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

Utilização de Multicast na Disseminação de Escritas em Ambientes Replicados

por
BERENICE FUCHS HOFSETZ

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

Taisy Silva Weber
Orientadora

Porto Alegre, Março de 1999

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Hofsetz, Berenice Fuchs

Utilização de Multicast para Disseminação de Escritas em Ambientes Replicados / por Berenice Fuchs Hofsetz - Porto Alegre, PPGC da UFRGS, 1999.

76f..: il.

Dissertação (mestrado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 1999. Orientadora: Weber, Taisy Silva.

1. Tolerância a falhas. 2. Replicação. 3. Comunicação de Grupo. 4. Multicast. I. Weber, Taisy Silva. II. Título.

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

Agradecimentos

Agradeço, primeiramente, ao meu marido Christian, por seu amor e companheirismo. Seu apoio incondicional foi de grande importância para a conclusão deste trabalho.

Um agradecimento especial para a minha orientadora, Taisy S. Weber, pela atenciosa e paciente orientação.

Agradeço aos meus pais Osmar Fuchs e Terezinha Lurdes Fuchs, pelo amor, dedicação e pela confiança que sempre depositaram em mim. Agradeço também o apoio do meu irmão Júlio, dos meus sogros Geraldo e Eluza, e das minhas cunhadas, Karen, Loren e Kelly.

Agradeço aos amigos e colegas da UFRGS, especialmente os do grupo de Tolerância a Falhas.

Ao CNPq, pelo apoio na realização deste trabalho.

Finalmente, agradeço a Deus, pois sem Ele, nada disso seria possível.

Sumário

Lista de Abreviaturas 7
Lista de Figuras 8
Lista de Tabelas 9
Resumo 10
Abstract 11
1 Introdução 12
1.1 Motivação 12
1.2 Seções 13
2 Replicação de arquivos 14
2.1 Grupos de replicação 14
2.2 Abordagem da cópia primária 15
2.3 Abordagem das cópias ativas 16
2.3.1 Máquina de estado 16
2.3.2 Difusão atômica 17
2.4 Disseminação de escritas 17
2.5 Pontos de falhas 18
2.6 Solução proposta 18
3 Comunicação de Grupo 20
3.1 Grupos de *multicast* 20
3.2 Propriedades de interesse na troca de mensagens 20
3.3 Falhas em sistemas distribuídos 21
3.4 Protocolos de comunicação de grupo 22
3.4.1 xAMp 22
3.4.1.1 Transmissão com resposta 23
3.4.1.2 Best-effort agreement 24
3.4.1.3 AtLeast agreement 24
3.4.1.4 Multicast Causal 25
3.4.1.5 Multicast atômico 25
3.4.1.6 Protocolo de membership 26
3.4.1.7 Endereçamento de grupos e detecção de falhas 28
3.4.2 Newtop 29
3.4.2.1 Ordenação total simétrica 30
3.4.2.2 Ordenação total assimétrica 31
3.4.2.4 Membership 32
3.4.2.5 Protocolo de Formação de Grupo 32
3.4.3 Transis 33
3.4.3.1 Operações particionáveis 33
3.4.3.2 Serviço de comunicação de grupo 35

3.4.3.3 Multicast confiável 36
3.4.3.4 Multicast causal 37
3.4.3.5 Multicast com acordo 37
3.4.3.6 Protocolo de membership 38

3.4.4 Horus 38
3.4.4.1 Modelo de Grupo do Horus 39
3.4.4.2 Camadas e protocolos 41
3.4.4.3 Multicast causal 42
3.4.4.4 Multicast totalmente ordenado 43
3.4.4.5 Membership 43

3.4.5 Totem 44
3.4.5.1 Protocolo Single-Ring 46
3.4.5.3 Protocolo de membership local 48
3.4.5.4 Protocolo Multiple-ring 49
3.4.5.5 Manutenção da topologia da rede 50

3.5 Comparação dos sistemas 51
3.5.1 Particionamento com partição primária 51
3.5.2 Reintegração após particionamento 51
3.5.3 Sobreposição de grupos de replicação 52
3.5.4 Primitivas de multicast confiável 52
3.5.5 Membership e visão de grupo 52

4 Protótipo para disseminação de escrita para arquivos replicados via multicast - PDERM 54
4.1 Descrição do modelo 54
4.1.1 Clientes 54
4.1.2 Servidor primário 55
4.1.3 Servidores secundários 55
4.1.4 Particionamento de rede e falha no servidor primário 55
4.2 Arquivos replicados e grupos de replicação 55
4.3 Ambiente 56
4.4 Utilização do xAMp 57
4.4.1 Execução do xAMp 57
4.4.2 Inicialização 57
4.4.3 Entrada e saída de grupos 58
4.4.4 Manipulação de serviço de grupo 59
4.4.4.1 Visão do grupo 59
4.4.4.2 Confirmações 59
4.4.4.3 Recebimento de mensagens 59
4.4.5 Serviço de mensagens 60
4.4.6 Demais rotinas utilizadas 60
4.5 Implementação do protótipo 61
4.5.1 Servidor primário 63
4.5.1.1 Tratamento da visão do grupo de servidores 64
4.5.1.2 Tratamento da visão do grupo de clientes 64
4.5.1.2 Tratamento de mensagens provenientes de clientes 65

4.5.1.3 Tratamento de mensagens de modificação em arquivos ____ 66
4.5.2 Servidores secundários ____ 67
4.6 Experimentos ____ 68
4.6.1 Execução do protótipo ____ 69
4.6.2 Falha de servidores secundários ____ 69
4.6.3 Falha do servidor primário ____ 70
5 Conclusão ____ 71
5.1 Replicação de arquivos ____ 71
5.2 Protocolos de comunicação de grupo ____ 71
5.3 Disseminação de escritas via *multicast* ____ 72
5.4 xAMp na disseminação de escritas ____ 72
5.5 Direções futuras ____ 72
Bibliografia ____ 74

Lista de Abreviaturas

AN	Abstract Network
ATM	Assincronous Transfer Mode
BC	Block Couter
CCS	Current Configuration Set
FIFO	First In First Out
GV	Group View
LAN	Local Area Network
LSE	Local Supeort Environment
MGS	Multicast Group of Stations
NEWTOP	NEWcastle Total Order Protocol
NFS	Network File Sistem, o sistema de arquivo do sistema operacional UNIX
PDERM	Protótipo para a Disseminação de Escritas em arquivos Replicados, através de *Multicast*
RNFS	Ssistema de arquivos distribuído e tolerante a falhas para UNIX
RPC	Remote Procedure Call
RRPC	Replicated Remote Procedure Call
UGI	Uniform Group Interface
xAMp	eXtended Atomic Multicast Protocol

Lista de Figuras

FIGURA 2.1- Grupos de Replicação..15

FIGURA 2.2 - Abordagem da cópia primária....................................... 15

FIGURA 3.1- Classificação de falhas... 21

FIGURA 3.2 - Sobreposição de grupos 30

FIGURA 3.3 - Camadas de protocolos do Horus................................. 39

FIGURA 3.4 - Anel lógico de passagem de *token* 46

FIGURA 3.5 - Múltiplos anéis conectados por *gateways*...................... 49

FIGURA 4.1 - Modelo simplificado de grupo de replicação 54

FIGURA 4.2 - Múltiplos grupos de replicação..................................... 56

FIGURA 4.3 - Intersecção de grupos de replicação............................. 56

FIGURA 4.4 - Modelo de replicação utilizado no protótipo.................. 62

Lista de Tabelas

TABELA 4.1 - Distribuição dos arquivos replicados............................. 61

TABELA 4.2 - Composição dos grupos... 62

TABELA 4.1 - Grupos abertos por cada servidor................................. 67

Resumo

Este trabalho trata da utilização de protocolos de comunicação de grupo para a disseminação de escritas em arquivos replicados.

A replicação de arquivos tem como objetivo aumentar a disponibilidade dos dados mesmo mediante a ocorrência de alguma falha. Existem duas abordagens principais para a replicação de arquivos: a da cópia primária e das cópias ativas. Em ambas as abordagens é necessário que seja mantida a integridade dos dados replicados, de forma que todos cópias dos arquivos replicados estejam no mesmo estado. Essa integridade pode ser mantida pela escolha correta de uma estratégia de disseminação de escritas.

Como os servidores que mantém cópias do mesmo arquivo formam um grupo de replicação, a disseminação de escritas pode ser feita através de comunicação de grupos. Neste trabalho são apresentados os sistemas de comunicação de grupo xAMp, da Universidade de Lisboa; Totem, Universidade da Califórnia; Transis da Universidade de Hebréia de Jerusalém; Horus, da Universidade de Cornell e Newtop da Universidade de Newcastle. Todos os sistemas descritos possuem características de comunicação de grupo e *membership* que permitem a sua utilização na disseminação de escritas para arquivos replicados.

Este trabalho descreve, também, o protótipo PDERM (Protótipo para a Disseminação de Escritas em arquivos Replicados, através de *Multicast*), implementado para analisar o comportamento de um sistema de comunicação de grupo, o xAMp, na disseminação de escritas em arquivos replicados pela estratégia da cópia primária. Foi analisado o aspecto da manutenção da integridade das réplicas mesmo na ocorrência de falha do servidor primário.

PALAVRAS-CHAVE: tolerância a falhas, sistemas distribuídos, replicação de arquivos, comunicação de grupo, *multicast*.

TITLE: "USING MULTICAST TO DISSEMINATE UPDATE OPERATIONS IN REPLICATED ENVIRONMENTS"

Abstract

This work discusses the use of group communication protocols to disseminate update operations in replicated files.

The main goal in file replication is to increase the data availability even during a fault occurrence. There are two main approaches of file replication: the primary site and the active replicas stratagy. The integrity of replicated data is required to both approaches, i. e., all replicated copies must be in the same state. This integrity can be obtained by choosing a correct strategy to disseminate update operations requests.

Considering that the servers which have copies of the same file form a replication group, the update operations dissemination can be done with group communication. This work presents the group communications systems namely: xAMP, from University of Lisboa, Newtop, from University of Newcastle, Transis, from The Hebrew University of Jerusalem; Horus, from Cornell University, and Totem, from University of California. All this systems have group communication and membership protocols which allow their use to disseminate update operations in replicated files.

This work also discusses the PDERM prototype (dissemination of update operations in replicated files using multicast). This prototype was implemented to analyse the behavior of the xAMp group communication system in the update operations dissemination with primary site strategy. The replicas integrity maintenance aspect was analysed even during a primary server fault.

KEYWORDS: Fault tolerance, distributed systems, file replication, group communication, multicast.

1 Introdução

Sistemas informatizados estão sendo cada vez mais utilizados em todas as áreas da vida moderna. Este fato faz com que o grau de dependência do homem por tais sistemas cresça de maneira considerável. Na mesma proporção em que aumenta a dependência do homem por sistemas computadorizados, cresce a necessidade de confiabilidade, segurança e disponibilidade desses sistemas, uma vez que as conseqüências em caso de colapso podem ser desastrosas.

A área da computação que tem como objetivo aumentar a confiabilidade dos sistemas é a Tolerância a Falhas. Neste ramo da computação são estudados métodos e técnicas para prover a garantia de que os sistemas vão funcionar conforme as suas especificações, mesmo mediante a ocorrência de alguma falha. Essa garantia é provida, em grande parte dos sistemas tolerantes a falhas, através da utilização de redundâncias em determinados componentes do sistema. Este trabalho trata da redundância de dados, através da replicação de arquivos, como forma de tolerar falhas.

1.1 Motivação

A replicação de arquivos aumenta de forma considerável a disponibilidade de dados a usuários. Esse aumento de disponibilidade, no entanto, traz consigo problemas de gerenciamento de réplicas: o resultado de operações de escrita (que modificam o estado do arquivo) feitas em arquivos replicados devem ser o mesmo que o resultado dessas operações executadas em arquivos não replicados. Trata-se da característica de serialização das operações de escrita, que tem como objetivo a manutenção da integridade das réplicas.

Para que todas as cópias dos arquivos replicados de um sistema estejam no mesmo estado, é necessário uma estratégia para disseminação das modificações feitas em uma cópia do arquivo para todas as outras cópias.

Servidores que mantém cópias de um mesmo arquivo formam um grupo de replicação e a comunicação desses servidores pode ser feita através de protocolos de comunicação de grupos.

A comunicação de grupos é um dos estudos que mais cresce dentro das pesquisas em sistemas distribuídos. Assim como utilizamos freqüentemente nomes coletivos para referenciar grupos de pessoas, como turmas de alunos, classes sociais ou faixas de idade, os grupos também podem ser utilizados em sistemas distribuídos, onde os processos podem ser organizados em grupos e as mensagens, endereçadas a este grupo [POW96]. A comunicação entre os membros de um grupo de processos é chamada de comunicação de grupo ou de comunicação *multicast*.

A maioria das aplicações, que utiliza a comunicação de grupo, exige que a entrega das mensagens seja feita de forma confiável e ordenada, sendo recebida em todos os seus destinatários de forma correta e na ordem necessária, mesmo na

presença de falhas [MOS96]. Este é o caso da disseminação de escrita para arquivos replicados. Por este motivo, foi escolhida, e será discutida e analisada a utilização de primitivas de comunicação de grupos para a disseminação de escrita em arquivos replicados.

O objetivo deste trabalho é analisar a adequação da utilização de um protocolo de comunicação de grupo para a disseminação de operações de escrita em arquivos replicados, de forma a garantir a serialização dessas operações de escrita, mesmo na presença de falhas.

1.2 Seções

O capítulo 2 apresenta a conceituação respeito de replicação de arquivos e grupos de replicação. Apresenta também as abordagens de replicação da cópia primária e das cópias ativas. No final do capítulo são discutidos os problemas existentes na disseminação de operações de escrita em arquivos replicados.

O capítulo 3 trata da comunicação de grupos. São apresentados sistemas de comunicação de grupos desenvolvidos em ambiente acadêmico: xAMp, da Universidade de Lisboa; Totem, Universidade da Califórnia; Transis da Universidade de Hebréia de Jerusalém; Horus, da Universidade de Cornell e Newtop da Universidade de Newcastle.

O capítulo 4 descreve o protótipo para disseminação de escritas para arquivos replicados através de *multicast* – PDERM, desenvolvido para avaliar aspectos de atomicidade na disseminação de operações de escritas utilizando primitivas de comunicação de grupos.

O capítulo 5 apresenta as conclusões do trabalho, assim as possíveis continuações da pesquisa aqui iniciada.

O capítulo 6 mostra a bibliografia consultada

2 Replicação de arquivos

A replicação de arquivos é uma das técnicas mais utilizadas em tolerância a falhas. Tem como objetivo aumentar a disponibilidade dos dados, para que estes possam ser acessados mesmo após a ocorrência de alguma falha. Desta forma mesmo que uma cópia de determinado arquivo fique inacessível, outras poderão estar disponíveis, mascarando a falha ocorrida.

A utilização de replicação de dados para tolerar falhas, apesar de ser uma idéia intuitiva e de fácil compreensão, tem uma implementação bastante difícil [GUE97]. A utilização de vários servidores replicados indiscutivelmente aumenta a disponibilidade do dados contidos nesses servidores. A replicação, no entanto, introduz problemas de consistência e gerenciamento de réplicas [JAL94].

O principal desafio na implementação de arquivos replicados em vários servidores está na manutenção da serialização das operações, ou seja, o resultado da execução de determinadas operações em dados replicados deve ser equivalente a execução serial dessas ações em dados não replicados.

Neste capítulo são explicados os grupos de replicação e suas possíveis composições. Em seguida, são abordadas as duas classes fundamentais de técnicas de replicação de arquivos: método da cópia primária e método das cópias ativas. Por fim são discutidos os problemas e possíveis soluções para a disseminação de escritas em ambientes replicados.

2.1 Grupos de replicação

No momento em que se decide manter várias cópias de um arquivo para aumentar a sua disponibilidade, alguns aspectos quanto ao número e localização das cópias devem ser considerados. Pode-se optar por manter cópias do arquivo em todos os servidores disponíveis ou em apenas um conjunto de servidores. Normalmente esta decisão é baseada em aspectos como taxa de utilização do arquivo, capacidade dos servidores, abordagem de replicação escolhida e proximidade dos principais usuários do arquivo.

Um grupo de replicação representa o conjunto de servidores nos quais um arquivo, ou conjunto de arquivos, está replicado. A figura 2.1 mostra como exemplo um sistema com cinco servidores dos quais apenas três fazem parte do grupo de replicação de determinado arquivo.

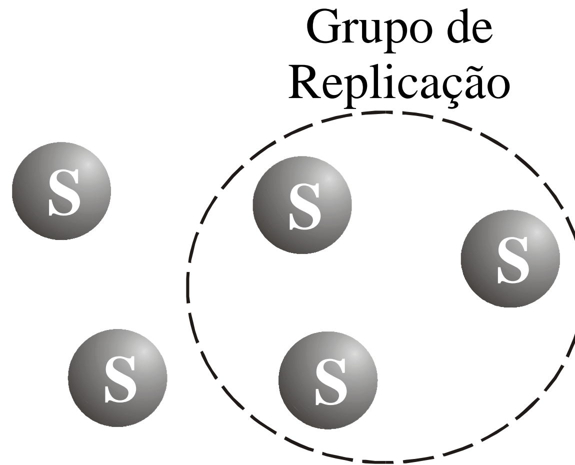

FIGURA 2.1 - Grupos de Replicação

2.2 Abordagem da cópia primária

Na abordagem de replicação com cópia primária, é considerada a existência de um servidor de dados primário e vários servidores secundários. Todos os clientes sabem qual é o servidor primário e comunicam-se apenas com este servidor. Isto traz a vantagem básica da serialização inerente, ou seja, em qualquer situação existirá uma ordem única em que as requisições de acesso são recebidas pelo servidor primário e repassadas às réplicas.

O servidor primário é responsável por irradiar para todas as réplicas cada procedimento de escrita, de forma síncrona (ou seja, um novo procedimento de escrita só começa quando o anterior tiver sido difundido para todas as réplicas), além de realizar estes procedimentos em sua cópia local. Os procedimentos apenas de leitura são realizados somente no contexto da cópia primária, e não necessitam ser irradiados. O passos seguidos nessa abordagem são os seguintes, conforme ilustra a figura 2.2:

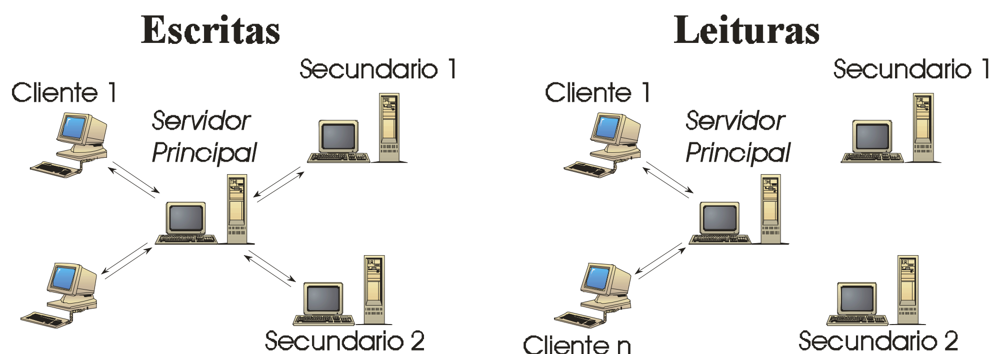

FIGURA 2.2 - Abordagem da cópia primária

a) cliente faz a requisição de serviço ao servidor primário;

b) se a requisição for de leitura, o servidor primário executa a operação e responde ao cliente;

c) se a requisição for de escrita, o servidor primário repassa a requisição para os servidores secundários, dos quais espera uma resposta. Quando receber as respostas esperadas, executa a operação de escrita e responde ao cliente.

Uma grande quantidade de trabalhos dedica-se ao método de cópia primária. O algoritmo básico está descrito em livros básicos da área ([TAN92], [JAL94]) e um estudo de seu desempenho é feito por Huang [HUA89].

Como por princípio todos os clientes conhecem qual servidor é o primário e comunicam-se apenas com este servidor, é necessário um algoritmo de mudança de servidor primário, que deve ser usado em caso de uma falha no servidor primário.

2.3 Abordagem das cópias ativas

Neste método de replicação, todas as réplicas de servidores são ativas e podem receber requisições de serviços dos clientes. A serialização nessa abordagem não é inerente, uma vez que as requisições de clientes não passam todas pelo mesmo servidor centralizador. Por essa razão, a garantia de serialização é mais complexa que na abordagem da cópia primária e é feita através do emprego de protocolos que garantam a serialização distribuída de todas as mensagens transmitidas dos clientes para os servidores. Os protocolos adequados para isso são, por exemplo, protocolos de difusão atômica.

Duas maneiras diferentes de garantir a consistência das múltiplas cópias na abordagem de replicação por cópias ativas podem ser citadas para ilustrar o problema: a abordagem da máquina de estado, de Shneider [SCH90] e a utilização de difusão atômica de Jalote [JAL89]. Ambas as abordagens utilizam a difusão (*broadcast*) de mensagens e consideram apenas um modelo *"fail-stop"* para os servidores, sem possíveis particionamentos na rede, e sem falhas nos clientes.

2.3.1 Máquina de estado

Nessa abordagem a requisição de um cliente é enviada para todos os servidores replicados através de uma mensagem *broadcast*. Todas as réplicas são equivalentes e executam as operações requisitadas. Os passos seguidos nessa abordagem são:

a) o cliente faz a requisição da operação para todos os servidores;

b) cada réplica processa a requisição, modifica o seu estado e responde ao cliente;

c) o cliente espera até receber a primeira resposta ou respostas iguais da maioria dos servidores.

Com a utilização de difusão confiável, é garantido que todos os servidores correntemente ativos, ou nenhum, receberão a mesma mensagem enviada por cada

cliente. Utilizando um protocolo para difusão de solicitações dos clientes para todos os servidores, e atribuindo para cada requisição um número único dentro de uma seqüência global fornecida por um relógio lógico executado entre os clientes, pode-se realizar perfeita serialização nos servidores, postergando operações recebidas "adiante" de outras já enviadas, mas ainda não recebidas.

2.3.2 Difusão atômica

Nessa abordagem, se uma operação é requisitada a um servidor replicado, todos as outras réplicas também devem executar a operação. Os passos seguidos em uma operação de escrita nessa abordagem são:

a) o cliente faz a requisição da operação para um servidor S_1;

b) o servidor S_1 transmite a requisição de operação para os demais servidores através de mensagem *broadcast* (ou *multicast*, se o arquivo estiver replicado em apenas alguns servidores);

c) cada réplica processa a requisição, modifica o seu estado e responde ao servidor S_1;

d) S_1 espera até receber a confirmação dos servidores e responde ao cliente.

A diferença básica em relação a abordagem anterior está no fato de que nesta abordagem o cliente faz a sua requisição a um servidor (o mais próximo, por exemplo) e este difunde a requisição, enquanto que na abordagem anterior a requisição era difundida aos servidores pelo próprio cliente.

2.4 Disseminação de escritas

Existem dois tipos de operações que podem ser requisitadas por um cliente a um servidor. Operações que não modificam o estado geral dos dados, chamadas de operações de leitura e operações que modificam o estado dos dados, as operações de escrita.

Em qualquer das abordagens descritas na seção anterior, operações de leitura podem ser executadas em apenas um dos servidores replicados, uma vez que não causam modificação nos dados acessados. Já as operações de escrita precisam ser executadas de maneira que seja garantido que a consistência dos dados será mantida.

A difusão de escritas em ambientes replicados, tanto através da abordagem de cópia primária quanto do método de cópias ativas, deve satisfazer as seguintes propriedades[GUE97]:

a) *ordenação*: requisições de operações feitas por clientes diferentes a um conjunto de servidores do grupo de replicação devem ser executadas em todos os servidores na mesma ordem;

b) *atomicidade*: se uma operação requisitada por um cliente for realizada em um servidor, deverá ser realizada em todos os outros servidores não falhos.

Na abordagem de cópia primária, como todos os clientes se comunicam sempre com o mesmo servidor, este funciona como um centralizador. Todas as mensagens de atualização passam por ele, o que garante a ordenação total naturalmente para todas as modificações dos dados.

Na abordagem de cópias ativas é necessário um protocolo que garanta a atomicidade e a ordenação, uma vez que as mensagens de atualização podem partir de vários pontos diferentes, não estando naturalmente ordenadas, ao contrário do que acontece na abordagem de cópia primária.

2.5 Pontos de falhas

Na abordagem de cópia primária, pode-se distinguir três pontos de falhas de um servidor primário [GUE97]:

a) antes de enviar a mensagem com a alteração requisitada às replicas;

b) depois de enviar a mensagem, mas antes de enviar a resposta ao cliente;

c) depois do cliente receber a resposta.

No terceiro caso, a falha é transparente ao cliente. No primeiro e no segundo caso, o cliente não receberá a resposta e poderá suspeitar da falha. O caso mais difícil de ser tratado é o segundo, onde algumas cópias podem ter recebido a mensagem e outras não.

2.6 Solução proposta

O sistema RNFS, sistema de arquivos distribuído e tolerante a falhas para UNIX [LEB96], que faz uso da replicação de dados como instrumento para tolerar falhas, implementa o método de replicação de cópia primária. A idéia original para a disseminação de escritas do RNFS era a utilização de RPCs na modalidade "*callback*". Nesta modalidade de RPC o servidor primário faz uma chamada remota aos secundários (com a operação de escrita a ser executada), que respondem imediatamente com uma mensagem nula. Desta forma o servidor primário prossegue suas operações e espera um RPC dos servidores secundários com o resultado das operações de escrita. Apesar do bom desempenho, esse método teria como desvantagens o grande número de mensagens e a complexidade do algoritmo, além de não levar em conta o aspecto da manutenção da atomicidade.

A disseminação de escritas implementada no RNFS funciona da seguinte forma: os servidores secundários são "montados" no servidor primário e as alterações

são feitas nesses diretórios montados, que serão copiados para os servidores secundários posteriormente.

A motivação para este trabalho foram as dificuldades relatadas no RNFS referente à manutenção da atomicidade na disseminação de escritas.

A proposta deste trabalho é analisar a possibilidade de utilização de um protocolo de multicast para a disseminação de operações de escrita na abordagem de cópia primária, garantindo, assim, a atomicidade nessa disseminação. Ao contrário do RNFS que usa RPC para manter coerência com o NFS, a solução proposta não se preocupa com nenhum sistema de arquivos particular. A única restrição é a disponibilidade de um mesmo serviço de comunicação de grupo em todos os nodos do grupo de replicação.

3 Comunicação de Grupo

Em sistemas distribuídos existem aplicações onde é necessário o envio de mensagens de um processo para vários outros processos. Nessas aplicações utiliza-se a chamada comunicação 'um para muitos'. Existem duas formas de comunicação desse tipo: *broadcast* e *multicast*. Na comunicação *broadcast*, o processo emissor envia mensagens para todos os outros processos do sistema, enquanto que na *multicast*, envia a mensagem para apenas um subconjunto de processos do sistema [JAL94].

A comunicação de grupo, ou comunicação *multicast*, tem sido um dos maiores alvos de estudos nos últimos anos dentro das pesquisas em sistemas distribuídos.

Este capítulo apresenta a conceituação básica sobre a grupos de *multicast*, algumas propriedades importantes na troca de mensagens *multicast*, assim como a classificação das falhas que podem ocorrer em sistemas distribuídos. Ainda neste capítulo são descritos alguns sistemas de comunicação de grupos desenvolvidos recentemente em ambientes acadêmicos.

3.1 Grupos de *multicast*

Segundo Garcia-Molina [GAR91], um grupo *multicast* é uma coleção de processos que são o destino da mesma seqüência de mensagens. Estas mensagens podem ser originárias de um ou mais nodos fonte e os processos destino podem estar executando em um ou mais nodos, não necessariamente distintos. Cada mensagem fonte é endereçada para o grupo *multicast* e o protocolo de comunicação deve garantir que as mensagens serão entregues aos processos apropriados.

Um *grupo* é uma abstração de endereçamento, utilizada para referenciar uma coleção de membros do grupo, e um membro do grupo é um ponto final de comunicação, que pode originar e entregar mensagens. Processos podem ser adicionados ao grupo, podem deixar o grupo, quando já cumpriram a sua função, ou podem ser retirados do grupo por causa de uma falha. Essas operações causam a modificação da visão do grupo e são, normalmente realizadas por um protocolo de *membership*. A visão de um grupo é uma fotografia do situação do grupo em um ponto específico na execução de um processo [REN95].

3.2 Propriedades de interesse na troca de mensagens

Segundo Pankaj Jalote [JAL94], quando uma mensagem é enviada de um processo para outros localizados em diferentes nodos do sistema, existem três propriedades de interesse: confiabilidade, ordenação consistente e preservação da causalidade.

A propriedade de confiabilidade exige que uma mensagem difundida seja recebida por todos os nodos ativos; a propriedade de ordenação consiste na entrega

das diferentes mensagens, originadas por diferentes nodos, na mesma ordem; e a propriedade de preservação da causalidade exige que a ordem na qual as mensagens foram enviadas aos nodos seja consistente com a causalidade entre os eventos de envio dessas mensagens.

Essas três propriedades resultam em três tipos diferentes de primitivas na entrega de mensagens *multicast*: *multicast* confiável, *multicast* atômico e *multicast* causal.

Um *multicast* confiável suporta apenas a propriedade de confiabilidade, ou seja, uma mensagem que foi enviada via *multicast* será entregue a todos os nodos ativos do grupo, mesmo se ocorrerem falhas no sistema. O *multicast* atômico, além da propriedade de confiabilidade, também suporta a propriedade de ordenação. O *multicast* causal garante que a ordem na qual as mensagens são entregues é consistente com a ordem que originou esta mensagens.

3.3 Falhas em sistemas distribuídos

As falhas que ocorrem em um sistema distribuído podem ser classificadas como [CAS86]:

Falhas de Colapso : componente pára de enviar mensagens;

Falhas de Omissão: o componente não responde a algumas entradas;

Falhas de Temporização: o componente responde ou muito cedo ou muito tarde;

Falhas Bizantinas ou Maliciosas: falhas que causam um comportamento arbitrário do componente

Conforme mostrado na figura 3.1, as falhas maliciosas englobam todas as outras falhas. Quanto mais interno for o círculo da figura, mais restrita é a falha.

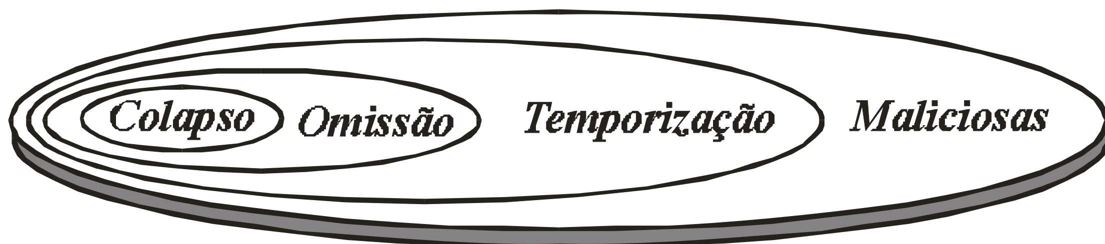

FIGURA 3.1 - Classificação de falhas

Quanto menos restritivo for o modelo de falhas utilizado por um sistema, maior será a complexidade deste sistema, pois maior será o número de situações distintas que deverá tratar.

3.4 Protocolos de comunicação de grupo

As seções que seguem descrevem sistemas que implementam protocolos de comunicação de grupo [HOF97]. Os sistemas oferecem protocolos para a troca de mensagens *multicast* de forma confiável e ordenada e também implementam protocolos de *membership*, cuja função básica é processar as mudanças de visão dos grupos.

Cada um dos sistemas apresentados tem características próprias e maneiras diferentes de garantir a entrega confiável e ordenada de mensagens *multicast*. Apesar de serem todos sistemas voltados para a comunicação em grupo, possuem enfoques e protocolos diferentes.

Os sistemas são recentes, sendo que os artigos utilizados como fonte de pesquisa possuem data superior a 1990 e todos foram desenvolvidos em ambiente acadêmico. O sistemas descritos são: xAMp, da Universidade de Lisboa; Newtop da Universidade de Newcastle; Transis da Universidade de Hebréia de Jerusalém; Horus, da Universidade de Cornell e Totem, da Universidade da Califórnia.

O sistema xAMp, primeiro a ser apresentado, está descrito de maneira mais completa e detalhada por ter sido escolhido para a fase experimental. Esta escolha se baseou, principalmente, na sua característica de flexibilidade uma vez que disponibiliza várias primitivas de comunicação de grupo, permitindo a escolha da primitiva a ser utilizada conforme as necessidades da aplicação. Outro ponto a favor do xAMp na escolha do protocolo para a experimentação foi o fato de estar disponível nos laboratórios da UFRGS.

3.4.1 xAMp

O xAMp (*eXtended Atomic Multicast Protocol*) é um serviço de comunicação de grupo com múltiplas primitivas, desenvolvido na Universidade de Lisboa por Luís Rodrigues e Paulo Veríssimo [ROD92a] do grupo INESC. Trata-se do seu predecessor AMp, primeiro suporte de comunicação do sistema DELTA-4, totalmente reprojetado [MUL93]. Tem como objetivo suportar o desenvolvimento de aplicações distribuídas com diferentes requisitos de dependabilidade, funcionalidade e desempenho.

O Serviço de Comunicação de Grupo xAMp é dividido em vários módulos [VOG92], um dos quais é específico para protocolos de comunicação de grupo. Os módulos do xAMp são:

LSE (Local Suport Environment) - ambiente independente para funções específicas, como alocação de memória, gerenciamento de *buffers*, etc;

AN (Abstract Network) - módulo que implementa todas as propriedades necessárias para suportar comunicação de grupo;

xAMp núcleo do protocolo, que implementa várias primitivas de comunicação *multicast*;

MGS (The Multicast Group of Stations protocol) - protocolo projetado para suportar *membership* e técnicas de endereçamento;

SYNC (The Clock Synchronization Service) - implementa algoritmos para obter relógios virtuais sincronizados, criando uma base de tempo global;

Dialog módulo de interface que permite o trabalho com interfaces padrões de comunicação UNIX.

O xAMp é executado sobre o módulo de rede abstrata (AN - *abstract network*) que oferece um serviço de *multicast* não confiável.

A seguir são explicados diversos protocolos do xAMp, também chamados de qualidades de serviço (protocolos que vão do *multicast* não confiável ao *multicast* seguro).

3.4.1.1 Transmissão com resposta

O procedimento de transmissão com resposta utiliza 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 [ROD92a].

A chamada da primitiva é feita da seguinte forma:

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 a ordenação na entrega das mensagens é relevante

send é um 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 é esperada

nr é o número de respostas esperadas.

OBS: normalmente **nr** = #**Pr** e **Pr** = **D(m)** sendo **D(m)** o conjunto de receptores

A primitiva realiza o seguinte: envia a mensagem através da rede (com os parâmetros acima descritos) e espera a resposta durante um intervalo de tempo predeterminado. Cada resposta que chega é inserida em uma bolsa de respostas. Se algumas respostas foram perdidas, a bolsa de respostas 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 essa ser reconhecida (*ack*) por todos os receptores em uma mesma transmissão. Quando a ordem não é necessária, a variável **ord** deve receber o valor “falso”. Neste caso, o procedimento pode ser otimizado de forma que a bolsa de respostas não necessite ser reinicializada entre o envio de duas mensagens.

3.4.1.2 Best-effort agreement

A primitiva tr-w-resp é utilizado no xAMp para proporcionar a entrega (*delivery*) confiável de frames (mensagem ou informações de controle do protocolo). O tr-w-resp é ativado pelo emissor da mensagem e este emissor 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 apenas uma chamada ao tr-w-resp. A escolha apropriada de **Pr** (conjunto de processadores dos quais a resposta é esperada) e **nr** (número de respostas esperadas) permite um retorno rápido em caso de omissão, quando nem todos os receptores endereçados necessitam receber a mensagem. Por exemplo quando o **nr** = 0, o procedimento imediatamente termina após o envio da mensagem sem esperar por respostas. Isso corresponde a um *multicast* não confiável.

3.4.1.3 AtLeast agreement

A primitiva *bestEffort* não é capaz de garantir a entrega de mensagens no caso de falha do emissor. Por isso, com a primitiva *AtLeast*, todos os receptores são responsáveis pela terminação do protocolo. Em conseqüência, o procedimento tr-w-resp é invocado tanto pelo emissor quanto pelos receptores. No entanto, para evitar retransmissões desnecessárias de mensagens, os receptores não executam a primeira etapa do procedimento tr-w-resp, utilizando o parâmetro booleano **send**. No caso da não ocorrência de falhas, a mensagem será reconhecida por todos os receptores destino, este reconhecimento será visto por todos os participantes e não será necessária retransmissão.

Como na primitiva *bestEffort*, algumas variantes podem ser obtidas pela escolha apropriada dos parâmetros **Pr** e **nr**. Por exemplo, se for escolhido **nr** de forma que **nr** seja menor do que o tamanho de **D(m)** (conjunto dos receptores), a primitiva vai assegurar que no mínimo **nr** dos processos endereçados receberão a mensagem. É interessante para protocolos baseados em quorum. Quando **Pr** = **D(m)** (ou seja, quando todos os membros do grupo recebem a mensagem), a primitiva é

também chamada de *multicast* confiável. Outras duas primitivas (causal e atômica) são baseadas nesta primitiva de *multicast* confiável.

As duas primitivas descritas acima auxiliam em determinadas aplicações distribuídas, onde funcionalidade de alto nível reduz os requisitos de ordenação e concordância, mas a necessidade da disseminação eficiente em grupo é mantida.

3.4.1.4 Multicast Causal

As primitivas de *multicast* confiável não impõem nenhuma condição de ordenação na troca de mensagens. No entanto, em alguns sistemas a ordem na qual as mensagens são entregues é relevante.

No xAMp a ordenação causal (ou ordenação lógica) é obtida com o uso de históricos de causalidade, ou seja, mantendo uma cópia das mensagens enviadas e recebidas entre processos e trocando estas informações junto com a mensagem 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 leva o histórico causal do seu emissor. Este histórico causal é modificado toda vez que uma mensagem é enviada ou entregue (*delivered*). 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 são adicionados ao histórico causal do receptor.

Para ser entregue, uma mensagem tem que se tornar estável. Uma determinada mensagem m é estável, em um dado processo k, se para todas as mensagens precedentes a m, o *flag* **entregue** já é verdadeiro em k. Para prevenir o crescimento infinito dos históricos causais, periodicamente são removidos identificadores estáveis. O tempo entre uma remoção e outra eqüivale ao tempo necessário entre o envio de uma mensagem e a sua entrega.

3.4.1.5 Multicast atômico

A propriedade de atomicidade garante a ordenação total de mensagens. Para preservar a ordem da rede, um mecanismo precisa garantir que as mensagens sejam entregues aos usuários respeitando a ordem em que foram colocadas na rede, e que se uma mensagem for retransmitida, apenas uma transmissão é usada para estabelecer esta ordem.

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 movida para o fim da fila, como se tivesse acabado de chegar (já que as mensagens são inseridas na fila por ordem de chegada).

O emissor invoca o procedimento tr-w-resp ativando o *flag* **ord**, requisitando assim a retransmissão da mensagem até que todos os receptores reconheçam a mesma tentativa. Quando for detectada uma transmissão de sucesso, onde todos os receptores confirmaram o recebimento da mensagem, o emissor emite um *frame* de aceitação, fazendo o *commit* da mensagem. Quando o *frame* de aceitação é recebido, os receptores marcam a mensagem associada como aceita e a entregam assim que ela alcançar o topo da fila.

Se o receptor não pode processar a mensagem, por motivos como falta de recursos ou espaço no *buffer*, por exemplo, ele notifica o emissor através de uma mensagem de reconhecimento não-ok. Com isso o emissor envia uma mensagem de rejeição de uma mensagem aceita 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 da decisão) o emissor decide enviar uma mensagem de aceitação ou de rejeição. 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 mecanismo de *timeout* e enviar um frame de requisito de decisão (Request-Decision). Usando este esquema, uma segunda rodada de reconhecimentos é evitada, aumentando o desempenho do protocolo.

Desde que em ambas as fases é o emissor que coordena o protocolo, é necessário um mecanismo para mascarar as falhas do emissor. No serviço atômico, estas falhas são suportadas com a utilização de um protocolo de terminação. Este protocolo é executado por uma função de monitoramento. Não existe um monitoramento permanente, só quando for necessário. O monitor inspeciona o emissor falho, coleta a informação sobre o estado da transmissão e dissemina a decisão a ser tomada (aceita ou rejeitada).

3.4.1.6 Protocolo de membership

O protocolo de *membership* do xAMp foi implementado na camada mais baixa do sistema de comunicação xAMp [ROD92b]. Isso permite que sejam designados endereços reduzidos (*short adresses*), compostos pela identificação compacta das estações, a cada nodo do sistema.

As ações que são destinadas a modificar a visão de grupo precisam sem realizadas através de funções especiais. As operações que podem ser realizadas em um grupo são: *join*, *leave*, *delete* e *check*.

Os principais objetivos do protocolo de *membership* são: manter uma lista completa e atualizada do grupo, chamada de MGS (*Multicast Group of Stations*) e manter uma função de mapeamento que traduz a identificação única dos nodos em endereços reduzidos. Este mapeamento é universal e estável (em todos os nodos o mesmo endereço reduzido corresponde à mesma estação e essa correspondência permanece durante toda a vida do sistema).O funcionamento do protocolo de *membership* do xAMp, o MGS, é o seguinte:

As funções mais complexas no protocolo são as que executam as operações de *join* e de *leave*. Para evitar o uso inconsistente de visões de grupo, essas operações precisam ser executadas como ações atômicas. No xAMp, durante um *join* ou um *leave* em determinado grupo, todos os componentes deste grupo ficam inacessíveis.

O protocolo de *membership* MGS mantém em cada membro do grupo uma réplica da tabela de estado global do MGS. Para que seja mantida a consistência das cópias, todas as alterações das tabelas são executadas de forma atômica, através de um mecanismo de *lock* distribuído. Antes de executar qualquer mudança na tabela de estado, um nodo precisa obter o *lock* e a cada instante apenas um nodo pode ter o *lock*.

O protocolo é composto por três atividades cooperantes: *guardian*, *changer* e *accepter*.

Guardian: responsável pela manutenção da cópia local da tabela de estado do MGS. Existe um guardião executando em cada nodo.

Changer: ativado em operações de *join*, *leave*, *delete* ou *check*, é o responsável pelas modificações na tabela de estado global do MGS.

Accepter: assegura a terminação apropriada em caso de falha do *changer*. É ativado quando uma cópia da tabela de MGS é modificada.

Na solicitação de uma operação os seguintes passos são seguidos:

1. O protocolo é iniciado pelo *changer,* que envia uma mensagem GetState para todos os *guardians*;

2. Cada *guardian,* após o recebimento do GetState, e se não houver outro *changer* com o *lock*, ativa a variável booleana *lock*, armazena o identificador do *changer* e envia uma mensagem MyState, contendo uma cópia da tabela de estado local. Se houver outro *changer* com *lock*, envia ao *changer* requisitante um *ack* negativo;

3. O *changer* espera todos os *acks*

a) se um *ack* negativo foi recebido de qualquer membro do grupo, significa que outro *changer* conseguiu o *lock*

b) se respostas foram perdidas, o GetState é retransmitido

c) se todos os membros responderam afirmativamente, o *changer* faz as modificações necessárias e dissemina o novo estado com uma mensagem NewState;

4. Quando um *guardian* recebe um NewState, modifica a sua cópia local, desliga a variável *lock,* inicializa um *accepter* e dissemina uma mensagem de RcState;

5. O *changer* (e os *accepters*) recebe as mensagens RcState. Se faltar alguma, a mensagem NewState é enviada até que todos os membros do grupo a reconheçam.

No MGS, a manipulação de estações falhas é feita da seguinte forma: falhas em um membro do grupo são apenas detectadas quando uma atividade *changer* é ativada. Existe uma operação especial que permite ativar um *changer* apenas para propósitos de verificação do grupo. Se o *changer* falha, o protocolo garante que todas as cópias das tabelas de estado estão em um estado consistente. O caso mais simples de detecção de falhas ocorre durante a execução do procedimento tr-w-resp. Se a ausência de reconhecimento de um membro do grupo persistir após o tempo estipulado na rede, então a falha do nodo é assumida. Se a falha é detectada por um *changer* antes da propagação do novo estado, esta informação será disseminada com a nova tabela. Senão, se detectada por um *changer* ou por um *accepter* durante a seqüência de propagação, o *changer* deve ser ativado novamente.

Um cenário mais elaborado ocorre quando um *changer* falha. Neste caso, a recuperação depende do estado do *changer* no instante da falha. Se o *changer* estiver com o *lock* e falhar antes de propagar o novo estado, a falha é detectada pelo *guardian* quando o tempo de *lock* terminar. Se o *changer* falhar durante a disseminação do novo estado, o protocolo de terminação é garantido pelo *accepter*. No último caso a falha do nodo será detectada pelo *accepter* durante a execução do procedimento tr-w-resp. Existe ainda uma primitiva *check* que permite que o *membership* seja verificado quando existe a suspeita de uma falha. Essa primitiva pode também ser executada periodicamente no sistema.

3.4.1.7 Endereçamento de grupos e detecção de falhas

Não existem restrições quanto ao conjunto de processos receptores das mensagens, ou grupo para o qual a mensagem deve ser enviada. O protocolo é muito genérico e aceita qualquer conjunto de processos como destino.

No xAMp, falhas são detectadas durante a troca de mensagens: um emissor detecta falhas de receptores durante a execução do procedimento tr-w-resp. No serviço atômico, falhas no emissor são detectadas por *timeouts*. Na detecção de uma falha, a identificação do nodo falho é imediatamente disseminado para todos os nodos do grupo. Uma mensagem atômica especial é enviada para cada grupo cujo

membership foi afetado pela falha, proporcionando ao usuário uma indicação de falha.

3.4.2 Newtop

O Newtop (NEWcastle Total Order Protocol) foi desenvolvido na Universidade de Newcastle. Trata-se de um protocolo de comunicação de grupo de propósitos gerais dinâmico e tolerante a falhas [MAC94]. Tem como objetivo preservar a ordenação na entrega de mensagens e a continuidade do sistema, mesmo na ocorrência de falhas.

O Newtop parte dos seguintes pressupostos:

1. processos podem pertencer simultaneamente a muitos grupos;
2. o tamanho dos grupos pode ser grande;
3. processos podem estar separados geograficamente se comunicando através da Internet;
4. o ambiente de comunicação é assíncrono, onde os tempos de transmissão não podem ser precisamente estimados;
5. o modelo de falhas considerado é *fail-stop*;
6. falhas de comunicação podem causar particionamento de rede, fazendo com que membros de um grupo possam ficar em partições diferentes e
7. existe uma camada de transporte que permite a transmissão de mensagens de forma seqüenciada e não corrompida.

Uma das principais características do Newtop é a de permitir a sobreposição de grupos. Como é mostrado na figura 3.2 (a), os processos P1 e P2 pertencem ao grupo g1 e também pertencem ao grupo g2 que é formado por P1, P2 e P3 (figura 3.2 (b)), de forma que os processos P1 e P2 pertencem a dois grupos simultaneamente.

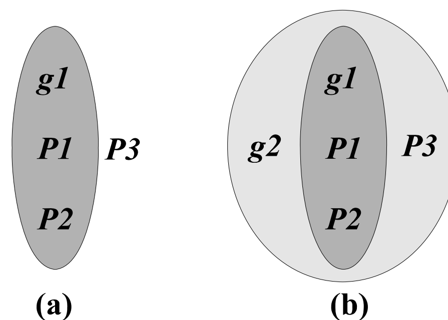

FIGURA 3.2 - Sobreposição de grupos

Em ambientes que suportam replicação de arquivos, a sobreposição de grupos é bastante útil uma vez que um servidor de arquivos pode fazer parte de mais de um grupo de replicação.

No Newtop, uma nova visão deverá sempre ser um subconjunto da visão antiga, uma vez que processos não podem entrar em um grupo do qual saíram. Em vez de um processo entrar em um grupo já existente, ele deverá formar um novo grupo, convidando os membros do grupo que lhe interessam. Isso eqüivale a entrar em um mesmo grupo com outro identificador.

O protocolo de *membership* do Newtop mantém a consistência da visão do grupo na presença de partições. Esta consistência é mantida de forma que:

- os processos não falhos dentro de um subgrupo (partição) devem ter a mesma visão do seu subgrupo;
- as visões de processos pertencentes a subgrupos diferentes , no momento em que se tornarem estáveis, não se interseccionarão, ou seja, a visão de um processo pertencente a um sub-grupo não conterá um processo pertencente a outro.

Quando um grupo particiona em subgrupos, os membros de cada subgrupo se consideram os únicos membros sobreviventes do grupo original, e não terão conhecimento da existência de outros subgrupos. O Newtop deixa a aplicação decidir quando deve ou não manter mais de um subgrupo. Essa flexibilidade torna o Newtop diferente dos protocolos de partição primária que garante a continuidade das operações apenas em uma partição (a primária).

Os três tópicos que seguem (ordenação total simétrica e ordenação total assimétrica) apresentam propriedades e primitivas de comunicação de dados atômicas do Newtop. São mostradas para fins de comparação com o xAMp e por serem necessárias em caso de utilização para replicação de arquivos pela abordagem das réplicas ativas. A sua leitura, portanto, não é essencial para a compreensão dessa dissertação, que tem como prioridade os aspectos de replicação da abordagem de cópias primária.

3.4.2.1 Ordenação total simétrica

Cada processo mantém um contador para as emissões e um para as recepções de mensagens. Mantém também um vetor chamado de vetor de recepção, com uma entrada para cada processo de sua visão, onde serão armazenados os valores do contador da última mensagem recebida pelo processo correspondente a entrada do vetor. Cada processo também possui um relógio lógico, utilizado para numerar as mensagens. Considerando-se uma variável **Dx** que representa o valor mínimo no vetor, Mensagens provindas de um processo são enviadas com números crescentes e recebidas na ordem FIFO, então o **Dx,i** de um processo P_i será menor ou igual ao relógio lógico de todos os outros processos P_j pertencentes a sua visão. É garantido,

assim, que P_i não receberá qualquer nova mensagem cujo contador for menor de **Dx,i**. As duas condições para entrega de uma mensagem **m** em P_i são:

1. uma mensagem **m** se torna entregável (*deliverable)* se seu contador for menor ou igual a **Dx,i**

2. as mensagens que podem ser entregues são entregues na ordem crescente de seus números, uma ordem fixa predeterminada é imposta para mensagens de número igual

Estas duas condições garantem que as mensagens recebidas serão entregues na ordem total.

Os procedimentos acima são adotados para o caso de processos pertencerem a um único grupo. Quando ocorre a sobreposição de grupos, cada processo mantém um vetor distinto para cada visão a qual pertence. **Di** representa o mínimo valor de todos o **Dx, i** dos grupos. Desta forma, para a entrega de uma mensagem **m**, é somente necessário modificar a primeira das duas condições descritas acima:

1. uma mensagem m se torna entregável se seu contador for menor ou igual a **Di.**

3.4.2.2 Ordenação total assimétrica

A versão assimétrica do Newtop utiliza um dos membros do grupo como um seqüenciador para ordenar mensagens. A idéia básica, para um grupo único, provinda do protocolo de Chang, foi estendida para a sobreposição de grupos.

Considerando-se que um processo $\mathbf{P_i}$ pertença a apenas um grupo: para enviar uma mensagem de *multicast*, primeiramente o processo envia a mensagem apenas para o seqüenciador, que numera a mensagem e a envia para todos os membros do grupo, inclusive para o emissor. O seqüenciador envia todas as mensagens que recebeu na ordem de recepção e os demais processos do grupo entregam a mensagem na ordem que receberam do seqüenciador.

Se $\mathbf{P_i}$ pertencer a mais de um grupo, necessita adiar o envio de sua mensagem **m** para o seqüenciador até que receba as mensagens que enviou anteriormente para este mesmo seqüenciador. Isso garante que o número dado à mensagem por $\mathbf{P_i}$ será maior que o número dado pelo seqüenciador à mensagem anterior. Assim, mensagens consecutivas disseminadas por $\mathbf{P_i}$ em grupos diferentes serão com certeza enviadas pelos respectivos seqüenciadores com números crescentes.

3.4.2.4 Membership

O processo de visão de grupo GV do processo P_i (GV_i) funciona da mesma forma no caso de P_i pertencer a um ou mais grupos[MAC94].

O GV_i (*group view*) utiliza uma primitiva de comunicação chamada *mcast(m)* para transmitir sua mensagem para todos os processos do seu grupo e as mensagens são entregues para os processos destino pela camada de transporte na ordem de envio. O GV_i possui um módulo chamado S_i, encarregado de suspeitar de falhas, que monitora a execução de todos os processos do grupo. Se S_i observar que nenhuma mensagem de *multicast* foi recebida de um determinado processo do grupo durante um determinado período de tempo, então ele suspeita a falha desse processo e notifica GV_i.

O algoritmo de GV possui dois componentes: concordância de *membership* e instalação de visão.

Concordância de *membership*

Uma notificação de S_i para GV_i terá o seguinte formato [P_k, **ln**], indicando a suspeita de que P_k esta em falha e o número (**ln**) da última mensagem de P_k recebida por P_i. GV_i mantém um conjunto de suspeitas onde as notificações de S_i são armazenadas. GV_i envia via *multicast* a mensagem de suspeita (**i**, ***suspect***, [P_k, **ln**]) para o processo GV de todos os processos da visão corrente. GV_i mantém as mensagens oriundas de P_k como pendentes para o caso da suspeita ser refugada. Neste caso, as mensagens pendentes serão encaradas como mensagens que acabaram de chegar, e serão manipuladas de forma apropriada. Se GV_i receber a confirmação que todos outros membros não suspeitos da visão também suspeitam de P_k, então P_k é adicionado a um conjunto chamado de $failed_i$ e P_i descarta qualquer mensagem recebida de P_k e GV_k. Quando GV_i confirma todas as suas suspeitas, ele modifica a variável booleana *consensus* para verdadeiro, e isto leva à instalação de uma nova visão, sem P_k.

Instalação de visão

O componente de instalação de visão utiliza a primitiva *updadeview*(**F,N**), que quando invocada será executada assincronamente e instalará uma nova visão antes de qualquer mensagem **m** (cujo contador **m.c** for maior do que **N**) ser entregue a **Pi**, ou seja antes da visão ser modificada, **Pi** deve entregar a última mensagem provinda do nodo falho, mas a nova visão será instalada antes da entrega das mensagens de Pi subsequentes.

3.4.2.5 Protocolo de Formação de Grupo

A formação de um novo grupo pode ser iniciada por qualquer processo. O processo **Pi** que iniciara um novo grupo deverá ter os nomes dos membros do novo

grupo e não pode ser membro de um grupo com a mesma formação do grupo que deseja criar. Funcionamento do protocolo:

1. **Pi**, que deseja formar um novo grupo, envia uma mensagem de formação de grupo para cada membro que deverá participar do novo grupo, os convidando para o novo grupo.

2. Quando um processo **Pj** recebe um convite para formar um grupo, difunde essa mensagem para cada um dos membros desejados no grupo, enviando na carona (*piggiback*) sua decisão.

3. Uma mensagem de 'não' age como um veto; **Pi** envia sua mensagem de 'sim' se recebeu 'sim' de todos os outros dentro de um determinado período de tempo, senão envia 'não'

4. Se um processo **Pk** receber uma mensagem de 'sim' de todos os membros do novo grupo, a visão para o novo grupo é inicializada. A primeira mensagem que **Pk** envia no novo grupo é uma mensagem especial, chamada de *start-group,* que contém um campo chamado de *start-number,* que recebe o valor do contador da mensagem.

5. **Pk** espera receber uma mensagem de *start-group* de cada **Pj** na sua visão corrente. Uma vez que todas as mensagens de *start-group* forem recebidas, mensagens normais poderão ser enviadas e recebidas no grupo.

3.4.3 Transis

O Transis, desenvolvido na Universidade Hebréia de Jerusalém, é um serviço de comunicação de grupo que tem como objetivo simplificar o desenvolvimento de aplicações distribuídas tolerantes a falhas. Possui uma abordagem substancialmente diferente de outros protocolos, tratando de partições de rede e provendo ferramentas para recuperá-las [DOL96]. O Transis foi projetado para suportar o que chama de operações particionáveis, nas quais múltiplos componentes da rede estão desconectados uns dos outros e operam de forma autônoma.

3.4.3.1 Operações particionáveis

A abordagem do Transis se distingue por permitir operações particionáveis e no suporte de um reagrupamento consistente após a recuperação. É assumido que a rede pode particionar e são pesquisadas semânticas que proporcionem à aplicação as informações acuradas dentro de cada elemento particionado, para que todas as partições possam continuar operacionais (sem a existência de uma partição primária).

As operações particionáveis tem como vantagem a alta disponibilidade de serviços, uma vez que não existe apenas um componente ativo. A manutenção de

múltiplos componentes ativos, no entanto, pode permitir a ocorrência de operações inconsistentes em partes diferentes do sistema.

Em algumas aplicações é mais importante para os usuários uma alta disponibilidade do que uma consistência absoluta. Por isso o Transis proporciona flexibilidade para construção de diversas aplicações tolerantes a falhas, algumas das quais se beneficiam de operações particionadas contínuas, outras não. Para o último caso, é possível prevenir a coexistência de múltiplos componentes particionados.

Algumas idéias da abordagem de operações particionáveis foram incorporadas por outros sistemas como Horus e Totem.

Depois de ocorrer um particionamento de rede, o ideal seria que houvesse pelo menos um componente capaz de fazer modificações, que todos as máquinas soubessem o estado das outras antes do particionamento e que durante a recuperação apenas as mensagens perdidas necessitassem ser reenviadas para levar as máquinas a um estado consistente. Como normalmente isso não acontece, é preciso operar com certa margem de incerteza, o que não significa necessariamente uma situação caótica.

Para as aplicações para as quais não é necessário que as operações continuem exclusivamente em um componente primário, o Transis permite que todas as partes da rede particionada continuem operando separadamente, e durante a recuperação, reagrupem. A difusão simples e não ordenada de mensagens entres componentes previamente desconectados é realizada depois que as partições são recuperadas, utilizando o que o autor chama de “mexerico”, onde as duas partições “conversam” sobre as modificações feitas em cada uma.

Existem, no entanto, muitas aplicações que permitem modificações apenas dentro de um componente primário. Para esse tipo de aplicações, a abordagem do Transis proporciona suporte a recuperação de um componente primário, se este for perdido. Os membros em todos os componentes podem se tornar operacionais e esperar para reagrupar com o componente primário ou gerar um novo primário, se este foi perdido. A dificuldade na recuperação de um componente primário é que as máquinas recuperadas podem estar em uma situação simétrica depois da recuperação. O Tansis assiste o desenvolvimento das aplicações nessa situação reportando visões escondidas e assim quebrando a aparente simetria. O exemplo a seguir explica a situação de simetria e como esta pode ser evitada utilizando visões escondidas:

Supondo que existam três máquinas no sistema, **A**, **B** e **C**, e que o componente (sub-grupo) que tiver duas ou mais máquinas é considerado primário: **A**, **B** e **C** iniciam como um componente conectado {**A**, **B**, **C**}. Depois, C desconecta de { **A**, B}. **A** reporta a visão local { **A**, **B** }, mas **B** desconecta de **A** antes de **B** reportar { **A**, B}. Em seguida, **B** reconecta com **C**, e **C** reporta sua visão {**B**, **C**}, **B** falha e **C** conecta com **A**. Considerando-se este cenário sem a visão escondida { **A**, **B**}, reportada por **B**, **A** e **C** podem estar em um estado simétrico, pois o histórico de modificações de visões de **A** contém { **A**, **B**, **C**}, { **A**, **B**} e o de **C** contém { **A**, **B**, **C**}, {**B**, **C**}. A menos que **B** tenha passado para **C** a informação sobre a possível visão escondida {**A**, **B**}, **A** e **C** não podem determinar qual deles tem o estado mais completo, uma vez que **B** está em falha. Por outro lado, se **B** realmente informou a

visão escondida {**A**, **B**}, então pode levar essa informação para a próximo componente conectado {**B**, **C**}.

As operações particionáveis, que permitem que duas partições de um grupo continuem operantes, podem ser úteis em um ambiente de replicação de arquivos em casos onde os aspecto da disponibilidade dos dados é mais importante do que a total confiança de que estes dados estão íntegros. Tais operações, no entanto, são mais adequada à abordagem de replicação das cópias ativas, uma vez que o usual é todos os componentes replicados estarem disponíveis para a requisição dos clientes. Após a reconexão das partições, todos os processos conversam entre si para chegar a um consenso sobre o estado atual das réplicas.

Em um ambiente replicado através da abordagem de cópia primária, operações particionáveis também podem ser utilizadas caso seja interessante que a partição que não está com o servidor primário continue ativa. Neste caso, primeiro deve ser escolhido um novo primário na partição e a decisão deve ser comunicada aos clientes. Logo após a reconexão, apenas os servidores primários de cada partição precisam chegar a um consenso sobre o estado dos arquivos replicados. Neste momento, também, deve ser escolhido um primário entre os dois.

3.4.3.2 Serviço de comunicação de grupo

O Transis proporciona um serviço de comunicação *multicast* no nível de transporte e está localizado entre as aplicações usuárias e a camada de rede. Trata-se de um gerenciador de mensagens e visões de grupo. Cada membro do grupo mantém uma visão local dos participantes da rede conectados e operacionais. Essa visão possui um certo tempo de vida, que inicia quando no momento em que é reportada para a aplicação e termina quando é modificada. Existe também o conceito de visões escondidas, que tem o mesmo formato das visões regulares mas que indicam ao usuário que a visão está falha.

Uma vez que a detecção de falhas em sistemas realísticos geralmente não é confiável, as visões locais podem não ser sempre verdadeiras. A cada instante do tempo as visões locais nas diferentes maquinas podem ser idênticas. No entanto a importância das visões locais assim como das visões escondidas está nas suas inter-relação nas diferentes máquinas.

Dentro do espaço de tempo de vida de cada visão local, o módulo faz a entrega de suas mensagens de *multicast*.

Vários tipos de serviços de comunicação *multicast* são suportados pelo Transis:

multicast FIFO: garante ordem de entrega FIFO baseada no emissor

multicast causal: preserva a ordem causal entre as mensagens

multicast com acordo: força uma ordenação única entre todos os pares de mensagens em todas as destinações

multicast seguro: garante ordenação de mensagens e além disso atrasa a entrega de mensagens até a mensagem ser reconhecida por todas as máquinas do grupo, garantindo atomicidade no caso de falhas de comunicação.

Para o modelo de replicação adotado nesta dissertação apenas FIFO e seguro são aplicados.

As entregas das mensagens e as informações de mudança de visão são coordenadas pelo serviço de grupo. Este principio é chamado sincronismo virtual e é estendido no Transis para ambientes particionáveis. O sincronismo virtual garante que uma visão local reportada para qualquer membro é reportada para qualquer outro membro a não ser que esse falhe. No caso de partições, é garantido esse mesmo comportamento dentro de cada componente isolado e comportamento coerente no momento em que os componentes reagruparem.

O Transis trabalha com um conceito de *clusters* de *multicast* [MAL95], utilizando um domínio de broadcast hierárquico, onde cada *cluster* representa um domínio de máquinas com comunicação via broadcast em hardware ou *multicast* seletivo em hardware. Um *cluster* pode estar dentro de um única LAN ou em múltiplas LANs conectadas por *gateways*. Os *clusters* são organizados em uma estrutura de grupo hierárquica e cada nível da hierarquia é um domínio de grupo que mantém o serviço de grupo internamente.

A seguir serão explicados os protocolos de *multicast* confiável, causal e com acordo, sendo o último implementado em um subsistema do Transis, com o objetivo de proporcionar ordenação total na entrega de mensagens.

A leitura dos itens que seguem não é necessária para a compreensão dessa dissertação.

3.4.3.3 Multicast confiável

O transporte confiável dentro de uma LAN, utiliza o hardware de broadcast disponível para a disseminação de mensagens em uma única transmissão. As mensagens de reconhecimento são enviadas ‘na carona’ (*piggyback*) de mensagens regulares e também são difundidas uma única vez. As mensagens formam uma corrente de reconhecimentos. As máquinas em um cluster de *multicast* podem detectar mensagens perdidas analisando as correntes das mensagens recebidas. Neste caso a máquina emite uma mensagem de reconhecimento negativo para que a mensagem perdida seja retransmitida.

3.4.3.4 Multicast causal

Como visto anteriormente, no Transis cada mensagem emitida contém reconhecimentos das mensagens anteriores. Esses reconhecimentos formam uma relação causal direta, uma vez que se **m'** contém um reconhecimento para **m**, então m causou **m'**. No caso de uma mensagem chegar a uma máquina e alguma de suas predecessoras estar perdida, esta mensagem não poderá ser entregue até a mensagem perdida ser recuperada. Desta forma, a entrega das mensagens é feita na ordem causal.

3.4.3.5 Multicast com acordo

Este serviço do Transis é implementado pelo protocolo ToTo, também desenvolvido na Universidade Hebréia de Jerusalém. Esse serviço de *multicast* é voltado para sistemas distribuídos e suporta um alto nível de coordenação entre grupos de processos em aplicações distribuídas.

O serviço de *multicast* com acordo estende a ordenação causal, explicada anteriormente, para a ordenação total. Parte do princípio de que todas as máquinas do grupo precisam concordar com uma ordem total única para a entrega das mensagens, apesar do fato de que mensagens podem ter sido transmitidas concorrentemente de diferentes *sites*, e podem levar tempos arbitrários para chegar às suas destinações.

Nessa abordagem, uma mensagem de *multicast* com acordo pode ser enviada espontaneamente por qualquer máquina, sem que seja forçada a ordenação e sem a intervenção de qualquer coordenador central. Essa mensagem é recebida pelos outros participantes do grupo, e é mantida até que possa ser entregue. Uma mensagem pode ser entregue por determinada máquina, quando tiver recebido das outras máquinas o reconhecimento (*ack*) da mensagem. Assim, cada máquina segue os seguintes passos para efetuar a entrega das mensagens:

1. espera por uma mensagem de cada máquina do grupo

2. entrega o conjunto de mensagens que não segue qualquer ordem causal, na ordem lexicográfica

Esse procedimento pode ser otimizado, uma vez que a espera pelo *ack* de todas as máquinas do grupo pode tornar o sistema extremamente lento. A otimização consiste na formação da ordem total durante a espera de mensagens apenas de um conjunto majoritário de máquinas (as que não estão neste conjunto, no entanto, também seguem a ordem total "decidida" pela maioria).

O protocolo ToTo é completamente assíncrono e pode operar em um ambiente dinâmico utilizando o serviço de *membership* do Transis.

3.4.3.6 Protocolo de membership

O Transis possui também um serviço de *membership*, que mantém uma visão consistente do chamado conjunto de configuração corrente (*current configuration set* - CCS), de todas as máquinas conectadas em um ambiente dinâmico.

O serviço de *membership* utiliza mensagens especiais que são enviadas juntamente com as mensagens regulares do sistema, que proporcionam informações sobre mudanças no conjunto e configuração corrente CCS.

Quando máquinas falham ou desconectam, a rede particiona ou reagrupa, as máquinas conectadas precisam reconfigurar e encontrar um novo acordo para o CCS. Falhas e desconexões são detectadas baseadas em timeouts. A adição de novas máquinas é espontâneo e é disparado no momento em que as máquinas são conectadas.

Na execução do protocolo de *membership* são seguidos os seguintes passos:

1. Início: inicia o procedimento de modificação de configuração

2. Acordo: todas as máquinas conectadas concordam com a modificação de configuração a ser feita

3. Flush: depois de todas as mensagens de acordo do passo anterior serem entregues, é gerada uma mensagem de mudança de configuração para ser entregue à aplicação.

3.4.4 Horus

O sistema Horus, desenvolvido na Universidade de Cornell, consiste de uma arquitetura de comunicação, que tem como objetivo proporcionar um modelo de comunicação de grupo flexível para desenvolvedores de aplicações [REN96]. Foi projetado para ser um sistema de comunicação de grupo portável, com poucas dependências de sistema, e que pode ser incorporado em sistemas operacionais distribuídos modernos como um serviço a nível de usuário ou como um subsistema a nível de *kernel*, ou ambos. O projeto do sistema integra idéias desenvolvida no Isis (clássico predecessor do Horus), Transis (explicado na seção anterior) e x-*kernel*.

O sistema Horus é menor que o seu predecessor Isis, proporciona maior flexibilidade e é mais rápido. Além disso oferece características de segurança, e pode tratar particionamentos de rede.

No Horus um protocolo é tratado como um tipo abstrato de dado. Camadas de protocolos podem ser empilhadas uma no topo de outra de várias formas,

em tempo de execução. A figura 3.3 mostra várias camadas de protocolos empilhadas como “tijolinhos Lego”.

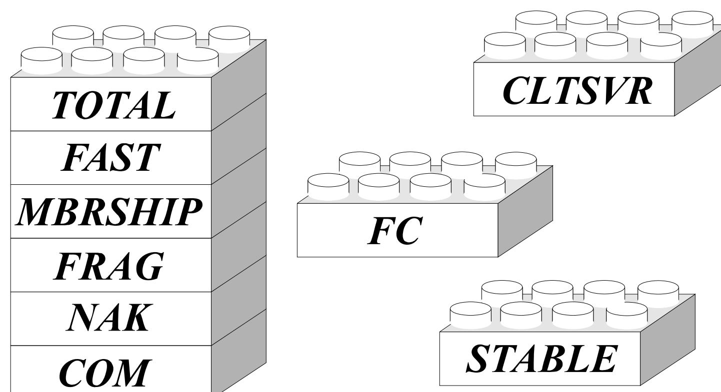

FIGURA 3.3 - Camadas de protocolos do Horus

3.4.4.1 Modelo de Grupo do Horus

Um *grupo* é uma abstração de endereçamento utilizada para referenciar uma coleção de membros do grupo [REN95]. Um membro do grupo é um ponto final de comunicação que pode originar e entregar mensagens. Processos podem ser adicionados ao grupo, podem deixar o grupo ou ser retirados do grupo por causa de uma falha. Essas operações causam a modificação do *membership* do grupo.

A visão de um grupo é uma fotografia do situação do grupo em um ponto específico na execução de um processo. Cada membro irá enxergar as modificação na situação do grupo como uma sucessão de visões e essas visões são relatadas aos membros do grupo concorrentemente e de forma assíncrona. Assim, em determinado instante do tempo real, os membros do grupo podem ter visões diferentes do grupo. Os protocolos do Horus tentam entregar a mesma seqüência de visões para cada membro, e se conseguirem, garantem que cada membro veja o mesmo conjunto de mensagens entre as visões. O Horus implementa uma variedade de modelos de execução de grupo de processos, incluindo o modelo de sincronismo virtual.

O Horus pode ser configurado para permitir continuidade das operações durante falhas transientes e partições de rede, utilizando uma variação dos protocolos propostos no Transis.

Quando ocorre uma falha de partição de rede, um único grupo pode ser dividido em múltiplos subgrupos: um primário e outros não primários. Os membros de subgrupos diferentes então terão seqüências diferentes de visões. Quando o particionamento é reparado um subgrupo não primário pode se recuperar unindo-se com o grupo primário.

As comunicações para um grupo são através de um *multicast de grupo*. Na ausência de falhas, uma mensagem de *multicast* é entregue para todos os membros do grupo presentes na última visão do emissor. Quando ocorrem falhas, se uma mensagem é entregue a algum membro não falho, deve ser entregue a todos os membros não falhos. Especificamente, quando um processo tem problemas na comunicação com outro processo de sua visão, o Horus tentará instalar uma nova visão excluindo esse membro.

O Horus sincroniza com os outros membros não falhos na visão, e assim todos os outros membros instalam a mesma nova visão. O Horus garante também que se dois processos estão em uma visão, e concordam em instalar uma nova segunda visão, então esses dois processos entregarão exatamente o mesmo conjunto de mensagens. Esse tipo de atomicidade de mensagens é chamada de "endereçamento de grupo virtualmente síncrono" no modelo Isis. Quando o Horus está configurado para suportar particionamento de rede, a execução do modelo resultante é o "modelo de sincronismo virtual estendido.

O Horus implementa o modelo de grupo, explicado na seção anterior, de uma maneira altamente reconfigurável e extensivamente dividido em camadas. O projeto do Horus permite que as aplicações paguem (em relação ao desempenho) apenas por aqueles aspectos que necessitam. Muito do projeto do Horus foi inspirado nos conceitos de vários sistemas modernos, como sistemas operacionais de *microkernel*, x-*kernel* e sistemas orientados a objetos.

Sistemas operacionais *microkernel* suportam um limitado número de primitivas básicas no nível de núcleo e serviços mais sofisticados nos níveis mais altos, permitindo flexibilidade. No Horus, um pequeno número de primitivas básicas foram identificadas e proporcionada por seu *microkernel*.

A abordagem global do Horus se assemelha ao x-*kernel*, que é uma *framework* para implementação de protocolos de rede. No x-*kernel*, cada protocolo implementa uma característica simples, e os protocolos podem ser ligados uns aos outros para suportar as necessidades da aplicação. O Horus adota esta idéia, mas sua interface é mais apropriada para protocolos de *multicast*. O Horus pode ser integrado ao x-*kernel*. Neste caso, poderia ser visto como uma extensão do x-*kernel*, especializado no caso de grupos de processos e *multicast*, mas pode também rodar dentro, ou sobre outros sistemas operacionais.

Em sistemas orientados a objetos, objetos sofisticados derivam de objetos básicos. No Horus, um "grupo básico" simples pode ser estendido com características como ordenação de mensagens ou controle de fluxo. Os grupos básicos não proporcionam visões virtualmente sincronizadas, ou mesmo troca de mensagens confiável. Ao invés disso, cada grupo básico tem um identificador e uma visão corrente do grupo. Cada membro do grupo pode ter sua própria visão de grupo, e é responsável pela manutenção e instalação de novas visões de grupo. Esse tipo de grupos básicos é suportado transparentemente sobre uma grande variedade de redes diferentes, como Ethernet e ATM, e é otimizado para o máximo desempenho.

Sobre o grupo básico existem várias características que podem ser seletivamente adicionadas para modificar a semântica do relato da visão de grupo e da comunicação de grupo. Cada característica é codificada como uma camada de software, que pode ser adicionada dinamicamente. Todas as camadas suportam a mesma interface chamada de *Uniform Group Interface* (UGI)

3.4.4.2 Camadas e protocolos

Mesmo suportando a mesma interface de grupo uniforme UGI, cada camada do Horus executa um protocolo diferente [REN94]. As aplicações podem especificar em tempo de execução qual camada ou camadas deseja utilizar. A seguir serão listadas e explicadas algumas das camadas implementadas no Horus:

COM - comunicação básica

Nessa camada não é executado protocolo algum, e o seu propósito é proporcionar a interface UGI sobre outras interfaces de comunicação de nível mais baixo (ATM, por exemplo)

NAK - comunicação FIFO

Proporciona *multicast* FIFO e comunicação ponto a ponto sobre mecanismos de comunicação não confiáveis.

STABLE - estabilidade de mensagem

Uma mensagem é chamada de ‘estável’ quando todos os membros do grupo enxergaram essa mensagem. A camada STABLE tem como função detectar quando uma mensagem foi entregue para todas as destinações podendo, conseqüentemente, ser descartada.

FC - controle de fluxo

Esta camada é a encarregada de controlar o fluxo de mensagens no Horus. Se baseia nas informações sobre estabilidade proporcionadas pela camada STABLE. Quando um número de mensagens não estáveis em um grupo exceder um número máximo, a camada FC começa a retardar (delay) as mensagens.

CLTSVR - *membership* cliente-servidor

Nessa camada, os membros do grupo são divididos em dois grupos: clientes e servidores. Os servidores executam sobre a camada MBRSHIP. Cada servidor é responsável por um conjunto de clientes.

Por definição, um cliente se comunica através de seu servidor responsável. Se um servidor falhar, outro servidor deverá adotar os clientes órfãos. Dessa maneira o protocolo simula exatamente o mesmo modelo de grupo da camada

MBRSHIP, com maior escalabilidade mas com um custo de maior latência média para os clientes. Esta camada é utilizada no suporte de RPC Replicados (RRPC), e na disseminação de dados de um pequeno conjunto de origens para um grande conjunto de processos destino.

Outras camadas:

No sistema Horus existem várias outras camadas, tais como:

XFER - transferência de estado

LWG - grupos pesos leves

FAST - aceleração de mensagens

PARCLD - disseminação hierárquica de mensagens

FRAG - fragmentação e remontagem de mensagens grandes

CRYPT - criptografia e decriptografia de mensagens

MERGE - localização e agrupamento de múltiplas instâncias de grupo

A seguir serão explicadas mais detalhadamente as camadas CAUSAL, responsável pelo *multicast* causal e TOTAL, encarregada do *multicast* com ordenação total. e MBRSHIP, camada que controla o *membership*. Apenas o entendimento da camada de *membership* é de interesse para o modelo de replicação adotado neste trabalho.

3.4.4.3 Multicast causal

Proporcionado pela camada CAUSAL, que traça a relação causal entre as mensagens usando vetor de *timestamp*.

Cada mensagem recebe um *label* com o vetor de *timestamp* do emissor, que conterá o número de mensagens provenientes dos outros membros e por ele entregues à aplicação [BIR96]. Exemplo:

O grupo G é formado por quatro processos e possui a visão [p_0, p_1, p_2 e p_3]. Considerando-se que processo p_1 possua o seguinte vetor [13, 0, 7, 6] (indicando que entregou 13 mensagens provenientes de p_0, 7 de p_2, 6 de p_3 e que não enviou nenhuma mensagem), no momento em que p_1 gerar e enviar uma mensagem **m1**, adicionará ao *label* dessa mensagem o seu vetor [13, 1, 7, 6]. Isso significa que, para que a mensagem **m1** seja entregue seguindo a ordenação causal, esta entrega deve ser adiada até que:

1. **m1** seja a próxima mensagem, na seqüência, proveniente do seu emissor;

2. as outras posições do vetor do processo receptor sejam iguais ou maiores às posições correspondentes do vetor que veio no seu *label*.

A entrega causal é proporcionada por uma camada separada (ORDER) que deve ser empilhada sobre a camada CAUSAL. A camada ORDER pode também ser usada para proporcionar entrega segura de mensagens (*safe delivery*).

3.4.4.4 Multicast totalmente ordenado

O *multicast* totalmente ordenado é garantido pela camada TOTAL, que utiliza um *token* para implementar a entrega totalmente ordenada de mensagens. Antes de uma mensagem ser enviada um membro precisa conseguir o *token*. O membro então coloca um número de seqüência em cada mensagem que envia, e as mensagens que chegam fora da seqüência são atrasadas até que possam ser entregues em ordem.

Todas as requisições e passagens de *token* são feitas através de piggyb*ack* quando existir um alto trafego de mensagens, o que resulta em uma comunicação totalmente ordenada eficiente. Ao invés de o *token* ficar fixo em um único ponto ou circular entre todos os membros do grupo, ele circula entre o conjunto corrente de emissores, uma abordagem que funciona bem se um conjunto de emissores apresenta um carga de tráfego alto e uniforme. Se, por outro lado, o carregamento do tráfego for baixo, a latência e *overhead* dessa abordagem podem se tornarem altas.

3.4.4.5 Membership

O *membership* e a atomicidade de mensagens são implementados na camada MBRSHIP, cujo núcleo é o protocolo Flush. O protocolo Flush é executado quando a falha de um membro é detectada, ou quando visões são reagrupadas. Sua intenção é finalizar a visão, assim todos os membros sobreviventes recebem o mesmo conjunto de mensagens. O protocolo funciona da seguinte forma:

1. um dos membros, normalmente o sobrevivente mais antigo, é eleito como o coordenador do protocolo

2. o coordenador difunde uma mensagem de FLUSH (contendo os membros falhos) para todos os membros não falhos da sua visão

3. todos os membros primeiro enviam ao coordenador todas as mensagem não estáveis dos membros falhos e em seguida uma mensagem de *reply* FLUSH_OK (é necessário que todos os membros coloquem no *log* todas as mensagens não estáveis)

4. subseqüentemente, os membros ignoram as mensagens que eles podem ter recebidos pelos supostos membros falhos, e esperam a instalação da nova visão

5. depois de ter recebido todas as mensagens de FLUSH_OK, o coordenador difunde qualquer mensagem de membros falhos que continua não estável. Nesse ponto, uma nova visão pode ser instalada.

Quando todas as mensagens estabilizarem, o protocolo Flush estará completado. Se processos falharem durante a execução do protocolo, uma nova rodada do protocolo pode iniciar imediatamente.

Embora a camada MBRSHIP seja capaz de fazer a recuperação de suas próprias falhas, também permite detecção externa de falhas. Nesse caso, um serviço externo compreende relatos de problemas de comunicação e outras informações de falha, e decide quando um processo deve ser considerado falho ou não.

Atualmente o sistema Horus está obsoleto, tendo o sistema Ensemble como seu sucessor. O Horus foi instalados nos laboratórios da UFRGS, mas as tentativas para fazê-lo funcionar não obtiveram êxito.

3.4.5 Totem

O sistema Totem, desenvolvido na Universidade da Califórnia, Santa Barbara, proporciona comunicação *multicast* confiável e totalmente ordenada em redes locais. O sistema Totem suporta aplicações tolerantes a falhas, nas quais processos distribuídos cooperam na execução de uma tarefa comum e na qual dados replicados precisam ser alterados de maneira consistente em um ambiente assíncrono e na presença de falhas [MOS95]. A ordenação total de mensagens simplifica a programação de aplicações distribuídas tolerantes a falhas. Se operações distribuídas são derivadas da mesma seqüência de mensagens na mesma ordem total, a consistência da informação replicada e mais fácil de ser mantida. O Totem é indicado para aplicações complexas, nas quais tanto tolerância a falhas quanto desempenho de tempo real são críticos.

São características do Totem:

1. alto *throughput* e baixa latência;

2. rápida detecção e recuperação de falhas;

3. ordenação total de mensagens, mesmo com particionamento de rede e com sobreposição de grupos de processos;

4. escalabilidade para grandes sistemas baseados em múltiplas LANs.

O sistema Totem proporciona *multicast* de mensagens confiável e totalmente ordenada para processos pertencentes a grupos de uma LAN (anel) ou de múltiplas LANs interconectadas por *gateways*. Proporciona a entrega de mensagens na presença de vários tipos de falhas de comunicação e processador, incluindo mensagens perdidas, particionamento de rede, falha de processador, assim como falhas de omissão e de temporização, só não garantindo essa entrega em caso de falhas completamente arbitrárias.

No Totem existem dois serviços confiáveis e totalmente ordenados de entrega de mensagens:

1. entrega com acordo - garante que os processos entreguem as mensagens em uma ordenação total consistente e que, quando um processo entrega uma mensagem, ele já entregou todas as mensagens precedentes originadas por processos da sua configuração atual e com *timestamp* válido.

2. entrega segura - garante que antes de um processador entregar uma mensagem, seja determinado que cada um dos outros processos da configuração atual recebeu a mensagem e vai entregá-la, a não ser que falhe. Esse serviço é útil em operações onde uma transação deve ser aceita por todos os processadores ou por nenhum.

O tipo de ordenação utilizado é definido pelo originador da mensagem.

Nem sempre um processador precisa ou consegue (em caso de falhas) entregar todas as mensagens. Quando um processador falha ou a rede particiona, torna-se impossível determinar quais mensagens foram entregues em qual ordem por quais processadores antes da falha, ou se mensagens foram entregues por processadores em outros componentes de uma rede particionada. O sincronismo virtual e o sincronismo estendido garantem que o objetivo dos serviços de entrega segura e com acordo será cumprido com qualquer configuração, mesmo se processadores falhos forem recuperados e se redes particionadas reagruparem. Quando uma falha ocorre, uma configuração provisória com um agrupamento reduzido é introduzido, sendo que todos os seus membros podem honrar a garantia de entrega.

O Totem possui um protocolo de comunicação *multicast* chamado de *single-ring*, que atua em cada um dos anéis, e um protocolo chamado de *multiple-ring*, que tem como objetivo controlar a comunicação entre os anéis. Existem também, dois protocolos de *membership*, um para controlar a configuração local, dentro de cada anel, e outro para controlar a topologia da rede, ou seja a configuração entre os anéis. Esses protocolos serão explicados a seguir, mas apenas o entendimento dos protocolos de *membership* e topologia de rede são de interesse para o modelo de replicação adotado neste trabalho.

3.4.5.1 Protocolo Single-Ring

O protocolo *single-ring* [AMI95], assume que não existem falhas maliciosas e que os canais de comunicação não alteram as mensagens em trânsito. Sobre uma rede com suporte a *broadcast* é construído um anel lógico com passagem de *token*. O *token* circula pela rede como uma mensagem ponto a ponto. A figura 3.4 mostra um anel lógico composto de cinco processos e a passagem do *token* no anel, bem como e os campos do *token*.

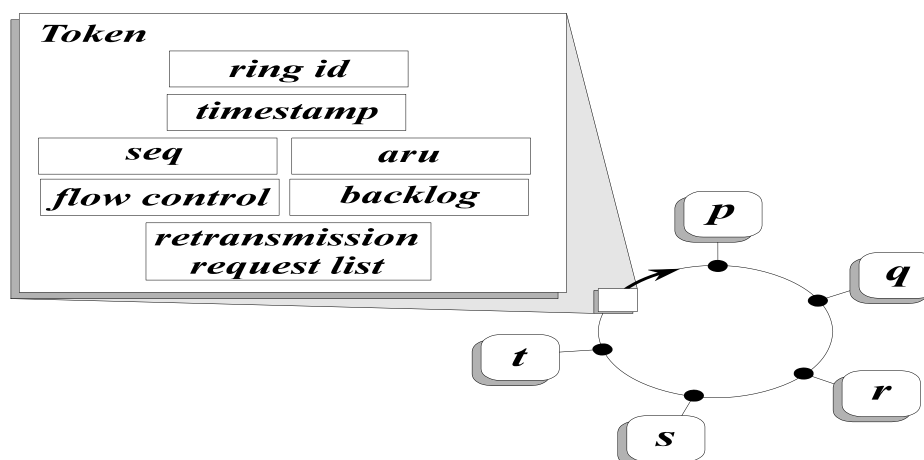

FIGURA 3.4 - Anel lógico com passagem de *token*

O *token* controla o acesso ao anel. Apenas o processo que está com o *token* pode difundir mensagens. Um processo pode enviar mais de uma mensagem cada vez que estiver com o *token*, sujeito aos limites impostos pelo mecanismo de controle de fluxo. O *token* continua a circular mesmo que nenhum processo tenha mensagens para enviar.

Quanto à ordenação de mensagens, o protocolo *single-ring* do Totem proporciona a entrega de mensagens com acordo e entrega segura de mensagens no anel lógico de passagem de *token*.

Cada cabeçalho de mensagem contém um campo de número de seqüência no *token*, chamado **seq**. Este campo **seq** proporciona uma seqüência única de números de seqüência para todas as mensagens difundidas no anel. A entrega de mensagens na ordem do número de seqüência significa a entrega de mensagens com acordo. A entrega segura de mensagens utiliza um campo adicional do *token*, o campo **aru** (*all-received-upto*) para determinar quando todos os processadores do anel receberam a mensagem.

O *token* possui ainda, entre outros, os seguintes campos:

***token*_seq** número seqüencial que permite o reconhecimento de cópias redundantes do *token*;

aru_id identificador do processo que alterou o valor do aru

rtr lista de retransmissão, contendo uma ou mais requisições de retransmissão.

Cada processo possui duas filas, uma para mensagens que estão esperando para ser enviadas e outra com as mensagens recebidas e ainda não entregues. Mantém ainda uma série de variáveis locais, incluindo,

my_*token*_seq valor do *token*_seq quando o processo repassou o *token* pela última vez

my_aru número da última mensagem recebida em ordem

O processo altera o valor de **my_*token*_seq** quando recebe o *token* e altera o valor do **my_aru** quando recebe mensagens.

Funcionamento do protocolo *single-ring*:

Quando um processo recebe o *token*, entrega as mensagens que podem ser entregues, faz a difusão das retransmissões e novas mensagens, atualiza o *token* e o transmite para o próximo processo do anel. Para cada nova mensagem que envia, o processo incrementa o campo **seq** do *token* e coloca este valor na mensagem.

Cada vez que um processo recebe o *token*, compara a variável **aru** do *token* com a variável **my_aru**. Se **my_aru** for menor, o processo troca o valor de **aru** para o valor de **my_aru** e altera o valor de **aru_id** para seu próprio identificador. Se **aru_id** é igual ao seu identificador, altera o valor de **aru** para **my_aru** (neste caso, o processador modificou o campo aru na última visita do *token*, e nenhum outro processo modificou este campo durante a rotação).

Se o campo **seq** do *token* for maior que **my_aru**, o processo não recebeu todas as mensagens que foram difundidas. Neste caso o processo anexa ao campo **rtr** do *token* os números de seqüência das mensagens perdidas, para que outro processo, que já as tenha recebido, faça as retransmissões, tirando, desta forma, seus números de seqüência do campo **rtr**.

Se um processador recebeu uma mensagem e todas as anteriores (indicado pelo número de seqüência da mensagem), pode entregar a mensagem como uma mensagem concordada (*agreed*).

Um processador pode entregar uma mensagem como uma mensagem segura, se o número de seqüência da mensagem é menor ou igual ao seu número de seqüência. Quando um processador entrega uma mensagem como segura, ele pode requisitar o espaço do disco utilizado pela mensagem porque ele nunca vai precisar retransmitir essa mensagem.

O protocolo não consegue prosseguir se o *token* for perdido. Um mecanismo de retransmissão foi implementado para reduzir a probabilidade de perda do *token*. Cada vez que um processo envia o *token*, ele dispara um *timeout* de retransmissão do *token*. Se recebe uma mensagem difundida, cancela o timeout. Se o tempo especificado no timeout chegar ao fim, o *token* é retransmitido ao próximo processo e o timeout é novamente disparado. O campo ***token*_seq** permite a detecção de *token* redundantes, um processo só aceita o *token* se o ***token*_seq** for maior que **my_*token*_seq**.

A circulação contínua do *token* não aumenta o *overhead* nem reduz o desempenho do sistema, se comparado com outros protocolos de *multicast* **[MOS95]**. No Totem, o *token* proporciona uma informação exata sobre o número de mensagens transmitidas durante a rodada anterior e essa informação é utilizada pelo mecanismo de controle de fluxo do Totem para limitar as transmissões e garantir que dificilmente ocorra *overflow* nos *buffers* de entrada. Isso permite ao Totem operar com maior *througput* do que os outros protocolos.

3.4.5.3 Protocolo de membership local

O protocolo *single-ring* ordenado do Totem é integrado a um protocolo de *membership*, que proporciona um serviço de configuração para LAN única, incluindo a adição de processadores novos e recuperados e exclusão de processadores falhos. Falhas nos processadores são detectadas por *timeouts*. Processadores novos ou em recuperação são detectados pelo aparecimento de mensagens na rede oriundas de processadores que não são membros do anel corrente. O Totem, manipula partições de rede e reagrupamento de componentes de uma rede particionada. O protocolo de *membership* do *single-ring* do Totem garante:

Consenso: todos os membros de uma configuração concordam com a nova configuração.

Terminação: todos os processadores instalam a configuração com a formação concordada dentro de um limite de tempo, a não ser que falhe dentro desse período de tempo.

Sujeito a esses requisitos de consenso e terminação, o protocolo de *membership* visa formar um *membership* o maior possível.

O detector de falhas do Totem utiliza *timeouts* e pode excluir um processador lento do grupo mesmo que ele não esteja realmente falho.

O consenso é garantido pelo protocolo de *membership* utilizando limites de tempo (*timeouts*), mesmo com a ocorrência de falhas remotas. A visão do grupo é reduzida até consenso ser alcançado. O protocolo pode resultar em um grupo de um único membro. No entanto, com uma escolha apropriada de *timeouts*, a probabilidade disso acontecer é muito pequena.

Depois de pesquisar o consenso no *membership*, o protocolo constrói um novo anel, no qual o protocolo de ordenação pode retomar as operações, gerar um novo *token* e recuperar mensagens ainda não recebidas quando a falha ocorreu. Para instalar uma nova configuração regular, o protocolo entrega duas mensagens de mudança da configuração (*Configuration Change*) ao invés da mensagem que devia estar sendo esperada. A primeira mensagem de mudança de configuração introduz uma configuração transitória de tamanho reduzido que exclui os processadores falhos ou inacessíveis. A entrega dessa mensagem significa que a entrega com acordo e a entrega segura agora apenas se aplicam à configuração transitória reduzida. Dentro da configuração transitória, as mensagens restantes da configuração antigas são entregues. Depois disso, a segunda mensagem de mudança de configuração é entregue, iniciando a nova configuração efetiva ou regular.

3.4.5.4 Protocolo Multiple-ring

O protocolo *multiple-ring* do Totem opera sobre múltiplas LANs interconectadas por *gateways* [MOS96]. Em cada LAN existe um anel lógico de passagem de *token*, onde o protocolo *single-ring* está operando. O protocolo *multiple-ring* proporciona essencialmente os mesmos serviços, com as mesmas propriedades do *single-ring*, mas para mais de um anel. A figura 3.5 mostra a união de três anéis através de *gateways*.

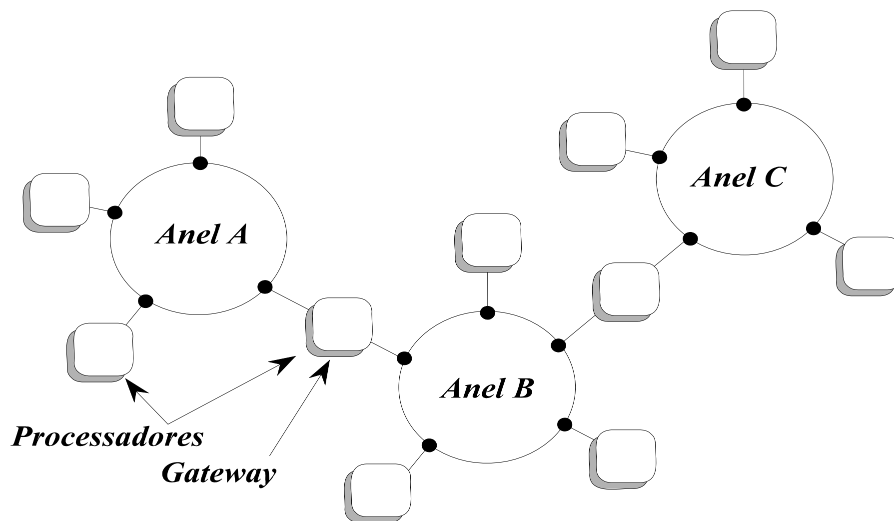

FIGURA 3.5 - Múltiplos anéis conectados por *gateways*

Para garantir uma ordenação total e global de mensagens, sobre todos os anéis é utilizado o conceito de *timestamp* de Lamport (se um processador recebe uma mensagem cujo *timestamp* excede o valor do seu próprio *clock*, o processador avança seu *clock* para um valor maior que o do *timestamp*) com entrega de mensagens por ordem de *timestamp*. As mensagens com o mesmo *timestamp* são entregues na ordem do identificador do anel de origem. Com essa ordem de *timestamp*, é garantida a consistência global na ordenação de mensagens.

Quando mensagens são geradas, são incrementados os *timestamps* e os números de seqüência em cada anel individual. Os *gatewais* repassam (*forward*) as mensagens por ordem de número de seqüência de um anel para o próximo. Quando um *gateway* difunde uma mensagem repassada, dá à mensagem um novo número de seqüência para o próximo anel então a mensagem pode ser entregue de modo confiável nesse anel. O *timestamp* da mensagem, no entanto, continua o mesmo. O número de seqüência do *single-ring* (que não contém lacunas) e o envio da mensagem na ordem do número de seqüência, garantem que não haverá mensagens perdidas.

Um processador mantém uma lista chamada de **rcv_msgs** para cada anel de qual poderia receber uma mensagem, e nessa lista mantém mensagens originadas nesse anel e recebidas pelo protocolo *single-ring*. Um processador pode entregar uma mensagem como uma mensagem com acordo, e removê-la da lista **recv_msgs**, se a mensagem tem o menor *timestemp* de todas as mensagens na lista. Uma vez que mensagens provindas do mesmo anel são repassadas na ordem de seus números de seqüência, e na ordem de seus *timestamps*, um processador sabe que nunca receberá uma mensagem com um *timestamp* menor daquele anel.

3.4.5.5 Manutenção da topologia da rede

No protocolo *multiple-ring* do Totem, cada gateway mantém uma estrutura de dados chamada *topology*, listando os anéis que estão ao seu alcance e os outro *gateways* que se interconectam com estes anéis. A topologia da rede pode ser completamente arbitrária. Como os *gateways* tem o conhecimento da topologia da rede, podem adaptar a estratégia de roteamento da mensagem para a topologia corrente. Um processador que não é um gateway precisa saber apenas o anel do qual pode esperar receber mensagens, ao invés de toda topologia da rede.

As falhas de processador e partições de rede são detectadas pelo protocolo *single-ring*, que gera uma mensagem de modificação de configuração para relatar a mudança no anel local. Cada *gateway* no anel analisa a mensagem de mudança de configuração para determinar o seu efeito na topologia. O protocolo *multiple-ring* gera e difunde uma mensagem de mudança da topologia (*Topology Change*), refletindo a mudança. Em particular, se uma *gateway* encontra um anel que está inacessível, o *gateway* remove o anel da sua topologia e notifica os outros processadores e *gateways* usando uma mensagem de modificação de topologia. Essa remoção do anel termina com a necessidade de esperar por mensagens deste anel e permite aos outros anéis de serem ordenados. Da mesma forma, uma mensagem de mudança de configuração, e sua conseqüente mensagem de mudança de topologia, podem ser utilizadas no caso de um novo anel ser adicionado à topologia.

Uma mudança na topologia deve ter o mesmo efeito para cada um dos processadores e *gateways* Apesar do fato de processadores e *gateways* poderem ficar sabendo das mudanças da topologia em tempos físicos diferente, eles precisam estar de acordo com um tempo lógico comum para a mudança da topologia e também no conjunto de mensagens entregues antes da mudança de topologia. Para isso, as mensagens de mudança de configuração e topologia levam o *timestamp* e são entregues na ordem desse *timestamp* com as mensagens normais.

3.5 Comparação dos sistemas

Os protocolos descritos nas seções anteriores foram comparados em relação a alguns aspectos que interessam ao modelo de replicação escolhido, o da cópia primária.

3.5.1 Particionamento com partição primária

No sistema xAMp, não é tratado o particionamento de rede, ou seja, não podem coexistir duas visões do mesmo grupo. No momento em que um processo fica incomunicável, ele é retirado do grupo. Isso equivale a uma partição primária, ou seja, apenas uma parte do grupo continua ativa.

No Newtop, mesmo na ocorrência de particionamento de rede, o seu protocolo de *membership* mantém a consistência da visão do grupo, permitindo assim, que um grupo de processos se particione em dois ou mais subgrupos. Quando isso ocorre, os membros de cada subgrupo se consideram os únicos membros sobreviventes do grupo original, e não terão conhecimento da existência de outros subgrupos. O Newtop deixa a aplicação decidir quando deve manter mais de um subgrupo ou apenas uma partição primária.

O sistema Transis tem como uma de suas principais características o tratamento de partições de rede, e o faz através de operações particionáveis, nas quais múltiplos componentes da rede estão desconectados uns dos outros e operam de forma autônoma. Permite também que no caso da ocorrência deste tipo de falha, as operações apenas tenham continuidade em uma partição primária, se essa opção for mais interessante para a aplicação.

O Horus, após um particionamento, um grupo pode ser dividido em várias partições: uma primária e várias secundárias. É permitido, no entanto, que a aplicação decida se quer usar apenas a partição primária ou prosseguir com a execução em todas as partições secundárias. É utilizada uma variação dos protocolos propostos no Transis.

O sistema Totem manipula partições de rede e reagrupamento de componentes de uma rede particionada. As partições de rede são detectadas pelo protocolo *single-ring*, e são tratadas pelos mecanismos de configuração local e configuração de topologia.

3.5.2 Reintegração após particionamento

Os sistemas que tratam da reintegração de servidores de forma automática após particionamento são os sistemas Totem e Transis e Horus. O Totem permite a reintegração de um servidor ao grupo através do reconhecimento de mensagens, externas ao grupo, que passam pelos anéis. No Transis adição de servidores novos ou recuperados é espontânea e é disparado no momento em que os servidores são

conectados. No Horus, quando a partição é reparada, uma partição secundária pode ser reintegrada à primária.

3.5.3 Sobreposição de grupos de replicação

A sobreposição de grupos, onde processos podem pertencer a vários grupos diferentes, é permitida no sistema xAMp, no Newtop e no Totem.

3.5.4 Primitivas de multicast confiável

Todos os sistemas apresentados possuem primitivas de multicast confiável, atômico e causal. Os sistemas Horus e xAMp, no entanto, são bastante flexíveis no que diz respeito a escolha do protocolo, permitindo que o usuário possa utilizar apenas a primitiva que lhe for necessária. Para a disseminação de escritas em arquivos replicados através da abordagem da cópia primária, apenas é necessária a primitiva de multicast confiável, uma vez que todas as mensagens partem do mesmo emissor, o servidor primário.

3.5.5 Membership e visão de grupo

Todos os sistemas apresentados trabalham com grupos de processos de forma dinâmica, ou seja, permitindo a mudança das visões do grupo. Esta característica é muito importante para a implementação de replicação de arquivos na abordagem da cópia primária, priorizada neste trabalho, uma vez que em caso de falha do servidor primário, este deve sair do grupo de replicação para que outro possa ser escolhido como primário. Além disso, deve ser permitida a sua reintegração ao grupo, após a sua recuperação

No xAMp, o mantém uma cópia a visão global em cada membro do grupo e as modificações na visão são executadas com a utilização de um *lock* distribuído que permite que apenas um membro faça modificações por vez na visão. As operações que podem ser realizadas em um grupo são: *join* para a entrada de processo no grupo, *leave* para a saída de processos no grupo, *delete* para retirada de processo falho e *check* disparada quando à suspeita de falha.

No Newtop, cada membro possui a visão completa do grupo, e só poderá instalar uma nova visão depois de conseguir o consenso. Uma observação importante é que no Newtop as visões de grupo não crescem, ou seja, só existem operações para retirada membros do grupo. A formação de um novo grupo, no entanto, pode ser iniciada por qualquer processo.

O protocolo de *membership* do Transis se baseia em acordo, de forma que todos os processos do grupo precisam concordar com a nova visão. O que distingue este protocolo de *membership* dos demais é a adição espontânea de novas máquinas, disparada no momento em que as máquinas são conectadas, sem a utilização de primitivas do tipo *join*.

O Horus possui uma camada que implementa o protocolo de *membership*, chamado de Flush. O protocolo é baseado em um coordenador que fará com que todos os membros sobreviventes do grupo recebem o mesmo conjunto de mensagens, durante a instalação de uma nova visão. Para a entrada de um novo membro no grupo é utilizado o protocolo de *merge*, também utilizado para reconfiguração após particionamento.

No Totem o protocolo de *membership* do protocolo *single-ring* garante consenso, ou seja a concordância de todos os membros de uma visão para uma nova visão ser instalada, e para isso, vai diminuindo o tamanho do anel, até conseguir o consenso. Novos processos e processos em recuperação são detectados pelo aparecimento de mensagens na rede oriundas de processadores que não são membros do anel corrente e podem passar a fazer parte do grupo.

4 Protótipo para disseminação de escrita para arquivos replicados via multicast - PDERM

Este capítulo descreve o protótipo desenvolvido. O objetivo do protótipo é simular um ambiente que utiliza replicação de dados e avaliar experimentalmente a utilização de um protocolo de comunicação de grupo na disseminação de operações de escritas para os arquivos replicados. O principal aspecto a ser avaliado é o da manutenção da atomicidade dos dados disseminados.

4.1 Descrição do modelo

O modelo utiliza a abordagem de replicação da cópia primária, onde existe um servidor primário, vários servidores secundários e alguns processos clientes que se comunicam com o servidor primário para fazer requisições de leitura ou escrita em arquivos. O servidor primário possui cópia de todos os arquivos. O sistema comporta vários grupos de replicação, cada um com o seu servidor primário. Para simplificar será considerado inicialmente apenas um grupo de replicação. A figura 4.1 mostra o modelo simplificado.

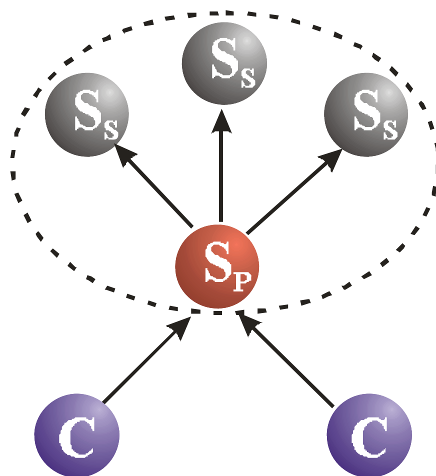

FIGURA 4.1 - Modelo simplificado de grupo de replicação

4.1.1 Clientes

Os processos clientes tem conhecimento apenas do servidor primário de um grupo de replicação com o qual se comunicam fazendo requisições de leitura ou escrita de dados em arquivos. A requisição enviada ao servidor primário possui três informações básicas:

arquivo: nome do arquivo que deve ser consultado ou modificado,

tipo: tipo de operação a ser feita (LEITURA OU ESCRITA)

dado: dado que será enviado, apenas em caso de operação de escrita.

A operação de escrita disponível no protótipo é a inclusão, ou seja, o dado enviado pelo cliente será incluído no arquivo determinado, caso a operação for escrita. As demais operações de atualização podem ser tratadas de forma semelhante.

4.1.2 Servidor primário

O servidor primário é um processo centralizador que se comunica tanto com os processos clientes quanto com os servidores secundários.

Quando recebe uma requisição de um cliente, o primário verifica o tipo de operação a ser feita. Em caso de leitura, acessa o arquivo desejado e imediatamente responde ao cliente, pois como essa operação não gera modificações no arquivo em questão, não precisa ser disseminada aos servidores que possuem cópia do arquivo.

Caso a requisição do cliente seja de escrita, esta é repassada através de *multicast* para o grupo de replicação no qual o arquivo em questão se encontra replicado, o que inclui, além de alguns secundários, o próprio servidor primário.

Se a operação de envio da mensagem via *multicast* obteve sucesso, significando que todos os servidores ativos do grupo de replicação receberam a mensagem com operação a ser realizada, o primário envia ao cliente uma resposta confirmando o sucesso da operação de escrita.

4.1.3 Servidores secundários

Cada servidor do grupo de replicação, após receber a mensagem do servidor primário, executa as modificações necessárias no arquivo que foi determinado.

4.1.4 Particionamento de rede e falha no servidor primário

Em caso de particionamento de rede, apenas uma partição, a que tiver o servidor primário, fica ativa. Os servidores secundários da outra partição param de receber mensagens de escrita e são excluídos do grupo de replicação.

O protótipo não mascara falhas no servidor primário, a não ser na entrega de mensagens *multicast* por ele iniciadas.

4.2 Arquivos replicados e grupos de replicação

A figura 4.2 mostra um sistema com vários grupos de replicação, cada qual com o seu servidor primário. Processos clientes, neste caso, conhecem os servidores primários de cada grupo.

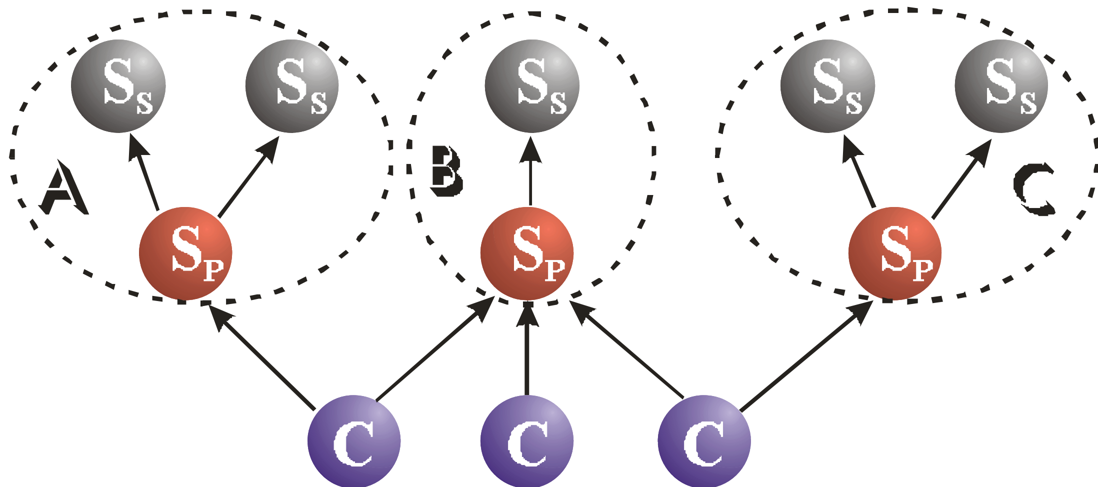

FIGURA 4.2 - Múltiplos grupos de replicação

A intersecção de grupos de replicação é possível. Conforme mostra a figura 4.3, um servidor pode ser o servidor secundário em mais de um grupo, assim como um servidor pode ser o primário de dois grupos de replicação.

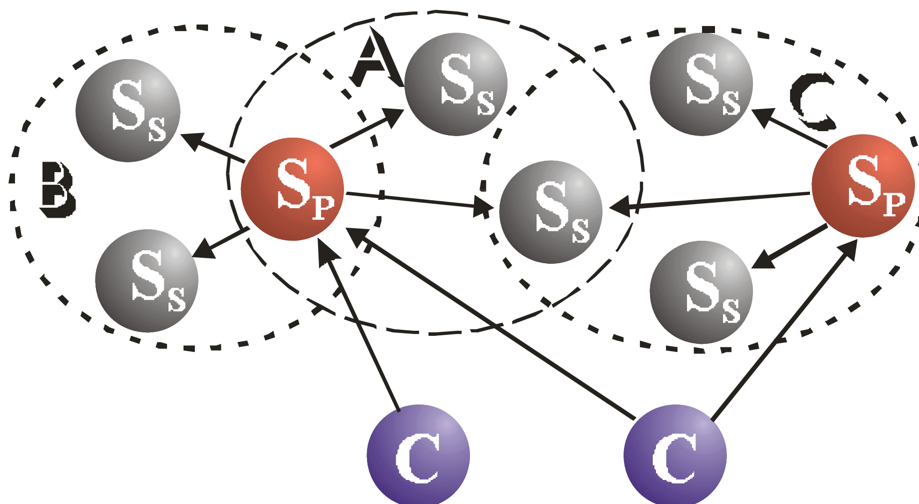

FIGURA 4.3 - Intersecção de grupos de replicação

Outra situação possível é a de um servidor ser o primário em um grupo e secundário em outro.

Nos dois esquemas mostrados, em caso de falha de um dos servidores primários um novo primário deve ser escolhido para o grupo e os clientes devem ser notificados da mudança.

4.3 Ambiente

O protótipo foi desenvolvido nas estações SUN, utilizando o sistema operacional UNIX, sobre o qual executa o núcleo do sistema xAMp, versão 3.1. As

aplicações foram criadas em linguagem C utilizando funções de gerenciamento e comunicação de grupo da biblioteca de funções do xAMp.

O sistema xAMp foi obtido para a instalação sob licença especial para ambiente acadêmico. O formulário para a licença pode ser encontrado na Internet no endereço http://formiga.di.fc.ul.pt/xAMp/LICENCE .

4.4 Utilização do xAMp

Na seção 3.4.1 foi descrito o funcionamento do xAMp, o que permitiu a escolha da utilização desse sistema para a solução do problema proposto: a utilização de protocolos multicast para a disseminação de escritas em ambientes replicados. O funcionamento interno do xAMp é transparente aos seus usuários, que só precisam saber utilizar as primitivas de comunicação de grupo disponíveis. Nas seções a seguir será explicado o funcionamento do xAMp na visão do usuário, bem como detalhes da utilização de algumas funções [CAS94] para gerenciamento e a comunicação de grupo.

4.4.1 Execução do xAMp

Para a execução do sistema xAMp, deve ser disparado um *daemon*, que fica residente na máquina. Esse *daemon* é disparado pelo usuário *root* da rede de estações de trabalho.

4.4.2 Inicialização

Todas as aplicações que se utilizam do xAMp necessitam do *include* xampuser.h, que define todas as constantes e estruturas de dados usadas como parâmetros para as funções.

As rotinas de inicialização disponíveis são xampOpen e xampClose, que servem, respectivamente para abrir e fechar a conexão da aplicação com o *deamon* do xAMp.

A rotina xampOpen possui três parâmetros:

nonblock: define se a conecção é aberta no modelo bloqueante (se for 0) ou não bloqueante, (se for diferente de 0);

async: define se as mensagens que chegam devem ser manipuladas de forma assíncrona (esta opção, no entanto, ainda não está disponível na versão utilizada);

iohandler: define como serão tratadas as mensagens de entrada. Se for igual a 0, será chamada a função padrão para tratamento de mensagens;

A seguir são mostrados exemplos de chamadas às rotinas xampOpen, abrindo a conexão do processo chamador ao núcleo do xAMp de maneira não bloqueante e síncrona, e xampClose, fechando a conexão com o xAMp.

```
xampOpen (1, 0, 0) ;

xampClose();
```

4.4.3 Entrada e saída de grupos

A criação ou extinção de grupos no xAMp não são feitas de maneira explícita, ou seja, não existem comandos para criação ou extinção de grupos. Para que um processo entre em um grupo esse processo precisa executar a rotina xampGroupOpen tendo como um dos parâmetros o grupo ao qual pretende se juntar. Fica implícito que caso este grupo não exista, ele será criado.

Se determinado processo deseja deixar um grupo, precisa executar a rotina xampGroupClose nesse grupo.

As rotinas xampGroupOpen e xampGroupClose servem, portanto, como *join* e *leave* em determinado grupo.

A versão do xAMp utilizada permite que um usuário abra mais de 50 grupos diferentes, ou abra o mesmo grupo multiplas vezes, desde que identificadores de objetos (ouid) diferentes sejam especificados a cada requisição.

São parâmetros para uma chamada à rotina xampGroupOpen:

group_id: identificação do grupo a ser aberto;

ouid: identificador do objeto que está abrindo o grupo;

key: identificador único criado pelo usuário, apenas usado em caso de ambiguidade (objeto que abre duas vezes o mesmo grupo mas precisa de uma confirmação em apenas um dos casos.);

além de três rotinas de manipulação de serviços de grupo que podem podem ser opcionalmente definidas pelo usuário (*viewhandler*, *confhandler* e *datahandler*) e que serão discutidas na próxima seção.

A rotina utilizada para fechar determinado grupo por algum objeto, xampGruopClose, possui três parâmetros (group_id, ouid, key) que são os parâmetros também utilizados da rotina xampGroupOpen, e foram descritos anteriormente.

Exemplos de chamadas às rotinas xampGroupOpen e xampGroupClose:

```
xampGroupOpen(GRUPO_A, my_oid, viewHandler,
              confHandler, dataHandler,  0);
```

```
xampClose(GRUPO_A, my_oid, 0);
```

4.4.4 Manipulação de serviço de grupo

As rotinas de manipulação de serviços de grupo são opcionalmente definidas pelo usuário e chamadas como parâmetro da rotina xampGroupClose. É através delas que é definido o tratamento das mensagens recebidas, assim como visões de grupo e confirmações de serviços. Caso forem definidas, tais rotinas são utilizadas nas seguintes situações descritas a seguir.

4.4.4.1 Visão do grupo

A visão de um grupo é tratada na rotina *viewhandler*, que é chamada toda a vez que a visão do grupo muda. As mudanças na visão do grupo são disseminadas de forma atômica para todos os membros do grupo. Novos membros só são considerados como membros depois que a nova visão do grupo é recebida. O formato da função é:

```
void viewhandler(group_id, view)
```

onde *group_id* é o identificador do grupo ao qual a visão se refere e *view* é um ponteiro para a *OuidList*, que é a lista dos identificadores dos objetos que pertencem ao grupo.

4.4.4.2 Confirmações

A rotina *confhandler* é chamada quando uma confirmação de requisição (*open* ou *close* em um grupo ou envio de mensagens) é recebida. O formato da função é:

```
void confhandler(group_id, ouid, mess_id, status,
                 data, data_size, key)
```

onde *group_id* é o identificador do grupo ao qual a confirmação se refere; *ouid* é a identificação do objeto que chama a confirmação; *mess_id* indica que tipo de requisição está sendo confirmada (OPEN_REQUEST, CLOSE_REQUEST, RELIABLE_SEND, ATOMIC_SEND, DELTA_SEND ou TIGHT_SEND); *status* indica o estado da requisição, (STATUS_OK ou STATUS_NOT_OK); *data* é um ponteiro para a informação enviada, caso a confirmação seja para requisições de envio de mensagem.

4.4.4.3 Recebimento de mensagens

A rotina *datahandler* faz o tratamento de mensagens recebidas e é chamada toda vez que dados são recebidos pelo grupo. O formato da função é:

```
void  datahandler(group_id,  oid,  mess_id,  info,
                            data, data_size)
```

onde *group_id* é o grupo para o qual a mensagem foi enviada; *ouid* é o objeto que enviou a mensagem*; mess_id* indica o tipo de dado que esta sendo recebido (RELIABLE_RECV, ATOMIC_RECV, DELTA_RECV, TIGHT_RECV); *data* é um ponteiro para um *buffer* que contém a mensagem transmitida e *data_size* é o tamanho deste *buffer*. O parâmetro *info* é sempre NULL nesta versão do xAMP.

Um detalhe importante é que o usuário não precisa utilizar primitivas de recebimento de mensagens, pois isso é feito pelo xAMp. A rotina *datahandler* (que é definida pelo usuário) precisa somente definir o que será feito com os dados recebidos pelo processo.

No sistema de simulação aqui apresentado apenas as rotinas do tipo *viewhandler* e *datahandler* foram implementadas.

4.4.5 Serviço de mensagens

Como o modelo implementado utiliza o método de replicação da cópia primária, o problema de manutenção da atomicidade na entrega das mensagens já se encontra resolvido. Todas as mensagens partem do mesmo processo centralizador (o servidor primário), o que garante a ordenação nas mensagens recebidas pelos servidores secundários. Por este motivo, é apenas necessário o uso de uma primitiva confiável de envio de mensagens, que garanta que todos os servidores do grupo realmente receberão a mensagem. Esta primitiva, no xAMp, é a rotina reliableSend, que garante o recebimento das mensagens por todos os membros ativos do grupo, mesmo em caso de falha do emissor. O formato da primitiva é o seguinte:

```
int reliableSend (group_id, ouid, to, data, size,
                            clab, key)
```

onde *group_id* é o grupo para o qual a mensagem está sendo enviada; *ouid* é identificador do objeto que envia a mensagem; to indica o subconjunto do grupo que deve receber a mensagem. Se for ALL_OBJ, todo o grupo recebe; *data* é um ponteiro para um *buffer* contendo os dados a serem transmitidos; *size* é o tamanho do *buffer*; *key* utilizada para facilitar as confirmações; *clab*, que é um *label*. É garantido que todas as mensagens com o mesmo *label* são entregues preservando a ordem FIFO.

A primitiva reliableSend envia de maneira confiável a mensagem especificada para todos os membros ativos do grupo ou sub-grupo especificado, mesmo em caso de falha do emissor.

4.4.6 Demais rotinas utilizadas

Além das rotinas acima detalhadas, o protótipo utiliza ainda:

ouidGetUniq () – gera um código um identificador único

xampMainloop() – proporciona um loop do servidor

ouidListSize (view) – retorna o tamanho da lista de objetos do grupo

ouidListElem (view, i) – acessa elementos da lista de objetos do grupo.

ouidToString(ouid, buff) – transforma o identificador único de um objeto em *string*

4.5 Implementação do protótipo

Como foi definido na seção 4.1.1, os processos clientes não precisam tomar conhecimento de outros servidores que não o primário, sendo este o único com o qual se comunicam. Esta comunicação poderia ser feita como troca de mensagens ponto a ponto. Neste modelo, no entanto, a comunicação entre clientes e servidor primário foi implementada com primitivas de comunicação *multicast*, mas valendo-se da flexibilidade oferecida pelo xAMp, que permite a escolha de apenas um membro do grupo para receber determinada mensagem. Desta forma, todos os processos clientes necessitam abrir um grupo chamado de GR_CLI, que também é aberto pelo servidor primário. Mensagens ao primário podem ser enviadas utilizando a primitiva reliableSend, mas optando pela configuração do parâmetro *to* com o identificador único do servidor primário, de forma que só este receba a mensagem, em vez de todos os membros do grupo.

Para teste do PDERM foram criados três arquivos para a replicação: DADOS_A, DADOS_B E DADOS_C. Cada arquivo se encontra replicado em um grupo de servidores diferentes, mas todos estão no servidor primário. A tabela 4.1 mostra a distribuição dos arquivos replicados nos servidores. Servidores neste treabalho chamados de tipo S1 são aqueles que apenas fazem parte do GRUPO_A, os do tipo S2 pertencem apenas ao GRUPO_B, os do tipo S3 pertencem apenas ao GRUPO_C e os do tipo S4 pertencem aos grupos GRUPO_A e GRUPO_B conforme mostra a tabela 4.3.

TABELA 4.1 - Distribuição dos arquivos replicados

TIPO DE SERVIDOR	ARQUIVOS ARMAZENADOS
PRIMÁRIO	DADOS_A, DADOS_B, DADOS_C
S1	DADOS_A
S2	DADOS_B
S3	DADOS_C
S4	DADOS_A, DADOS_B

Desta forma, existem três grupos de replicação: GRUPO_A, grupo de servidores onde se encontram as cópias o arquivo DADOS_A; GRUPO_B, onde se encontram as cópias do arquivo DADOS_B e GRUPO_C, grupo de replicação de DADOS_C. A tabela 4.2 mostra os três grupos criados e seus componentes.

TABELA 4.2 - Composição dos grupos

GRUPO	COMPONENTES (TIPO DE SERVIDORES)
GRUPO_A	PRIMÁRIO, S1, S4
GRUPO_B	PRIMÁRIO, S1, S4
GRUPO_C	PRIMÁRIO, S3

A figura 4.4 mostra o esquema com a intersecção de três grupos de replicação, onde o primário é o mesmo para os três grupos. Em caso de colapso do servidor primário, deve ser escolhido um novo servidor primário para cada grupo. Neste caso o sistema ficaria com três servidores primários. Na reintegração o nodo falho deverá ser, num primeiro momento, reintegrado de forma independente em cada grupo. Posteriormente pode ser novamente escolhido como primário em todos os grupos. A reintegração de nodos falhos, no entanto, não é tratada neste trabalho.

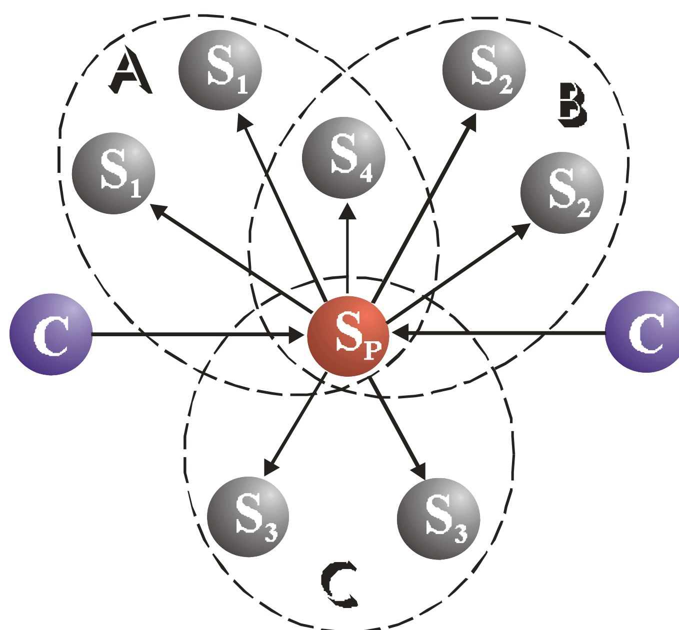

FIGURA 4.4 - Modelo utilizado no protótipo

Este modelo foi escolhido para teste o protótipo por representar um caso usual, onde o servidor escolhido como primário é o de maior desempenho no tratamento de operações I/O.

Note que S1, S2, S3 e S4 são tipos de servidores, sendo que em no sistema podem haver vários servidores de cada tipo.

Cada tipo de servidor de arquivos (PRIMÁRIO, S1, S2, S3 e S4), executa um código que difere apenas na abertura dos grupos a que pertence. A seguir serão descritos os algoritmos básicos utilizados no servidor primário e nos servidores secundários.

4.5.1 Servidor primário

No programa principal ,o servidor primário precisa, basicamente: abrir o descritor do xAMp, ou seja, abrir uma conexão do seu processo com o núcleo do sistema xAMp; gerar o seu identificador único que será usado na abertura de grupos e na troca de mensagens; abrir os diferentes grupos aos quais pertence. Essas operações são mostradas nos trechos de programa do protótipo PDR para o modelo mostrado na figura 4.3.

Abertura da conexão do processo com o núcleo do xAMp

```
if ((xamp_fd = xampOpen (1, 0, 0)) < 0)
    {
          printf ("Error %d Opening xAMp\n", xamp_fd);
          exit ();
    }
```

Geração de uma identificação única para o processo

```
    my_oid  = ouidGetUniq ();
```

Abertura dos grupos de replicação GRUPO_A, GRUPO_B , GRUPO_C e do grupo de clientes GR_CLI

```
xampGroupOpen(GRUPO_A, my_oid, servViewH, 0, dataHandler, 0);
xampGroupOpen(GRUPO_B, my_oid, servViewH, 0, dataHandler, 0);
xampGroupOpen(GRUPO_C, my_oid, servViewH, 0, dataHandler, 0);
xampGroupOpen(GR_CLI , my_oid, cliViewH, 0, dataHCli,0);

xanpMainLoop();
```

Todos os grupos de replicação são abertos tendo como parâmetros o nome do grupo, a identificação do processo e as rotinas servViewH, que tratará a visão do grupo que a chamou, e dataHandler, que será chamada toda a vez que chegar alguma mensagem para o grupo.

O grupo de clientes é aberto tendo como parâmetros o nome do grupo, a identificação do processo e as rotinas *cliViewH*, que será chamada toda a vez que a visão do grupo de clientes mudar e a rotina de manipulação de dados *dataHCli*, que será chamado toda vez que chegar alguma mensagem do grupo de clientes.

As rotinas *servViewH*, *dataHandler*, *cliViewH* e *dataHCli*, chamadas na abertura de grupos, não fazem parte do sistema xAMp. O sistema xAMp apenas chama estas rotinas, se implementadas pelo usuário, em momentos específicos como

mudança de visão de grupo e recebimento de mensagem do grupo. As implementações dessas rotinas no protótipo, são descritas nas seções que seguem.

4.5.1.1 Tratamento da visão do grupo de servidores

A rotina *servViewH* será chamada toda a vez em que houver uma modificação no grupo em cuja abertura foi passada como parâmetro. Sua incumbência é percorrer todo a lista de elementos do grupo e imprimir o nome dos elementos do grupo. Esta rotina serve, no protótipo PDERM, para teste e acompanhamento experimental.

Imprime o nome do grupo e o número de membros

```
    printf ("\nGrupo <%d>:\n", group);
    printf ("  %d membros\n", ouidListSize (view));
```

Percorre a lista de membros do grupo e escreve os seus identificadores

```
    printf ("  Membros = [");
    for (i=0; i < ouidListSize (view); i++)
    {
         ouidToString (ouidListElem (view, i), str);
         printf ("%s, ",str);
    }
    printf ("]\n");
```

4.5.1.2 Tratamento da visão do grupo de clientes

A rotina *cliViewH* será chamada toda a vez em que houver uma modificação no grupo de clientes. No momento que a visão do grupo de clientes se modificar, será enviada uma mensagem de *multicast* para todos os clientes do grupo, com o número do identificador único de objeto (ouid) do servidor primário. Desta forma os clientes saberão para quem devem enviar suas mensagens de *multicast* com as requisições de leitura ou escrita de dados. Além do envio de mensagens essa rotina também percorre a lista de elementos do grupo e a imprime.

Imprime o nome do grupo e o número de membros
```
    printf ("\nGrupo <%d>:\n", group);
    printf ("  %d membros\n", ouidListSize (view));
```

Percorre a lista de membros do grupo e escreve os seus identificadores

```
    printf ("  Membros = [");
    for (i=0; i < ouidListSize (view); i++)
    {
          ouidToString (ouidListElem (view, i), str);
          printf ("%s, ",str);
    }
    printf ("]\n");
```

Envia para o grupo de clientes o seu identificador único
```
          ouidToString  (my_iod, mensagem);
          reliableSend(GR_CLI,my_oid,ALL_OBJ,                mensagem,
strlen(mensagem), 0, 1);
```

4.5.1.2 Tratamento de mensagens provenientes de clientes

A rotina *dataHCli*, chamada pelo xAMp toda a vez que é recebida uma mensagem do grupo de clientes, executa os seguintes passos:

- extrai da mensagem recebida as informações de **tipo** de operação (LEITURA ou ESCRITA), o nome do **arquivo** no qual deve ser realizada a operação (A, B ou C) e o **dado** a ser escrito, caso o tipo de operação for escrita;

- se o **tipo** de operação for de ESCRITA

  - monta a mensagem a ser enviada para o grupo de servidores, colocando nela o **arquivo** a ser modificado e o **dado** a ser incluído;

  - envia mensagem com o dado para o seu grupo de replicação, (se o arquivo for A, GRUPO_A , se for B, para GRUPO_B e se for C, para GRUPO_C;

Extrai informações da mensagem recebida do cliente
```
      sscanf(data,"%d %c %s",&type, arquivo, dado);
```

Se o tipo de operação for escrita, monta e
```
      if (type==ESCRITA)
                  sprintf(mensagem,"%c %s\n",arquivo,dado);
```

Envia a mensagem para o grupo de replicação do arquivo requisitado

```
      switch (arquivo){
          case 'A':
               reliableSend(GRUPO_A,  my_oid,  ALL_OBJ,  mensagem,
strlen(mensagem), 0, 1);
          case 'B':
               reliableSend(GRUPO_B,  my_oid,  ALL_OBJ,  mensagem,
strlen(mensagem), 0, 1);
          case 'C':
               reliableSend(GRUPO_C,  my_oid,  ALL_OBJ,  mensagem,
strlen(mensagem), 0, 1);
          }
```

4.5.1.3 Tratamento de mensagens de modificação em arquivos

Como o servidor primário faz parte dos grupos de replicação de todos os arquivos de sistema, ele só pode fazer uma modificação em um arquivo se todos os outros membros do grupo de replicação deste arquivo também o fizerem. Por isso, a mensagem com a disseminação da operação de escrita é enviada a todo o grupo de replicação, o que inclui o próprio servidor primário. Se a mensagem, enviada através da primitiva reliableSend, chegar, significa que também chegou em todos os secundários, portanto a modificação do arquivo pode ser realizada.

A rotina que trata do recebimento de mensagens de modificações em arquivos é a rotina *dataHandler*, que será chamada toda vez que o chegar alguma mensagem ao grupo (grupo em cuja abertura foi passada como parâmetro).

Os passos executados pela rotina dataHandler são os seguintes:

- extrai da mensagem as informações recebidas (**arquivo –** arquivo a ser modificado e **dado** – dado a ser inserido no arquivo;
- abre para atualização o arquivo DADOS_A, DADOS_B ou DADOS_C, conforme o conteúdo da variável **arquivo** (A, B ou C);
- escreve o **dado** no arquivo;
- fecha o arquivo aberto;

Extrai informações da mensagem recebida do cliente

```
      sscanf(data,"%c %s", arquivo, dado);
```

Abre o arquivo requisitado, faz a modificação e fecha o arquivo

```
      switch (arquivo){
         case 'A':
                  strcpy(arq,"A");
            strcat (filename, arq);
                  if ((fp = fopen(filename,"a"))==NULL)
                  {printf("Erro   na   abertura   do   arquivo   %s
                          \n",filename);}
            fprintf(fp,"%s\n",dado);
               fclose(fp);
                  ...
```

Repetindo a mesma operação para os demais arquivos (B e C)

4.5.2 Servidores secundários

O código dos servidores secundários é semelhante ao código do servidor primário, exceto pela abertura do grupo de clientes e tratamento de mensagens deste recebidas. Cada tipo de servidor secundário abre apenas um ou dois grupos de replicação, conforme mostra a tabela 4.3. As rotinas de manipulação, utilizadas como parâmetro para a abertura dos grupos, também são as mesmas nos servidores secundários e no primário. A tabela a seguir mostra quais grupos são abertos cada tipo de servidor.

TABELA 4.3 - Grupos abertos por cada servidor

TIPO DE SERVIDOR	**GRUPO**
PRIMÁRIO	GRUPO_A, GRUPO_B, GRUPO_C, GRUPO_CLI
S1	GRUPO_A
S2	GRUPO_B
S3	GRUPO_C
S4	GRUPO_A, GRUPO _B

O exemplo mostra o programa principal do servidor do tipo S1, e a abertura para comunicação dos grupos a que pertence.

Abertura da conexão do processo com o núcleo do xAMp

```
if ((xamp_fd = xampOpen (1, 0, 0)) < 0)
    {
         printf ("Error %d Opening xAMp\n", xamp_fd);
         exit ();
    }
```

Geração de uma identificação única para o processo

```
    my_oid  = ouidGetUniq ();
```

Abertura dos grupos de replicação ao qual o servidor pertence

```
xampGroupOpen(GRUPO_A, my_oid, servViewH, 0, dataHandler, 0);

xampMainloop();
```

No protótipo implementado, a abertura de grupos é feita de forma estática, uma vez que está descrito no próprio código do servidor quais grupos deverão ser por ele abertos.

Em um sistema real pode haver uma dinâmica de grupo, de forma que um servidor pode ora pertencer a um grupo de replicação, ora a outros. Para isto, no momento em que o servidor é inicializado, ele pode receber a informação de que grupo ou grupos deve abrir, além das cópias atualizadas dos arquivos replicados neste(s) grupo(s).

4.6 Experimentos

O objetivo da etapa de experimentação é analisar o comportamento de um modelo de replicação de arquivos na abordagem da cópia primária que usa primitivas de *multicast* para disseminar as operações de escrita requisitadas ao servidor primário. Para alcançar este objetivo não é necessário, no entanto, a utilização de mais de uma estação de trabalho, uma vez que todos os processos (clientes e servidores) podem executar na mesma máquina. Assim, os experimentos foram realizados em apenas uma máquina da rede de estações de trabalho SUN.

O modelo testado é o modelo mostrado na figura 4.4.

Para que os vários secundários responsáveis pelas cópias dos arquivos não escrevessem no mesmo arquivo, foi criado um esquema especial para os nomes dos arquivos. Desta forma, o servidor primário escreve nos arquivos DADOS_A, DADOS_B e DADOS_C e cada servidor secundário escreve no arquivo cujo nome é composto pelo nome original (DADOS_A por exemplo) acrescido de um índice que identifica o servidor secundário. Este índice é designado ao servidor secundário como argumento no momento em que é disparado.

4.6.1 Execução do protótipo

Todo os processos clientes e servidores são disparados através de linha de comando. Os servidores secundários, necessitam um argumento que serve para a sua identificação, uma vez que vários servidores do mesmo tipo podem ser disparados.

Em um primeiro teste, foi disparado um processo cliente, um servidor primário e sete servidores secundários, dois de cada tipo (S1, S2, S3) e um do tipo S4. A figura 4.4 visualiza a situação.

Em um primeiro momento o cliente, que é um processo interativo, enviou ao primário um pedido de inserção de informações no arquivo DADOS_A.

O servidor primário, ao receber a mensagem do cliente, repassou a modificação a ser feita para o GRUPO_A (grupo de replicação do arquivo DADOS_A) e assim que a primitiva de envio se completou, respondeu ao cliente.

No final desta primeira etapa, o resultado esperado era que todas as cópias do arquivo DADOS_A estivessem iguais. O resultado foi alcançado.

O experimento foi repetido diversas vezes, inclusive com um grande número de servidores secundários. No final das execuções todos os arquivos replicados estavam íntegros, contendo as mesmas informações na cópia principal e nas réplicas.

Para o teste do protótipo foram injetadas falhas de *crash* num primeiro momento em servidores secundários e, em seguida, no servidor primário. A maneira utilizada para introduzir a falha foi simplesmente derrubar (através do comando *kill* ou CTRL+C) o processo em questão. As falhas dos servidores secundários eram injetadas em qualquer tempo da execução do protótipo. Já as falhas do servidor primário ocorriam no primeiro instante após o envio de mensagem de alteração das réplicas.

4.6.2 Falha de servidores secundários

O comportamento do protótipo mediante a falha de um servidor secundário foi equivalente ao esperado: todos os servidores não falhos realizaram as modificações e o servidor falho foi retirado do(s) grupo(s) a que pertencia.

No xAMp, falhas de processos receptores de mensagens são detectadas pelo emissor, conforme explicado na seção 3.4.1.7. Um processo falho é retirado dos grupos a que pertence e a nova visão é distribuída de forma atômica para todos os membros que restaram no grupo. Essa operação é realizada pelo protocolo de *membership* do xAMp, conforme o explicado na seção 3.4.1.6.

4.6.3 Falha do servidor primário

Conforme explicado na seção 2.5, a pior situação de falha em um servidor primário é a da falha que ocorre após o servidor primário haver enviado a mensagem para os secundários, mas antes de responder ao cliente, de forma que não há como saber se o envio de mensagens se completou ou não.

O xAMp, através da primitiva de envio de mensagem *reliableSend,* garante que mesmo em caso de falha do processo emissor (no caso o servidor primário) as mensagens são recebidas por todos os membros ativos do grupo. A maneira como isso é garantido está explicado na seção 3.4.1.3.

O comportamento esperado após a ocorrência de falha no servidor primário foi conforme as expectativas. Todos os servidores secundários receberam as mensagens e fizeram a modificação requerida no arquivo. Desta forma a integridade dos arquivos replicados foi mantida, o que permitiria a escolha de um novo servidor primário para cada grupo de replicação entre os secundários.

5 Conclusão

Este capítulo apresenta as conclusões do trabalho em relação à replicação de arquivos, protocolos de comunicação de grupos e utilização de comunicação de grupo para a disseminação de escritas para arquivos replicados. São discutidas também algumas idéias para a continuação deste trabalho.

5.1 Replicação de arquivos

A grande vantagem da utilização de arquivos replicados para tolerar falhas é o considerável aumento na disponibilidade dos dados, mesmo na ocorrência de falhas. A desvantagem é a dificuldade na manutenção da integridade dos dados replicados. Sistemas que comportam a replicação de dados precisam prover, ou utilizar, mecanismos de controle de réplicas para garantir a confiabilidade e a serialização na disseminação de operações de escrita, para que o conteúdo de todas as réplicas de um arquivo sejam iguais.

Na abordagem da cópia primária a característica de serialização das operações é garantida pelo fato de haver um processo centralizador que dissemina todas as operações de escrita. Neste caso só é necessário um mecanismo para garantir a confiabilidade nessa disseminação de escritas. O grande problema dessa abordagem reside na possibilidade de falha do servidor primário. Neste caso deve ser escolhido um novo servidor primário até que o antigo se recupere, e esta decisão deve ser comunicada aos clientes.

Na abordagem de cópias ativas, a implementação da disseminação de escritas é um pouco mais complicada do que na abordagem de cópia primária, pois as requisições de escritas podem partir de qualquer um dos servidores. Neste caso, além da ordenação, a estratégia adotada deve garantir, além da confiabilidade, a ordenação na disseminação de escritas. A falha de um dos servidor, no entanto, é mais fácil de ser mascarada, uma vez que processos clientes não conhecem apenas um servidor.

Apesar dos custos computacionais envolvidos, a replicação de arquivos é a melhor opção para sistemas cujas necessidades de alta disponibilidade de dados são importantes.

5.2 Protocolos de comunicação de grupo

Os sistemas apresentados no capítulo 3 implementam protocolos de comunicação de grupo que garantem entrega de mensagens *multicast* de forma ordenada e confiável.

Cada sistema tem suas características próprias em relação a tratamento de falhas, particionamento de rede, sobreposição de grupo, recuperação de processos e protocolos de *membership*.

5.3 Disseminação de escritas via *multicast*

A disseminação de escritas em ambiente replicados pode ser implementada de forma adequada utilizando comunicação de grupo.

Parte-se da idéia de que o conjunto de servidores que mantém cópia de um determinado arquivo forma um grupo de replicação. As alterações feitas neste arquivo são disseminadas ao grupo através de comunicação de grupo.

A abordagem de replicação da cópia primária, implementada no protótipo PDERM, exige do protocolo de comunicação de grupo apenas a garantia de confiabilidade (ou todos os servidores ativos recebem as modificações ou nenhum recebe). A garantia de atomicidade (ordenação) já é garantida pela característica de serialização, inerente à abordagem, de que todas as mensagens de alteração partem do mesmo servidor.

A abordagem de replicação de réplicas ativas, como não possui um processo centralizador, de onde partem todas as mensagens, necessita protocolos de comunicação de grupo que garantam além da confiabilidade, a atomicidade na entrega de mensagens *multicast*.

A disseminação de escritas em ambientes replicados, em qualquer das abordagens, pode ser implementada por qualquer um dos sistemas de comunicação de grupo apresentados, uma vez que todos implementam a comunicação entre processos de forma confiável e totalmente ordenada.

Tratando-se, particularmente da abordagem neste trabalho utilizada, a da cópia primária, nota-se que, como cada sistema de comunicação de grupo apresentado possui algumas peculiaridades no que diz respeito ao tratamento de falhas e *memebership*, podem haver variâncias em alguns aspectos do modelo. O objetivo principal na disseminação de escritas, de manutenção da integridade dos dados, pode ser alcançado através da utilização de qualquer dos protocolos analisados.

5.4 xAMp na disseminação de escritas

O sistema xAMp se adapta perfeitamente ao modelo de replicação escolhido, pois permite a sobreposição de grupos e a utilização de uma primitiva confiável de comunicação do servidor primário com os secundários. Os resultados alcançados no teste do protótipo foram satisfatórios, mesmo na ocorrência de falha do servidor primário.

5.5 Direções futuras

O protótipo apresentado no capítulo 4 implementa o algoritmo de replicação de cópia primária, e se preocupa apenas com a característica de seriabilidade de operações de escrita, de forma a manter a integridade dos dados replicados.

Um sistema que implementa a abordagem de replicação da cópia primária, no entanto, precisa se preocupar, além do algoritmo de replicação, com técnicas para o mascaramento de falhas do servidor primário.

Em caso e falha do servidor primário é indispensável para a manutenção da disponibilidade dos dados a escolha de um novo servidor primário, uma vez todas as operações (leitura, escrita e disseminação de escritas) são por ele realizadas. Uma vez que todos os servidores secundários de um grupo de replicação possuem os dados replicados no mesmo estado, qualquer um está apto a ser escolhido como servidor primário. A escolha do novo servidor primário deve ser comunicada aos clientes do sistema.

No momento em que o servidor primário se recupera da falha, é interessante que este se reintegre ao grupo de replicação e colete informações para a atualização do seu sistema de arquivos.

Após a reintegração do antigo servidor primário ao grupo, pode ser interessante que este retome o seu posto de servidor primário, por motivos como maior desempenho de I/O, por exemplo. Para isso deve ser desencadeada a escolha de um novo servidor primário.

A reintegração de servidor primário já foi tratada em um trabalho recentemente desenvolvido na UFRGS [PAS98]. As demais técnicas de mascaramento, no entanto, podem ser tratadas em trabalhos futuros, dando continuidade a este trabalho.

Bibliografia

[AMI 95] AMIR, Y. et al. The Totem single-ring ordering and membership protocol. **ACM Transactions on Computer System**, New York, v.13, n.4, p. 312-342, Nov. 1995.

[BIR 96] BIRMAN, K. P. **Building secure and reliable network applications**. Ithaca, N.Y.: Cornell University, 1996.

[CAS 86] CRISTIAN, F.; AGHILI, H.; STRONG, R. Clock Synchronization in the Presence of Omissions an Performance Faults, and Processor Joins. In: INTERNATIONAL SYMPOSIUM ON FAULT TOLERANT COMPUTING SYSTEMS, 16., 1986. **Proceedings...** [S.l.:s.n.],1986.

[CAS 94] CASIMIRO, Antônio. **xAMp Reference Manual (for version 3.1)**. Lisboa: INESC, University of Lisboa, 1994.

[DOL 96] DOLEV, D.; MALKI, D. The Transis approach to high availability Cluster Communication. **Communications of the ACM**, NewYork, v.39, n.4, Apr. 1996.

[GAR 91] GARCIA-MOLINA, H.; SPAUSTER, A. Ordered and reliable multicast communications. **ACM Transactions on Computer System**, New York, v.9, n.3, Aug. 1991.

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

[HOF 97] HOFSETZ, B. F. **Estudo de protocolos de multicast confiável:** Trabalho Individual. Porto Alegre: CPGCC da UFRGS, 1997.

[HUA 89] HUANG, Y.; JALOTE, Pankaj. Avaliability Analysis of the Primary Site Approach for Fault Tolerance. **Acta Informatica,** Berlin, n. 26, p.543-557,1989.

[JAL 94] JALOTE, P. **Fault tolerance in distributedc systems.** [S.l.]: Prentice Hall, 1994.

[LEB 96] LEBOUTE, Mário. **RNFS - Um sistema de arquivos distribuído tolerante a falhas para o UNIX.** Porto Alegre: CPGCC da UFRGS, 1996. Dissertação de Mestrado.

[MAC 95] MACÊDO, R. J. A.; EZHILCHELVAN, P. D. Reability aspects of multicast protocols. In: SIMPÓSIO DE COMPUTADORES TOLERANTES A FALHAS, 6., 1995, Canela, RS. **Anais**... Canela: [s.n.], 1995.

[MOS 95] MOSER, L. E. et. al. **The Totem system.** California: University of California, Dept. of Electrical an Computer Engineering, 1995. Technical Report.

[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, Apr. 1996.

[MUL 93] MULLENDER, S. **Distributed systems**. New York: Addison-Wesley, 1993. p 448 - 490.

[PAS 98] PASIN, Marcia, **Reintegração de servidores em sistemas distribuídos.** Porto Alegre: CPGCC da UFRGS, 1998. Dissertação de Mestrado.

[POW 96] POWELL, D., Group Comunications. **Communications of the ACM**, New York, v.39, n.4, Apr. 1996.

[REN 95] RENESSE, R. V., BIRMAN, K. P. **Protocol composition in Horus.** [S.l.]: Cornell University, Dept. of Computer Science, 1995. (Technical Report 95-1505).

[REN 96] RENESSE, R. V., BIRMAN, K. P., MAFFEIS, S., Horus: a flexible group communication system. **Communications of the ACM**, New York, v.39, n.4, Apr. 1996.

[ROD 92a] RODRIGUES, L.; VERÍSSIMO, P. xAMp: a multi-primitive group communications service. In: SYMPOSIUM ON RELIABLE DISTRIBUTED SYSTEMS, 11.,1992. **Anais...** Houston, Texas: IEEE, 1992.

[ROD 92b] RODRIGUES, L.; VERÍSSIMO, P.; RUFINO, J. **A low-level processor group membership protocol for LANS.** Lisboa: INESC, 1992. (RT/92).

[SCH 90] SHNEIDER, F. B. Implementing fault-tolerance services using the state machine approach: a tutorial. **ACM Computing Surveys**, New York, v.22, n.4, Dec. 1990.

[TAN 92] TANENBAUM, Andrew S. **Modern Operating Systems.** Englewood Clifs, NJ.: Prentice-Hall, 1992. 728p.

[VOG 92] VOGELS, W.; RODRIGUES, L.; VERÍSSIMO, P. Fast group communications for standard workstations. In: OPENFORUM, 1992. **Proceedings...** Utrecht Netherlands:[s.n.], 1992.