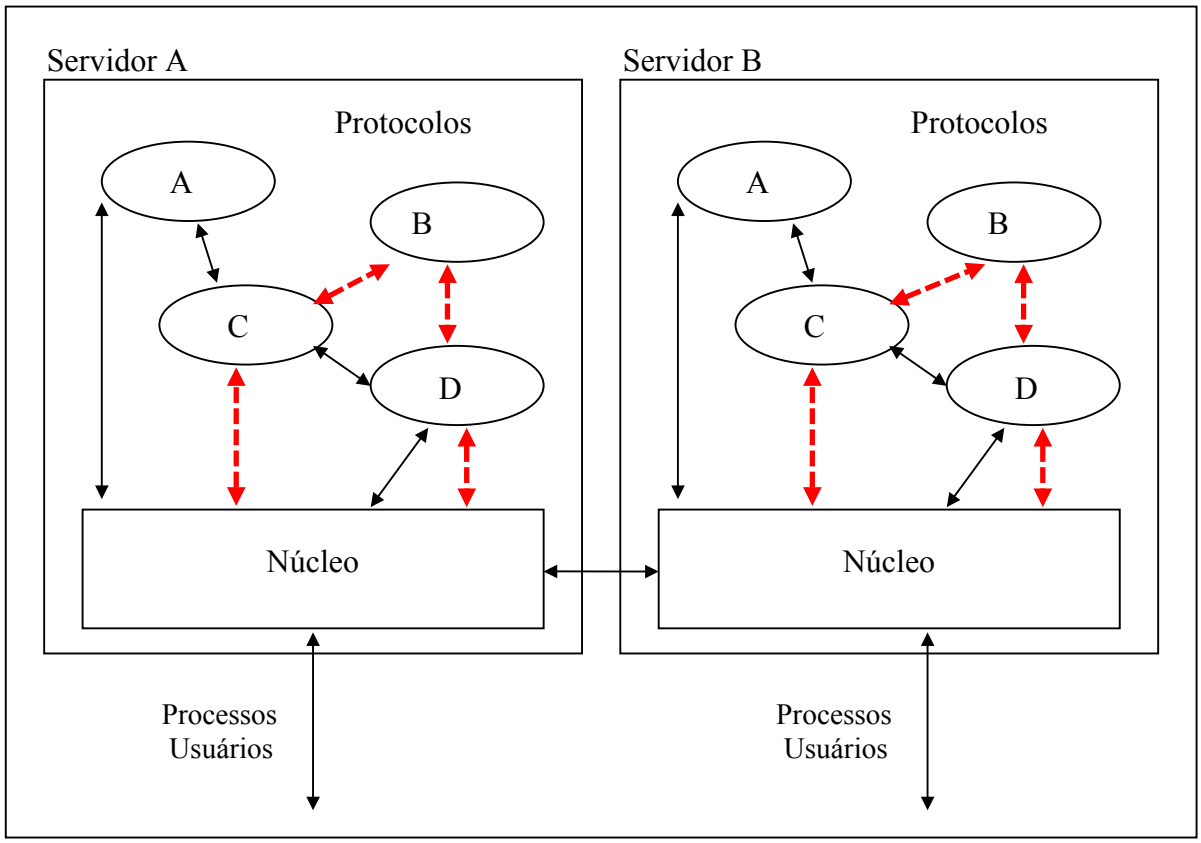

FIGURA 3.2 - Estrutura do Servidor de Comunicação em Grupo

Na figura 3.2, o fluxo das mensagens enviadas e recebidas está indicado pelos arcos entre os diversos componentes. Neste caso existem dois grupos ativos no sistema. O primeiro grupo utiliza os subprotocolos A, C e D, nesta ordem, enquanto que o segundo utiliza, em ordem, C, B e D. Cada grupo opera independentemente, mesmo possuindo subprotocolos em comum, como no exemplo acima.

3.4.1 Núcleo do Servidor

O núcleo é o principal componente do servidor, sendo o único a ser disparado automaticamente na inicialização do servidor, sendo que os subprotocolos somente são inicializados quando forem necessários. Ao ser disparado, o núcleo precisa obter informações sobre os outros servidores já inicializados, por isso é necessário informar o nome de uma máquina onde o servidor já esteja rodando. O servidor desta máquina é considerado como "coordenador" da entrada do novo servidor no sistema, sendo responsável por informar a configuração atual dos servidores e dos grupos existentes, bem como repassar possíveis mudanças nesta visão, devido a entrada ou saída de servidores ou a criação ou destruição de grupos, para o novo servidor, até que todos os outros servidores tenham sido informados desta entrada e passem a enviar as mensagens diretamente para ele. Se ocorrer um erro no coordenador que faça com que ele saia do grupo de servidores, antes da confirmação da entrada do novo servidor, esta é abortada, pois alguma mensagem de alteração dos grupos pode não ter sido repassada pelo coordenador durante a entrada do novo servidor. Mesmo que o novo servidor tenha conseguido se comunicar com os outros servidores que permanecem ativos, a sua

entrada no grupo não será efetuada se o coordenador falhar, garantindo desta forma consistência na visão dos grupos. Os algoritmos utilizados na manutenção do grupo de servidores são descritos na seção 3.5.

O núcleo possui informações sobre os grupos existentes no sistema, tais como nome, processos pertencentes ao grupo, última mensagem recebida, e sobre os processos locais que participam de algum grupo. Para cada objeto *Group* instanciado por um processo e inicializado na biblioteca, é gerado um objeto *Client* correspondente no núcleo, contendo as informações sobre o processo, o grupo que ele representa e as últimas mensagens recebidas. Estas informações estão disponíveis para os subprotocolos que podem requisitá-las ao núcleo.

O núcleo é responsável ainda por disparar os subprotocolos quando um processo local criar ou entrar em um grupo, caso o subprotocolo ainda não tenha sido criado, e também informá-los sobre a criação ou destruição de grupos que os utilizem. Além disso, o núcleo comunica aos subprotocolos da entrada ou saída de processos nos grupos que os utilizam.

A comunicação entre os diversos núcleos é por difusão confiável, utilizando um algoritmo com confirmações positivas e limites de tempo, que será descrito em mais detalhes nas próximas seções. Porém esta difusão confiável somente é utilizada para mensagens geradas pelos próprios núcleos para garantir a visão consistente da composição do grupo de servidores. As mensagens enviadas pelas aplicações ou pelos subprotocolos são enviadas através de difusão simples, sem garantia de entrega, ficando a difusão confiável a cargo de um subprotocolo, se assim for desejado.

Mecanismos de armazenamento de mensagens para retransmissão não são necessários, pois as mensagens ponto-a-ponto do HetNOS são confiáveis. Portanto, devido a esta característica da camada de comunicação HetNOS, se um processo não recebeu uma mensagem, ocorreu um erro ou no nodo emissor ou no nodo receptor e um dos dois servidores será retirado do grupo, não existindo a necessidade de retransmissão. Subprotocolos que tiverem necessidade de retransmissão devem implementar este tipo de fila internamente ao subprotocolo.

3.4.2 Subprotocolos

Os subprotocolos são quem realmente implementa a comunicação em grupo. Cada subprotocolo realiza uma tarefa específica, ou assegura uma propriedade da comunicação em grupo. Podem existir subprotocolos, por exemplo, para garantir a difusão confiável (inclusive mais de um, utilizando algoritmos diferentes), para garantir ordenação FIFO, causal ou total, para realizar a gerência dos grupos, para garantir a atomicidade na comunicação, para garantir sincronismo virtual e outros.

Os subprotocolos são processos independentes que implementam um serviço específico de comunicação em grupo. Para cada tipo de serviço existirá um subprotocolo correspondente em cada núcleo. Diversos grupos podem utilizar os mesmos subprotocolos, porém existirá somente uma instância do processo que implementa o subprotocolo em cada máquina da rede para atender a todos os grupos que o utilizam.

Os subprotocolos são organizados como uma pilha, para cada grupo existente no ambiente, conforme o apresentado na figura 3.2. Para um determinado subprotocolo, o subprotocolo imediatamente abaixo na pilha de subprotocolos da máquina onde está rodando é chamado de subprotocolo **inferior**. Da mesma forma, o subprotocolo situado imediatamente acima na pilha é chamado de subprotocolo **superior**. Para o primeiro subprotocolo da pilha, o subprotocolo inferior é o próprio núcleo e, para o último

subprotocolo, o subprotocolo superior também será o núcleo, fechando o caminho que a mensagem deve percorrer para ser enviada ou recebida. A pilha de subprotocolos é replicada em todas as máquinas que possuem o FlexGroup ativos, os protocolos semelhantes nas máquinas remotas são denominados de subprotocolos **pares**.

Eles podem ser colocados em qualquer ordem pelo grupo, na sua criação, porém nem todas as combinações possíveis fazem sentido ou até mesmo funcionam. Cada subprotocolo se preocupa apenas com a execução de sua tarefa. Os subprotocolos devem ser derivados de uma classe virtual já existente *Protocol* que já possui funções para gerenciar para cada grupo a comunicação com os subprotocolos superior e inferior, com o núcleo e com os subprotocolos pares nas outras máquinas.

Cada subprotocolo, no envio de uma mensagem por um processo local, recebida através do núcleo, agrega suas informações ao cabeçalho da mensagem (se houver informações a serem agregadas) e envia para o subprotocolo imediatamente inferior, até a mensagem chegar novamente ao núcleo, quando será difundida para as outras máquinas. Na recepção, o ciclo é inverso. Ao receber uma mensagem, o subprotocolo extrai as informações do cabeçalho, verifica se ela pode ser entregue para os processos locais e, em caso positivo, ela é passada para o subprotocolo imediatamente superior, até chegar ao núcleo, quando será armazenada na fila de mensagens associada a cada cliente do Servidor para futuro envio à aplicação.

Os subprotocolos também podem enviar mensagens diretamente para os subprotocolos pares em outras máquinas, dependendo do tipo de algoritmo que implementarem. Este tipo de mensagem não passa pela pilha, e é enviada diretamente para o núcleo que a encaminha para as outras máquinas. Normalmente, somente mensagens de controle ou sincronização do subprotocolo são enviadas desta maneira.

Os subprotocolos também podem receber mensagens diretamente do núcleo. Este tipo de mensagem pode ser uma mensagem enviada pelo mesmo subprotocolo em outra máquina, conforme explicado acima, ou pode ser uma mensagem gerada pelo próprio núcleo para informar uma alteração nos grupos, ou para informar que o protocolo não é mais utilizado e pode ser encerrado.

3.5 Funcionamento do Sistema

A seção anterior descreveu a estrutura do sistema, em função das funcionalidades oferecidas, do fluxo de mensagens e das características apresentadas pelo ambiente. Nesta seção, serão apresentadas as soluções adotadas para atender a estes requisitos.

Apesar de deixar a implementação das características de difusão confiável, gerenciamento de grupos, tolerância a falhas e outras para os subprotocolos, o Servidor precisa manter uma visão consistente do grupo de servidores e dos grupos existentes, além dos processos vinculados a estes grupos.

A seguir, são descritas as principais operações efetuadas pelo ambiente e os principais protocolos utilizados pelo ambiente.

3.5.1 Camada de Comunicação

Conforme o descrito anteriormente, o FlexGroup não implementa uma camada de comunicação, utilizando as primitivas de envio e recepção de mensagens ponto-a-ponto oferecidas pelo HetNOS [BAR93]:

- h_send – Primitiva de envio de mensagens de maneira assíncrona

- h_receive – Primitiva de recepção de mensagens de maneira síncrona

Toda a comunicação existente entre a biblioteca do FlexGroup com o Servidor de Comunicação e entre os servidores é feita utilizando estas primitivas.

A comunicação do núcleo com os subprotocolos e dos subprotocolos uns com os outros também utiliza estas primitivas, porque eles estão implementados como processos devido à impossibilidade atual de utilizar threads com o HetNOS. Idealmente os subprotocolos e o núcleo deveriam ser implementados como threads do processo Servidor e possuiriam memória compartilhada, otimizando a comunicação entre eles.

As primitivas de comunicação ponto-a-ponto do HetNOS são confiáveis, ou seja, se uma mensagem enviada por um processo para outro processo não é recebida, assume-se que ocorreu uma falha em um dos dois processos, e nenhuma outra mensagem subseqüente trocada por estes processos será recebida pelo outro processo. Esta propriedade, porém, não é válida para a difusão de mensagens, pois a mensagem pode ser entregue a alguns destinatários e a outros não, caso ocorra uma falha no processo origem durante a difusão. Além disso, as mensagens enviadas por um mesmo processo são serializadas, ou seja, são sempre recebidas na mesma ordem em que foram enviadas, porém esta característica não é válida para mensagens enviadas por processos distintos.

O HetNOS não permite a existência de erros transientes na comunicação entre os nodos, ou seja, a camada de comunicação garante que, se uma mensagem não puder ser enviada entre duas máquinas rodando o HetNOS, uma destas máquinas será considerada como falha e excluída da rede HetNOS. Esta propriedade faz com que o Servidor de Comunicação em grupo não precise se preocupar com particionamento da rede e nem com mensagens inválidas ou repetidas que possam vir a ser enviadas por Servidores que falharem.

3.5.2 Difusão Confiável entre os Servidores

Os diversos Servidores de Comunicação do FlexGroup atuam como um grupo e precisam ter uma visão consistente da composição deste grupo. Para assegurar esta propriedade, é necessário que as mensagens que informam sobre modificações nesta configuração sejam enviadas e recebidas de maneira atômica, sendo necessário um mecanismo de difusão confiável.

Existem diversos protocolos de difusão confiável disponíveis na literatura, com diferentes graus de desempenho e complexidade. Como o número de mensagens trocadas pelos servidores tende a não ser grande, pois não se espera que ocorram muitas mudanças na configuração da rede HetNOS depois que ela tenha sido inicializada, foi decidido adotar um protocolo baseado em confirmações e timeouts de transmissão com o objetivo de simplificar o projeto do protótipo.

Somente são enviadas por difusão confiável as informações sobre alterações na configuração dos Servidores de Comunicação em Grupo e as mensagens de criação e destruição de grupos. As outras mensagens trocadas entre os servidores, os subprotocolos e as aplicações são enviadas pela primitiva de difusão comum do HetNOS, não confiável.

O protocolo empregado é baseado no protocolo Trans, descrito em [JAL94], porém com modificações para aproveitar as características das primitivas de comunicação do HetNOS e atender as necessidades do ambiente. O funcionamento da difusão confiável entre os servidores é o seguinte:

1) O processo origem da mensagem difunde uma mensagem através da primitiva de multicast disponível no HetNOS (não confiável). As mensagens que utilizam a difusão confiável recebem um identificador, composto pelo nome do servidor de origem e um contador que serve para identificar unicamente a mensagem, este contador é incrementado a cada mensagem enviada pelo servidor.
2) Cada servidor ao receber uma mensagem deste tipo, incrementa o valor do contador associado ao servidor origem da mensagem e guarda o tempo da chegada da mensagem. Porém esta mensagem ainda não é tratada, pois depende da entrega em todos os outros servidores.
3) Ao difundir uma mensagem qualquer (através da difusão confiável, ou da difusão comum), o servidor anexa a esta mensagem uma confirmação da maior mensagem recebida dos outros servidores. Isto permite que mais de uma mensagem seja confirmada em uma difusão. Se não houver nenhuma mensagem a ser difundida em um intervalo de tempo x (timeout), o servidor envia uma mensagem nula contendo apenas esta confirmação.
4) O servidor que difundiu a mensagem espera por confirmação de todos os outros servidores. Se receber esta confirmação, ele considera que a mensagem foi difundida com sucesso. Se ele não receber esta confirmação de algum servidor em um intervalo de tempo 2*x, ele difunde uma mensagem de suspeita sobre o servidor que não respondeu conforme será descrito a seguir. Se o servidor origem da mensagem não receber nenhuma confirmação dentro do intervalo de tempo 2*x, ele assume que sofreu uma falha na sua camada de comunicação e encerra sua execução por não conseguir mais se comunicar com os outros servidores.
5) Um servidor somente trata uma mensagem enviada por difusão confiável quando receber uma confirmação de todos os outros servidores que não tenham falhado. Se o servidor que enviou a mensagem for considerado como falho, todas as suas mensagens que estejam esperando confirmação são descartadas.

Na figura 3.3 é apresentado um exemplo de difusão confiável entre três servidores de comunicação:

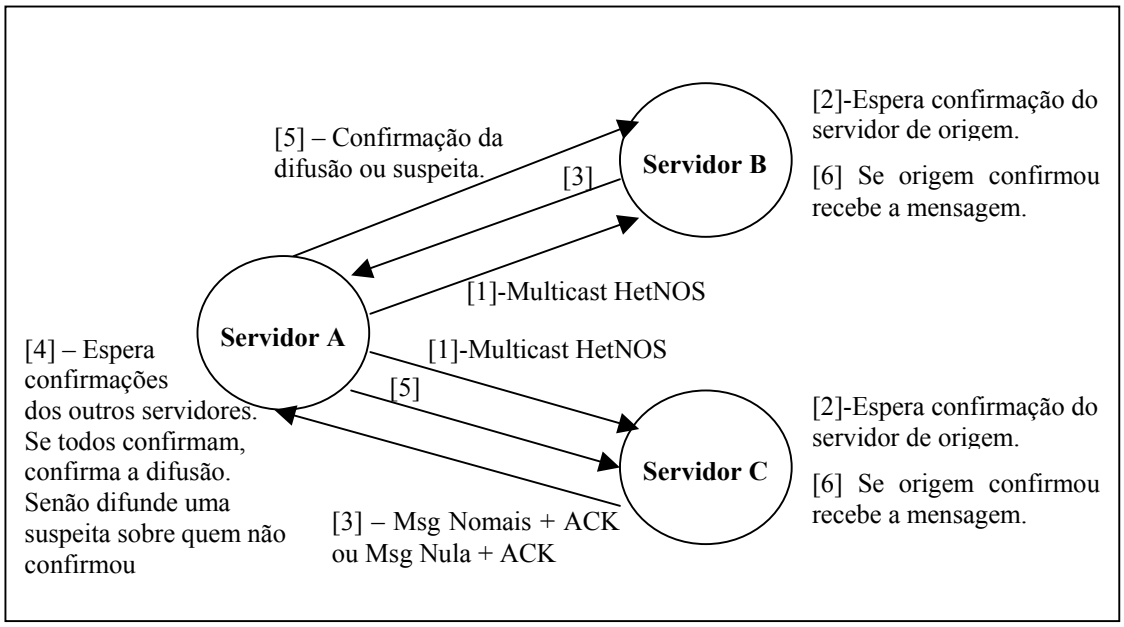

FIGURA 3.3 - Difusão confiável entre os servidores de comunicação em grupo

O tratamento de falhas nos servidores é disparado pela rotina de difusão confiável, conforme descrito acima. Dada a característica de confiabilidade das mensagens HetNOS, um servidor que não recebe uma confirmação de outro servidor dentro do intervalo de tempo estipulado, assume que ocorreu uma falha no servidor remoto ou na máquina onde está rodando. Ele então dispara o seguinte procedimento:

1) Difunde uma mensagem de suspeita sobre cada servidor de quem não recebeu resposta pela difusão normal. Este mecanismo somente é válido para mensagens enviadas entre os servidores por difusão confiável, ou seja, mensagens de alteração no grupo de servidores e de criação ou destruição de grupos. As outras mensagens não possuem tratamento de falhas, que deve ser implementado por um subprotocolo específico.
2) Um servidor que recebe uma mensagem de suspeita age da seguinte maneira: se dentro de um intervalo de tempo x tiver recebido uma mensagem do servidor suspeito, envia uma mensagem negando a suspeita, caso contrário envia uma mensagem confirmando a suspeita.
3) Se o servidor que difundiu a suspeita receber confirmação de todos os outros servidores, ele retira o servidor suspeito da sua lista de servidores e difunde uma mensagem de maneira confiável, comunicando esta modificação ao grupo de servidores.
4) Se o servidor que difundiu a suspeita receber uma mensagem negativa da exclusão do grupo, ele verifica se recebeu a confirmação da difusão anterior que gerou a suspeita (neste caso, não ocorreu uma falha, somente um atraso) e se a difusão tiver sido confirmada, ele continua operando. Se a difusão não tiver sido confirmada, ele assume que ele próprio falhou e se encerra. Este comportamento é necessário para garantir a sincronização entre os servidores, pois se um servidor não consegue difundir uma mensagem de alteração no grupo de servidores, a sua visão do grupo passa a ficar inconsistente em relação aos outros servidores e ele deve encerrar a sua

execução. Os outros servidores que permanecem corretos irão detectar esta saída e excluí-lo do grupo de servidores.

Este algoritmo não garante que processos que ficaram sem se comunicar durante um longo tempo e tenham sofrido um atraso muito grande no envio de suas mensagens de confirmação das difusões não possam ser considerados falhos pelos outros e sejam retirados do grupo. Porém ele garante que sempre que ocorrer uma falha de comunicação que impeça um servidor de receber mensagens dos outros servidores, este servidor será retirado do grupo. O mecanismo de detecção de falhas em um grupo de quatro servidores é exemplificado na figura 3.4. O servidor A não recebeu confirmação do servidor D para uma mensagem enviada por difusão confiável e então difundiu uma suspeita sobre D.

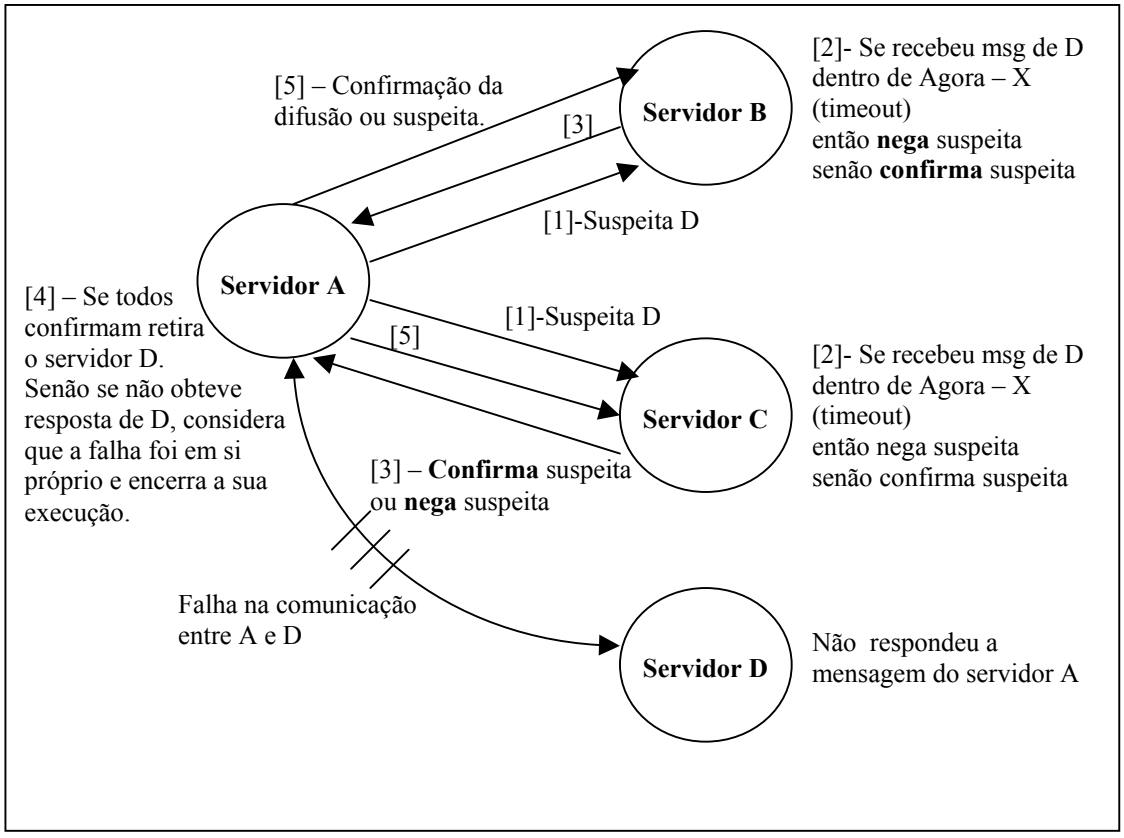

FIGURA 3.4 - Detecção de falhas entre os servidores de comunicação em grupo

3.5.3 Inicialização do Servidor

Quando da inicialização do servidor, podem acontecer dois casos: ele é o primeiro servidor do ambiente, ou já existem outros servidores no ambiente. No primeiro caso, o servidor simplesmente inicializa as suas estruturas e, como ainda não existe nenhum grupo, fica esperando as requisições de criação de grupos e de entrada de novos servidores.

No segundo caso, o comando que carrega o novo Servidor deve informar o nome de uma máquina onde já exista um Servidor FlexGroup executando. Se este parâmetro não for informado, o Servidor assume que ele é o primeiro e age conforme o descrito acima.

O processo de entrada de um novo servidor pode ser visualizado na figura 3.5.

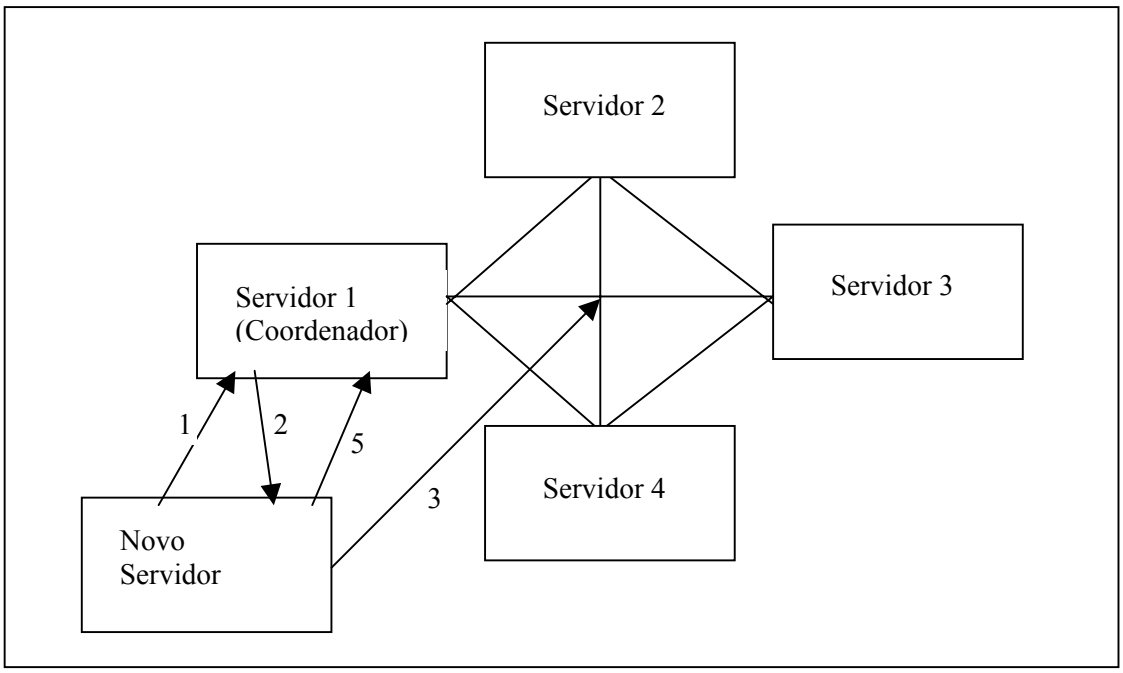

FIGURA 3.5 - Algoritmo de entrada de um novo servidor

O algoritmo de entrada de um novo servidor segue os seguintes passos:

1) O novo servidor envia uma mensagem para o servidor da máquina indicada como parâmetro e este servidor remoto passa a ser o coordenador da entrada do novo servidor.
2) O coordenador responde para o novo servidor, informando a lista dos servidores e a visão dos grupos. Este coordenador também é responsável por repassar para o novo servidor todas as mensagens de alteração dos grupos ou dos servidores que venha a receber ou enviar, durante o processo de entrada do novo servidor.
3) O novo servidor, ao receber a mensagem do coordenador, **difunde** de forma confiável (conforme descrito acima) uma mensagem para todos os outros servidores, informando a sua entrada.
4) Caso ocorra um erro na difusão, de acordo com o algoritmo de detecção de falhas descrito na seção anterior, a entrada do novo servidor é abortada. Da mesma forma, se durante a difusão for detectado um erro no coordenador, a entrada do novo servidor é abortada, pois ele pode não ter recebido alguma mensagem do coordenador, não garantindo a visão consistente dos grupos.
5) Se a difusão ocorre com sucesso, o novo servidor comunica este evento ao processo coordenador, que pára de repassar as mensagens de alteração para ele e passa a fazer parte do grupo de servidores.

3.5.4 Encerramento de um Servidor

O encerramento do Servidor de Comunicação não implica o encerramento das aplicações de comunicação em grupo que o utilizam, porém estas aplicações deixam de

fazer parte dos grupos a que pertenciam e qualquer operação da biblioteca de comunicação em grupo que tentarem disparar retornará um erro.

O servidor pode ser encerrado por um comando do usuário no console de onde foi disparado o servidor ou quando for considerado falho. No primeiro caso, o núcleo do servidor executa o seguinte procedimento:

1) Envia uma mensagem para todas as aplicações locais comunicando a sua finalização.
2) Envia uma mensagem para todos os subprotocolos comunicando o seu encerramento.
3) Espera uma resposta dos subprotocolos confirmando a saída. Isto é necessário, pois os subprotocolos podem necessitar executar operações próprias de finalização, inclusive enviar mensagens para os subprotocolos das outras máquinas.
4) Envia por difusão confiável uma mensagem para todos os outros servidores comunicando a sua saída.
5) Desaloca as estruturas internas e as áreas de informação sobre os grupos e sobre as aplicações e encerra a execução.

No caso de ocorrência de falha, o Servidor faltoso não consegue se comunicar com algum dos outros servidores e ele tentará enviar uma mensagem de suspeita sobre o Servidor com o qual não conseguiu se comunicar. Como não receberá a confirmação desta suspeita, ele considerará que a falha ocorreu nele próprio, porque as mensagens ponto-a-ponto do HetNOS são confiáveis. Desta forma, ele executará o mesmo procedimento descrito acima, com exceção do passo 4.

Os outros servidores detectarão a sua falha ao tentarem se comunicar com ele e enviarão mensagens de suspeita, que serão em algum momento confirmadas pelos outros servidores, pois o servidor que falhou deixará de se comunicar com eles. Os outros servidores o retirarão então da sua visão do grupo.

3.5.5 Criação de um Grupo

A criação de um novo grupo é disparada por um processo aplicativo através da biblioteca do FlexGroup, porém o responsável pela criação do grupo é o núcleo do Servidor de Comunicação em Grupo. O Servidor local deve garantir que não exista outro grupo com o mesmo nome no sistema e que os subprotocolos indicados como parâmetro da criação do grupo existam e possam ser executados. Para verificar esta condição, é necessário que ele obtenha dos outros servidores remotos uma confirmação da possibilidade de criação do novo grupo. As mensagens difundidas pelo Servidor local utilizam a difusão confiável já descrita na seção 3.4.1.

O grupo inicialmente não possui processos associados a ele, portanto o Servidor não precisa se preocupar com a composição do grupo durante a sua criação, ao contrário de protocolos onde na criação pode-se já informar uma lista de processos componentes. Na criação do grupo, é informado o processo responsável (owner) do grupo, somente este processo poderá destruir o grupo. Se nenhum responsável for informado na criação do grupo, qualquer membro do grupo poderá solicitar a sua destruição. Se o processo responsável for encerrado ou sair do grupo antes de destrui-lo, o grupo deixa de ter um responsável, podendo igualmente ser destruído por qualquer outro membro do grupo.

O procedimento de criação de um novo grupo é o seguinte:

1) A aplicação chama o método CreateGroup da biblioteca FlexGroup, informando o nome do grupo e os subprotocolos a serem utilizados.
2) A biblioteca envia uma mensagem ao Servidor local solicitando a criação do novo grupo
3) O servidor local verifica as condições para a criação do grupo (não existe outro grupo com o mesmo nome, os subprotocolos informados são válidos, não existe um pedido pendente de criação de grupo com o mesmo nome e outras condições). Caso uma delas não seja válida, retorna um erro para a biblioteca e aborta a operação.
4) Se o pedido de criação passou pela validação do passo anterior, o núcleo do Servidor local difunde uma mensagem para os outros Servidores, solicitando a criação do novo grupo.
5) Os servidores remotos respondem confirmando ou negando a criação do novo grupo, após terem executado o mesmo processo de validação descrito no passo 3. Se dois processos tentarem criar um grupo com o mesmo nome, ao mesmo tempo, pode ser que nenhum dois consiga, pois os servidores não tem como definir qual pedido teria preferência. Porém ao receber o retorno da função com o código de erro de pedido pendente, a aplicação pode tentar novamente criar o grupo, após um pequeno intervalo de tempo.
6) O servidor local difunde aos outros servidores a decisão da consulta.
7) Se as respostas foram todas positivas, o servidor local dispara os subprotocolos que ainda não estejam ativos e informa aos que já estavam ativos sobre a criação do novo grupo, retornando para a aplicação uma mensagem confirmando a criação do grupo. Neste momento são criadas as estruturas de dados necessárias para gerenciar o grupo, como o descritor do grupo e a fila de mensagens associadas ao grupo. Os outros servidores criam o grupo, porém não disparam os subprotocolos, que somente serão executados quando algum processo local passar a fazer parte do grupo.

Caso ocorra uma falha no Servidor local antes dele difundir a mensagem de confirmação, a criação do grupo será abortada, pois os outros servidores não receberão a confirmação, e em algum momento detectarão a falha do servidor, excluindo-o do grupo de servidores. Se a falha ocorrer após esta confirmação, o grupo será criado, pois a mensagem de confirmação utiliza difusão confiável, garantindo que todos os outros servidores que não falharam criaram o grupo. Falhas em outros servidores durante o processo são tratadas pelo próprio algoritmo de difusão confiável, que tentará excluir os servidores que falharam.

3.5.6 Destruição de um grupo

A destruição de um grupo é ativada pela chamada da função de destruição de grupo na biblioteca do FlexGroup. Se o grupo possuir um processo responsável, informado na sua criação, somente este processo poderá chamar a função de destruição do grupo, caso contrário, qualquer membro do grupo poderá solicitar a sua destruição. Se dois processos solicitarem concorrentemente a destruição do grupo em máquinas remotas, o grupo será destruído e os dois processos que solicitaram a destruição receberão como retorno sucesso na destruição do grupo.

A biblioteca informa o Servidor de Comunicação do pedido de destruição e o núcleo do servidor é o responsável por executar esta operação. Ele deve comunicar os outros servidores e os subprotocolos da destruição do grupo, devendo também encerrar

a execução dos subprotocolos que não sejam mais necessários e liberar as estruturas associadas ao grupo, como fila de mensagens, descritor e outras. O seguinte procedimento é executado pelo núcleo:

1) Difunde uma mensagem para os outros servidores, informando a destruição do grupo, através do protocolo de difusão confiável descrito na seção 3.4.1.
2) Envia uma mensagem para cada um dos subprotocolos utilizados pelo grupo, informando-os da destruição do grupo. O subprotocolo se detectar que nenhum grupo mais o utiliza, encerra a sua execução e comunica sua saída ao núcleo.
3) O núcleo desaloca todas as estruturas associadas ao grupo, como a fila de mensagens e o descritor do grupo.

Durante as fases 1 e 2, as aplicações de comunicação em grupo ainda podem receber as mensagens que estejam na fila de mensagens liberadas para entrega (já passaram por todos os subprotocolos). As outras operações, como o envio de mensagens e a inclusão no grupo, por exemplo, não podem mais ser executadas a partir do momento em que o núcleo receber a mensagem solicitando a destruição do grupo.

3.5.7 Entrada de um processo no grupo

Um processo, para participar de um grupo, chama a função da biblioteca JoinGroup, informando o nome do grupo e o seu nome, normalmente o mesmo nome utilizado pelo HetNOS para identificar os processos.

A biblioteca envia o pedido para o Servidor de Comunicação em Grupo, que determina se pode incluir o novo processo ao grupo. Em caso positivo, esta inclusão deve então ser comunicada para os outros servidores e subprotocolos. O mecanismo de entrada de um processo em um grupo funciona da seguinte maneira:

1) O Servidor de Comunicação em Grupo recebe o pedido do processo que deseja entrar no grupo, verifica a existência do grupo e se o processo não faz parte do grupo; se e a inclusão for possível, retorna uma confirmação à aplicação, senão retorna um erro.
2) O servidor difunde uma mensagem para os outros servidores contendo esta modificação no grupo por difusão não confiável.
3) Um servidor, ao receber uma mensagem de alteração da composição de um grupo, atualiza a sua visão do grupo e informa os subprotocolos locais (se existirem) da modificação.
4) O núcleo do servidor verifica se os subprotocolos necessários ao grupo já estão sendo executados na máquina, pois se não houver nenhum processo pertencente ao grupo na máquina, os subprotocolos não são executados. Se não existirem os subprotocolos, o núcleo dispara a sua execução.
5) Se os subprotocolos já existirem, o núcleo então informa os subprotocolos da entrada de um novo processo no grupo.

Este procedimento não assegura que os servidores tenham uma visão consistente da composição do grupo, pois as mensagens de alteração são enviadas por difusão não confiável. Deve ser implementado um subprotocolo que trate este tipo de atomicidade na alteração dos grupos e outras propriedades, como, por exemplo, sincronismo virtual entre a entrega de mensagens e a mudança na composição dos grupos, conforme visto no capítulo 2. Este mecanismo foi adotado para não sobrecarregar o núcleo com

funcionalidade de gerenciamento da composição dos grupos, que deverá feita por um subprotocolo se desejado pela aplicação, e para evitar aumentar o número de mensagens trocadas entre os servidores por difusão confiável, que exigem a execução do protocolo descrito na seção 3.5.1.

Os subprotocolos que não têm interesse na composição dos grupos simplesmente ignoram as mensagens de mudança da composição dos grupos. Os subprotocolos que tratam diretamente da composição do grupo, têm somente a garantia que cada servidor tem uma visão correta dos processos locais que pertencem ao grupo, e devem implementar um mecanismo para garantir a consistência desta visão globalmente.

3.5.8 Saída de um processo do grupo

Para sair de um grupo, um processo chama a função LeaveGroup, informando o nome do grupo e o nome pelo qual era identificado. A biblioteca envia uma mensagem ao Servidor de Comunicação em Grupo, pedindo a saída do processo e ela também bloqueia qualquer nova operação sobre este grupo para o processo.

O Servidor, ao tomar conhecimento do pedido de saída de um processo realiza a seguinte operação:

1) Difunde uma mensagem para os outros servidores, informando da modificação no grupo, através de difusão não confiável.
2) Remove o processo da sua visão do grupo informado.
3) Confirma para a aplicação a saída do processo do grupo.
4) Envia uma mensagem para os subprotocolos pertencentes ao grupo, comunicando a saída do processo.
5) Os servidores remotos, ao receberem a mensagem comunicando a saída de um processo de um grupo, removem o processo de sua visão do grupo e comunicam aos subprotocolos correspondentes na sua máquina.

Além da saída normal de um processo do grupo, através da função LeaveGroup, um processo também pode ser retirado do grupo devido a falhas. Se um servidor falha e é retirado do grupo de servidores, todos os processos que estavam naquela máquina são considerados como falhos e retirados dos grupos dos quais participam. Além disso, se o servidor não conseguir se comunicar mais com algum processo, ele assume que este processo falhou e o retira dos grupos de que participa, pois a comunicação ponto-a-ponto é confiável. Desta forma, mesmo se um processo for encerrado sem ter se desligado de algum grupo, o grupo continuará a funcionar sem ficar bloqueado esperando por algum processo que tenha falhado ou encerrado extemporaneamente.

Ao contrário da destruição de um grupo, os subprotocolos não encerram a sua execução se não existir mais nenhum processo local que os utilize. Como podem surgir outros processos que utilizem o grupo e o custo de encerrar e reinicializar um subprotocolo é relativamente grande, os subprotocolos continuam a executar até que todos os grupos de que participam sejam destruídos. Este processamento é muito pequeno, pois os subprotocolos ficarão bloqueados esperando mensagens para serem enviadas ou recebidas, que não existirão até que um novo processo local entre no grupo. O subprotocolo somente precisará responder as mensagens dos subprotocolos pares nas outras máquinas, caso o subprotocolo assim o exija (depende do protocolo a ser implementado).

3.5.9 Envio de mensagem

Para enviar uma mensagem para um grupo, um processo somente precisa chamar o método Send do objeto que representa este grupo informando a mensagem a ser difundida para o grupo. O envio da mensagem é assíncrono, ou seja, assim que o Servidor de Comunicação local receber a mensagem, o processo que a enviou é liberado para prosseguir a sua execução. Este processo não tem garantia de que a mensagem será efetivamente entregue, pois podem acontecer erros no próprio servidor. Porém ele tem garantia de que se a mensagem for entregue, ela obedecerá as condições impostas pelos subprotocolos que estão associados ao grupo.

O envio de uma mensagem por uma aplicação é esquematizado na figura 3.6. A figura representa os passos de 1 a 4 do algoritmo de envio de mensagem descrito a seguir, que é executado pelo servidor da máquina onde a aplicação que deseja enviar a mensagem está executando.

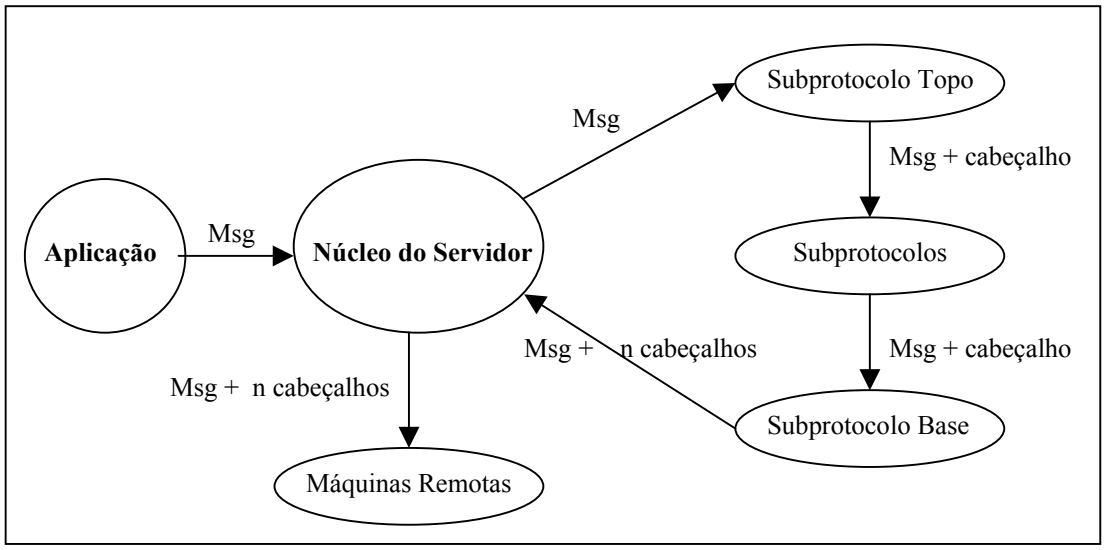

FIGURA 3.6 - Envio de uma mensagem por uma aplicação

O núcleo do servidor, ao receber um pedido de envio de mensagem, executa o seguinte procedimento:

1) Verifica se o grupo existe, responde à aplicação informando sucesso se o grupo existe e erro se não existir mais e libera a aplicação que enviou a mensagem.
2) Envia a mensagem para o subprotocolo **topo** da pilha associada ao grupo e se libera para tratar outras requisições.
3) Cada subprotocolo, ao receber uma mensagem a ser enviada, coloca as informações que deseje enviar para os subprotocolos em outras máquinas no cabeçalho da mensagem e envia a mensagem para o protocolo imediatamente inferior na pilha. O subprotocolo **base** da pilha envia a mensagem de volta ao núcleo do servidor, marcada como pronta para enviar.
4) O núcleo do servidor, ao receber a mensagem de volta, após ter passado por todos os subprotocolos da pilha, a difunde de maneira não confiável para os outros servidores.

Todos os servidores (inclusive o servidor origem da mensagem) executam o mesmo procedimento ao receber uma mensagem de aplicação. Este procedimento está representado na figura 3.7.

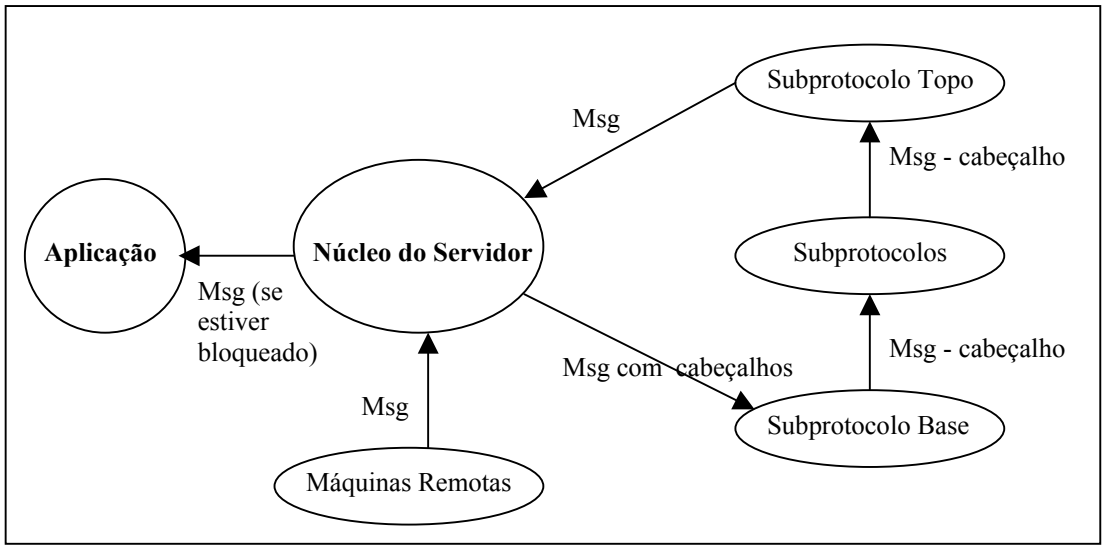

FIGURA 3.7 - Recepção de mensagem pelo Servidor de Comunicação

Cada servidor ao receber uma mensagem de aplicação difundida por qualquer servidor (inclusive ele próprio) executa os seguintes passos:

1) Um servidor, ao receber uma mensagem de aplicação, se possuír processos locais associados ao grupo de destino da mensagem, a envia para o protocolo **base** da pilha para que ele trate a recepção da mensagem.
2) Cada subprotocolo, ao receber uma mensagem, testa sua condição de entrega, que depende do subprotocolo que implementa. Quando esta condição for satisfeita, a mensagem é enviada para o subprotocolo imediatamente superior na pilha. O último subprotocolo da pilha, quando tiver sua condição de entrega satisfeita, envia a mensagem para o núcleo marcada como pronta para entregar. Se a condição de entrega não for satisfeita, a mensagem é colocada na fila de mensagens, ficando a espera do atendimento desta condição. A cada nova mensagem recebida, é feito novamente o teste para todas as mensagens que estejam na fila de mensagens com um indicador de não entregue ao próximo nível.
3) Quando o núcleo receber esta mensagem de volta, ele coloca-a na fila de mensagens associada a cada processo local que pertence ao grupo, com exceção do processo origem da mensagem se ele for local. Se o processo estiver bloqueado, esperando uma mensagem, envia esta mensagem para o processo, liberando-o.

Cada subprotocolo é responsável por acrescentar e retirar as informações necessárias ao seu funcionamento no cabeçalho da mensagem, caso contrário o protocolo seguinte não conseguirá reconhecer o seu cabeçalho e não poderá tratar a mensagem.

O processo servidor possui uma fila de mensagens para cada processo que tenha se conectado a ele independente do número de grupos no qual participe. Esta conexão é feita durante a inicialização da biblioteca do FlexGroup e mantida até o encerramento do processo.

3.5.10 Recepção de uma mensagem

Um processo recebe uma mensagem de um grupo, chamando o método Receive de um objeto grupo, informando somente a área onde o conteúdo da mensagem será armazenado. A recepção de mensagens segue o modelo síncrono, ou seja, o processo fica bloqueado até que uma mensagem esteja disponível.

Ao ser invocado o método Receive, a biblioteca envia uma mensagem ao servidor solicitando a recepção de uma mensagem do grupo a que está associada e fica bloqueada esperando a resposta do servidor contendo a mensagem ou um erro como, por exemplo, a não existência do grupo.

O Servidor de Comunicação, quando recebe um pedido de recepção de mensagem, verifica a fila de mensagens associada ao processo que fez a requisição. Se houver uma mensagem na fila destinada ao grupo informado, o servidor envia esta mensagem para a aplicação que pode prosseguir o seu processamento. Se não existir uma mensagem para ser entregue a aplicação, o servidor atualiza o indicador de bloqueado da aplicação para verdadeiro e prossegue o seu processamento, ficando a aplicação bloqueada.

Quando houver uma mensagem em condição de ser entregue, conforme o descrito na seção anterior, este indicador de bloqueado é atualizado para falso e a mensagem é enviada para a aplicação que pode prosseguir no seu processamento.

4 Implementação

Com o propósito de validar as idéias e a arquitetura propostas no capítulo anterior, bem como mensurar a complexidade envolvida na implementação do ambiente de comunicação em grupo e dos subprotocolos, foi desenvolvido um protótipo do sistema. Durante a implementação, a tomada de decisões relativas ao protótipo procurou priorizar a clareza na interface entre os diversos módulos e a simplicidade de utilização do ambiente para desenvolver aplicações em grupo e novos subprotocolos.

Como plataforma de hardware foi utilizado um conjunto de estações de trabalho Sun SPARC interligadas por uma rede Ethernet. Cada estação utiliza os sistemas operacionais SunOS ou Solaris como sistema operacional nativo e o sistema de rede heterogêneo HetNOS como sistema de comunicação e gerenciamento de processos.

A seguir serão descritos os mecanismos utilizados na implementação das diversas partes que compõem o sistema.

4.1 Biblioteca

A biblioteca do ambiente Flexlib é composta de um arquivo objeto (flexlib.o) que deve ser ligado à aplicação de comunicação em grupo na compilação do programa e um arquivo de cabeçalho (flexlib.h) que deve ser incluído na aplicação. O principal componente da biblioteca é uma classe *Group*, a qual possui todos os métodos necessários para utilizar as funcionalidades do ambiente.

A definição da classe Group é a seguinte:

```
class Group {
public:
        int GroupExists(char *GroupName);
        int CreateGroup(char *GroupName, char *Protocols, char *Owner);
        int DestroyGroup();
        int JoinGroup(char *GroupName, char *ProcessName);
        int LeaveGroup(char *ProcessName);
        int GetGroupInfo(char *GroupName, char *Info);
        int Send(char *msg);
        int Receive(char *msg);
private:
        char GroupName[20];                 /* Nome do Grupo  */
        char *Protocols;            /* Subprotocolos utilizados pelo grupo */

        char Owner[20];             /* Processo responsável pelo grupo */
        ....
        virtual Group();            /* Construtor do grupo */
        virtual ~Group();           /* Destruidor do grupo */
        SendToServer(int *TipoOperacao,
                char *msg);   /* Função que realiza a comunicação com o
        servidor */
        ...
}
```

A tabela a seguir apresenta os métodos disponíveis na classe *Group*:

TABELA 4.1 – Métodos da classe *Group*

Função	Descrição
GroupExists(char *GroupName)	Verifica a existência de um grupo identificado por *GroupName*.
CreateGroup(char *GroupName, char *Protocols, char *Owner)	Cria um novo grupo, identificado por *GroupName* e utilizando os protocolos informados no parâmetro *Protocols*, e possuindo como **responsável** o processo informado no parâmetro *Owner*. Este método associa o objeto *Group,* no qual o método foi invocado, ao novo grupo criado, se a criação for bem sucedida. Se o grupo já existir ou algum dos parâmetros for inválido, retorna um erro para a aplicação.
DestroyGroup()	Destrói o grupo associado ao objeto *Group* no qual o método foi invocado. Se o objeto não estiver associado a nenhum grupo retorna um erro.
JoinGroup(char *GroupName, char *ProcessName)	Insere um novo processo, identificado por *ProcessName*, no grupo informado no parâmetro *GroupName*. Este método associa o objeto *Group,* no qual o método foi invocado, ao grupo informado. Se o grupo não existir ou o processo não existir, retornar um erro.
LeaveGroup(char *ProcessName)	Retira o processo identificado por *ProcessName* do grupo associado ao objeto *Group* no qual o método foi chamado. Se o processo não fizer parte do grupo, ou o objeto *Group* não estiver associado a um grupo é retornado um erro.
GetGroupInfo(char *GroupName, char *Info)	Busca as informações relativas a um grupo (responsável e subprotocolos utilizados).
Send(char *msg)	Envia a mensagem *Msg* (em formato string ASCII), para o grupo associado ao objeto *Group* no qual o método foi chamado.
Receive(char *msg)	Recebe uma mensagem destinada ao grupo associado ao objeto *Group* e a coloca na área de armazanamento indicada por *Msg*

Todos os métodos da classe *Group* retornam 0 em caso de sucesso e em caso de erro retornam um código de erro. Os principais códigos de erro estão descritos na flexlib.h e devem ser tratados pela aplicação.

Além desta classe, o biblioteca ainda implementa duas funções, **flexgrp_init** e **flexgrp_terminate**, que devem ser chamadas na inicialização e no término do aplicativo, respectivamente.

A função **flexgrp_init** é responsável por inicializar a camada de comunicação, através da função h_init do HetNOS e por estabelecer a comunicação com o Servidor de Comunicação existente na máquina. Caso não consiga estabelecer uma destas duas conexões, a função retorna um erro e não permite o funcionamento de nenhum dos métodos descritos acima.

A função **flexgrp_terminate** comunica o Servidor de Comunicação da finalização do aplicativo e encerra a conexão com o HetNOS, através da função h_terminate.

A biblioteca de comunicação em grupo para poder funcionar corretamente, precisa incluir na compilação as bibliotecas do HetNOS. Visando facilitar a implementação das aplicações, está disponível um exemplo de arquivo de compilação (Makefile), contendo as diretivas de compilação necessárias para gerar uma aplicação para o FlexGroup que deve ser utilizado como base para as aplicações a serem desenvolvidas.

Além disso, aplicações HetNOS precisam ser rescritas para utilizarem as funções flexgrp_init e flexgrp_terminate ao invés das funções h_init e h_terminate, pois os HetNOS só permite uma conexão por processo. As outras funções do ambiente HetNOS podem ser utilizadas normalmente após o estabelecimento da conexão.

4.2 Núcleo do Servidor

O núcleo do servidor é o componente central da arquitetura proposta para o FlexGroup. Ele é o responsável pela interface como a biblioteca de programação, com os subprotocolos e com os outros servidores remotos e por gerenciar o funcionamento do ambiente como um todo. Os diversos núcleos distribuídos nas máquinas da rede compõem, eles próprios, um grupo que utiliza os protocolos descritos no capítulo anterior para assegurar o seu correto funcionamento.

O núcleo é implementado como um processo HetNOS comum e não como um novo servidor do sistema. Esta política, apesar de apresentar um maior overhead de comunicação, devido à menor integração do ambiente FlexGroup com a camada de comunicação, foi adotada por oferecer maior portabilidade e simplicidade no projeto do servidor e por não exigir a alteração do código do próprio HetNOS. Desta forma, o Servidor de Comunicação em Grupo fica totalmente desvinculado da camada de comunicação, podendo inclusive ser utilizada uma outra camada de comunicação como PVM, MPI ou diretamente sobre os sockets do UNIX.

O núcleo do servidor é composto basicamente de uma classe chamada *Dispatcher*. Esta classe executa um laço "infinito", de espera de mensagens (requisições), até que o servidor seja encerrado, via pedido do usuário (Crtl + c) ou pela ocorrência de uma falha. Estas mensagens podem vir de uma aplicação local, através da biblioteca Flexlib, de um subprotocolo local ou de um outro servidor remoto. Quando recebe uma mensagem, o servidor executa os procedimentos correspondentes e volta a esperar por novas requisições. Na implementação atual, devido a limitações impostas pelo ambiente HetNOS na utilização de threads, cada requisição é tratada seqüencialmente, devido ao núcleo ser estruturado como um único processo. No futuro, pretende-se transformar cada pedido em uma thread independente, o que permitiria a execução simultânea de duas ou mais requisições, aumentando a eficiência e o desempenho do servidor.

Para disparar o Servidor de Comunicação em Grupo é necessário executar o seguinte comando no shell do HetNOS:

- h *flexsrv_maquina* flexsrv –*servidor_remoto*, onde
  - h é o indicador de que se deseja disparar uma aplicação HetNOS

- *flexsrv_máquina* é o nome simbólico atribuído ao servidor que está sendo disparado, onde máquina deve ser o nome simbólico da máquina onde se está disparando o servidor.
- flexsrv é o nome do executável HetNOS que implementa o Servidor de Comunicação em Grupo do FlexGroup.
- *servidor_remoto* é o nome de uma máquina onde o ambiente já esteja executando, ou vazio se este for o primeiro servidor.

A classe Dispatcher é a responsável pela execução do servidor. Na sua inicialização, ela inicializa a conexão com o HetNOS e comunica-se com o servidor informado na linha de comando (se não for o primeiro servidor), para tentar entrar no grupo de acordo com o protocolo descrito no capítulo anterior. O núcleo do servidor precisa armazenar informações sobre os outros servidores, sobre as aplicações, sobre os grupos e sobre os subprotocolos. A figura 4.1 mostra quais as classes e informações utilizadas por cada um dos componentes do ambiente FlexGroup. As classes utilizadas pelo núcleo e pelos subprotocolos, bem como os seus atributos, são descritos posteriormente na seqüência do capítulo.

Biblioteca
Classe *Group*
Nome do grupo
Protocolos
Responsável
Conexão com o servidor
Nome HetNOS
Nome servidor
Nome máquina

Núcleo do Servidor
Classe *RemoteServer*
Nome servidor remoto
Última mensagem recebida do servidor
Hora de recepção da última mensagem
Tempo de espera (timeout) do servidor
Indicador de suspeita.
Classe *KernelGroup*
Nome do grupo
Protocolos
Protocolo base da pilha
Protocolo topo da pilha
Lista de processos pertencentes ao grupo
Classe *Client*
Nome do processo
Listas dos grupos onde o processo participa
Buffer de mensagens a serem entregues
Indicador de processo bloqueado na recepção
Grupo do qual o processo espera mensagens

Subprotocolos
Classe *Protocol*
Nome do protocolo
Nome da máquina
Grupos que utilizam o protocolo
Buffer de mensagens
Classe *ProtocolGroup*
Nome do grupo
Protocolos
Responsável
Processos do grupo
Protocolo superior na pilha do grupo
Protocolo inferior na pilha do grupo
Classe *Message*
Origem da mensagem
Identificador
Corpo da mensagem
Indicador de bloqueado esperando condição de entrega

FIGURA 4.1 - Informações utilizados pelos componentes do FlexGroup

Após entrar no grupo de servidores, o novo servidor passa a armazenar informações sobre os outros servidores de maneira a possibilitar a implementação da difusão confiável e dos protocolos de entrada e saída do grupo de servidores descritos no capítulo anterior. Para cada servidor remoto, são guardadas as seguintes informações através de um objeto da classe *RemoteServer*:

- ServerName : Nome simbólico do servidor remoto, no formato *flexsrv_*maquina, utilizado para envio e recepção de mensagens.
- LastMessage: Identificador seqüencial da última mensagem (enviada por difusão confiável) recebida do servidor. Esta informação é enviada na confirmação das mensagens envidas por difusão confiável.
- TimeLastMessage : Horário local do recebimento da última mensagem (qualquer) do servidor remoto. Esta informação é utilizada para responder aos pedidos de suspeita sobre o servidor.
- TimeoutMessage : Tempo da última resposta envida pelo servidor remoto a uma difusão confiável. Este tempo é utilizado para determinar o timeout que será utilizado na espera de confirmação dos servidores.
- Suspected : Booleano que indica se há uma suspeita de falha sobre o servidor remoto.

Além das informações sobre os servidores remotos o núcleo também precisa de informações sobre os grupos existentes no sistema. Para cada grupo existente no sistema, estas informações são encapsuladas em um objeto *KernelGroup* que contém os seguintes atributos:

- GroupName: Nome simbólico do grupo. O nome do grupo deve ser único no sistema, por isso requisições de criação de novos grupos devem comparar o nome informado com o nome dos grupos já existentes.
- Protocols : Nome simbólico dos protocolos utilizados pelo grupo. O servidor precisa conhecer os protocolos utilizados pelo grupo para inicializá-los
- FirstProtocol: Nome HetNOS do primeiro protocolo da pilha (base), para onde são envidas as mensagens recebidas tendo o grupo como destinatário.
- LastProtocol : Nome HetNOS do último protocolo da pilha (topo), para onde são enviadas as mensagens enviadas localmente para o grupo e que devem passar pelos subprotocolos antes de serem enviadas para as outras máquinas.
- ProcessList : Lista que contém informações sobre os processos que compõem o grupo. Esta lista somente tem garantia de estar atualizada para os processos locais, pois as mensagens de alteração dos grupos não são enviadas de maneira confiável. As informações sobre os processos são as seguintes:
    - Name : Nome simbólico do processo.
    - Local : Booleano que indica se o processo é local ou remoto
    - LastMsgSent :Identificador da última mensagem enviada pelo processo para o grupo. Somente é válido para processos locais.

Estas informações podem ser consultadas pelos subprotocolos, através do envio de um pedido de informação sobre um grupo.

Outra função do núcleo é gerenciar a comunicação com as aplicações clientes do serviço de comunicação em grupo. Cada aplicação deve conectar-se ao servidor através

do método *flexgrp_init*, descrito na seção anterior. Ao receber um pedido de conexão, o núcleo cria um objeto da classe *Client* para armazenar as informações relativas à aplicação. Esta classe cliente possui os seguintes atributos:

- ProcessName : Utilizado para responder aos pedidos da aplicação.
- GroupName[MAXGROUPS] : Array contendo os diversos grupos do qual a aplicação faz parte.
- MsgBuffer[MAXMSGS] : Buffer onde ficam armazenadas as mensagens que podem ser entregues à aplicação. Cada mensagem é composta dos seguintes campos:
    - Group : Grupo ao qual a mensagem é destinada. Este campo é verificado quando a aplicação solicita a recepção de uma mensagem de um grupo.
    - Sender : Processo que enviou a mensagem.
    - Msg: Corpo da mensagem.
- Blocked : Campo booleano que identifica se a aplicação está bloqueada esperando uma mensagem de um grupo.
- GroupBlocked : Identifica o grupo do qual a aplicação espera uma mensagem quando estiver bloqueado. Ao ser recebida e liberada para entrega pelos subprotocolos uma mensagem destinada ao grupo, ela é enviada à aplicação e estes campos é reinicializado.

O núcleo também precisa conhecer os subprotocolos que executam na máquina, para poder enviar-lhes mensagens como, por exemplo, mensagens de alteração dos grupos a que pertencem ou de encerramento do servidor. Como as informações sobre os protocolos já existem nos objetos Group existentes, o núcleo armazena somente o nome HetNOS dos subprotocolos para poder enviar mensagens que não passam pela pilha de subprotocolos.

Conforme o descrito anteriormente, o servidor permanece executando um laço infinito, até ser encerrado. A cada nova mensagem que recebe, o servidor executa os procedimentos adequados para tratar a mensagem recebida e volta a esperar por mensagens. O fluxo de mensagens recebido e enviado pelo servidor pode ser visualizado na figura 4.2.

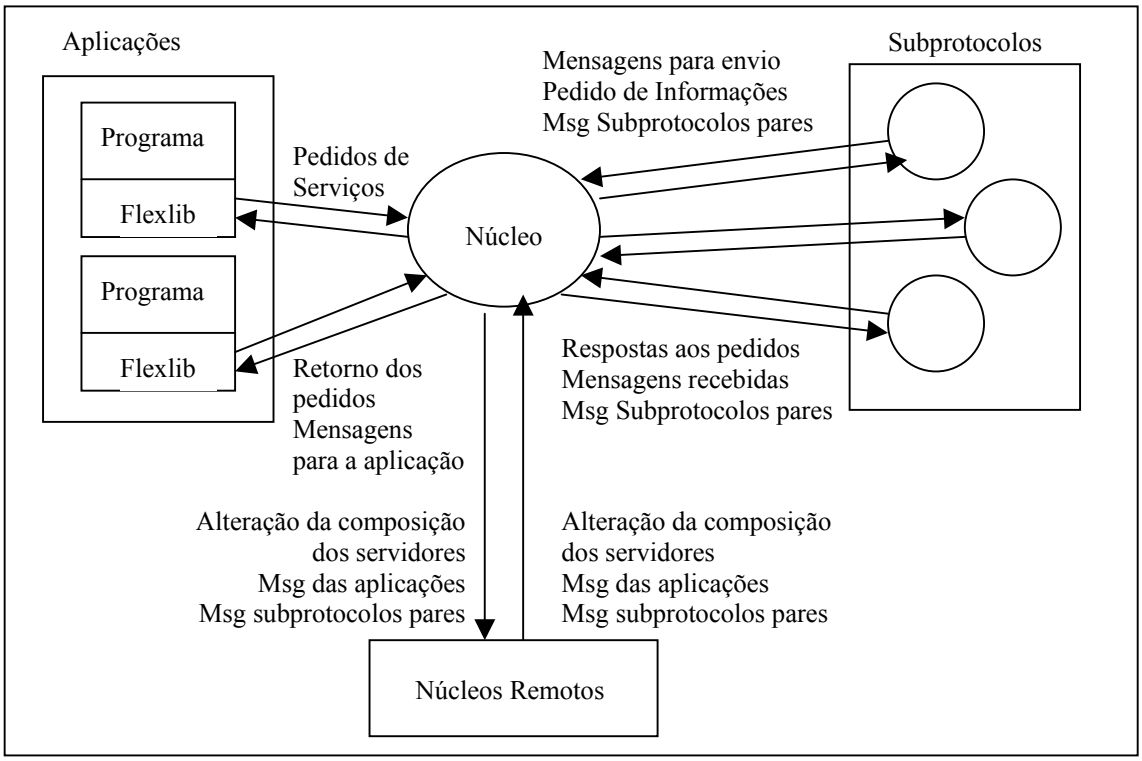

FIGURA 4.2 - Fluxo de Mensagens do Núcleo do Servidor FlexGroup

As mensagens recebidas pelo servidor podem ser divididas em três tipos:

- Mensagens dos Servidores
- Mensagens dos Subprotocolos
- Mensagens das Aplicações

4.2.1 Mensagens dos Servidores

As mensagens dos servidores são as requisições enviadas pelos outros servidores remotos. Estas requisições podem tratar tanto da composição do grupo de servidores (enviadas por difusão confiável), como conter mensagens difundidas por uma aplicação ou ainda conter mensagens enviadas por um subprotocolo remoto para os outros subprotocolos. As principais mensagens que o núcleo recebe de um servidor remoto são as seguintes:

- **ServerCoordenate** - Recebida de um novo servidor que deseja entrar no grupo, indica que o núcleo deste servidor passa a ser o coordenador desta entrada. Ele responde enviando para o novo servidor uma mensagem contendo a lista dos servidores e outra contendo a lista dos grupos. Enquanto não receber uma mensagem de **ServerJoinOK** do novo servidor repassa todas as mensagens de manipulação dos grupos e dos servidores para ele. Se a mensagem de **ServerJoinOK** não for recebida dentro do intervalo de espera do servidor, ele considera que o novo servidor falhou e difunde uma mensagem para os outros servidores, abortando a entrada.

- **ServerJoin** - Recebida por difusão confiável de um novo servidor que deseja entrar no grupo, porém o núcleo local não é o coordenador desta operação. Acrescenta o nome do servidor na lista de servidores e envia uma mensagem para ele confirmando a sua entrada.
- **ServerLeave** - Recebida por difusão confiável de um servidor que esteja encerrando a sua execução. O núcleo então retira o servidor origem da mensagem da sua visão do grupo de servidores.
- **SuspectServer** - Recebida de um servidor que não conseguiu se comunicar com outro servidor e difundiu uma suspeita sobre o servidor omisso. Procura na classe *RemoteServer* correspondente ao servidor suspeito, o intervalo decorrido desde a recepção da última mensagem deste servidor suspeito. Se este intervalo for maior que o tempo de espera de comunicação responde com uma mensagem **ReplySuspect** com conteúdo True, senão responde com **ReplySuspect** False.
- **ReplySuspect** - Mensagem enviada pelos outros servidores para aceitar ou rejeitar um pedido de suspeita enviado pelo servidor local. Se todos os servidores responderem com True o servidor suspeito é retirado da lista e uma mensagem **ServerLeave** é difundida para os outros servidores.
- **CanCreateGroup** - Informa o núcleo da criação de um novo grupo. Se o grupo já existir localmente o núcleo envia uma mensagem **ReplyCreateGroup**, indicando erro na criação, caso contrário envia uma mensagem **ReplyCreateGroup** confirmando a criação.
- **ReplyCreateGroup** - Mensagem enviada pelo servidores remotos para confirmar ou abortar a criação de um novo grupo iniciada pelo servidor local. Se todos os servidores responderam OK, o núcleo difunde uma mensagem de CreateGroup confirmando a criação do grupo. Caso contrário, o grupo não pode ser criado e o núcleo retorna um erro para a aplicação.
- **CreateGroup** - Mensagem que confirma a criação de um novo grupo. O núcleo aloca então as estruturas necessárias para armazenar as informações do grupo.
- **UpdateGroup** - Mensagem que informa o servidor local de uma atualização gerada em outros servidor. São chamados os mesmos métodos da classe grupo usados para os pedidos locais vindos de uma aplicação, porém sem a propagação para os outros servidores.
- **PeerProtocol -** Mensagem enviada por um subprotocolo em outra máquina. A mensagem é enviada para o subprotocolo local correspondente.
- **AppMessage** - Este é o tipo normal de mensagem a circular pelo ambiente, ou seja, mensagens difundidas por um processo de aplicação para um grupo. O núcleo repassa a mensagem para o subprotocolo base da pilha correspondente ao grupo destino da mensagem.

4.2.2 Mensagens dos Subprotocolos

Mensagens dos subprotocolos são as requisições recebidas pelo núcleo de um subprotocolo local. Estas requisições podem ser a liberação de uma mensagem para entrega, a liberação de uma mensagem para envio, ou ainda uma consulta às informações do núcleo ou o envio de uma mensagem para os subprotocolos correspondentes em outra máquina. As principais mensagens e o seu tratamento são:

- **DeliverMessage** - Indica que uma mensagem destinada a um grupo atendeu os requisitos de entrega de todos os subprotocolos do grupo e pode ser entregue para os processos locais pertencentes ao grupo. Esta mensagem é enviada pelo subprotocolo que ocupa o topo da pilha referente ao grupo. O núcleo ao receber esta mensagem, extrai o corpo da mensagem e para cada processo local verifica se ele está bloqueado esperando uma mensagem para este grupo. Se estiver bloqueado, o núcleo envia a mensagem para ele e o desbloqueia; caso contrário, o núcleo coloca a mensagem na fila de mensagens do processo.
- **SendMesssage** - Mensagem recebida do subprotocolo base da pilha de um grupo, indica que uma mensagem enviada por um processo local já passou por todos os subprotocolos e pode ser difundida para as outras máquinas, tendo as informações necessárias sido colocadas no seu cabeçalho.
- **SendToPeer** - Subprotocolo deseja enviar uma mensagem para um (ou todos) os subprotocolos correspondentes em outra máquina. O núcleo não realiza nenhuma ação local ao receber esta mensagem, simplesmente envia uma mensagem de **PeerProtocol** para o servidor ou servidores indicados, contendo a mensagem e o subprotocolo origem.
- **GetGroupInfo** - Mensagem enviada por um subprotocolo que deseja obter as informações sobre a visão do grupo disponíveis no núcleo. O núcleo responde enviando a sua visão do grupo indicado na mensagem.
- **GetServerInfo** - Mensagem enviada por um subprotocolo que deseja obter as informações sobre a visão do grupo de servidores. O núcleo responde enviando a sua visão do grupo de servidores. Este tipo de informação é útil para subprotocolos saberem em quais outras máquinas existem subprotocolos pares executando.

4.2.3 Mensagens das Aplicações

As funções da biblioteca descritas na seção 4.1 são responsáveis por gerar as mensagens de aplicação que o núcleo recebe. Estas mensagens contém requisições das operações permitidas pelo serviço de comunicação em grupo e são tratadas da seguinte maneira.

- **CreateGroup** – Pedido de criação de um novo grupo. O núcleo difunde então, de maneira confiável, uma mensagem de **CanCreateGroup** para os outros servidores e espera pelas respostas. Somente após receber as respostas ele responde para a aplicação, que permanece bloqueada durante este procedimento. O núcleo porém não fica bloqueado durante esta espera pois ele pode atender outras requisições.
- **DestroyGroup** – Destrói um grupo. O núcleo responde imediatamente para a aplicação, com um código de sucesso se o grupo existir ou erro se ele não existir mais. O núcleo então difunde uma mensagem **UpdateGroup** contendo o pedido de encerramento do grupo. O núcleo também comunica os subprotocolos locais da destruição do grupo através do envio de uma mensagem de **GroupDestroy** a eles. Se nenhum outro grupo utilizar mais um determinado subprotocolo, o núcleo envia também uma mensagem de **ProtocolDestroy** para este subprotocolo, solicitando o seu encerramento.
- **JoinGroup** – Mensagem que solicita a inserção de um novo processo em um grupo. O núcleo retorna imediatamente a resposta para a aplicação e, caso o

pedido possa ser atendido, envia uma mensagem de **UpdateGroup** para os outros servidores e para os subprotocolos locais.

- **LeaveGroup** – Pedido de saída de um processo de um grupo. O núcleo verifica se o grupo existe e se possui o processo informado. Caso isto ocorra, ele retorna uma resposta positiva para a aplicação, caso contrário, ele envia uma resposta negativa. Se a operação puder ser executada, o núcleo difunde uma mensagem de UpdateGroup para os outros servidores e para os subprotocolos locais.
- **Send** – Requisição de envio de mensagem. O núcleo responde imediatamente para a aplicação, liberando-a, pois o envio de mensagens no FlexGroup segue o modelo assíncrono. A aplicação não tem garantia da entrega da mensagem, pois o núcleo pode falhar antes de difundi-la, porém tem a garantia de que a mensagem obedecerá as propriedades implementadas pelos subprotocolos pertencentes ao grupo. O núcleo envia uma mensagem do tipo **Send** para o subprotocolo topo da pilha e espera que todos os subprotocolos a tratem para poder enviar aos outros servidores.
- **Receive** – Requisição de recepção de mensagem de um grupo. A recepção de mensagens no FlexGroup utiliza o modelo síncrono. O núcleo verifica se existe uma mensagem para o grupo disponível na fila de mensagens; caso exista uma mensagem disponível o núcleo envia a mensagem para a aplicação que pode prosseguir seu processamento. Caso não exista uma mensagem disponível, o núcleo atualiza o estado da aplicação para bloqueado. A aplicação será desbloqueada quando ocorrer a liberação de uma mensagem para o grupo, através do recebimento pelo núcleo de uma mensagem **DeliverMessage** dos subprotocolos, já explicada na seção 4.2.2.

4.3 Subprotocolos

Os subprotocolos são os responsáveis pela implementação dos protocolos de comunicação em grupo. Uma das principais motivações do ambiente é permitir que possa se construir novos protocolos a partir dos já existentes no sistema. Os subprotocolos apresentam uma interface simples com o restante do sistema. Eles somente precisam se comunicar com o núcleo e com os subprotocolos adjacentes na pilha dos grupos em que fazem parte, através de um interface bem definida que é descrita a seguir.

Os subprotocolos são implementados como processos HetNOS, se comunicando com o núcleo e com os outros subprotocolos através das funções *h_send* e *h_receive*, já descritas anteriormente. O subprotocolo é composto basicamente de um objeto derivado da classe *Protocol*. Esta classe não implementa nenhum subprotocolo especificamente, mas sim contém a funcionalidade básica de um subprotocolo e as funções e mensagens que devem ser tratadas ao desenvolver um protocolo. Muitas destas funções são métodos virtuais, que devem ser reescritos quando da criação de um subprotocolo real.

O objetivo desta classe é permitir que o desenvolvedor de um novo subprotocolo somente tenha que codificar as questões relativas às características que o seu protocolo deseja abordar, não precisando tratar outras questões.

A classe *Protocol* possui informações sobre os grupos de que faz parte e sobre as mensagens que estão esperando por alguma condição de entrega. Além disso, novos subprotocolos podem precisar armazenar outras informações, que devem ser determinadas quando da criação de uma classe derivada desta. Os principais atributos da classe são mostrados no quadro 4.1.

Estas classes podem e devem ser derivadas para atender as necessidades específicas dos subprotocolos. As mensagens, por exemplo, somente contêm o processo origem e o identificador seqüencial gerado pelo núcleo para este processo. Isto dificilmente será suficiente para um protocolo que deseje implementar uma ordenação total das mensagens. Estas outras informações devem ser colocadas pelos protocolos quando do envio das mensagens e retiradas quando uma mensagem é recebida, de maneira a não interferir no funcionamento dos outros protocolos.

Além destes atributos, a classe Protocol possui também uma interface para encapsular o envio de mensagens para os protocolos inferiores e superiores e para os subprotocolos correspondentes em outras máquinas através dos métodos:

- SendtoNext(char *Group, msg) - Envia para o protocolo superior na pilha uma mensagem que atendeu a condição de entrega. A função identifica o subprotocolo seguinte na pilha do grupo informado e envia uma mensagem do tipo **Deliver** para o subprotocolo contendo a mensagem a ser entregue.
- SendtoPrevious(char *Group, msg) - Envia para o nível imediatamente inferior na pilha uma mensagem que está sendo enviada e à qual ele já agregou as informações necessárias. A função busca o subprotocolo anterior na pilha do grupo informado e envia uma mensagem do tipo **Send** para ele contendo a mensagem a ser enviada.
- SendtoPeer(char *msg, char *machines) – Envia uma mensagem para os subprotocolos correspondentes nas outras máquinas. Se o parâmetro *machines* for informado, a função envia somente para as máquinas especificadas e se ele for nulo a função envia para todas as outras máquinas. Esta mensagem é enviada diretamente ao núcleo, através de uma mensagem de **SendToPeer**. O núcleo envia então para os outros servidores uma mensagem do tipo **PeerProtocol**.

```
class Protocol{
        char *ProtocolName;                         /* Nome do subprotocolo */
        char *MachineName;      /* Nome da máquina onde o subprotocolo  está sendo executado */
        ProtocolGroup Groups[MAXGROUPS];   /* Lista de objetos da classe Group na qual
                                                    o subprotocolo faz parte */
        Message Messages[MAXMESSAGES]; /* Lista de mensagens que estão esperando
                                                    para serem entregues   */
        ....}
        class ProtocolGroup  {              /* Representa um grupo que utiliza os subprotocolos */
                char *GroupName;            /* Nome do grupo */
                char *Protocolos;           /* Lista de subprotocolos utilizados pelo grupo */
                char *Owner;                /* Responsável pelo grupo */
                struct GroupProcess {       /* Processos que compõem o grupo (sem garantia de estar
                                               atualizado) */
                   char *ProcessName;       /* Nome do processos */
                   char *LastMsgReceived; /* Última mensagem recebida do processo */
                    ... } Process[MAXPROCESS];
```

```
        char *NextProtocol;        /* Nome do processos que implementçça o próximo
                                   subprotocolo na pilha de subprotocolos do grupo */
        char *PreviousProtocol;    /* Nome do processo que implementa o subprotocolo
                                   anterior na pilha de subprotocolos do grupo */
        ...}
class Message {                    /* Fila de mensagens de aplicação recebidas pelo
                                      subprotocolo */
        char *Sender;              /* Processo origem da mensagem*/
        int  Id;                   /* Identificador da mensagem */
        char *Msg;                 /* Corpo da mensagem (contendo os cabeçalhos dos
                                     subprotocolos acima na pilha)  */
        int  Blocked;              /* Flag que identifica se a mensagem está esperando
                                      alguma condição para poder ser entregue */
        ...}
```

QUADRO 4.1 – Descrição das classes utilizadas na implementação dos subprotocolos

O funcionamento esperado de um protocolo é bastante similar ao do núcleo, na medida em que ele deve ficar esperando mensagens, chamar o procedimento adequado e voltar a esperar uma mensagem. Os subprotocolos porém recebem somente mensagens do núcleo local e dos subprotocolos adjacentes nas pilhas em que fazem parte. O fluxo de mensagens dos subprotocolos é esquematizado na figura 4.3.

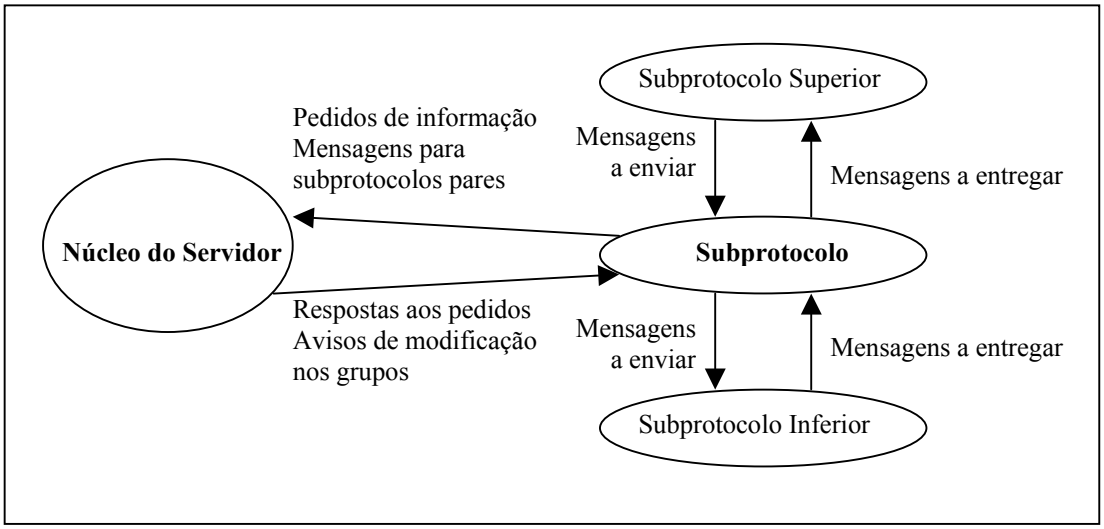

FIGURA 4.3 - Fluxo de mensagens de um subprotocolo

As mensagens recebidas diretamente do núcleo são as mensagens que dizem respeito ao gerenciamento dos grupos ou dos próprios subprotocolos. As principais mensagens recebidas diretamente do núcleo e o tratamento padrão dado pela classe *Protocol* a elas são os seguintes.

- **ProtocolDestroy** – Comunica ao protocolo que ele não é mais necessário e que deve encerrar a sua execução, pois nenhum grupo mais o utiliza. Permite que o subprotocolo execute alguma rotina de finalização antes de encerrar o processo. O tratamento da classe Protocol é liberar as estruturas alocadas para os atributos da classe (descritos acima), através do método FreeProtocol e atribuir o valor verdadeiro para variável que indica o fim do laço principal e consequentemente do processo.
- **GroupCreate** – Comunica o subprotocolo da criação de um novo grupo que o utiliza. O subprotocolo deve então criar um novo objeto da classe *ProtocolGroup* para representar este novo grupo. O tratamento padrão é criar um novo objeto *ProtocolGroup* e inicializar os atributos iniciais (nome, protocolo, owner) através do método CreateGroup da classe *Protocol*. Este método é virtual e deve ser rescrito quando for criado um novo subprotocolo.
- **GroupDestroy** – Comunica o subprotocolo da destruição de um grupo que o utilizava. O subprotocolo deve executar uma rotina de finalização para aquele grupo. A classe Protocol simplesmente desaloca o objeto ProtocolGroup correspondente ao grupo sendo destruído. Este método é virtual e deve ser rescrito quando for criado um novo subprotocolo.
- **GroupUpdate** – Informa o subprotocolo que houve alguma alteração na visão do grupo e este deve atualizar a sua visão. A classe Protocol não realiza nenhum tipo de tratamento para esta mensagem. Protocolos de gerência de grupo podem utilizar estas mensagens para obter informações sobre os grupos e garantir propriedades como atomicidade de visão do grupo e sincronismo das mensagens difundidas e das mudanças de visão.
- **GroupInfo** – Mensagem enviada pelo núcleo local contendo as informações disponíveis no núcleo em resposta a um pedido de **GetGroupInfo** enviada pelo subprotocolo.
- **PeerMessages** – Mensagem enviada pelo subprotocolo correspondente em uma máquina remota. A classe Protocol não implementa nenhum tipo de tratamento para estas mensagens. Estas mensagens atendem a necessidades específicas do subprotocolo que se deseje implementar.

As mensagens de aplicação são recebidas dos subprotocolos adjacentes (ou do próprio núcleo no caso da base e do topo da pilha de subprotocolos). A classe Protocol trata as seguintes mensagens recebidas através da pilha:

- **Send** – Mensagem recebida do protocolo imediatamente superior na pilha e que contém uma mensagem de aplicação que está sendo enviada. O subprotocolo deve acrescentar ao cabeçalho da mensagem as informações necessárias ao seu funcionamento que deseje enviar para as outras máquinas e repassar a mensagem para o subprotocolo imediatamente inferior, através do método SendtoPrevious. O tratamento padrão da classe Protocol é simplesmente chamar a função SendtoPrevious.
- **Deliver** - Mensagem recebida do protocolo imediatamente inferior na pilha e que contem uma mensagem que esta sendo recebida. O subprotocolo deve extrair as informações correspondentes os seu protocolo do cabeçalho da mensagem, verificar se a mensagem atendeu a sua condição de entrega e, em caso positivo, repassá-la para o nível imediatamente superior através da função SendtoNext. Se a mensagem não atender a condição de entrega, ela deve ser armazenada na fila de mensagens, com o indicador de bloqueada,

até que a condição necessária seja atendida, após o recebimento das mensagens que estejam faltando para a condição de entrega ser atendida. Quando a condição for atendida, a mensagem será então enviada para o próximo subprotocolo. A classe Protocol simplesmente chama a função SendtoPrevious, repassando imediatamente a mensagem para o nível superior.

Para validar o ambiente proposto foram construídos algum subprotocolos básicos, que servem como base para a derivação de novos subprotocolos e como exemplo de como criar um subprotocolo. Inicialmente foram desenvolvidos subprotocolos de difusão confiável, difusão causal e difusão atômica.

4.3.1 Subprotocolo de Difusão Confiável

A difusão confiável das mensagens para todos os participantes é um requisito exigido pela maioria das aplicações de comunicação em grupo. Além disso, estes protocolos são utilizados, na maioria das vezes, como base para construção de outros protocolos mais restritivos quanto à ordenação das mensagens ou quanto à sincronização entre os participantes. Existem muitos algoritmos de difusão confiável disponíveis na literatura, que apresentam técnicas diversas para assegurar a confiabilidade da difusão como, por exemplo, difusão em árvore [SGS84], difusão para os nodos vizinhos [HAD93], utilização de camada de rede com suporte a broadcast [AMI92]. Entre eles, foi escolhido para a implementação o algoritmo utilizado pelo protocolo Trans, descrito em [JAL94], devido à sua simplicidade e por não exigir nenhum software ou hardware específico para sua implementação.

Este protocolo utiliza o mecanismo de enviar nas próprias mensagens difundidas pelos processos as confirmações negativas e positivas de recebimento das mensagens anteriores, simplificando a detecção de mensagens perdidas e minimizando a necessidade de confirmações explícitas. Foram feitas algumas adaptações para adequar o protocolo ao FlexGroup, como, por exemplo, ao invés das confirmações serem enviadas por cada processo do grupo, elas são enviadas pelo subprotocolo, representando os processos locais. Esta simplificação pode ser feita, porque, se o subprotocolo local receber uma mensagem, esta mensagem será entregue aos processos locais, dado que eles prossigam o seu processamento e requeiram a entrega das mensagens.

O protocolo assume que, quando um nodo difunde uma mensagem, através de alguma primitiva de difusão não-confiável (como a oferecida pelo FlexGroup), alguns nodos a recebem e outros não. A idéia básica do protocolo é enviar confirmações positivas e negativas através das próprias mensagens difundidas. Cada mensagem carrega junto, no cabeçalho acrescentado pelo subprotocolo, a identificação do nodo origem e um número de seqüência único para a mensagem naquele nodo, que formam o identificador da mensagem, e as confirmações positivas e negativas do subprotocolo difusor. As confirmações positivas compõem uma lista contendo os identificadores das últimas mensagens recebidas. As confirmações negativas compõem uma lista com os identificadores das mensagens que o servidor detectou que deixou de receber e para as quais ele solicita retransmissão. Quando a retransmissão é pedida por um nodo, qualquer outro nodo que a recebeu pode transmiti-la novamente para o nodo que a solicitou, desde que envie as mesmas informações da mensagem original. Isto pode fazer com que um nodo receba mais de uma vez a mesma mensagem, porém as mensagem duplicadas podem ser descartadas através do seu número de seqüência.

Desta forma, cada subprotocolo possui uma ack-list (lista de confirmações positivas), uma nack-list (lista de confirmações negativas), uma received-list (lista de mensagens recebidas) e uma retransmission-list (lista de mensagens para retransmitir). A ack-list contém os identificadores das mensagens (nodo e número seqüencial) para as quais o subprotocolo deve enviar uma confirmação positiva. A nack-list contém os identificadores das mensagens para as quais o subprotocolo deve enviar uma confirmação negativa. A received-list contém as mensagens enviadas ou recebidas pelo subprotocolo e que podem necessitar retransmissão. A retransmission-list contém os identificadores das mensagens cuja retransmissão foi solicitada por algum outro subprotocolo.

O algoritmo original não leva em conta a existência de diversos grupos, porém no FlexGroup, o mesmo subprotocolo pode estar sendo utilizado por diversos grupos ao mesmo tempo. Porém, como para os subprotocolos não são levados em conta os processos do grupo e sim os subprotocolos de cada nodo, as estruturas descritas acima não precisam ser replicadas para cada grupo, podendo ser únicas para cada subprotocolo em cada nodo. Desta forma, a informação do grupo ao qual pertence a mensagem não é utilizada por este subprotocolo durante a execução do seu algoritmo. A informação do grupo somente é utilizada para determinar o caminho na pilha que a mensagem deve seguir. Além disso, para otimizar o desempenho, as mensagens retransmitidas são enviadas diretamente ao subprotocolo que requisitou a retransmissão, através do método SendToPeer.

Quando um subprotocolo difunde uma mensagem **m,** ele deve executar os seguintes passos:

- Colocar a identificação do nodo, o número seqüencial da mensagem, a ack-list e a nack-list no cabeçalho da mensagem
- Limpar a ack-list
- Difundir *m* (enviando para o próximo subprotocolo na pilha, correspondente ao grupo origem da mensagem).

Além de difundir suas próprias mensagens, o subprotocolo em cada nodo também envia as mensagens na retransmission-list através do mecanismo de SendToPeer do FlexGroup. Se o subprotocolo não receber uma confirmação positiva de uma mensagem de nenhum outro subprotocolo par, dentro de um determinado intervalo de tempo, ele a difunde novamente.

Ao receber uma mensagem, o subprotocolo executa as ações descritas no Quadro 4.2.

```
acrescenta m.id a ack-list          /* Para confirmar o recebimento da mensagem */
acrescenta m a received-list        /*  Coloca m na lista de mensagens recebidas  */
se m.id ∈ nack-list então remove m.id da nack-list   /* Se m estava na lista de
                                                        mensagens perdidas, retira-na desta lista */
se m.id ∈ retransmission-list então remove m.id da retransmission-list
                                    /* Se m estava na lista de retransmissão, retira-na
                                    desta lista, porque o nodo origem já a transmitiu  */
para todo id ∈ m.acks faça          /* Processa a lista de confirmações  */
    se id ∈ ack-list então remove id de ack-list        /*  Mensagem já confirmada,
                                                        não precisa confirmar novamente */
    se a mensagem com este id ∉ received-list então acrescenta id a nack-list
                                    /* Mensagem sendo confirmada não foi recebida,
                                    coloca o id na lista de confirmações negativas */
para todo id ∈ m.nacks faça         /* Processa a lista de mensgens perdidas pelo processo
                                    origem *//
    se a mensagem com este id ∈ received-list  /* Se já recebeu a mensagem */
        então acrescenta a mensagem com este id a retransmission-list
                                    /* Coloca a mensagem na lista de retransmissão */
    senão acresenta id a nack-list   /* Se não recebeu a mensagem, também a
                                    acrescenta a sua lista de mensagens perdidas */
```

QUADRO 4.2 – Recepção de uma mensagem pelo subprotocolo de difusão confiável

Conforme o descrito no quadro 4.2, quando recebe uma mensagem *m*, o subprotocolo a salva na received-list e o seu id é acrescentado a ack-list, representando que *m* deve ser confirmada positivamente. Se m.id estiver na nack-list, ele é retirado desta lista, pois a mensagem já foi recebida. Da mesma forma, se m está na lista de retransmisssão, ela é excluída desta lista, pois o nodo origem já executou este pedido de retransmissão.

Após isto, as confirmações positivas e negativas contidas no cabeçalho extraído da mensagem são processados. Todas as mensagens para os quais existe uma confirmação positiva não precisam ser confirmadas por este subprotocolo e devem ser retiradas da ack-list, se estiverem nesta lista. Se alguma das mensagens para os quais existe confirmação não estiver na received-list, esta mensagem foi perdida e o seu identificador é acrescentado a nack-list, para que algum subprotocolo a retransmita. Se alguma mensagem é confirmada negativamente em *m*, então, se esta mensagem foi recebida ou difundida pelo nodo, ela é acrescentada à retransmission-list para ser futuramente retransmitida para o processo origem de *m*. Se a mensagem confirmada negativamente não tiver sido recebida pelo subprotocolo, o seu identificador é acrescentado à nack-list.

Além destas ações, se o número de seqüência de **m** indicar que alguma mensagem difundida pelo subprotocolo do nodo originador da mensagem foi perdida, então o identificador desta mensagem perdida também é acrescentado a nack-list. Para fazer este teste, cada subprotocolo possui um vetor com o número seqüencial das mensagens recebidas de cada subprotocolo par. Este vetor é independente dos grupos e processos que utilizam o subprotocolo, pois a estrutura do FlexGroup garante que, se o subprotocolo receber uma mensagem difundida, todos os processos locais, que continuarem ativos durante tempo suficiente, finalmente receberão esta mensagem.

Desta forma, para qualquer mensagem, se algum subprotocolo que permanece ativo (que não falhe) receber uma mensagem difundida, então todos os subprotocolos e conseqüentemente todos processos que não falharem em algum momento receberão esta mensagem. Este protocolo assume que todos os nodos prosseguem enviando mensagens, o que é implementado através de mensagens nulas, enviadas através do mecanismo de SendToPeer, disponível no FlexGroup. Isto ocorre se o subprotocolo não enviou nenhuma mensagem válida durante um intervalo de tempo. Desta forma a difusão confiável é garantida, pois cada subprotocolo detecta as mensagens perdidas através da lista de confirmação dos outros subprotocolos e recupera as mensagens, garantindo as propriedades da difusão confiável, pois se um subprotocolo correto recebe uma mensagem todos os outros subprotocolos corretos também a receberão. Mensagens são entregues uma, e apenas uma vez, devido ao mecanismo de números seqüenciais para cada mensagem de cada nodo.

4.3.2 Subprotocolo de Difusão Causal

Conforme foi descrito na seção 2.1, a ordenação causal é um requisito necessário para muitas aplicações distribuídas, que precisam assegurar o respeito a uma possível relação de causa-efeito entre as mensagens difundidas no sistema. Desde o início dos anos oitenta, diversos protocolos de difusão causal tem sido propostos na literatura. Um dos protocolos mais conhecidos dentro desta classe de protocolos é o algoritmo de ordenação causal do Horus [BIR91b]. Este protocolo foi projetado para utilizar a estrutura modular, também oferecida pelo Horus, assumindo que existe um serviço de multicast confiável sobre o qual é estabelecida uma ordenação causal, se adaptando muito bem, portanto, à implementação no FlexGroup, sendo utilizado com o subprotocolo de difusão confiável descrito na seção anterior.

O protocolo é baseado em um vetor-de-relógios (*VT*) que é mantido por cada processo do grupo, e cujas entradas indicam o número de mensagens enviadas pelo processo e recebidas de cada um dos outros processos. Desta forma, em um grupo com *n* processos, cada processo *Pi* possui um vetor *VT*(*Pi*) com *n* posições. Em um processo *j*, a i-ésima posição deste vetor representa o número da última mensagem recebida do processo *i*, exceto a entrada *j*, que é a sua própria entrada, e contém o número da última mensagem por ele enviada. No FlexGroup, cada subprotocolo armazena este vetor para todos os processos locais pertencentes ao grupo e atualiza este vetor de acordo com as regras do algoritmo, pois todas as mensagens envidas e recebidas pelos processos locais pertencentes ao grupo passam pelo subprotocolo.

Quando um processo envia uma mensagem para o grupo, o seu vetor-de-relógios é enviado junto com a mensagem no cabeçalho acrescentado pelo subprotocolo para os processos destinatários, onde será avaliado pelo subprotocolo. Este protocolo utiliza a fila de mensagens implementadas através da lista de objetos da classe *Message*, para armazenar as mensagens já avaliadas pelo sistema e que ainda não podem ser entregues ao processo destino.

O vetor usado neste protocolo mantém informações suficientes para representar uma implicação entre mensagens, ou seja, uma ordem lógica em que transmissões e recepções devem ocorrer. Um vetor associado a uma mensagem *m*, informa o número de mensagens relacionadas que precedem *m*. Para isso, os vetores mantidos pelos processos são manipulados da seguinte maneira :

1. Quando um processo inicia sua execução, através do pedido de inclusão em um grupo que utiliza o subprotocolo, o vetor associado a este processo no subprotocolo de difusão causal da mesma máquina onde está o processo tem todas as suas entradas inicializadas com zero pelo subprotocolo.
2. Quando um processo *Pi* deseja enviar uma mensagem, a entrada no seu vetor local $VT(Pi)[i]$ é incrementada de um, e o vetor *VT* é colocado no cabeçalho do subprotocolo enviado junto com a mensagem para o nível inferior da pilha de subprotocolos.
3. Quando um processo recebe uma mensagem de outro processo *j*, ele retira o vetor do cabeçalho, atualiza o vetor dos processos locais, atribuindo para cada posição o máximo entre o vetor local e o vetor recebido com a mensagem. Se a mensagem atender a condição de entrega, ela é enviada para o nível superior na pilha de subprotocolos (ou para o núcleo se o subprotocolo estiver no topo da pilha). Também é verificada a condição de entrega para todas as outras mensagens que estiverem em estado de espera na fila de mensagens.

O protocolo deve decidir se uma mensagem que chega atende à condição de entrega ou se terá sua entrega retardada, devido à falta de alguma mensagem causalmente relacionada. Na recepção de uma mensagem *m*, enviada por *Pi* com um vetor $VT(m)$, o subprotocolo retarda a entrega de *m* até que para todos os vetores locais Pj (que são idênticos, pois as mensagens são recebidas pelo subprotocolo e depois repassadas em ordem para os processos locais) [BIR91b]:

- Para qualquer *k* entre 1 e n: $VT(m)[k] = VT(Pj)[k] + 1$, se $k = i$; e
  $VT(m)[k] \leq VT(Pj)[k]$, se $k \neq i$.

A primeira condição simplesmente indica que a mensagem é a próxima mensagem esperada do processo *i*, isto é, não estão faltando mensagens de *i*. Esta condição existe porque mensagens de um mesmo emissor sempre são causalmente relacionadas.

A segunda condição garante que o emissor já entregou todas as mensagens que podem ser causalmente relacionadas a esta mensagem, ou seja, todas as mensagens que o processo origem da mensagem recebeu antes de difundir a mensagem e que poderiam ter influenciado causalmente esta mensagem.

Se a mensagem passa por ambos os testes, o protocolo pode repassá-la ao próximo subprotocolo. Caso contrário, a mensagem deve ser armazenada para posterior análise, após a chegada das mensagens que faltam. A figura 4.4 mostra como funciona o protocolo para um grupo com 4 processos (A, B, C e D).

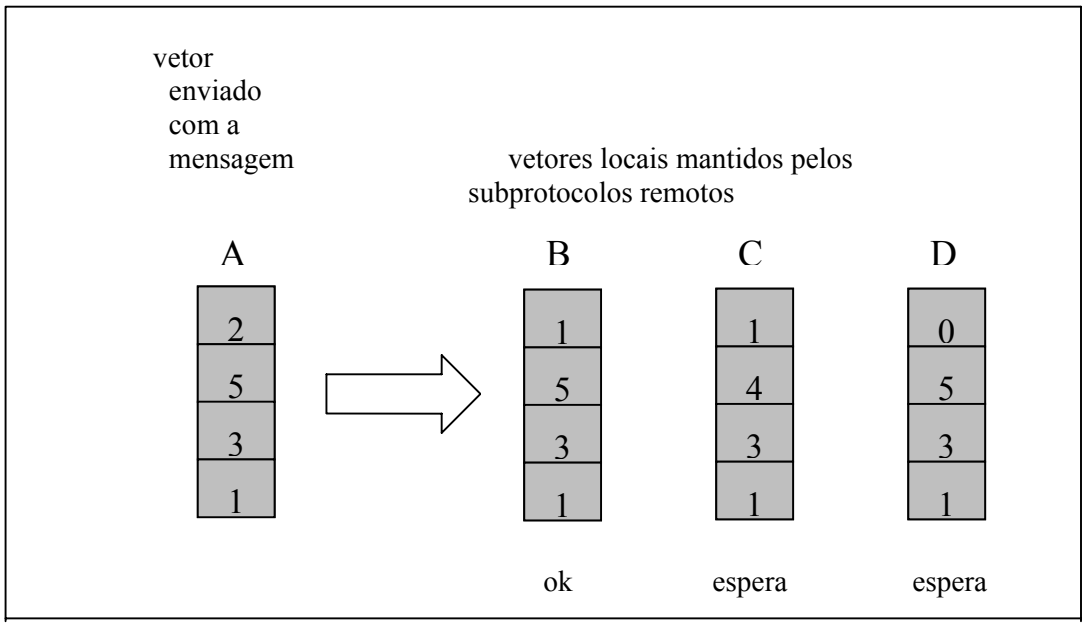

FIGURA 4.4 - Exemplo de comunicação, utilizando o subprotocolo de difusão causal

Neste exemplo, o processo A difunde uma mensagem com número de seqüência 2 e envia junto o vetor indicado na figura, contendo o valor das últimas mensagens recebidas dos outros processos. O processo B ao receber a mensagem de A pode entregá-la, porque todas as mensagens recebidas por A já foram recebidas por B e nenhuma mensagem de A deixou de ser recebida por B (como o valor do vetor para A é 1, a próxima mensagem esperada de A é a 2). Já processo C não poderá entregar a mensagem de A imediatamente porque ainda não recebeu a mensagem 5 de B, recebida por A antes de enviar sua mensagem. A mensagem de A fica então esperando a chegada desta mensagem para poder ser entregue. O processo D também não poderá entregar a mensagem neste momento, pois a mensagem que espera de A é a de identificador 1. Portanto D, deverá adiar a entrega desta mensagem, até que a mensagem anterior de A tenha sido recebida.

4.3.3 Subprotocolo de Difusão Atômica

Uma das principais propriedades que pode ser oferecida por um protocolo de comunicação em grupo é a atomicidade na entrega das mensagens para todos os membros do grupo. Muitos algoritmos para garantir a difusão atômica foram apresentados na literatura como, por exemplo, o AbCast [BIR91b], o Totem [MOS96] e o Newtop [MAC95b].

Foi escolhido o algoritmo do Newtop para ser implementado como um subprotocolo no protótipo do ambiente FlexGroup, devido à sua simplicidade, facilidade de entendimento e por não exigir nenhum tipo de hardware ou software específico. A seguir serão descritos sucintamente o protocolo e a sua implementação no FlexGroup.

O protocolo Newtop possui primitivas para entrega de mensagens com ordenação total e preservação da causalidade para os membros do grupo, permitindo que processos pertençam a vários grupos, garantindo a ordenação total também para estes processos.

Foi implementado o protocolo simétrico proposto no Newtop, onde todos os nodos executam o mesmo algoritmo. Uma adaptação feita para utilizar o protocolo no FlexGroup é que, em vez de cada processo executar o algoritmo, cada nodo da rede executa o algoritmo e atua em nome dos processos locais pertencentes ao grupo, mantendo as informações necessárias a cada processo. Cada processo i pertencente a G mantém um relógio lógico (um contador) representado por **RLi**, que é usado para numerar as mensagens enviadas por i. Este relógio é incrementado antes do envio de uma mensagem e seu valor é atribuído a um campo m.c do cabeçalho adicionado pelo subprotocolo a mensagem. Ao receber uma mensagem m, o subprotocolo extrai este valor do cabeçalho e altera o valor de RLi para max(RLi, m.c). Este mecanismo garante a ordenação causal das mensagens em função do campo m.c.

Cada processo i mantém um Vetor de Recepção (**VRi**). Este vetor contém um campo inteiro para cada processo j pertencente à atual visão de G. Este campo guarda o valor da última mensagem recebida de j. O valor **Di** representa o mínimo valor em VRi. Como Di é menor que qualquer RLj, o processo i tem certeza de não receber nenhuma mensagem com m.c menor que Di. Desta forma, i pode entregar todas as mensagens m recebidas, tais que m.c ≤ Di. Além disso, todas as mensagens que podem ser entregues devem ser entregues em ordem não descendente de seus números e um algoritmo determinístico é aplicado para decidir a ordem de entrega de mensagens de mesmo valor.

Estas condições garantem a entrega das mensagns com ordenação total, se for garantido que elas se tornem passíveis de entrega. Garante-se que uma mensagem seja passível de entrega somente se os processos do sistema continuam vivos, enviando mensagens, fazendo com que Di aumente com o passar do tempo. O Newtop faz com que cada processo utilize um mecanismo simples, chamado tempo de silêncio, que faz com que os processos se mantenham vivos através do envio de mensagens nulas, durante o período em que não enviam mensagens computacionais. Mensagens nulas contêm somente informações referentes ao protocolo e são enviadas através do mecanismo de envio de mensagens para os protocolos pares (SendToPeer) existente no FlexGroup. Quando um processo envia ou recebe uma mensagem nula, os RLs são incrementados de maneira normal, porém a mensagem não é colocada na fila de mensagens do subprotocolo.

O subprotocolo mantém um único RLi para todos os grupos em que participa e este relógio é avançado segundo as regras de relógio lógico, independente do grupo do qual recebe a mensagem. O subprotocolo mantém um VRi separado para cada processo que faz parte de um grupo que o utiliza, mesmo que o processo participe de múltiplos grupos. Ele garante uma ordenação total para todos os grupos, fazendo com que Di passe a ser o mínimo valor dos VRi de todos os grupos de que o processo faz parte e que utilizam o subprotocolo de difusão atômica. Uma mensagem com m.c ≤ Di pode ser entregue porque nenhuma mensagem com número menor que Di virá a ser recebida de qualquer grupo. O mecanismo de tempo de silêncio é utilizado independentemente para cada grupo, garantindo o aumento de Di com o passar do tempo. Como cada subprotocolo mantém estruturas separadas para cada grupo, a implementação é direta, bastando acrescentar estes campos à classe ProtocolGroup.

Como é preciso que um processo possa recuperar as mensagens perdidas enviadas por algum outro processo, eles guardam as mensagens já entregues, mas que podem ter que ser retransmitidas. Desta forma, é necessário um mecanismo que determine quando uma mensagem recebida pode ser descartada com segurança. No protocolo Newtop isto é feito através do conceito de estabilidade de mensagens. Uma

mensagem é considerada estável em um processo i quando este processo reconhece que todos os outros processos pertencentes à visão atual receberam esta mensagem.

As informações sobre a estabilidade das mensagens são enviadas também no cabeçalho das mensagens transmitidas ou nas mensagens nulas. Cada mensagem m transmitida por um processo i contém um campo **m.ldn** que recebe o valor atual de Di. Para identificar mensagens estáveis, cada processo mantém um Vetor de Estabilidade (**VE**), onde para cada processo i, o respectivo campo VEi[j] representa o último valor de m.ldn recebido de um processo j. Todas as mensagens recebidas por i com valor menor ou igual ao mínimo valor encontrado em VEi são estáveis e podem ser descartadas depois de serem entregues.

O protocolo do Newtop também apresenta características de tratamento de falhas e de gerenciamento da visão dos grupos, que poderiam ser implementados como outros subprotocolos independentes, porém devido a limitações de tempo e de escopo deste trabalho elas não foram implementados no FlexGroup.

4.4 Utilização do FlexGroup

O projeto do ambiente do FlexGroup visa oferecer uma interface de fácil utilização para a construção de aplicações que utilizem comunicação em grupo. A aplicação do FlexGroup deve ser também uma aplicação HetNOS, devido à necessidade de comunicação entre a biblioteca do sistema, ligada à aplicação, e o núcleo do servidor de comunicação em grupo.

Um dos princípios fundamentais de programação do ambiente HetNOS é que nomes (textuais) carregam maior significado e são mais portáveis do que valores ou códigos. Assim sendo, no HetNOS utiliza-se elementos textuais como elemento de transmissão de informações entre processos, bem como para identificação de objetos do sistema. Tais símbolos são implementados através de cadeias de caracteres ASCII. Esta mesma idéia foi adotada na construção do ambiente FlexGroup, devido à necessidade de interagir com as funções do HetNOS de comunicação e gerência de processos. Maiores detalhes sobre as vantagens e desvantagens da utilização de strings para identificação dos objetos do sistema e para envio das mensagens podem ser encontrados em [BAR93].

O processo de criação de uma aplicação de comunicação em grupo, utilizando o FlexGroup é composto das seguintes etapas:

1) Criar uma aplicação distribuída HetNOS, de acordo com o descrito em [BAR93], que utilize mecanismos de comunicação em grupo.
2) Incluir a biblioteca FlexLib no programa através do arquivo de cabeçalho flexlib.h.
3) Instanciar um objeto da classe Group (classe que encapsula a comunicação em grupo para a aplicação).
4) Criar um novo grupo, indicando os protocolos a serem utilizados pelo grupo; ou entrar em um grupo já existente através dos métodos disponíveis na Classe Group, descritos na seção 4.1.
5) Utilizar as funções de envio e recepção de mensagens para o grupo oferecidas pelo ambiente para atender as necessidades de comunicação da aplicação que está sendo construída.

A seguir é apresentado um exemplo de programação utilizando o ambiente FlexGroup:

```
#include "flexlib.h"  // Biblioteca para utilização do FlexGroup
#include "usrlib.h"  // Biblioteca para utilização do HetNOS
……                  //  Outras referências necessárias a aplicação

void main(char *argc, int argv) {
        int nOk;
        Group *mygroup;    // Objeto da classe Group
        char group_name[MAX_GROUP_NAME];
        char *my_name;
        char protocol_list[MAX_LEN_PROTOCOLS];
        …                       // Outras variáveis

        nOk = h_init();     // Inicialização do HetNOS
        if nOk <= 0 {
                printf("Erro na inicialização do HetNOS")
                exit(1);
        }
        nOk = flexgrp_init();    // Inicialização do FlexGroup
        if nOk < 0 {
                printf("Erro na inicialização do FlexGroup")
                exit(1);
        }
        ….                          // Outras inicializações
        mygroup = New (Group);     // Cria o objeto Group
        my_name = h_get_name();     // Retorna o nome HetNOS do processo
        strcpy(group_name, "GrupoExemplo");
        if (mygroup->GroupExists(group_name ))  {    // Testa se o grupo já existe
                mygroup->GroupJoin(my_name);    // Se já existe, tenta entrar no grupo
        }
        else {                                              // Se não existe, tenta criar o grupo
                strcpy(protocolo_list, "RELIABLE ATOMIC");
                                                    // Define os protocolos
                mygroup->GroupCreate(group_name, my_name, protocol_list, NULL);
                                                //      Cria  o  grupo,  identificado  por
group_name
        }
        ….     // Implementação da aplicação
        my_group->Send(msg);
        printf("Mensagen enviada: %s, msg);
        …
        my_group->Receive(*msg);
        printf("Mensagem recebida de %s :  %s", sender, msg):
        ….
        If (DestruirGrupo) my_group->GroupDestroy();   // Destrói o grupo
        Else my_group->LeaveGroup();                        // Sai do grupo, sem destruí-lo
        ….
        free(my_group);                 // Destrói o objeto Group
        h_terminate();                  // Encerra a conexão ao HetNOS
        exit(0);
```

}

Para executar a aplicação, é necessário executar os seguintes procedimentos:

- Disparar a execução do HetNOS nas máquinas onde se deseje executar a aplicação [BAR93].
- Disparar a execução do Servidor de Comunicação em Grupo do FlexGroup em cada máquina onde se deseje rodar a aplicação, através do seguinte comando no prompt do UNIX:
  flexsrv nome_host
  onde *nome_host* é o nome de uma máquina onde o Servidor já esteja rodando ou nulo no caso de ser a primeira máquina.
- Abrir uma sessão do shell do HetNOS através do programa *hlogin* e disparar a aplicação, através do seguinte comando:
  h nome_simbólico nome_executável
  onde *nome_executável* é o nome do programa e *nome_simbólico* é o nome HetNOS que o processo receberá.

4.5 Criação de um Subprotocolo

A criação de novos protocolos de maneira rápida e simplificada, a partir dos subprotocolos já existentes no sistema e enfocando somente as características necessárias ao protocolo que se queira desenvolver, é umas das principais propostas do projeto do FlexGroup. Os subprotocolos possuem uma interface pré-definida de comunicação com os outros subprotocolos e com o núcleo, que não precisa ser reimplementada, a não ser que o desenvolvedor do subprotocolo assim o deseje.

A seguir são descritos os principais passos para a construção de um novo subprotocolo:

- Criar uma nova classe que implemente o subprotocolo a partir da classe *Protocol*, ou de um outro subprotocolo já existente.
- Estender os métodos de construção e destruição da classe (se necessário). O construtor deve obter os parâmetros necessários para a execução do subprotocolo, como os grupos que o utilizam e os processos pertencentes a cada grupo.
- Criar uma classe derivada da classe Group, contendo as informações necessárias sobre os grupos para o subprotocolo, além das já existentes na classe Group. Estes atributos devem ser inicializados no construtor da classe.
- Criar uma classe derivada da classe *Message*, contendo as informações necessárias sobre as mensagens enviadas e recebidas pelo protocolo. Estas informações devem ser atualizadas na recepção e envio de mensagens. Esta classe implementa as entradas no buffer de mensagens do subprotocolo. Este buffer pode conter mensagens recebidas, mas não entregues, mensagens a serem enviadas, ou mensagens já entregues, mas que podem precisar retransmissão, dependendo do protocolo que se deseje implementar.
- Criar o tratamento das mensagens que o subprotocolo envia diretamente para os subprotocolos pares nas outras máquinas (**ProtocolMessages**). Este tratamento só precisa ser feito se for necessária a troca direta de informação entre as diversas instâncias do subprotocolo.

- Estender os métodos virtuais que tratam as mensagens de Send e Receive para acrescentar e retirar as informações necessárias do header da mensagem no envio e na recepção de mensagens. Além disso, o subprotocolo deve também implementar métodos para determinar quando uma mensagem atende à condição de entrega, ou seja, quando ela pode ser repassada para o nível superior da pilha (através do método SendNext) com a garantia de atender as propriedades implementadas pelo subprotocolo.
- Estender o tratamento das mensagens de **GroupCreate**, **GroupDestroy** e **GroupUpdate** para manter a visão do grupo necessária ao subprotocolo.

Cada subprotocolo é também uma aplicação HetNOS, portanto precisa incluir as bibliotecas do HetNOS e chamar as funções de inicialização e encerramento da conexão com o HetNOS da mesma maneira que descrito na seção anterior. Da mesma forma, para executar o Servidor de Comunicação em Grupo em uma estação, é necessário que esta estação esteja executando o HetNOS.

Futuramente, pretende-se que os subprotocolos passem a ser implementados como threads do Servidor de Comunicação, melhorando o desempenho e deixando de ser aplicações HetNOS. Dessa forma não necessitaria mais os procedimentos acima para dispará-los, pois passariam a fazer parte da DCL do HetNOS, sendo disparados automaticamente pelo servidor de inicialização do HetNOS.

4.6 Análise dos Resultados

O overhead acrescentado pela plataforma de desenvolvimento (Unix e HetNOS), juntamente com as simplificações adotadas na implementação do protótipo, produziram tempos de execução muito acima do esperado, inviabilizando a comparação dos resultados com o desempenho de outros ambientes de comunicação em grupo descritos na literatura. Mesmo assim, os testes realizados com o protótipo do servidor do ambiente de comunicação em grupo FlexGroup e com os subprotocolos implementados como parte do protótipo permitiram algumas constatações a respeito dos serviços oferecidos para a construção de aplicações de comunicação em grupo e também permitiu detectar-se os pontos de estrangulamento (críticos ao desempenho). Tais resultados apontam quais operações do ambiente merecem uma maior atenção em futuras revisões do projeto.

Seria importante, para validar o sistema, a construção de diversas aplicações de comunicação em grupo, que utilizassem diferentes composições de subprotocolos, permitindo uma melhor avaliação da integração dos subprotocolos e uma comparação do desempenho das aplicações, utilizando diferentes subprotocolos, além de submeter o ambiente a diferentes volumes de envio e recepções de mensagens.

A seguir são realizadas avaliações do ambiente em relação às principais características exigidas por aplicações de comunicação em grupo.

.

4.6.1 Tolerância a Falhas

A determinação das restrições quanto às falhas suportadas é de extrema importância tanto no desenvolvimento como na utilização de um protocolo de comunicação em grupo ou difusão confiável. Quanto menos restritiva for a semântica de falhas considerada, mais robusto, caro e complexo torna-se o sistema. O FlexGroup suporta somente falhas de crash para os servidores que executam em cada nodo, ou seja, quando ocorrer uma falha, o processo falho simplesmente pára de enviar mensagens.

Isto pode ser assumido, pois a camada de comunicação do HetNOS assegura este tipo de comportamento, mas não garante porém que atrasos na comunicação por parte de um processo sejam interpretados como uma falha, retirando este processo do sistema, mesmo que ele não tenha realmente falhado. Esta semântica é a mais restritiva em relação a falhas, porém o custo e a complexidade dos protocolos é bastante reduzida se for possível utilizar este modelo de falhas.

Para os subprotocolos, nenhum outro tipo de falhas é tratado explicitamente. Pode-se, entretanto, tratar outros tipos de falhas através de um novo subprotocolo que apresente um tratamento de falhas de temporização ou mesmo falhas arbitrárias. Este tipo de subprotocolo poderia garantir detecção de falhas para todos os outros subprotocolos que estivessem acima na pilha de subprotocolos.

4.6.2 Desempenho

Um dos fatores críticos de uma aplicação distribuída é o seu desempenho. Conseqüentemente, os serviços utilizados por aplicações distribuídas devem ter uma preocupação em oferecer serviços com alta confiabilidade, porém sem comprometer de forma muito acentuada o desempenho do sistema. Conforme o que já foi dito anteriormente, o overhead devido à utilização do HetNOS como camada de comunicação impede a comparação do ambiente com outros softwares de comunicação existentes.

Outro fator que prejudica o desempenho do FlexGroup é a implementação dos subprotocolos como processos Unix independentes que se comunicam via troca de mensagens do HetNOS. Mesmo as mensagens sendo locais, isto acarreta um retardo muito grande no envio e na recepção de mensagens pois, para cada mensagem difundida por uma aplicação, são enviadas *n* mensagens do HetNOS entre cada um dos subprotocolos da pilha, onde *n* é o número de subprotocolos, até que a mensagem chegue de volta ao núcleo e seja difundida para os outros nodos. Da mesma forma, na recepção da mensagem, são enviadas outras *n* mensagens em cada nodo, até que a mensagem chegue ao seu destino. A melhor solução para este problema seria a implementação dos subprotocolos como threads ao invés de processos, o que permitiria que eles compartilhassem memória, suprimindo a necessidade do envio dessas mensagens. No atual estágio de implementação do ambiente HetNOS, porém, ainda não é possível o desenvolvimento de aplicações que utilizem múltiplas threads de execução.

Da mesma forma, o núcleo do servidor teria uma grande melhora de performance se fosse implementado através de múltiplas threads, pois cada pedido da aplicação poderia ser atendido por uma thread independente, fazendo com que vários pedidos fossem atendidos em paralelo e diminuindo o tempo de latência na entrega de mensagens.

4.6.3 Custo e Complexidade

Protocolos de comunicação em grupo sempre introduzem custos adicionais aos sistemas onde são utilizados. Para determinar o custo de um protocolo, dois fatores são determinantes: o overhead inserido nas mensagens difundidas e as mensagens adicionais introduzidas pelo processo. No caso do FlexGroup, o overhead introduzido é diretamente dependente dos subprotocolos que são utilizados, que colocam as suas informações como cabeçalho da mensagem sendo difundida. Caso muitos subprotocolos sejam empregados por um grupo, este cabeçalho tende a ser maior do que seria se um único protocolo fosse utilizado, pois vários desses subprotocolos podem enviar a mesma

informação na mensagem. O núcleo do servidor ainda acrescenta a esta mensagem o nome do grupo a que se destina e as informações necessárias à confirmação positiva ou negativa da recepção de mensagens dos outros servidores, conforme o que foi descrito na seção que descreve a difusão confiável entre os servidores. As mensagens HetNOS são sempre enviadas em pacotes múltiplos de 1024 bytes. Desta forma, as informações adicionais não causam uma grande impacto adicional ao sistema, se as mensagens enviadas forem bem maiores do que este tamanho, quando um pacote adicional não seria significativo; ou bem menores, quando as informações caberiam em um único pacote. Elas, porém, podem influenciar negativamente se as mensagens tiverem um tamanho próximo do tamanho do pacote HetNOS, pois novos pacotes podem ter de ser enviados contendo somente informações de controle.

Em relação às mensagens adicionais, foi visto na seção anterior que devido aos subprotocolos serem implementados como processos, são necessárias $n * p$ mensagens locais adicionais para cada mensagem difundida, onde n é o número de nodos e p o número de subprotocolos utilizados pelo grupo. Além destas mensagens, existe o custo das mensagens nulas enviadas pelos núcleos, contendo somente as informações necessárias para a difusão confiável entre os núcleos, no caso de não haver mensagens de aplicação para serem difundidas.

4.6.4 Escalabilidade e Flexibilidade

É importante na análise de um protocolo determinar para quais aplicações este protocolo foi projetado. Também é importante analisar o comportamento deste protocolo quando o tamanho ou o número de grupos cresce bastante.

O FlexGroup foi projetado tendo a flexibilidade como um dos principais pontos do projeto. Ele permite a utilização e a implementação de diversos tipos de protocolos, atendendo a um grande número de aplicações. Ele permite ainda diversas composições entre os subprotocolos, criando na prática um número muito grande de serviços que podem ser oferecidos pelo ambiente, mesmo que algumas destas composições não façam sentido ou não tenham utilidade prática.

Em relação à escalabilidade, o sistema é sensível ao crescimento no número de máquinas onde está sendo executado. O ambiente operacional utilizado não permitiu um teste com um número muito grande de máquinas, porém ao aumentarmos o número de máquinas, notou-se uma queda de desempenho não proporcional ao aumento de máquinas, fato este que se deve em grande parte à utilização do HetNOS, que também apresenta este tipo de comportamento ao crescer o número de estações onde está sendo executado, devido principalmente à sua estrutura em forma de anel.

Quanto ao crescimento do número de grupos e processos que o utilizam, o sistema se apresentou bastante estável, pois os subprotocolos são executados independentemente dos grupos que os utilizam, e os subprotocolos disponíveis não levam em conta os processos pertencentes aos grupos e sim os nodos existentes no sistema.

5 Conclusão

Este trabalho apresentou a proposta de um ambiente flexível para a construção e utilização de protocolos de comunicação em grupo. Dois aspectos foram considerados na busca deste objetivo: a modularização dos protocolos, dividindo-os em subprotocolos e a definição de uma interface única para as aplicações de comunicação em grupo acessarem estes subprotocolos. A comunicação em grupo é um paradigma bastante utilizado por aplicações que precisam realizar uma tarefa de maneira cooperativa, onde cada processo pode executar uma parte do processamento, visando um melhor desempenho, ou todos podem executar o mesmo processamento, visando aumentar a confiabilidade e a tolerância a falhas do sistema. Por outro lado, este mecanismo de envio de mensagens a todos os membros de um grupo gera uma complexidade adicional ao sistema, pois precisam assegurar a garantia de recepção das mensagens por todos os membros do grupo.

As características de confiabilidade, desempenho e custo, que são fundamentais em sistemas distribuídos, estão intimamente associadas em ambientes de comunicação em grupo. Quanto maior for a classe de falhas suportada por um protocolo, maior será sua confiabilidade, porém maiores serão também os seus custos, e pior será seu desempenho. Outra característica fundamental de um protocolo de difusão é a ordenação, que quanto mais rígida for, maior será a confiabilidade e também o custo do protocolo resultante. Ordenações menos rígidas permitem menor custo e melhor desempenho, porém podem ser inaceitáveis para aplicações que necessitem garantir a consistência de maneira distribuída.

De tudo isso, pode-se concluir que não existe um sistema que possa ser apontado como o melhor, ou como um caminho a ser seguido. Cada protocolo possui uma classe de aplicações à qual atende plenamente e outras às quais não atende tão satisfatoriamente. Dessa forma, mecanismos que permitam a utilização de protocolos com diferentes características em um mesmo ambiente têm sido propostos na literatura. Entretanto, mesmo estas hierarquias de protocolos não oferecem a possibilidade de adaptar os protocolos à necessidade das aplicações, além de não poderem ser expandidos para suportar novos protocolos.

O presente trabalho propôs a construção de um ambiente para o desenvolvimento e utilização de protocolos de comunicação em grupo. A idéia central do ambiente é dividir as características componentes de um protocolo, criando subprotocolos que implementem cada uma dessas características. Dessa forma, facilita-se a criação de novos protocolos, que podem utilizar os subprotocolos já existentes, criando apenas subprotocolos para implementar os serviços ainda não disponíveis no ambiente. Além disso, as aplicações que utilizam o paradigma de comunicação em grupo podem escolher em tempo de execução os subprotocolos que melhor atendem às suas necessidades de comunicação, tolerância a falhas e desempenho.

O ambiente proposto é composto por uma biblioteca e um servidor, subdividido em um núcleo e subprotocolos. A biblioteca é ligada às aplicações do usuário, fornecendo uma visão única dos serviços de comunicação em grupo. O núcleo é o responsável por gerenciar toda a comunicação entre os nodos e entre os componentes do ambiente e por atender as requisições de serviços das aplicações de usuário. Os subprotocolos implementam os serviços de comunicação em grupo propriamente ditos, sendo que eles podem ser utilizados em diversas combinações para oferecer o tipo de serviço desejado por uma aplicação.

Para validar a concepção do ambiente foi construído um protótipo do servidor de comunicação em grupo. O protótipo implementa as principais funcionalidades descritas no projeto do ambiente, possuindo também alguns subprotocolos básicos de comunicação em grupo, que implementam serviços como difusão confiável, ordenação causal e ordenação total das mensagens. O desenvolvimento destes subprotocolos possibilitou a validação do ambiente como ferramenta de construção de novos protocolos e oferece uma base para o desenvolvimento futuro de um repositório de subprotocolos.

O desempenho do protótipo desenvolvido foi inferior ao desejado, devido à plataforma de desenvolvimento e às simplificações adotadas no protótipo. Isto, no entanto, não inviabiliza a utilização do projeto para construção de novos protocolos, para a comparação entre os diversos protocolos existentes, ou para a construção e validação de aplicações de comunicação em grupo em geral. Ele pode ser utilizado também como uma ferramenta para determinar o melhor protocolo para uma determinada aplicação, pois permite compararmos o funcionamento de uma aplicação com diferentes protocolos, em tempo de execução.

Futuramente, devem ser construídas novas aplicações de comunicação em grupo, para testar e validar o funcionamento do ambiente, permitindo uma melhor avaliação sobre o modelo proposto e sobre os subprotocolos existentes.

Da mesmo forma, faz-se necessário implementar novos protocolos visando enriquecer o ambiente e permitir uma melhor análise do funcionamento e correção do sistema. Além disso, estes subprotocolos permitirão que novas aplicações de comunicação em grupo possam ser construídas para utilizar o ambiente.

Existem diversas melhorias que podem ser realizas no ambiente FlexGroup como,m por exemplo, a transformação dos subprotocolos em threads do servidor ao invés de processos separados, melhorando a performance, e o porte para utilização com outras camadas de comunicação, como PVM ou MPI.

Além disso, como forma de avaliação prática para posterior reestudo dos mecanismos propostos, faz-se necessária a aplicação do sistema em situações reais de uso, como em aplicações que utilizem comunicação em grupo para manter dados replicados em diversas máquinas distribuídas em uma rede. Tal experimento pode ser realizado a partir do refinamento da presente implementação.

Por fim, convém salientar o conhecimento e experiência adquiridos com este trabalho, oriundos do contato teórico com tópicos de diversas áreas de pesquisa (tolerância a falhas, sistemas distribuídos, orientação a objetos) e do exercício de programação distribuída experimentado no desenvolvimento do projeto.

Bibliografia

[AMI 92] AMIR, Yair et al. **Total Ordering of Messages in Broadcast Domains**. Jerusalem: Hebrew University of Jerusalem, 1992. (Technical Report CS92-9).

[AMI 95] AMIR, Y. et al. The Totem Single-Ring Ordering and Membership Protocol. **ACM Transactions on Computer Systems**, New York, v. 13, n.4, p. 312-342, Nov. 1995.

[ANC 95] ANCEAUME, Emmanuelle et al. **On the formal specification of group membership services.** Ithaca, New York: Computer Science Department, Cornell University, 1995. (Technical Report 9-1534.)

[BAR 92] BARCELLOS, Antônio Marinho P. **HetNOS - Um Sistema Operacional de Rede Heterogêneo**. Porto Alegre: CPGCC da UFRGS, 1993. Dissertação de Mestrado.

[BAR 93] BARCELLOS, Antônio Marinho P. **HetNOS - Um Sistema Operacional de Rede como Ferramenta de Apoio ao Desenvolvimento de Sistemas Distribuídos**. Porto Alegre: CPGCC da UFRGS, 1994. Relatório de Pesquisa.

[BIR 91a] BIRMAN, Kenneth P.; COOPER, Robert; GLEESON, Barry. **Design Alternatives for Process Group Membership and Multicast**. Ithaca, New York: Cornell University, 1991. (Technical Report TR-91-1185.)

[BIR 91b] BIRMAN, Kenneth; SCHIPER, André; STEPHENSON, Pat. Lightweight Causal and Atomic Group Multicast. **ACM Transactions on Computer Systems**, New York, v.9, n.3, p.272-314, Aug. 1991.

[BIR 96] BIRMAN, Kenneth. **Building Secure and Reliable Network Applications**. USA: Manning Publications, 1996.

[CHA 91] CHANDRA, Tushar; TOUEG, Sam. Unrealiable Failure Detectors for Asynchronous Distributed Systems. In: ANNUAL ACM SYMPOSIUM ON

PRINCIPLES OF DISTRIBUTED COMPUTING, 1991. **Proceedings...** [S.l.]: ACM Press, 1991. p. 325-340.

[CHA 95] CHANDRA, Tushar et al. **On the Impossibility of Group Membership**. Ithaca, New York: Computer Science Department, Cornell University, 1995. (Technical Report 95-1533.)

[DOL 92] DOLEV, Danny et al. Membership algorithms for multicast communication groups. In: INTERNATIONAL WORKSHOP ON DISTRIBUTED ALGORITHMS, WDAG6, 1992**. Proceedings...** [S.l.:s.n.], 1992. p. 292-313.

[HAD 93] HADZILLACOS, Vassos; TOUEG, Sam. Fault Tolerant Broadcasts an Related Problems. In: MULLENDER, Sape (Ed.). **Distributed Systems.** 2.ed. New York: ACM Press, 1993.

[HAD 94] HADZILLACOS, Vassos; TOUEG, Sam. **A Modular Approach to Fault-Tolerant Broadcasts and Related Problems**. Toronto: University of Toronto, 1994. Technical Report.

[JAL 94] JALOTE, Pankaj. **Fault Tolerance in Distributed Systems**. Englewood Cliffs: Prentice-Hall, 1994.

[MAC 95a] MACEDO, R.; EZHILCHELVAN, P.; SHRIVASTAVA, S. K. Newtop: A Fault-Tolerant Group Communication Protocol. In: INTERNATIONAL CONFERENCE ON DISTRIBUTED SYSTEMS, 1995. **Proceedings...** Vancouver: [s.n.],1995.

[MAC 95b] MACEDO, R.; EZHILCHELVAN, P.; SHRIVASTAVA, S. K. Flow Control Schemes for a Fault-Tolerant Multicast Protocol. In: IEEE PACIFIC RING INTERNATIONAL SYMPOSIUM ON FAULT-TOLERANT SYSTEMS, 1995. **Proceedings...** Newport Beach, California: [s.n.],1995.

[MOS 96] MOSER, L. E. et al. Totem: A Fault-Tolerant Multicast Group Communication System. **Communications of the ACM**, New York, v. 39, n. 4, p. 54-63, Apr. 1996.

[REN 96] RENESSEE, Robbert van et al. **Horus**: A Flexible Group Communication System. Ithaca, New York: Department of Computer Science, Cornell University, 1996. Technical Report.

[RIV 96] RIVERA, Rodrigo Dias. **Estudo e Análise de Protocolos de Difusão Confiável e Comunicação em Grupo**: trabalho individual. Porto Alegre: CPGCC da UFRGS, 1996. (TI-601).

[ROD 92] RODRIGUES, L.; VERÍSSIMO, P. xAmp: a multi-primitive group communication service. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 1992. **Proceedings...** Houston, Texas: IEEE Press, 1992.

[ROD 95] RODRIGUES, Luis et al. **A Transparent Light-Weight Group Service.** Portugal: INESC, 1995. Technical Report.

[SCH 95] SCHIPER, André; GUERRAOUI, Rachid. **Total Order Multicast to Multiple Groups**. Lausanne: Ecole Polytechnique Fédérale de Lausanne, 1995. Technical Report.

[TAN 92] TANENBAUM, Andrew. **Modern Operating Systems**. Englewood Cliffs: Prentice-Hall, 1992.

[TAN 95] TANENBAUM, Andrew. **Distributed Operating Systems**. Englewood Cliffs: Prentice-Hall, 1995.

[VER 89] VERÍSSIMO, Paulo. **Comunicação em Grupo Fiável em Sistemas Distribuídos**. Portugal: INESC, 1989. Tese de Doutorado.

[VOG 92] VOGELS, W.; RODRIGUES, L.; VERÍSSIMO, P. Fast Group Communications for Standard Workstations. In: OPEN-FORUM, TECHNICAL CONFERENCEe, 1992. **Proceedings...** Utrecht, Netherlands: [s.n.], 1992.