Port System
Sistema de Comunicação em Grupo
para o Ambiente Xchart

Este exemplar corresponde à redação final da tese devidamente corrigida e defendida pelo Sr. **Edilmar Lima Alves**, e aprovada pela Comissão Julgadora.

Campinas, **15** de **março** de **1996**

Prof. Dr. _____
Orientador

Prof. Dr. _____
Co-Orientador

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, UNICAMP, como requisito parcial para a obtenção do Título de **MESTRE** em Ciência da Computação.

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Alves, Edilmar Lima

AL87p Port system: sistema de comunicação em grupo para o ambiente Xchart / Edilmar Lima Alves. -- Campinas, [S.P. :s.n.], 1996.

Orientadores : Hans Kurt Edmund Liesenberg, Luiz Eduardo Buzato.

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Matemática, Estatística e Ciência da Computação.

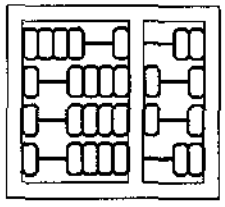
1. Sistemas de comunicação. 2. Sistemas operacionais distribuídos (Computadores). 3. Interfaces de usuário (Sistema de computador). I. Liesenberg, Hans Kurt Edmund. II. Buzato, Luiz Eduardo. III. Universidade Estadual de Campinas. Instituto de Matemática, Estatística e Ciência da Computação. IV. Título.

Tese de Mestrado defendida e aprovada em 15 de março de 1995 pela Banca Examinadora composta pelos Profs. Drs.

Prof (a). Dr (a).

Prof (a). Dr (a).

Prof (a). Dr (a).



Universidade Estadual de Campinas
Departamento de Ciência da Computação

Instituto de Matemática, Estatística e Ciência da

Computação

Port System
Sistema de Comunicação em Grupo para o Ambiente Xchart

Edilmar Lima Alves

Prof. Dr. Hans Kurt Edmund Liesenberg (orientador)

Prof. Dr. Luiz Eduardo Buzato (co-orientador)

Cidade Universitária Zeferino Vaz, Campinas, 16 de fevereiro de 1996

Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, Universidade Estadual de Campinas, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

Port System
Sistema de Comunicação em Grupo para o Ambiente Xchart[1]

Edilmar Lima Alves[2]

Departamento de Ciência da Computação

IMECC - UNICAMP

Banca Examinadora:

- Luiz Eduardo Buzato (Co-Orientador)[3]
- Maria Beatriz Felgar de Toledo[4]
- Leo Pini Magalhães[5]
- Ricardo de Oliveira Anido (Suplente)[6]

[1] Dissertação apresentada ao Instituto de Matemática, Estatística e Ciência da Computação, Universidade Estadual de Campinas, como requisito parcial para obtenção do título de Mestre em Ciência da Computação.

[2] Bacharel em Ciência da Computação pela Universidade Federal de Mato Grosso do Sul.

[3] Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

[4] Professora do Departamento de Ciência da Computação - IMECC - UNICAMP.

[5] Professor do Departamento de Computação e Automação - FEE - UNICAMP.

[6] Professor do Departamento de Ciência da Computação - IMECC - UNICAMP.

Agradecimentos

À minha esposa, Michelle, que sempre soube compreender as dificuldades e os desafios de uma pós-graduação, e me deu um apoio fundamental para que eu pudesse chegar a este ponto.

Aos meus pais (Iza e Edemar), de quem sempre recebi o carinho e os ensinamentos mais puros, e que nunca perderão seu valor.

À minha família e amigos de Campo Grande que, apesar da distância que nos separa, conseguiram manter um estreito relacionamento com a minha pessoa e nunca serão esquecidos durante a minha vida.

Aos meus colegas de curso, Luciana, Maurício "Eras", Ronaldo, Paulo Drummond, Arturo, Erasmo, Raul, Edmar e Loren, que sempre me ajudaram a superar os momentos de solidão em Campinas.

Aos professores Hans Liesenberg, Luiz Eduardo Buzato e Ricardo Anido, e ao aluno de doutorado Fábio Nogueira de Lucena que, em momentos diferentes, sempre ajudaram no desenvolvimento do corrente trabalho.

Ao Conselho Nacional de Pesquisa (CNPq) e à Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), por financiarem este projeto, permitindo que eu tivesse a tranquilidade necessária para o desenvolvimento deste trabalho.

Aos meus amigos do "Céu", presença marcante em toda a minha vida.

Resumo

O objetivo do corrente trabalho é fornecer um sistema de comunicação em grupo que servirá de base para o desenvolvimento de outros subsistemas do Ambiente Xchart.

O Ambiente Xchart contém um conjunto de ferramentas que permitem a especificação e a implementação de interfaces homem-computador concorrentes, ou seja, interfaces que podem ser divididas em várias sub-interfaces. Cada sub-interface pode ser executada em um computador diferente de um sistema distribuído. Sendo assim, é necessária a existência de um ambiente de controle da execução distribuída de sub-interfaces, denominado Gerente de Distribuição.

Este trabalho implementa o subsistema de mais baixo nível do Gerente de Distribuição, e é formado pelos seguintes módulos: Sistema de Comunicação (responsável pela troca de mensagens com o sistema distribuído), Sistema de Detecção de Falhas (detecta falhas na comunicação e avisa o Sistema de Comunicação), Servidor de Grupos (responsável pela manutenção de grupos e envio de *multicast* para um grupo de processos), e Servidor de Nomes (fornece a transparência de localização de recursos compartilhados no sistema distribuído).

Abstract

The aim of this work has been the implementation of a group communication system, that will be the base for development of other subsystems of the Xchart Environment.

The Xchart Environment provides a set of tools for specification and implementation of concurrent human-computer interfaces, interfaces that may be divided in many sub-interfaces. Each sub-interface may be run in a different computer of a distributed system. Thus, an environment to control the distributed execution of the sub-interfaces, named Distribution Manager, is needed.

This work implements the lowest level subsystem of the Distribution Manager, and it is composed of the following modules: Communication System (responsible for the exchange of messages with the distributed system), Failure Detection System (it detects failures in the communication and notifies the Communication System), Group Server (responsible for keeping groups and for multicast sending for a group of processes), and Name Server (it provides location transparency of shared resources of a distributed system).

Conteúdo

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

A disponibilidade de computadores equipados com componentes multimídia (monitores *bitmapped*, *mouse*, alto-falantes, microfones, etc) tem permitido a construção de interfaces homem-computador mais amigáveis.

A evolução dos sistemas de computação, com a criação de processadores menores e mais potentes e *hardware* de comunicação, permitiu a construção de sistemas distribuídos. Aplicações distribuídas são aquelas onde as várias funções computacionais (processos) e os vários recursos (memórias, arquivos, impressoras, etc) estão espalhados em um ambiente de rede de computadores.

Dentro do contexto de interfaces e também distribuição, o Projeto Xchart está desenvolvendo um ambiente de especificação e implementação de interfaces homem-computador, o ambiente Xchart, que facilita o desenvolvimento de interfaces complexas, com vários componentes concorrentes e possivelmente distribuídos.

Port System é o subsistema do ambiente Xchart responsável pela implementação dos serviços de comunicação entre grupos de objetos e resolução de nomes. É importante ressaltar que, apesar deste trabalho participar da composição do ambiente Xchart, suas rotinas podem ser utilizadas de forma independente por aplicações que necessitam de grupos e/ou nomes, sem a necessidade de adição de outros subsistemas presentes neste ambiente.

1.1 Projeto Xchart

O Projeto Xchart visa a construção de um ambiente de programação para interfaces homem-computador concorrentes. Uma interface concorrente é uma composição de sub-interfaces ativas concorrentemente, possivelmente distribuídas, e que interagem através de algum mecanismo de comunicação. Sistemas interativos responsáveis por manter várias apresentações distintas de uma mesma informação, ou ainda aqueles que permitem a interação contínua com o usuário enquanto a aplicação é executada concorrentemente, são exemplos comuns destas interfaces. A linguagem Xchart é utilizada na especificação de interfaces desta natureza e o ambiente Xchart contém ferramentas que apóiam o projeto e a implementação dos programas para estas interfaces.

O projeto da linguagem Xchart decorre de experiências anteriores com *Statecharts* [23] no desenvolvimento do controle de interfaces [8]. As dificuldades encontradas no uso de *Statechart* motivaram extensões/alterações que constituem a base para a definição de Xchart. A notação *Statechart* foi desenvolvida para especificar sistemas reativos em geral, e as interfaces homem-computador constituem um subconjunto de sistemas reativos, com certas peculiaridades. A Seção 1.2 define sistemas reativos de maneira geral. Em seguida, a Seção 1.3 descreve as características que tornam a linguagem Xchart apropriada ao desenvolvimento de interfaces.

1.2 Sistemas Reativos

Os sistemas podem ser classificados, em termos de seu comportamento, como transformantes ou reativos:

- Transformante (orientado a dados): obtém a saída através de transformações (funções) aplicadas à entrada (ex. compiladores, folhas de pagamento, previsão meteorológica, etc).

  Um sistema transformante (Figura 1.1*i*) obtém a saída $S = F(E)$ através da aplicação da função F (transformação) à entrada E e, em seguida, termina sua execução.

- Reativo (controlado por eventos): continuamente responde (reage) a estímulos externos e internos. Em geral, não computa uma única

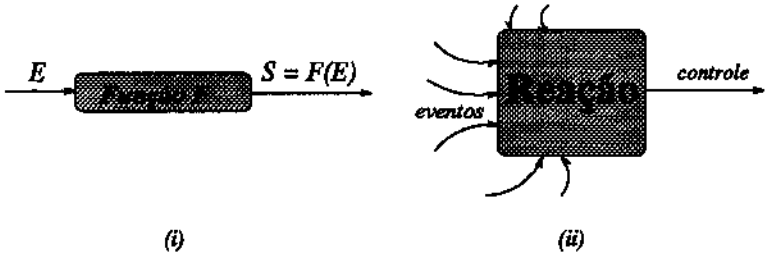


Figura 1.1: Sistemas Transformante *(i)* e Reativo *(ii)*

função, mas mantém um relacionamento ininterrupto com o ambiente onde é executado, ou seja, normalmente não possui uma condição de término. A reação (saída) é conseqüência do tratamento da entrada (eventos) e compreende o controle do sistema, isto é, seqüência de operações a serem realizadas (ex. sistemas de comunicação, interfaces homem-computador, caixa automático de bancos, etc).

Um sistema reativo (Figura 1.1*ii*) gera sinais de controle para o ambiente conforme a ocorrência de eventos de entrada.

1.2.1 Camadas de um Sistema Reativo

Um sistema reativo está continuamente esperando por eventos. Quando um evento ocorre, ele analisa o contexto no qual o evento ocorreu e executa algum processamento. Por exemplo, em uma interface homem-computador, o evento "pressionar botão esquerdo do *mouse*" gera uma ação que depende da posição da tela onde ele ocorreu (contexto da interface).

Conceitualmente, sistemas reativos podem ser subdivididos três camadas [1] (Figura 1.2):

- Apresentação: responsável pela captação de eventos de entrada e emissão de eventos de saída, traduzindo eventos externos em internos e vice-versa. Esta camada processa eventos de entrada e de saída gerados no ambiente (análise léxica).

- Diálogo ou Núcleo Reativo: contém a estrutura de controle (lógica ou comportamento) do sistema reativo. Decide qual a reação a ser produzida em função do evento de entrada e do contexto atual do sistema (análise sintática).

- Aplicação: contém a funcionalidade do sistema, ou seja, as ações passíveis de serem disparadas pelo Diálogo (ação semântica).

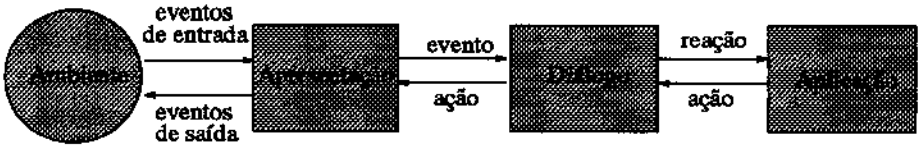


Figura 1.2: Camadas de um Sistema Reativo

Um sistema reativo pode ser descrito segundo a relação:

reação (evento, configuração) = (ação, nova configuração)

Uma configuração de um sistema reativo corresponde a um contexto atual em que o sistema se encontra em um determinado momento. Quando ocorre um evento, o sistema reage de acordo com a dupla *(evento, configuração)*. Esta reação pode resultar na execução de uma ação e na transição do sistema para uma nova configuração. Devido ao efeito de uma reação ser dependente da dupla *(evento, configuração)*, em princípio é necessário analisar todas as combinações válidas de eventos de entrada e configurações, a fim de modelar a reação. Como sistemas reativos costumam apresentar uma grande variedade de eventos de entrada e configurações possíveis, o controle do processo de reação (diálogo) é complexo [9]. Daí surge a necessidade da utilização de uma notação que facilite a especificação de diálogos de sistemas reativos.

1.3 Linguagem Xchart

A linguagem Xchart pretende atender aos requisitos de especificação de diálogos de interfaces homem-computador concorrentes.

1.3.1 Caracterização da Linguagem Xchart

Existem alguns conceitos básicos sobre os quais a linguagem Xchart é baseada. São eles:

- **Estado**: representa um contexto, uma situação de momento, em que se encontra um sistema, ou parte dele, em um intervalo entre a ocorrência

de dois eventos sucessivos. Um estado é representado graficamente por um retângulo com cantos arredondados, sendo seu nome escrito em negrito. Existem três tipos de estados. Estados do tipo **OR** possuem sub-estados com exclusão mútua de ativação, indicados por retângulos de contorno contíguo (Figura 1.3*i*). Estados do tipo **AND** possuem sub-estados concorrentes, ativados e desativados simultaneamente, indicados por retângulos de contorno tracejado (Figura 1.3*ii*). Finalmente, estados do tipo **BASIC** são estados sem sub-estados, indicados por retângulos com contorno contíguo. Na Figura 1.3, os estados **Or-A**, **Or-B**, **And-A**, **And-B** e **And-C** são do tipo **BASIC**.

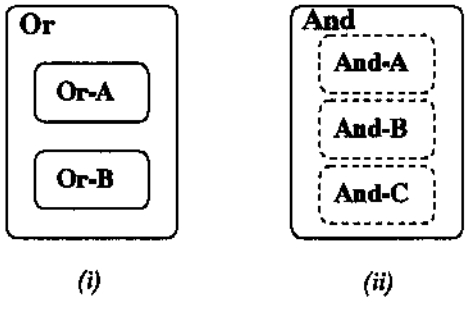


Figura 1.3: Tipos de Estados: *(i)* **OR** e *(ii)* **AND**

- **Configuração**: conjunto de estados ativos em um determinado instante de tempo.
- **Evento**: consiste em uma entrada ou saída do sistema reativo, que varia com a natureza do sistema modelado. Compreende qualquer ocorrência notável por um sistema reativo.
- **Aresta**: arco orientado usado para denotar a ocorrência de um evento e a transição entre estados.
- **Ação**: é gerada após a ocorrência de um evento, sendo um processo atômico que, conceitualmente, não consome tempo. Permite a execução de tarefas associadas a transições ou à ativação e desativação de estados.
- **Atividade**: corresponde a um processo não-atômico, que consome tempo e permite a execução de tarefas durante o período em que se encontra ativo o estado ao qual está associado (semântica de um estado).

Conceitualmente, uma atividade é iniciada no instante da ativação do estado associado e é interrompida por ocasião de sua desativação, caso não tenha sido concluída.

- **Rótulo**: corresponde às condições de disparo da transição descrita por uma aresta. Sua sintaxe é $\alpha[\sigma]/\beta$, onde:
  - α é o evento necessário para que a transição ocorra, se o estado origem da aresta estiver ativo;
  - σ é a condição de guarda que inibe o disparo da transição, caso seja falsa;
  - β é a ação executada quando ocorre o disparo da transição.

  Os três elementos de um rótulo são opcionais. Se o rótulo é vazio, a transição é dita automática e é disparada quando concluída a atividade associada ao estado origem da aresta. Uma transição sem estado origem é dita *default* (Figura 1.4*ii*) e indica o estado a ser ativado, caso não haja um candidato de ativação pré-determinado. Um estado destino de uma transição *default* é chamado estado inicial.

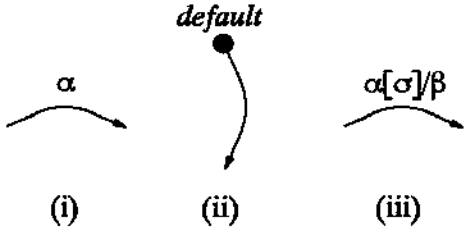


Figura 1.4: Arestas e Alguns Rótulos Típicos

- **Hierarquia**: o conjunto de estados e seus sub-estados podem ser dispostos em níveis, sob a forma de estados encadeados, formando a hierarquia de estados do sistema. A Figura 1.5 mostra as hierarquias de estados referentes aos Xcharts da Figura 1.3.
- **Concorrência**: modelada por estados **AND** que são ativados e/ou desativados concorrentemente.

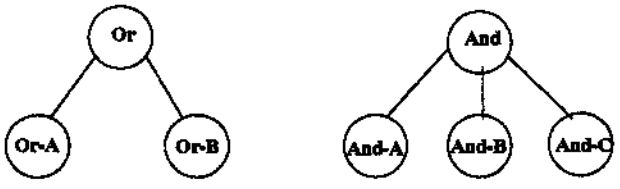


Figura 1.5: Hierarquia de Estados

- **Comunicação**: a comunicação (e sincronização) entre estados concorrentes se dá através da geração de eventos internos. Um evento interno é gerado por uma atividade de um estado concorrente, e pode ter reflexos em outros estados.

- **Variáveis**: uma variável é um repositório para um valor inteiro, possivelmente utilizado em expressões e combinado com operações matemáticas básicas. Memória pode ser compartilhada entre estados de um Xchart através de variáveis. Na Figura 1.6, a variável x é comum aos estados **Entrada** e **Segurança**. Após o nome do usuário ser fornecido, a senha é verificada e, se for inválida, o evento correspondente é gerado. Na quarta vez, o estado **Segurança** detecta esta situação pela condição $[x > 3]$ e a transição para **Procedimento especial** ocorre.

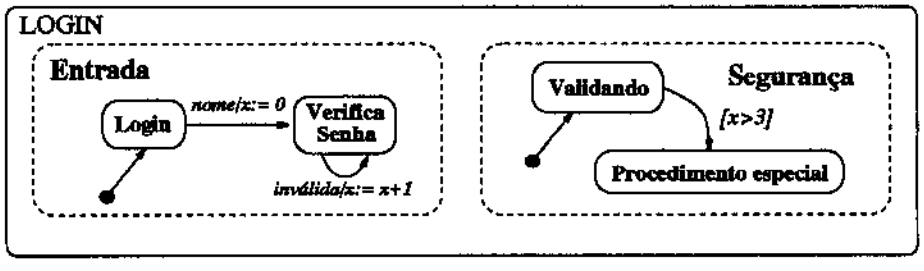


Figura 1.6: Compartilhamento de Variáveis

As ações disparadas quando da ocorrência de eventos são executadas atomicamente. Na Figura 1.7, suponha que os estados **A** e **D** estejam ativos no Xchart **X**. Se ocorrer o evento e, as transições exibidas serão avaliadas. Como reação, as ações que compartilham as variáveis x e y serão executadas. Logo, deve existir um controle de concorrência implícito nas reações e os novos valores devem ser $x' = x-4$ e $y' = y+4$.

Ao final das reações, ou os valores de x e y serão os citados acima, ou $x' = x - 2$ e $y' = y + 2$ (uma das ações é abortada) ou ainda, não serão alterados (ambas abortadas).

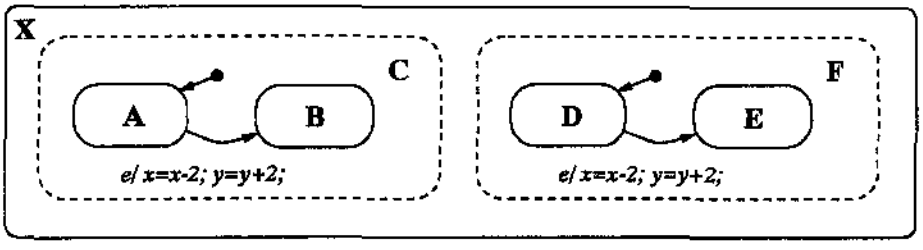


Figura 1.7: Ações Atômicas

A linguagem Xchart é definida por um vocabulário visual que inclui diagramas hierárquicos de estados, transições e outros elementos visuais que representam diversos modos de interação e execução de diálogos.

1.3.2 Especificação de Interfaces Concorrentes

Como visto anteriormente, uma interface concorrente é formada por um conjunto de sub-interfaces ativas simultaneamente. A linguagem Xchart permite a especificação do diálogo de cada sub-interface e da interação entre sub-interfaces. A especificação de cada sub-interface corresponde a um Xchart $\mathbf{x_i}$ ($1 \leq i \leq n$, onde n é o número de sub-interfaces da interface concorrente). Segundo a Figura 1.8, o sistema completo pode ser representado conceitualmente por um estado **X**, onde cada um dos seus sub-estados representa o Xchart de uma das sub-interfaces ativas.

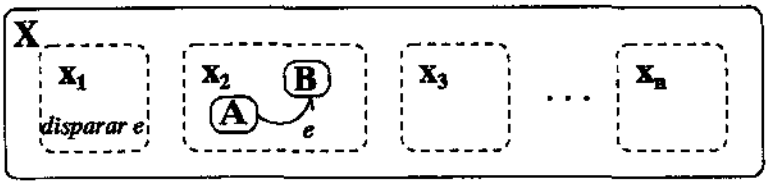


Figura 1.8: Especificação de uma Interface Concorrente

É possível que ocorra alguma interação entre sub-interfaces, através de um mecanismo de comunicação. A Figura 1.8 ilustra a transição de **A** para **B** no Xchart $\mathbf{x_2}$, causada pela ocorrência do evento e em $\mathbf{x_1}$.

A seguir, é descrito o modelo de execução de Xchart, de forma a mostrar como a especificação de uma interface concorrente comporta-se durante a execução.

1.4 Modelo de Execução de Xchart

As sub-interfaces de uma interface podem estar dispersas em um sistema distribuído, interagindo através de um Ambiente de Execução. A Figura 1.9*i* apresenta m sub-interfaces, possivelmente distribuídas, sendo executadas em n processadores $P_1, P_2, ..., P_n$, com $m \neq n$, em geral. A Figura 1.9*ii* mostra as camadas do Ambiente de Execução, que deve possuir um Núcleo Reativo para controlar o comportamento das sub-interfaces, e um Gerente de Distribuição para apoiar a execução distribuída das sub-interfaces.

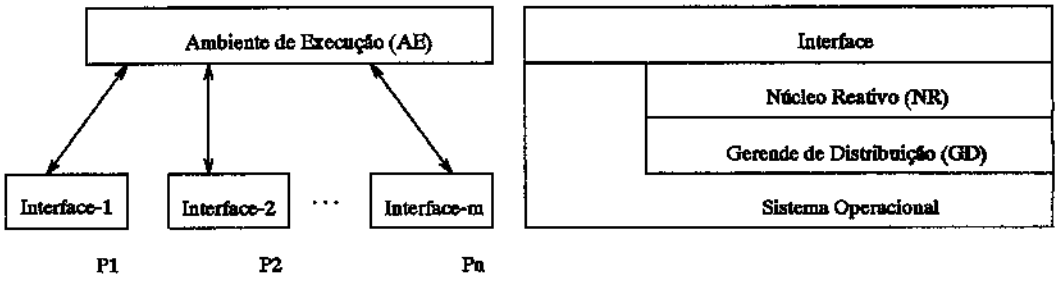


Figura 1.9: Arquitetura do Sistema de Execução

As interfaces utilizam os recursos do Ambiente de Execução, em particular o Núcleo Reativo e o sistema operacional. Quando há a necessidade de alguma operação distribuída, o Núcleo Reativo interage com o Gerente de Distribuição.

O cenário mais simples de execução é aquele onde a interface é composta por apenas uma sub-interface. Neste caso, não é necessária a presença do Gerente de Distribuição, pois todas as operações são locais e internas ao Xchart que compõe a interface. Um caso mais complexo pode ser aquele onde existem duas ou mais sub-interfaces, mas em execução em um único computador. O Gerente de Distribuição também não é necessário aqui, pois

as interações entre os Xcharts podem ser otimizadas localmente pelo Núcleo Reativo. Somente no caso mais geral, no qual existem várias sub-interfaces espalhadas por vários computadores, é que se torna necessária a utilização do Gerente de Distribuição.

1.4.1 Requisitos de Distribuição

Uma interface concorrente pode apresentar características que justificam uma implementação distribuída. Por exemplo, a edição compartilhada de um texto por dois usuários torna necessária a existência de, pelo menos, duas sub-interfaces, e um controle de concorrência no acesso ao texto. Neste caso, é necessário que o Gerente de Distribuição forneça um conjunto de funcionalidades que permitam o controle, por parte dos Núcleos Reativos, da interação entre as sub-interfaces distribuídas.

O Gerente de Distribuição implementa os seguintes componentes para a utilização pelo Núcleo Reativo:

- Sistema de Comunicação em Grupo: implementa mecanismos para a comunicação entre sub-interfaces distribuídas de uma interface concorrente. Por exemplo, a ocorrência de um evento em uma sub-interface pode ser difundida através de troca de mensagens entre o Núcleo Reativo local e o grupo de Núcleos Reativos que controla as outras sub-interfaces.

- Sistema de Processamento de Transações Atômicas: permite a utilização de variáveis compartilhadas e transações atômicas entre sub-interfaces distribuídas.

- Serviço de Nomes: existem recursos criados durante a especificação de uma interface concorrente que devem ser conhecidos globalmente no ambiente de execução. Por exemplo, um evento que ocorre em uma sub-interface e pode causar uma transição em uma sub-interface de outro computador, ou uma variável compartilhada entre sub-interfaces, são recursos típicos para serem cadastrados em um Serviço de Nomes. Desta forma, qualquer sub-interface pode localizar e ter acesso a um recurso a partir de seu nome global.

É importante ressaltar que o Núcleo Reativo implementa internamente comunicação e transações, quando a interação entre as sub-interfaces é local. O Gerente de Distribuição é usado normalmente para operações remotas.

1.5 Ambiente Xchart

Até o momento, descreveu-se apenas os componentes necessários à especificação e implementação do diálogo de interfaces concorrentes. Contudo, para o desenvolvimento completo de uma interface é necessário implementar a apresentação e a aplicação. O ambiente Xchart possui ferramentas para auxiliar a confecção da apresentação e do diálogo. O código da aplicação é de responsabilidade do projetista do sistema.

Para o desenvolvimento da apresentação de uma interface, o ambiente Xchart fornece um construtor de apresentações denominado *Grace*. O projetista especifica graficamente a apresentação e *Grace* gera um conjunto de classes C++ [31] que a implementa em um determinado ambiente operacional (Figura 1.10).

Figura 1.10: Especificação da Apresentação

A especificação do diálogo de uma interface é baseada em um construtor de diálogos chamado *Smart*. O projetista utiliza a linguagem Xchart para especificar o diálogo e *Smart* gera a descrição textual correspondente, denominada TeXchart, que é independente de arquitetura e de linguagem de programação. Em seguida, a descrição textual passa por um interpretador de TeXchart que pode gerar, por exemplo, código C++ equivalente (Figura 1.11).

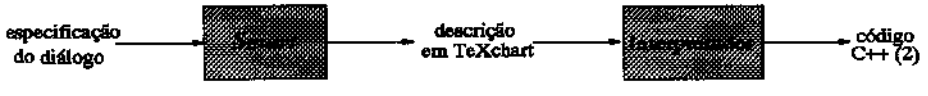


Figura 1.11: Especificação do Diálogo

Dado o resultado da especificação (código da apresentação gerado por *Grace*, código do diálogo gerado por *Smart* e código da aplicação feito pelo projetista), é necessário reunir e organizar estas informações de forma a produzir código executável. A Figura 1.12 descreve o fluxo de operações até o código final. Em primeira instância, o código C++ gerado por *Smart* é pré-processado pelo *Xchart Stub*, para gerar uma estrutura distribuída do mesmo, com lado cliente e lado servidor. Este passo é necessário se existe interação entre sub-interfaces no ambiente distribuído. O próximo passo é a compilação dos códigos C++ e a geração dos códigos-objeto. Em seguida, os códigos-objeto são ligados com as bibliotecas Xchart (Núcleo Reativo e, opcionalmente, Gerente de Distribuição) para gerar um ou mais executáveis. Apenas um executável é necessário se todas as sub-interfaces estarão no mesmo computador. Se existir duas ou mais sub-interfaces em diferentes computadores, então o *Xchart Stub* monta a estrutura cliente/servidor e o Gerente de Distribuição é ligado ao código-objeto.

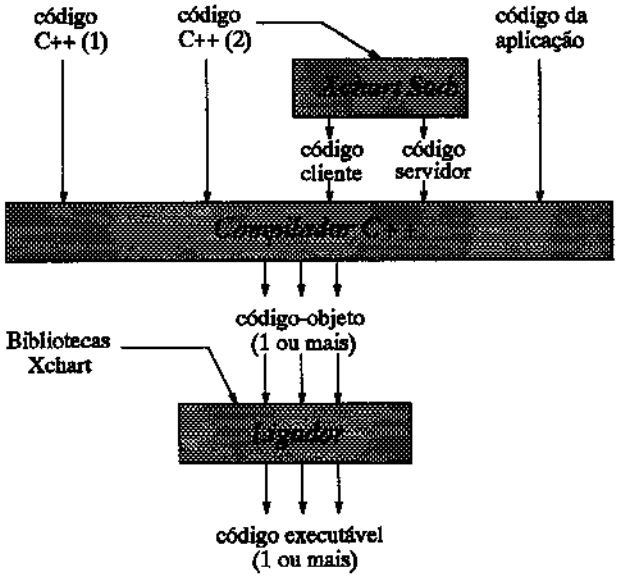


Figura 1.12: Geração dos Programas da Interface

Para controlar a execução da interface concorrente gerada, as bibliotecas Xchart realizam as seguintes tarefas:

- **Núcleo Reativo**: controla o diálogo da interface, ou seja, o processo

de ativação/desativação de estados e a transição de estados internos a cada Xchart, realiza as interações entre Xcharts locais e utiliza o Gerente de Distribuição para processar as interações entre um Xchart local e outro remoto.

- **Gerente de Distribuição**: apóia as interações não locais. O Gerente de Distribuição é dividido em dois componentes:

  1. **Sistema de Processamento de Transações Atômicas**: conjunto de rotinas para inicialização, finalização e aborto de transações atômicas, e controle de concorrência de acesso aos recursos usados nas ações internas da transação [4].
  2. **Serviço de Grupos** e **Serviço de Nomes**: implementam as funcionalidades (grupos e nomes) requeridas pelo Sistema de Processamento de Transações Atômicas e pelo Núcleo Reativo. A implementação destes serviços corresponde ao corrente trabalho (*Port System*).
     (a) **Serviço de Grupos**: rotinas de criação/remoção de grupos, inserção/remoção de membros de um grupo e envio de mensagens para grupos.
     (b) **Serviço de Nomes**: centralizador das informações sobre localização de todos os recursos usados no ambiente distribuído. É acionado quando da necessidade de inserção/remoção/localização de recursos.

1.6 Resumo

Neste capítulo, foi descrito o contexto onde *Port System* se situa. O ambiente Xchart pretende fornecer um conjunto de ferramentas que permitam especificar e implementar interfaces concorrentes. Como as sub-interfaces que compõem uma interface deste tipo podem estar distribuídas por uma rede de computadores, é necessária a presença de um Gerente de Distribuição, para controlar as interações entre sub-interfaces. O sistema *Port System* implementa o Serviço de Grupos e o Serviço de Nomes, que fazem parte do Gerente de Distribuição.

Capítulo 2

Sistemas Distribuídos

Neste capítulo são discutidos conceitos de sistemas distribuídos. Inicialmente, são descritas a estrutura e a funcionalidade de sistemas distribuídos em geral. Em seguida, destaca-se a arquitetura de um sistema operacional distribuído, e descreve-se os seus principais componentes.

2.1 Caracterização de Sistemas Distribuídos

Atualmente, um sistema de computação pode ser classificado, segundo a topologia do ambiente de execução, em centralizado ou distribuído. Um sistema centralizado é executado em um único computador, sendo seu processamento independente da interação com outros sistemas. Um sistema distribuído é formado por um conjunto de subsistemas interligados, que interagem através de um sistema de comunicação para realizar uma tarefa. Com isso, o que distingue um sistema distribuído de um sistema centralizado é a presença de um sistema de comunicação e a ausência de memória compartilhada [28].

Um sistema distribuído possui vantagens e desvantagens. Entre as vantagens estão [32]:

- crescimento incremental: módulos de *hardware* e *software* podem ser gradativamente inseridos no sistema, em pequenos incrementos, aumentando o desempenho do mesmo;
- confiabilidade: um sistema distribuído pode ser implementado de forma a tolerar a ocorrência de falhas parciais, ou seja, a falha de

algum de seus componentes. Um sistema distribuído tolerante a falhas deve possuir redundância de funcionalidade entre seus componentes de forma que, na falha de um componente, o sistema possa continuar sua execução;

- compartilhamento de recursos: recursos como bases de dados, impressoras, e CD-ROMs podem ser compartilhados mais facilmente pelos usuários de um sistema distribuído.

Entre as desvantagens na distribuição estão [32]:

- poucas ferramentas de desenvolvimento de aplicações distribuídas;
- dificuldade com a administração e a segurança do ambiente distribuído.

Para limitar estas desvantagens, é necessário prover ferramentas para facilitar o desenvolvimento e a administração de sistemas distribuídos. Além disso, o desenvolvimento de novas tecnologias de processamento e, principalmente, de comunicação de dados, podem colaborar para a melhoria de sistemas distribuídos.

2.1.1 Projeto de Sistemas Distribuídos

Um sistema distribuído deve atender a dois requisitos fundamentais: transparência e tolerância a falhas.

Transparência

Um sistema implementado de forma a esconder dos seus usuários a forma como as operações são executadas no ambiente distribuído é dito um sistema transparente. A transparência é um requisito básico no projeto de sistemas distribuídos, pois ela permite a construção da abstração de sistema centralizado, facilitando a visão do usuário.

Existem vários tipos de transparências, relacionadas a diferentes aspectos de um sistema distribuído. Entre as mais comuns estão:

- localização: um usuário de um recurso do sistema não precisa preocupar-se em saber onde o recurso está localizado. A partir de uma referência ao recurso (nome ou identificador), o sistema encarrega-se de localizá-lo para o usuário.

- acesso: um usuário não se preocupa com a maneira como um recurso é manipulado fisicamente em um determinado computador. O sistema mascara diferenças de representação de dados e mecanismos de interação entre computadores de diferentes arquiteturas.

- concorrência: os usuários podem compartilhar recursos sem se preocupar com a concorrência e a manutenção da consistência dos mesmos.

- replicação[1]: podem existir várias cópias de um mesmo recurso espalhadas no ambiente distribuído, para fins de tolerância a falhas e/ou facilidade de acesso, sem que o usuário tome conhecimento disso.

- falhas: o sistema mascara a ocorrência de falhas no ambiente distribuído.

Várias combinações destas transparências são encontradas em sistemas distribuídos. Porém, a adição de cada uma delas em um sistema distribuído tem o seu respectivo custo de implantação, gerando um aumento da complexidade do projeto.

Tolerância a Falhas

Um sistema de computação pode ser decomposto em um conjunto de componentes que interagem para realizar uma tarefa. Para garantir uma execução confiável do sistema, sua especificação deve tratar aspectos de tolerância a falhas das operações executadas em cada componente. Uma falha de um componente do sistema ocorre quando, durante sua execução, seu comportamento difere do que foi especificado [13]. Quando um componente falha, o sistema passa para um estado de erro. Um estado de erro pode levar o sistema a apresentar um defeito [27].

A Figura 2.1 representa a execução de um sistema através de diagramas de transição de estados, sendo possível especificar pontos passíveis de ocorrência de defeitos através da análise destes diagramas. Uma transição errônea é uma transição de estado para a qual um defeito subseqüente pode ser atribuído, caso uma ação corretiva não seja executada. Um estado errôneo é um estado que pode gerar um defeito. Nesta figura, é mostrada uma transição errônea

[1]Durante a dissertação este termo é usado como um sinônimo de duplicação (cópias), em conjunto com o verbo replicar. Em inglês: *Replication*.

T_1 e uma possível ação corretiva T_2. Caso T_2 não seja executada, o sistema tenderá a uma situação de defeito.

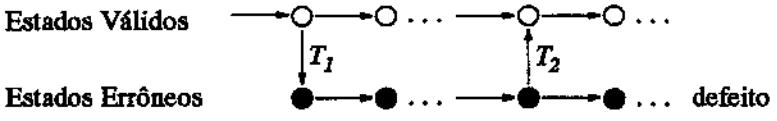


Figura 2.1: Diagrama de Transição de Estados e a Especificação de Falhas

— Classificação de Falhas

Uma possível classificação de falhas de componentes de um sistema é a seguinte [13, 26, 30]:

- falha-e-parada (*fail-stop*): componente do sistema de computação pára, após notificar o sistema que está em um estado errôneo. É o tipo mais simples de falha, quando um problema em um componente causa reflexos bem definidos na execução do resto do sistema.
- omissão (*omission*): componente nunca responde a um pedido gerado por outro. Isto pode ocorrer por problemas com o componente ou por sobrecarga de tarefas mais urgentes.
- temporização (*timing*): componente produz um resultado antes ou depois do prazo especificado.
- arbitrária (*bizantine*): componente responde de modo arbitrário às solicitações que recebe. É o caso mais geral de falha.

2.2 Arquitetura de um Sistema Operacional Distribuído

Uma aplicação distribuída pode ser implementada utilizando-se funções fornecidas por subsistemas básicos de um sistema operacional distribuído. A Figura 2.2 ilustra a arquitetura de um sistema operacional típico; os subsistemas estão arranjados em camadas, onde as linhas horizontais representam interfaces. Assim, uma aplicação distribuída tem acesso aos diversos níveis

de serviços implementados pelo sistema operacional distribuído, desde aqueles oferecidos pelos componentes do núcleo até os oferecidos por componentes de nível superior como, por exemplo, um sistema de arquivos.

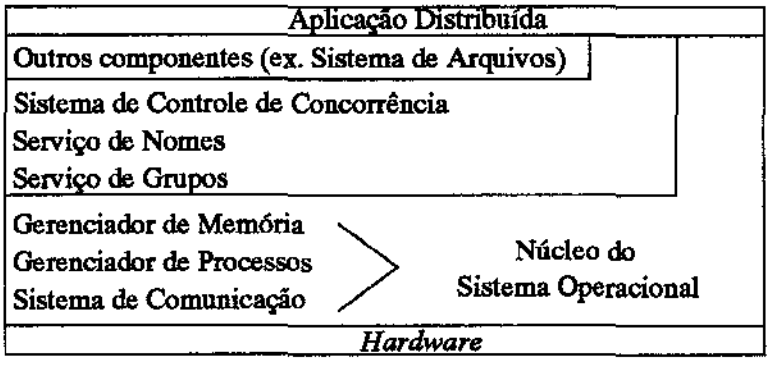


Figura 2.2: Arquitetura de um Sistema Operacional Distribuído

No nível inferior, encontra-se o *hardware*, controlado diretamente pelo núcleo do sistema operacional, através do Gerenciador de Memória (carga/descarga dos processos na memória), Gerenciador de Processos (escalonamento de unidades executáveis em um processador) e Sistema de Comunicação (interface para a interação entre processos locais ou remotos).

No nível imediatamente acima do núcleo, aparecem as implementações de mecanismos mais complexos. O Sistema de Controle de Concorrência é necessário para organizar a utilização de recursos compartilhados por processos do sistema distribuído (ex. região de memória compartilhada, impressora, terminal de vídeo). O Serviço de Nomes fornece transparência de localização dos recursos compartilhados. O Serviço de Grupos permite a criação de grupos de processos, controla as interações internas ao grupo e as interações de um grupo com outros serviços ou grupos.

No nível superior, podem existir outros subsistemas, construídos a partir dos serviços das camadas inferiores a eles. Um exemplo é um Sistema de Arquivos, que organiza e controla a forma de armazenamento de arquivos em memória secundária (disco, fita, etc). Em um sistema distribuído, toda a memória secundária pode ser controlada por um único Sistema de Arquivos Distribuído, que usa o Serviço de Nomes para localizar os arquivos e o Sistema de Controle de Concorrência para manter a consistência e a integridade de arquivos compartilhados.

As aplicações distribuídas utilizam, direta ou indiretamente, os serviços fornecidos pelos níveis inferiores, a fim de executar uma tarefa.

A seguir, são descritos os componentes principais de um sistema operacional distribuído, com ênfase nos componentes desenvolvidos em *Port System.*

2.2.1 Sistema de Controle de Concorrência

Dois ou mais processos podem realizar um acesso concorrente a um recurso compartilhado. Se não houver uma disciplina de acesso a esse recurso, a atividade dos processos pode tornar inconsistente o estado mantido pelo recurso. Para manter a consistência e a integridade do recurso compartilhado, é necessário um Sistema de Controle de Concorrência. Por exemplo, se dois processos compartilham o recurso memória e podem executar as operações de leitura e de escrita sobre uma mesma posição, o Sistema de Controle de Concorrência pode permitir apenas leituras concorrentes. Uma leitura e uma escrita não podem ocorrer concorrentemente porque um processo pode começar sua leitura e, em seguida, o outro começa a escrita, não sendo certo que o valor lido será consistente. Duas escritas concorrentes podem até deixar a região de memória com um valor formado em parte por uma escrita e em parte por outra, sendo ainda pior. Através do Sistema de Controle de Concorrência, é possível garantir que a região de memória sempre terá um conteúdo consistente.

2.2.2 Serviço de Nomes

Nomes podem ser usados para facilitar o compartilhamento de recursos entre processos. Um nome pode representar um referência implícita à localização do recurso compartilhado, sendo utilizado por aqueles que desejam manipular o recurso [29]. Desta forma, nomes podem ser vistos como identificadores de recursos. Estes identificadores referenciam univocamente recursos de um sistema e nunca são reutilizados. São aplicados quando da necessidade de transparência de localização, ou seja, um recurso do sistema pode ser encontrado a partir de seu identificador lógico, e não de sua localização.

O Serviço de Nomes permite às aplicações distribuídas [35]: inserir informações gerais sobre um recurso a ser compartilhado (a princípio, nome e localização), remover estas informações e, quando uma aplicação desejar ter acesso ao recurso, localizá-lo pelo nome. Com isso, para o cliente do recurso,

a única informação que deve ser conhecida é o nome do recurso, que normalmente corresponde a algo legível, enquanto sua localização é capturada de forma transparente através do Serviço de Nomes.

Acoplamentos (*Bindings*)

Um cliente precisa conhecer a localização de um recurso compartilhado antes de utilizá-lo. Para tanto, ele deve realizar um acoplamento entre o nome do recurso e a localização. Um acoplamento simples consiste no mapeamento direto *nome* $\rightarrow$ *localização*, segundo uma função *f(nome)* = *localização*.

Os acoplamentos compostos são aqueles onde o mapeamento *nome* $\rightarrow$ *localização* é indireto, passando por um conjunto de funções compostas $f_1(f_2(...(f_n(nome))...))$ = *localização*. Tais funções são como filtros que, a partir do nome de entrada, vão restringindo a faixa de possíveis valores, até chegar ao resultado final único.

A escolha do nível de acoplamento requer da aplicação a definição de quais recursos são compartilhados e como deve ser o acesso a eles.

Arquitetura do Serviço de Nomes

Em sistemas centralizados, a resolução de nomes pode ser implementada, de forma simplificada, através do acoplamento simples. Esta abordagem pode ser generalizada para sistemas distribuídos, com a implementação de um único Servidor de Nomes. Este servidor centraliza a resolução de nomes em um único nó do sistema distribuído. O acoplamento composto pode ser implementado com o melhoramento do processamento de consulta, de acordo com as necessidades das aplicações. A estrutura centralizada é mais simples de implementar, porém, possui desvantagens como a não tolerância a falhas, pois não possui replicação da base de dados, e a diminuição do desempenho do serviço com o aumento do número de solicitações ao mesmo.

Quando o serviço é distribuído, a complexidade de implementação é maior. Se a tabela de nomes é replicada em todos os servidores participantes, a manutenção da consistência das réplicas é uma tarefa não-trivial. Caso a tabela de nomes seja particionada entre os servidores (cada um controla uma subtabela de nomes local), o processo de acoplamento torna-se mais complexo, exigindo a análise de todas as tabelas, no pior caso, para encontrar ou não um recurso.

Portanto, a definição da arquitetura do Serviço de Nomes depende dos requisitos de desempenho, disponibilidade e tolerância a falhas da aplicação.

2.2.3 Serviço de Grupos

Um grupo consiste de um conjunto de processos que compartilham a mesma funcionalidade e é visto como uma única entidade lógica, sem expor para os usuários sua estrutura interna [18]. Grupos são apropriados a aplicações com características e comportamentos compartilhados entre os seus componentes, os quais interagem entre si para realizar uma tarefa.

O Serviço de Grupos é encarregado de fornecer a abstração de grupos às aplicações distribuídas. O Serviço de Grupos oferece as seguintes funcionalidades: entrada no grupo (e criação automática do grupo, caso ele ainda não exista), saída do grupo (e remoção automática do grupo, caso não existam mais membros no grupo) e mecanismos de interação entre os membros do grupo.

Classificações de Grupos

Pode-se classificar um grupo sob vários aspectos:

- Interação com o grupo

  Um grupo fechado é aquele onde somente seus membros enviam requisições para o grupo. Já um grupo aberto pode receber requisições tanto de seus membros quanto de processos fora do grupo.

- Grupos sobrepostos

  Ocorrem quando processos pertencem, simultaneamente, a diferentes grupos.

- Classificação Estrutural

  Um grupo possui dados compartilhados e operações sobre estes dados. Esta classificação define a maneira como os membros do grupo implementam o compartilhamento de dados [18]:

  - Dados e Operações Homogêneas: todo membro mantém uma réplica completa do conjunto de dados e implementa o mesmo conjunto de operações sobre os mesmos.

Ex. Sistema de Arquivos Replicado.

- Somente Operações Homogêneas: os dados são particionados entre os membros do grupo, sendo que cada membro possui parte do estado global e todos os membros dão suporte ao mesmo conjunto de operações.

  Ex. Serviço de Nomes Distribuído.

- Somente Dados Homogêneos: os membros compartilham um conjunto de dados (através de memória compartilhada ou replicação) mas cada membro provê um conjunto de operações sobre os dados; deve existir um controle de concorrência para o acesso aos dados.

  Ex. Grupos Cooperativos.

- Heterogêneo: dados e operações heterogêneas, cooperação entre os membros não-obrigatória e estado interno de cada um independente dos demais.

  Ex. *E-mail*, *news groups*, video-conferência.

- Classificação Comportamental

  - Grupo Determinístico: cada membro recebe e atua sobre cada requisição, requerendo coordenação e sincronização entre os membros. Todos os membros são equivalentes em comportamento. É usado em replicação de dados e tolerância a falhas.

    Ex. Bases de Dados Distribuídas.

  - Grupo Não-Determinístico: relaxa aspectos de consistência, comportamento e sincronização. Normalmente, os membros não são equivalentes pois, dada uma requisição, não é necessário que todos os membros atuem, e cada membro pode responder diferentemente.

    Ex. Serviço Distribuído de Tempo.

A organização de um grupo é dependente das necessidades da aplicação. Dados os exemplos, verifica-se que o conceito de grupo pode ser usado na solução de diversos problemas de sistemas distribuídos e, para cada problema, as variantes de organização devem ser analisadas de forma a verificar a que se enquadra melhor.

Arquitetura do Serviço de Grupos

A arquitetura do Serviço de Grupos pode ser centralizada ou distribuída. A implementação centralizada exige a presença de um coordenador de grupo. O coordenador de grupo é responsável pelo cadastro de membros e pelo controle das interações no grupo. Desta forma, os clientes do Serviço de Grupos limitam-se a emitir requisições de operações ao coordenador de grupo e a processar as interações. Este tipo de implementação é mais simples, porém, não é tolerante a falhas e não possui boa escalabilidade com o crescimento do número de solicitações ao serviço.

Um Serviço de Grupos distribuído não possui a figura do coordenador central. Todos os membros do grupo cooperam no sentido de controlar a consistência das operações sobre o mesmo. A entrada, a saída e as interações no grupo necessitam o consentimento de todos os membros do grupo, para serem processadas. Tal abordagem implica em uma maior tolerância a falhas, pois as informações de controle do grupo estão replicadas ou divididas entre os membros. Em contrapartida, aumenta a complexidade do serviço, devido à necessidade de troca de informações de controle entre os membros do grupo, para realizar cada operação.

2.2.4 Gerenciador de Memória

O Gerenciador de Memória é responsável basicamente pela carga/descarga de processos da memória principal, controle da utilização de memória principal pelos processos e controle de memória secundária (usada para implementar memória virtual).

Quando um processo inicia sua execução, ele é carregado em uma região de memória principal. Se o processo como um todo (código, dados e pilha) não puder ser completamente carregado na sua região de memória, o Gerenciador de Memória pega parte do processo, a ser utilizada no momento, e carrega na memória principal. As outras partes ficam em memória secundária e somente vão para a memória principal se necessárias, quando é feita uma troca entre a parte em memória principal e a parte requerida em memória secundária. Quando o processo termina sua execução, ele é retirado da memória principal.

Um processo pode, em determinado momento, requerer memória para dados ao Gerenciador de Memória. Então, o Gerenciador de Memória verifica

se existe espaço na região de memória do processo para conter os dados. Se não existe, ele não aloca espaço para o processo. É importante ressaltar que a memória de cada processo é protegida da utilização por outros processos, ou seja, mesmo que um processo tenha espaço sobrando em sua região de memória, este espaço não é repassado para outro processo.

2.2.5 Gerenciador de Processos

Existem sistemas operacionais que dão suporte a processos e outros que dão suporte a processos leves como unidades de execução. Contudo, não importando qual é a unidade escalonável para execução em um processador, o Gerenciador de Processos é necessário para controlar e monitorar a execução de todos os processos.

Quando um processo tem a sua execução iniciada, o Gerenciador de Processos pode fornecer a ele todos os recursos do computador, para que ele execute até o fim (escalonamento simples), ou permitir que ele execute por um período de tempo e, em seguida, seja colocado em uma fila de prioridade contendo processos a executar (escalonamento priorizado). Em qualquer caso, o Gerenciador de Processos sempre sabe qual é o estágio de execução de todos os processos: iniciando, terminando, em execução, bloqueado ou escalonado.

2.2.6 Sistema de Comunicação

Os sistemas de comunicação entre processos correspondem à base de implementação de sistemas distribuídos. Sem um Sistema de Comunicação, alguns componentes de um sistema distribuído não conseguiriam fornecer seus serviços para os componentes que se encontrassem em nós remotos. O Sistema de Comunicação de sistemas operacionais distribuídos é baseado no tráfego de mensagens entre processos concorrentes [5]. Uma mensagem é transferida através da utilização de uma rede de computadores e de um protocolo de comunicação, que permite que o emissor e o receptor da mensagem consigam trocar informações.

Troca de Mensagens

A troca de mensagens pode ser classificada como orientada à conexão ou não, síncrona ou assíncrona, ponto-a-ponto ou *multicast*:

- Conexão

  A forma orientada à conexão consiste dos seguintes passos [7]: emissor envia mensagem ao receptor requerendo início de conversação, e espera resposta; receptor envia resposta estabelecendo a conexão; processos emissor e receptor trocam informações por mensagens (e retornam uma mensagem de reconhecimento, um **ACK**[2], por mensagem recebida); processo emissor ou receptor envia mensagem de término de conexão e espera resposta; processo que recebeu mensagem de término envia resposta e encerra a conexão.

  O esquema não orientado à conexão consiste no tráfego direto de mensagens, sem necessitar o estabelecimento de uma conexão entre o emissor e o receptor, sendo unidirecional. Pode ser subdividido em: sem reconhecimento (datagrama), onde o receptor não envia resposta sobre a chegada da mensagem, ou com reconhecimento, quando um **ACK** é retornado ao emissor [7].

- Sincronização

  Tanto o emissor quanto o receptor podem ser síncronos (bloqueantes) ou assíncronos (não-bloqueantes). Um emissor **síncrono envia uma** mensagem e fica bloqueado até receber uma resposta. Um emissor assíncrono envia uma mensagem e continua seu processamento. Na recepção, o tipo síncrono indica que o receptor deve ficar bloqueado enquanto não houver mensagens para ele, ao passo que o assíncrono requer que o sistema de comunicação detecte a chegada de alguma mensagem e gere uma interrupção para avisar o receptor.

  A Figura 2.3 ilustra as combinações possíveis para modos de comunicação entre um par emissor-receptor [15]. Quando a comunicação é totalmente síncrona, tem-se o mecanismo de rendez-vous. No outro extremo, a comunicação totalmente assíncrona requer a criação de dois processos, um para enviar e outro para receber a mensagem, pois o emissor e o receptor originais não ficam bloqueados enquanto a mensagem está sendo transmitida. O mecanismo de passagem de mensagem necessita a recepção síncrona. Por último, o mecanismo de CPR é

[2]ACK (*acknowledgement*): sinal de retorno do receptor que indica que a mensagem foi recebida corretamente.

síncrono na emissão e assíncrono na recepção, e permite a criação de uma abstração de mais alto nível: chamada de procedimento remoto (também conhecida como CPR), que dá suporte à chamada de procedimentos existentes no sistema distribuído.

Emissor / Receptor	Síncrono	Assíncrono
Síncrono	rendez-vous	passagem de mensagem
Assíncrono	CPR	criação de processo

Figura 2.3: Combinações Possíveis entre Emissor e Receptor

- Topologia

  Uma comunicação com troca de mensagens pode envolver dois ou mais processos. A comunicação ponto-a-ponto é aquela onde apenas dois processos estão envolvidos e, em um determinado momento, um processo é o emissor de mensagem e o outro é o receptor de mensagem (Figura 2.4*i*). A comunicação *multicast* tem um emissor e vários receptores (Figura 2.4*ii*). Um tipo especial de *multicast* (*broadcast*) é aquele onde todos os processos do sistema distribuído são receptores da mensagem (Figura 2.4*iii*).

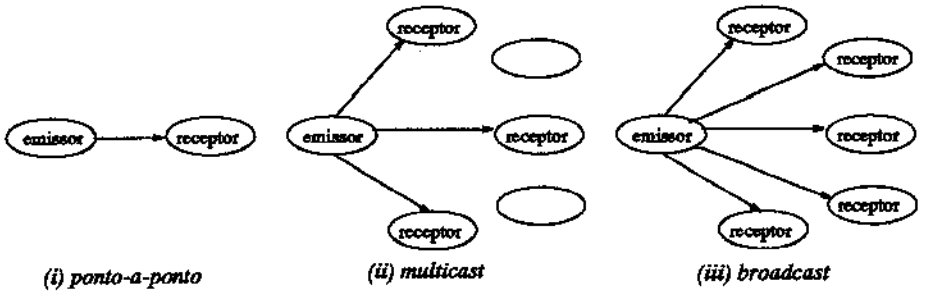


Figura 2.4: Topologia de Comunicação entre Processos

Comunicação em grupo

Comunicação em grupo é aquela que ocorre entre um conjunto determinado de processos. Seu objetivo é oferecer uma abstração para o projetista de uma aplicação baseada em grupo, pois a interação com o grupo é através de uma interface única e não através de uma interface para cada membro do grupo.

O mecanismo de comunicação em grupo é a parte mais básica de um Serviço de Grupos. Toda a funcionalidade que um grupo oferece a uma aplicação é limitada pelos recursos de comunicação em grupo. Um grupo fechado necessita de restrições na comunicação em grupo, enquanto um grupo aberto simplifica a comunicação. Quando da necessidade de grupos sobrepostos na aplicação, pode-se utilizar algoritmos de comunicação em grupos que tomam proveito desta característica. Em termos de estrutura e comportamento de grupos, pode-se requerer comunicação com conexão ou baseada em datagramas.

A comunicação em grupo pode ser implementada por mensagens ponto-a-ponto, sendo que cada mensagem para um grupo é, na verdade, internamente derivada em n cópias da mensagem enviadas ponto-a-ponto (n processos no grupo); por *multicasts*, através do uso de um endereço de *multicast* reconhecido por todos os membros e envio da mensagem para tal endereço; ou por *broadcasts*, com o uso da facilidade de endereço de *broadcast* oferecida pelo *hardware* de rede, sendo o envio da mensagem semelhante a um *multicast*, com a diferença de que todas as máquinas de um sistema distribuído recebem tal mensagem.

Quando da implementação de comunicação em grupo, duas propriedades podem ser necessárias na entrega de mensagens aos membros do grupo: atomicidade e ordenação. A propriedade da atomicidade indica que cada um dos n processos do grupo somente receberá uma mensagem se e somente se os outros $n-1$ processos também a receberem. Esta propriedade é desejável porque, quando qualquer membro envia uma mensagem para o grupo, ocorrem apenas duas situações: mensagem chegou a todos os membros ou não chegou a nenhum, não existindo situações intermediárias do tipo mensagem chegou a um subconjunto dos membros do grupo.

A propriedade da ordenação de mensagens é usada para organizar a entrega de mensagens simultaneamente enviadas por diferentes membros de um grupo. Dado que dois ou mais membros de um grupo podem enviar mensagens aleatoriamente para o grupo, no decorrer do tempo, é possível que estes

membros enviem uma mensagem ao mesmo tempo. Por exemplo, sejam dois processos que enviam, respectivamente, a mensagem A e a mensagem B para o grupo. Se alguns membros do grupo recebem a seqüência $[A, B]$ e outros a seqüência $[B, A]$, então pode ocorrer alguma inconsistência nos processamentos das mesmas, pois diferentes membros do grupo vão efetuar operações diferentes. Grupos com ordenação de mensagens geram a mesma seqüência de processamento para todos os membros, no caso $[A, B]$ ou $[B, A]$. A atomicidade e a ordenação não são características obrigatórias para um sistema de comunicação em grupo, mas elas facilitam a visão de grupo como uma entidade lógica e o estabelecimento de um paralelo entre a comunicação em grupo e a comunicação ponto-a-ponto, que intrinsicamente possui estas duas propriedades.

A partir da análise do tipo de aplicação alvo, pode-se decidir por implementar um grupo fechado ou aberto, com suporte ou não à sobreposição, com necessidade ou não de atomicidade e ordenação e, por fim, de acordo com as necessidades de desempenho e com as instalações físicas para a execução da aplicação, o tipo de troca de mensagens (ponto-a-ponto, *multicast* ou *broadcast*).

Algoritmos de comunicação em grupo

Existe uma série de algoritmos, centralizados ou distribuídos, para a implementação da comunicação em grupo. Os algoritmos de Chang e Maxemchuk [6] (centralizado), Birman e Joseph (ISIS) [3] (distribuído) e Kaashoek [19] (centralizado) estão entre os algoritmos estudados durante o projeto.

A. Algoritmo de Chang e Maxemchuk

O objetivo do algoritmo é implementar um protocolo de *broadcast* confiável. Ele supõe que a rede local de computadores provê *broadcast* em *hardware*, mas que este mecanismo não é confiável, ou seja, não garante que a mensagem enviada é recebida por todos os membros na rede. Com a adição do protocolo, é possível assegurar a confiabilidade dos *broadcasts*. Além disso, cada *broadcast* enviado é atômico e ordenado.

No contexto de falhas, o algoritmo somente analisa falha parcial do tipo falha-e-parada. Para detectar a falha de um membro, o protocolo realiza R tentativas de envio de mensagem para ele e, em seguida, supõe a falha do

mesmo. Este valor R é um compromisso entre a necessidade de detecção rápida (valor pequeno) e o cuidado em não realizar uma detecção errônea de falha (valor grande).

A maneira mais direta de garantir a confiabilidade de entrega dos *broadcasts* é construir um protocolo que retransmite o *broadcast* até que um **ACK** é recebido de cada um dos membros do grupo, confirmando a chegada da mensagem. Com isso, cada mensagem enviada para n membros do grupo requer n **ACK**s. O algoritmo de Chang e Maxemchuk permite o envio de *broadcasts* confiáveis sem a necessidade de retorno de n **ACK**s.

O protocolo pode estar em uma de duas fases: fase normal e fase de reforma.

Durante a fase normal, cada membro do grupo pode enviar *broadcasts* para o grupo. Alguns membros podem perder mensagens, devido a erros de transmissão, mas o protocolo garante a confiabilidade. O *broadcast* de uma mensagem é realizado em três etapas: transmissão, ordenação e consumo. Durante a etapa de transmissão, o emissor retransmite o *broadcast* até que o coordenador do grupo (inicialmente um membro aleatório e, posteriormente, escolhido pelo algoritmo), chamado *token site*, retorne um *broadcast* **ACK**. Se isto não ocorrer durante R tentativas sucessivas, o emissor supõe que o coordenador falhou; caso contrário, segue-se para a etapa de ordenação. Um número de seqüência é gerado pelo coordenador para acompanhar o **ACK**. Quando o **ACK** chega a cada membro do grupo, ele verifica se o número de seqüência corresponde ao próximo esperado e, em caso negativo, o membro pede por retransmissão de mensagens ao coordenador. Somente após todos os membros do grupo terem recebido todas as mensagens, e estarem com o mesmo número de seqüência, é que a mensagem atual é consumida, durante a etapa de consumo.

Além de enviar **ACK**s e retransmissões, o coordenador é responsável por transferir para outro membro do grupo a função de coordenador. Isto ocorre quando, dentro de um intervalo de tempo T, nenhuma mensagem de *broadcast* é enviada e, conseqüentemente, nenhum **ACK** é necessário. A função de coordenador é transferida ao próximo membro do grupo (supondo que os membros do grupo formam uma fila circular), que deve retornar um **ACK**, aceitando a função. Se um **ACK** não é retornado após R tentativas, supõe-se que o receptor da função falhou, sendo a operação de transferência feita para o próximo membro da fila.

A fase de reforma é executada quando uma falha ou uma recuperação de falha de um dos membros é detectada. Uma falha é detectada quando do envio de uma mensagem R vezes para o membro defeituoso, enquanto uma recuperação de falha é detectada quando um membro do grupo, supostamente falho, tenta interagir com o grupo. Durante esta fase, redefine-se quais são os membros válidos (atualmente ativos) do grupo, elege-se um novo coordenador entre os membros ativos do grupo e reestrutura-se o ambiente de forma a confirmar todas as mensagens antigas enviadas ao grupo e reiniciar o processo a partir de novas mensagens. Para realizar estas tarefas, o protocolo executa um algoritmo de eleição de coordenador (semelhante ao algoritmo de Garcia Molina [22]) e as fases de transmissão, ordenação e consumo em caráter extraordinário, para resolver o problema das mensagens pendentes.

B. Algoritmo empregado por ISIS

ISIS é um ambiente de computação distribuída desenvolvido por Birman, Joseph e um grupo de pesquisa da *Cornell University*. Este ambiente implementa três primitivas de comunicação em grupo (ABCAST, CBCAST e GBCAST), para *broadcast* atômico.

O conceito básico sobre o qual são implementadas as três primitivas é o de sincronismo. Um sistema síncrono é aquele onde os eventos acontecem seqüencialmente, ou seja, cada evento (no caso cada envio de *broadcast*) gasta tempo zero para ocorrer. Por exemplo, se um processo A envia, no instante t_0, uma mensagem para B, C e D, a mensagem chega instantaneamente aos destinos. Uma subseqüente mensagem de D para os outros membros do grupo, enviada no instante $t_0 + \Delta t$, chega no mesmo instante aos destinos. Desta forma, o sistema pode ser visto como um conjunto de eventos discretos.

Sistemas síncronos são impossíveis de implementar na prática, sendo necessário analisar sistemas com requisitos temporais mais fracos. Um sistema fracamente síncrono é aquele no qual os eventos gastam uma quantidade finita de tempo, mas todos os eventos aparecem na mesma ordem para todos os destinos. Em particular, processos recebem todas as mensagens na mesma ordem.

Existem aplicações que aceitam semânticas temporais mais fracas do que as citadas acima. Tais aplicações podem ser sistemas virtualmente síncronos, quando a ordenação temporal é relaxada, mas, sob certas circunstâncias, ela ocorre. Para entender estas circunstâncias, é necessário expressar a diferença

entre eventos causalmente relacionados e eventos concorrentes. Dois eventos de um sistema distribuído são causalmente relacionados se a ocorrência do segundo evento pode ter sido influenciada pelo primeiro. Dois eventos não relacionados são ditos concorrentes. O sincronismo virtual indica que, se duas mensagens são causalmente relacionadas, todos os processos devem recebê-las na mesma (correta) ordem. Se elas são concorrentes, nenhuma garantia é dada de que as mensagens serão entregues em alguma ordem para diferentes processos.

As primitivas fornecidas por ISIS são: ABCAST, que provê comunicação fracamente síncrona e é usada para transmitir dados para os membros de um grupo; CBCAST, comunicação virtualmente síncrona e transmissão de dados para o grupo; e GBCAST, semelhante à ABCAST, mas é usada para o cadastro de membros no grupo.

Ao usar ABCAST, o emissor A atribui um número de seqüência para a mensagem e a envia para os membros do grupo. Cada membro, ao receber a mensagem, retorna uma resposta, com um número de seqüência maior que qualquer outro enviado ou recebido, para A. Quando A recebe os números de seqüência de todos os membros do grupo, ele escolhe o maior e envia uma mensagem de consumo, com este número de seqüência, para o grupo. Ao receber uma mensagem de consumo, o membro do grupo consome a mensagem. A ordenação é baseada em ordem crescente de números de seqüência, sendo que os membros do grupo consomem mensagens baseados nestes números.

A primitiva de CBCAST é menos complexa e mais eficiente. Seu funcionamento é mostrado a seguir. Se um grupo possui n membros, cada um mantém um vetor com n componentes. O i-ésimo componente do vetor é o número da última mensagem recebida em seqüência pelo membro i. Os vetores são inicializados com zero. Quando um membro tem uma mensagem para enviar, ele incrementa sua própria entrada no vetor e envia o vetor como parte da mensagem. Suponha que um processo A envia uma mensagem M_1, com o vetor $(1,0,0)$ para os membros B e C. Quando M_1 chega a B, uma verificação do primeiro componente é feita, para ver se o primeiro componente do vetor em B é menor que o da mensagem e se os outros componentes são iguais. Caso contrário, se o primeiro componente é maior, a mensagem é uma duplicação; e se os outros componentes são diferentes, alguma mensagem foi perdida. Agora, suponha que B envia uma mensagem M_2 (com vetor $(1,1,0)$), que chega a C antes de M_1. Neste caso, C verifica que B já recebeu uma mensagem de A antes de M_2. Então, C enfileira M_2 até

que M_1 chegue. Com isso, obtém-se uma ordenação causal, onde a falta de mensagens é detectada a partir do relacionamento entre os valores do vetor.

C. Algoritmo de Kaashoek

É um algoritmo para *broadcast* atômico e ordenado, sem apoio à sobreposição de grupos e restrito a grupos fechados. Pode ser aplicado a grupos determinísticos, devido à confiabilidade da comunicação, e também a grupos não-determinísticos, devido à simplicidade e desempenho das operações de *broadcast*. Contudo, é um algoritmo centralizado. É possível alterar o algoritmo de forma a prover tolerância a falhas [24].

— Protocolo e Algoritmo

O algoritmo original de Kaashoek foi concebido para implementar um protocolo de *broadcast* confiável.

O ambiente básico de execução supõe que existe apenas um membro do grupo em cada nó da rede, e que esta rede suporta *broadcast* de mensagens. Cada mensagem enviada para o grupo é um *broadcast* na rede. Esta rede é local, de forma que não há preocupação com roteamento entre subredes em uma rede de longa distância.

Cada nó possui um núcleo que manipula a comunicação e que é usado pelo membro do grupo para a manipulação de *broadcasts* confiáveis. Quando um membro deseja enviar um *broadcast*, ele transfere a mensagem para o núcleo que a envia ponto-a-ponto para um núcleo especial, o coordenador do algoritmo (*sequencer*). Ao receber a mensagem, o coordenador envia um *broadcast* com a mensagem e o próximo número de seqüência de mensagem. Supondo que mensagens não são perdidas, é fácil verificar que, se dois processos emitem um pedido de *broadcast* simultaneamente, um deles chegará primeiro ao coordenador e será processado, enquanto que o outro será enfileirado. Com isso, a ordenação é garantida.

Todos os processos devem receber o *broadcast* e consumi-lo, incrementando o número de seqüência local de mensagens recebidas. Porém, podem ocorrer falhas durante a transmissão, de forma que algum processo não receba esta mensagem. Ao receber o *broadcast* seguinte, o núcleo que não possui o *broadcast* anterior compara seu número de seqüência com o da mensagem e descobre uma falha. Os números de seqüência devem ser iguais. O núcleo então envia uma mensagem ponto-a-ponto para o coordenador pedindo por

retransmissão de uma ou mais mensagens pendentes. Para ser capaz de responder a estes pedidos, o coordenador armazena os *broadcasts* já enviados em um *buffer* de histórico e retorna mensagens ponto-a-ponto (uma ou mais) para o núcleo emissor do pedido.

A confiabilidade e a atomicidade do *broadcast* é baseada nos números de seqüência. Cada pedido de *broadcast* de um núcleo para o coordenador possui um campo número de seqüência da última mensagem recebida pelo atual emissor. Desta forma, o coordenador pode manter uma tabela com a dupla (membro do grupo, número de seqüência recebido), utilizada para confirmar mensagens recebidas por todos os membros e retirá-las do *buffer* de histórico.

Se um processo não deseja enviar *broadcasts* por um certo tempo, o coordenador não consegue saber quais *broadcasts* ele recebeu. Neste caso, esta informação será obtida durante a fase de sincronização, analisada posteriormente.

A sobrecarga do protocolo é baixa, já que apenas duas mensagens são necessárias para enviar um *broadcast* confiável, atômico e ordenado: uma mensagem ponto-a-ponto do emissor para o coordenador e uma mensagem *broadcast* para todos os nós. Mensagens de **ACK** de resposta a cada *broadcast* não são necessárias, devido ao mecanismo de confiabilidade do protocolo.

O algoritmo pode estar em dois modos de operação: Normal e Sincronização.

Operação Normal

Estado em que os membros do grupo enviam pedidos de *broadcast*, recebem *broadcasts* e emitem pedidos de retransmissões, sendo que o coordenador processa todos os tipos de mensagens (*broadcasts* ou retransmissões) e guarda todos os *broadcasts* enviados no *buffer* de histórico. As interações entre cada membro do grupo (emissor e receptor) e o coordenador são:

- emissor $\stackrel{msg}{\rightarrow}$ coordenador (pedido de *broadcast*)

Campos de *msg*	Significado
src	endereço de rede do emissor
dest	endereço de rede do coordenador
DATA	tipo de mensagem
LastSeqReceived	número de seqüência da última mensagem recebida
MessageID	identificador de mensagem (controle de duplicação)
buffer	dados para o *broadcast*

Quando a mensagem chega ao coordenador, ele executa os seguintes passos:

se (mensagem é uma duplicação)
  então retornar um aviso ao emissor;
  senão
    atualizar a tabela de números de seqüência
    com *LastSeqReceived* do emissor;
    se (*buffer* de histórico está totalmente ocupado
      e não se consegue compactá-lo[3])
      então
        iniciar Operação de Sincronização;
      senão
        mensagem é inserida neste *buffer*;
        mensagem é enviada posteriormente para
        todos os membros do grupo (com o próximo
        número de seqüência de mensagem *broadcast*);
    fim se
fim se

[3]Para compactar o *buffer* de histórico, o coordenador verifica se ao menos uma das mensagens *broadcast* enviadas foi confirmada por todos os membros do grupo. Ele analisa a tabela de números de seqüência e captura o menor *LastSeqReceived*. Com isso, todas as mensagens neste *buffer* cujo número de seqüência é menor que este *LastSeqReceived* são removidas.

- coordenador $\stackrel{msg}{\rightarrow}$ receptores (*broadcast* para o grupo)

Campos de *msg*	Significado
src	endereço de rede do coordenador
broadcast	endereço de *broadcast* da rede
DATA	tipo de mensagem
NextSeqtoUse	próximo número de seqüência de mensagem
buffer	dados do *broadcast*

Em cada receptor:

se (*LastSeqReceived* local = *NextSeqtoUse* da mensagem)

então

receptor consome a mensagem;

LastSeqReceived $\leftarrow$ *LastSeqReceived* + 1;

senão

receptor envia um pedido de retransmissão ao coordenador;

fim se

- receptor $\stackrel{msg}{\rightarrow}$ coordenador (pedido de retransmissão)

Campos de *msg*	Significado
src	endereço de rede do receptor
dest	endereço de rede do coordenador
RETRANS	tipo de mensagem
LastSeqReceived	número de seqüência da última mensagem recebida
MessageID	identificador de mensagem (controle de duplicação)

Quando o coordenador recebe este tipo de mensagem, ele opera como segue:

se (mensagem é uma duplicação)

então

retornar um aviso ao receptor;

senão

atualizar a tabela de números de seqüência

com *LastSeqReceived*;

enviar mensagens *LastSeqReceived* a $NextSeqtoUse - 1$

ponto-a-ponto para o membro do grupo,

a partir do *buffer* de histórico;

fim se

- coordenador $\stackrel{msg}{\rightarrow}$ receptor (retransmissão para um membro)

Campos de *msg*	Significado
src	endereço de rede do coordenador
dest	endereço de rede do receptor
DATA	tipo de mensagem
seq	varia de *LastSeqReceived* a $NextSeqtoUse - 1$
buffer	dados do índice *seq* do *buffer* de histórico

Operação de Sincronização

O tamanho do *buffer* de histórico é limitado por questões práticas de utilização de memória e também para certificar a confiabilidade do protocolo. Quando o *buffer* chega ao limite de ocupação e nenhuma das mensagens nele armazenadas teve sua chegada confirmada por todos os membros do grupo, então é necessário sincronizar a execução de todos os membros, enviando todas as mensagens pendentes e esvaziando o *buffer*.

O algoritmo usa um protocolo do tipo *two-phase commit* para garantir que todos os membros recebam todas as mensagens [17].

Na primeira fase, o coordenador interage com os membros do grupo da seguinte maneira:

- coordenador $\stackrel{msg}{\rightarrow}$ receptores

Campos de *msg*	Significado
src	endereço de rede do coordenador
broadcast	endereço de *broadcast* da rede
SYNC (PHASE1)	tipo de mensagem
$NextSeqtoUse - 1$	número de seqüência da última mensagem enviada

O coordenador envia este *broadcast* e fica esperando uma resposta de cada membro do grupo. Ao receber este tipo de mensagem, o receptor executa o seguinte:

se $(LastSeqReceived - 1 = NextSeqtoUse - 1)$

então

retornar um **ACK-SYNC** para o coordenador;

senão

retornar um pedido de retransmissão de mensagens;

fim se

Quando o coordenador recebe um **ACK-SYNC** de um membro do grupo, ele atualiza a tabela de números de seqüência no respectivo membro com *NextSeqtoUse*. Se receber um pedido de retransmissão, ele envia uma ou mais mensagens ponto-a-ponto para o emissor do pedido, como na Operação Normal.

Após receber todas as mensagens pendentes, cada membro do grupo retorna um **ACK-SYNC** para o coordenador. O coordenador passa para a segunda fase após receber **ACK-SYNC** de todos os membros do grupo.

Na segunda fase, o coordenador envia um **COMMIT** para o grupo:

- coordenador $\stackrel{msg}{\rightarrow}$ receptores

Campos de *msg*	Significado
src	endereço de rede do coordenador
broadcast	endereço de *broadcast* da rede
COMMIT (PHASE2)	tipo de mensagem

Quando recebe um **COMMIT**, cada membro do grupo retorna um **ACK-COMMIT** para o coordenador e volta para a Operação Normal. O coordenador espera um **ACK-COMMIT** de cada membro do grupo e, a seguir, volta à Operação Normal.

Comparação entre os algoritmos

O algoritmo de Kaashoek foi escolhido como o algoritmo de comunicação em grupo para *Port System*, por questões de simplicidade e desempenho.

Em comparação com o algoritmo de Chang e Maxemchuk, o algoritmo de Kaashoek não é tolerante a falhas, mas ele é otimizado para o caso comum de não ocorrência de falhas. Os dois são centralizados, mas em Chang e Maxemchuk a figura do coordenador de grupo é volátil, sendo que todos os membros do grupo, em algum instante de execução, tornam-se o coordenador. Em Chang e Maxemchuk, todas as mensagens são *broadcasts*, enquanto Kaashoek utiliza mensagens ponto-a-ponto quando possível.

Comparando as primitivas de comunicação em grupo de ISIS com o algoritmo de Kaashoek, ISIS é distribuído e tolerante a falhas. Contudo, o número de mensagens de controle que ISIS utiliza é muito maior que em Kaashoek. ABCAST usa mais mensagens de controle que CBCAST. ABCAST apresenta atomicidade e ordenação semelhante à funcionalidade de Kaashoek, enquanto CBCAST, mais eficiente que ABCAST, possui apenas ordenação causal.

2.3 Resumo

Durante este capítulo, foi dada uma visão geral de alguns dos principais conceitos pertinentes a sistemas distribuídos.

Quando da descrição da arquitetura de um sistema operacional distribuído, foi dada uma maior ênfase ao Serviço de Nomes, Serviço de Grupos e Sistema de Comunicação, que correspondem à proposta de *Port System*. Dentro do Sistema de Comunicação, aprofundou-se a descrição do algoritmo de Kaashoek, escolhido para implementar a comunicação em grupo.

Capítulo 3

Trabalhos Relacionados

O objetivo deste capítulo é descrever vários sistemas de suporte ao desenvolvimento de aplicações distribuídas. Cada sistema fornece um conjunto de funcionalidades gerais ou específicas a certos tipos de aplicações. O estudo destas funcionalidades em conjunto com os requisitos de distribuição do ambiente Xchart ajudou o projeto de *Port System.*

Os sistemas descritos podem ser divididos em dois grupos: sistemas operacionais distribuídos e ambientes de programação distribuída.

3.1 Sistemas Operacionais Distribuídos

Nesta seção, são descritos os sistemas operacionais distribuídos Amoeba [21], Mach [33] e Chorus [14], os quais tentam, de formas diferentes, prover algumas das transparências discutidas no capítulo anterior.

3.1.1 Amoeba

A principal idéia envolvendo Amoeba é a de que não existem computadores pessoais. Quando um usuário inicia sua operação em um computador, os processos disparados por ele podem ser executados em qualquer computador do sistema distribuído; o sistema executará o processo no computador onde existirem mais recursos disponíveis [21, 34].

O sistema operacional Amoeba possui dois tipos básicos de componentes: um micronúcleo e os servidores.

Micronúcleo

O micronúcleo está presente em todos os computadores da rede e executa tarefas básicas de todo sistema operacional: gerenciamento de memória, gerenciamento de processos e comunicação entre processos.

O modelo de memória de Amoeba é bastante simples. Cada processo ocupa um ou mais blocos de memória, chamados segmentos, com código, dados ou pilha. Um processo pode criar, remover, ler ou escrever em um segmento. Quando um processo é criado, ele possui pelo menos um segmento, e todos os segmentos do processo devem ser carregados em memória, para que ele possa executar, pois não existe esquema de paginação ou troca de segmentos com a memória secundária.

Amoeba dá suporte a processos leves como unidades de execução escalonáveis. Cada processo pode ser formado por um ou mais processos leves, que compartilham um mesmo espaço de endereçamento de memória. Quando criado, um processo possui pelo menos um processo leve principal. Os processos são criados e controlados pelo Servidor de Processos. Cada computador deve possuir um Servidor de Processos.

Duas formas de comunicação estão disponíveis em Amoeba: CPR e comunicação em grupo. Toda comunicação ponto-a-ponto consiste de um cliente que envia uma mensagem para um servidor e fica bloqueado, esperando pela resposta. O micronúcleo oferece três primitivas para CPR: `trans` (usada pelo cliente para enviar uma mensagem e esperar por resposta), `get_request` (usada pelo servidor para receber a próxima mensagem) e `put_reply` (usada pelo servidor para retornar resposta). Quando um servidor inicia sua execução, ele cadastra seu endereço (um número conhecido como porta) em uma tabela local de portas. Então, o cliente deve enviar, junto com a mensagem, a porta destino do servidor, que deve ser previamente conhecida. Ao chamar `get_request`, o servidor indica para o micronúcleo que ele está disposto a receber mensagens em uma determinada porta. Quando uma mensagem chega nesta porta, o servidor é desbloqueado e realiza a tarefa requisitada. No final, o servidor chama `put_reply`, que retorna a resposta relacionada com a última chamada de `get_request`. Isto ocorre porque o micronúcleo controla o mecanismo de comunicação para que a resposta retorne para o endereço correto. Com isso, cada chamada à `get_request` deve ser posteriormente seguida por uma chamada à `put_reply`.

Quanto à comunicação em grupo, Amoeba provê o mecanismo para

a cooperação entre vários processos que realizam uma tarefa em comum. O micronúcleo oferece as seguintes primitivas para grupos: `CreateGroup` (criar um novo grupo), `JoinGroup` (inserir um novo membro no grupo), `LeaveGroup` (remover um membro do grupo, e o próprio grupo, se estiver vazio), `SendToGroup` (enviar um *broadcast* atômico e ordenado para o grupo) e `ReceiveFromGroup` (receber uma mensagem do grupo). As primeiras três primitivas estão relacionadas com a manutenção de grupos, enquanto as outras duas referem-se ao tráfego de mensagens. O algoritmo descrito na Seção 2.2.6-C é utilizado por Amoeba para implementar comunicação em grupo.

Servidores

Todas as funcionalidades não providas pelo micronúcleo são implementadas através de servidores. Servidores são processos executados em modo usuário, utilizados para estender a funcionalidade do micronúcleo. Desta maneira, pode-se expandir os recursos do sistema operacional através da criação de servidores, que possuem uma interface de utilização bem definida através de chamadas às primitivas de CPR.

O conceito básico relacionado aos servidores é o de objeto. Um objeto em Amoeba é um conjunto de dados que pode ser processado através de operações bem definidas. Quando um processo deseja criar um certo objeto, ele faz uma CPR para um servidor que controla este tipo de objeto. Cada servidor deve publicar os serviços que os objetos fornecem. Assim, quando um cliente deseja usar um objeto, ele executa uma CPR com a identificação do serviço requerido. Por exemplo, o Servidor de Arquivos do Amoeba pode realizar operações como criação, remoção, leitura, escrita, etc, sobre cada objeto arquivo, identificado por um nome global conhecido em todo o ambiente distribuído.

3.1.2 Mach

Os objetivos do sistema operacional Mach são: fornecer uma base para a construção de outros sistemas operacionais, permitir o acesso transparente aos recursos da rede, explorar conceitos de paralelismo nas aplicações e tornar o sistema operacional portável para o maior número de arquiteturas possível [33].

Assim como o Amoeba, Mach é baseado no conceito de micronúcleo, que fornece o gerenciamento de memória, o gereciamento de processos e comunicação. Outras funcionalidades, como o Sistema de Arquivos, Serviço de Nomes e Serviço de Tempo, são implementadas a nível de processo usuário.

O gerenciamento de memória é feito sobre objetos de memória, que são estruturas de dados que podem ser mapeadas no espaço de endereçamento dos processos. Um objeto de memória pode ocupar uma ou mais páginas, que são mapeadas em memória virtual. Assim como em sistemas operacionais centralizados, Mach oferece um completo sistema de controle de execução de processos em memória virtual.

O gerenciamento de processos fornece recursos semelhantes aos de Amoeba: controle de processos e processos leves, sendo os processos leves as unidades de execução escalonáveis.

Comunicação em Mach é apenas ponto-a-ponto do tipo Passagem de Mensagem. O mecanismo é otimizado para interação local, em detrimento da comunicação remota. O objeto porta é a base do sistema de comunicação, funcionando como um *pipe* do UNIX (esquema unidirecional). Quando um processo deseja se comunicar com outro, ele escreve dados na porta do receptor, que posteriormente faz a leitura. As portas permitem uma comunicação confiável, pois é garantido que as mensagens serão entregues ao destino. Ao criar uma porta, o processo recebe como retorno um identificador, usado em operações subseqüentes sobre a porta como, por exemplo, ler a próxima mensagem e remover a porta.

Mach possui uma série de primitivas para gerenciar portas. As mais importantes são: `Allocate` (criar uma porta para leitura de dados), `Deallocate` (remover a porta) e `Set_qlimit` (indicar o número máximo de mensagens consecutivas que podem ser enfileiradas na porta). Para enviar ou receber mensagens da porta é usada a primitiva `mach_msg`, que possui, entre outros parâmetros, o identificador da porta. O emissor envia a mensagem e é liberado; o receptor chama esta função e, se não existir mensagem a receber, é bloqueado por um intervalo de tempo especificado como parâmetro.

No micronúcleo, existe uma tabela de portas usada para identificar se a porta destino de uma mensagem é local ou remota. Na comunicação remota, Mach utiliza o Servidor de Mensagens de Rede, que estende o conceito de portas para a rede. Antes de transmitir a mensagem, o Servidor de Mensagens de Rede interage com outros servidores remotos para localizar a porta destino. Em seguida, ele envia a mensagem ao servidor remoto, que se encarrega

de enfileirá-la na porta destino.

3.1.3 Chorus

O micronúcleo Chorus [14] foi desenvolvido com o objetivo de dar suporte ao desenvolvimento de outros sistemas operacionais e ambientes de programação distribuída, a nível de servidores especializados. Ele é formado por um supervisor, módulos de memória virtual, tempo-real e comunicação. O supervisor atua em contato direto com o *hardware*, tratando interrupções.

O módulo de memória virtual controla a memória local e permite a criação da abstração de atores a nível de modo usuário. Um ator é uma unidade lógica de distribuição e de encapsulamento de recursos (comparável a um processo), e define um espaço de memória protegido. Cada computador do ambiente distribuído pode conter vários atores.

O módulo de tempo-real dá suporte a processos leves como unidades de execução. Um processo leve é criado dentro do contexto de um ator, e compartilha com os demais processos leves deste ator todos os seus recursos.

A comunicação entre processos leves é totalmente transparente, independente dos processos leves estarem no mesmo ator, em atores diferentes do mesmo computador ou em diferentes computadores. Um processo leve envia uma mensagem para outro através do uso de portas (filas de entrada de mensagens). Uma porta é associada a um único ator, sendo compartilhada entre seus processos leves. Além disso, é suportado o conceito de grupo de portas. Um grupo representa um conjunto de portas, que recebem transparentemente mensagens enviadas a ele. As portas de um grupo podem estar em diferentes atores e diferentes computadores, e são inseridas e removidas dinamicamente.

3.1.4 Comparação entre os Sistemas

Os três sistemas operacionais discutidos apresentam algumas semelhanças quanto ao gerenciamento de memória, processos e comunicação (Tabela 3.1).

Conceitualmente, o sistema que melhor atende às necessidades de aplicações distribuídas em geral é Amoeba, por ser genuinamente distribuído.

Todos os três usam a estrutura de micronúcleo e servidores de modo usuário.

Característica	Amoeba	Mach	Chorus
Transparência de Distribuição	√		
Micronúcleo	√	√	√
Memória Virtual		√	√
Processos Leves	√	√	√
Passagem de Mensagem		√	√
CPR	√		
Grupo	√		√

Tabela 3.1: Quadro Comparativo dos Sistemas Operacionais Distribuídos

Em termos de comunicação ponto-a-ponto, Amoeba provê CPR, enquanto Mach e Chorus fornecem exatamente o contrário, em termos de sincronização entre emissor e receptor, ou seja, Passagem de Mensagem. A comunicação em grupo é fornecida por Amoeba e Chorus.

3.2 Ambientes de Programação Distribuída

Ambientes de programação distribuída têm sido desenvolvidos com o objetivo de fornecer ferramentas que facilitem a programação de aplicações distribuídas. As interfaces de programação desses ambientes fornecem abstrações de programação de alto nível ao projetista da aplicação.

3.2.1 COOL (*Chorus Object-Oriented Layer*)

O ambiente COOL [20] foi implementado sobre Chorus com o objetivo de fornecer um ambiente de execução de programas orientados a objetos independente de linguagens de programação. Compiladores de diversas linguagens de programação podem utilizar o ambiente de execução fornecido por COOL como o seu ambiente de execução. COOL aproxima as abstrações oferecidas por Chorus (atores, processos leves, portas e grupos) dos mecanismos de execução de programas requeridos por linguagens orientadas a objetos como, por exemplo, criação de objetos, remoção de objetos e chamada de métodos.

Este sistema é formado por três camadas: o COOL-base (atua diretamente sobre o Chorus, tornando-o um micronúcleo orientado a objetos), o Sistema de Execução Genérico (implementa a abstração de objeto, com os conceitos de instanciação, encapsulamento, herança, interface de métodos, etc) e o Sistema Dependente de Linguagem (permite o suporte adequado à semântica de objetos de uma linguagem de programação específica, fazendo o mapeamento entre estes objetos e os objetos genéricos).

3.2.2 ISIS

ISIS possui um conjunto de ferramentas para dar apoio à construção de aplicações distribuídas, através da utilização de grupos como base de programação [2]. Grupos são utilizados em ISIS para uma série de aplicações, como a construção de serviços tolerantes a falhas, cooperação entre um conjunto de processos para realizar uma tarefa, lista de discussão sobre um determinado tópico, etc.

O sistema de comunicação em grupo de ISIS é implementado sobre uma camada de transporte que possui um protocolo de comunicação confiável e um sistema de detecção de falhas do tipo falha-e-parada. Assim, é garantido que as mensagens transmitidas sempre são entregues a todos os membros operacionais do grupo. A partir disso, são implementadas as seguintes primitivas (Seção 2.2.6-B): ABCAST (*broadcast* atômico e ordenado), CBCAST (*broadcast* baseado em ordenação causal de eventos) e GBCAST (*broadcast* de mensagens de controle e manutenção do grupo).

Quaisquer aplicações baseadas em grupos podem ser implementadas sobre ISIS. Contudo, ele é otimizado para processar a comunicação dos seguintes tipos de grupos:

- Grupo pontual: grupo fechado de processos que cooperam para realizar uma tarefa (ex. replicação de dados);
- Grupo cliente/servidor: grupo aberto que possui como membros os servidores. Um processo que envia mensagens ao grupo é registrado como cliente;
- Grupo de difusão: grupo cliente/servidor onde o cliente registra-se no grupo e um membro do grupo pode enviar mensagens para um conjunto de clientes, que são processos passivos à espera de informações do grupo;

- Grupo hierárquico: é mantida uma estrutura de árvore de grupos, a partir de um grupo raiz, e pode-se enviar uma mensagem para toda a hierarquia, para uma subárvore ou apenas um subgrupo.

As primitivas de comunicação em grupo podem ser utilizadas para a implementação de serviços tolerantes a falhas, como serviços de replicação de dados ou um monitor de falhas, para facilitar a análise de falhas em processos, processadores, linhas de comunicação, etc.

3.2.3 Regis/Darwin

Regis é um ambiente de programação distribuída que considera um sistema distribuído como um conjunto de componentes e conexões [25]. Componentes podem ser primitivos ou compostos. Um componente primitivo consiste de um objeto escrito em C++, enquanto um componente composto é criado a partir de outros componentes (primitivos ou compostos) usando uma linguagem de configuração de componentes chamada Darwin. Um sistema Regis possui normalmente uma árvore de componentes, onde a raiz e todos os nós não-folhas são escritos em Darwin e as folhas são objetos C++. Um gerador de código é usado para transformar a estrutura de árvore de componentes em um programa executável, cujo nome é o mesmo do componente raiz.

O ambiente Regis suporta comunicação inter-componentes (em um mesmo sistema) e inter-sistemas, utilizando os seguintes objetos de comunicação:

- `port` (porta de entrada): contém uma fila de entrada de mensagens de um determinado tipo, definido durante a instanciação de um objeto. Entre os métodos fornecidos estão: `port` (instancia um objeto `port` para um tipo de mensagem e recebe como parâmetro opcional um valor numérico, que representa a identificação única do serviço na rede), `in` (bloqueia o invocador até que uma mensagem seja recebida), `arm` (adiciona a porta a uma lista de portas, usada posteriormente pela função *friend* `sel`), função `sel` (retorna uma referência à primeira porta da lista que possui uma mensagem enfileirada, sendo útil para servidores que atendem a várias portas) e `length` (retorna o número de mensagens enfileiradas para a porta).

- `portref` (porta de saída): é uma referência a um objeto `port`, ou seja, um apontador para um `port`, usado para enviar mensagens para ele. Entre os métodos fornecidos estão: `portref` (instancia um objeto `portref` e, opcionalmente, pode receber como parâmetro um apontador para um objeto `port`, se este objeto estiver na mesma máquina, ou a identificação única do serviço na rede, caso contrário), `out` (envia uma mensagem para um `port` e retorna para o invocador) e `bound` (retorna `TRUE` se `portref` está conectado a um `port`).

Para criar um sistema distribuído em Regis existem duas opções. Uma delas é usar diretamente os objetos `port` e `portref`, escrever código C++ para cada componente do sistema e ligar com o sistema de execução Regis (que implementa os objetos `port` e `portref`), gerando os executáveis dos componentes. A outra é escrever os componentes e as conexões entre eles em Darwin, para depois criar os executáveis correspondentes a partir do gerador de código. A segunda opção oferece uma abstração de mais alto nível. Através da linguagem Darwin, é possível definir componentes (através do comando `component`) e criar conexões entre componentes (através do comando `bind`, que conecta um `portref` de um componente a um `port` do outro). Existe uma série de comandos para criar um sistema com vários componentes e conexões [16].

Para suportar sistemas distribuídos escritos em Regis, são necessários dois serviços do sistema: *RND* (*Regis Name Daemon*), que provê o acoplamento entre um `portref` e um `port`, e a conseqüente transparência de localização, através da utilização da identificação única de rede, e *RED* (*Remote Execution Daemon*), que permite a execução de um sistema em uma máquina remota e interage com *RED*s remotos para transferir mensagens de dados para um componente ou mensagens de controle para o próprio *RED*.

3.2.4 OMNI

O sistema OMNI [11] encontra-se em desenvolvimento no Departamento de Ciência da Computação da UNICAMP, com o objetivo de facilitar a criação e a comunicação entre processos distribuídos. Ele fornece uma biblioteca de funções para a criação de sistemas e um conjunto de servidores que dão apoio à execução destes sistemas.

OMNI é formado pelos seguintes componentes [11]:

- Servidor de Nomes: onde os nomes de todos os recursos do OMNI são cadastrados e associados a um identificador (*OMNIid*). Este servidor é local a cada máquina, e armazena os nomes cadastrados por processos locais. Quando um processo deseja utilizar um objeto, ele aciona o servidor local, que procura pelo nome em sua tabela de nomes. Se encontra o nome, o servidor retorna o *OMNIid* para o processo; caso contrário, o servidor envia um *broadcast* a todos os outros servidores na rede, questionando a existência do nome. Se algum servidor remoto possuir aquele nome, ele retorna o *OMNIid* para o servidor local, que repassa ao processo cliente; caso contrário, é retornado um *OMNIid* nulo. Este esquema de localização é o mais genérico, mas OMNI permite a criação de escopos para nomes (nome único local, nome único global ou nome não necessariamente único), restringindo a busca apenas ao computador local ou a todo o ambiente distribuído.

- Servidor de Processos: processos em OMNI estendem conceitos do sistema operacional UNIX para ambientes distribuídos. Este módulo oferece serviços para a criação de processos, envio e recebimentos de sinais. Através do Servidor de Processos, um sistema pode requerer qualquer um dos serviços acima, passando como parâmetro, entre outras coisas, a máquina destino. Então, os servidores local e remoto interagem para realizar o serviço de forma transparente.

- Portas: correspondem ao mecanismo de comunicação entre processos OMNI. Podem existir dois tipos de portas: conectáveis e não-conectáveis. Portas conectáveis usam o esquema orientado à conexão para troca de mensagens. Sendo assim, se dois processos desejam interagir, cada um deve executar as primitivas: `om_createConPort` (criar uma porta), `om_connect` (conectar-se à outra porta), `om_read` (receber uma mensagem) ou `om_write` (enviar uma mensagem) e, para terminar a interação, `om_disconnect` (desconectar-se da outra porta) e `om_destroyConPort` (destruir sua porta). Além disso, as portas conectáveis podem conter conectores especiais, permitindo que um processo envie uma mensagem para várias portas (*broadcast*) ou receba mensagens de várias portas (*mailbox*).

  As portas não-conectáveis permitem que processos interajam sem a necessidade da conexão. São fornecidas as primiti-

Característica	COOL	ISIS	Regis/Darwin	OMNI
Interface	objeto	procedural	objeto	procedural
Passagem de Mensagem	√		√	√
Grupo	√	√		√
Tolerância a falhas		√		

Tabela 3.2: Quadro Comparativo dos Ambientes de Programação Distribuída

vas: `om_createConLessPort` (criar uma porta não-conectável), `om_destroyConLessPort` (destruir esta porta), `om_send` (enviar uma mensagem) e `om_receive` (receber uma mensagem). Portas não-conectáveis podem formar um grupo. Para tal, existem as primitivas `om_createPortGroup` (criar um grupo), `om_insertPort` (inserir uma porta no grupo), `om_removePort` (remover uma porta do grupo) e `om_destroyPortGroup` (destruir o grupo).

3.2.5 Comparação entre os Ambientes

A comparação entre estes ambientes é difícil de ser feita diretamente. Isto porque existem diferentes objetivos principais entre os sistemas (Tabela 3.2). COOL preocupa-se em fornecer uma interface orientada a objetos que utilize da melhor maneira os recursos de Chorus, e não inclui funcionalidades extras como, por exemplo, CPR. ISIS fornece um ambiente de programação de sistemas tolerantes a falhas baseado em grupo. Regis/Darwin e OMNI permitem comunicação do tipo Passagem de Mensagem e definem portas de forma semelhante.

3.3 Relação entre os Ambientes e *Port System*

Esta Seção traz um estudo das características dos ambientes Amoeba, Mach, Chorus/COOL, ISIS, Regis/Darwin e OMNI, que foram aproveitadas no desenvolvimento do sistema *Port System*.

3.3.1 Amoeba

Amoeba fornece três funções para CPR: `trans`, usada pelo cliente, `get_request` e `put_reply`, usadas pelo servidor. *Port System* implementa objetos cliente e servidor, com funcionalidade semelhante. Contudo, Amoeba não usa um serviço de nomes para localizar o destino da mensagem, que deve ser conhecido, enquanto *Port System* possui o Servidor de Nomes, que fornece transparência de localização.

Assim como Amoeba, *Port System* implementa a comunicação em grupo baseado no algoritmo de Kaashoek, com algumas extensões e alterações (Seção 4.3.6). As primitivas de comunicação em grupo do Amoeba (`CreateGroup`, `JoinGroup`, `LeaveGroup`, `SendToGroup` e `ReceiveFromGroup`) podem ser mapeadas em métodos dos objetos emissores e receptores de mensagens do grupo, fornecidos por *Port System*. Os outros sistemas que suportam grupos (Chorus, ISIS e OMNI) possuem uma implementação distribuída de comunicação em grupo. Basicamente, a escolha do algoritmo centralizado levou em conta a simplicidade e a eficiência do mesmo, em detrimento da tolerância a falhas.

3.3.2 Mach

Comunicação em Mach é otimizada para interação local, e dá suporte apenas ao esquema ponto-a-ponto de Passagem de Mensagem. *Port System* trata da mesma forma a comunicação local e remota, e permite vários esquemas ponto-a-ponto, implementados como um subconjunto de comunicação em grupo.

A principal contribuição de Mach a *Port System* foi a ilustração do mecanismo de portas de comunicação, apesar de sua implementação ser muito simples. *Port System* implementa comunicação baseada em portas de entrada e de saída de mensagens.

3.3.3 Chorus/COOL

O ambiente formado por Chorus e COOL possui conceitos interessantes de como aproximar um sistema operacional de uma linguagem de programação orientada a objetos. Tais conceitos foram observados durante o projeto de *Port System* que, apesar de não ter o objetivo de fornecer um completo am-

biente distribuído orientado a objetos, utilizou estes conceitos para fornecer uma interface de programação orientada a objetos para fácil utilização dos mecanismos de comunicação em grupo.

3.3.4 ISIS

O objetivo do estudo de ISIS foi fazer uma comparação entre algoritmos centralizados e distribuídos para comunicação em grupo e analisar as aplicações de um ambiente de programação distribuída baseado em grupos.

3.3.5 Regis/Darwin

Este ambiente foi o que mais influenciou na definição dos conceitos de portas e na criação de uma interface de programação orientada a objetos. As classes `port` e `portref` de Regis podem ser mapeadas quase que diretamente nas portas de entrada e de saída de *Port System*. Além disso, apesar de Regis/Darwin não possuir comunicação em grupo, estendeu-se as funcionalidades de `portref` para criar o conceito de porta de saída de *multicast* em *Port System*.

3.3.6 OMNI

O estudo de OMNI, em conjunto com Regis/Darwin, serviu como base para a criação de um sistema baseado em portas e a necessidade de um serviço de nomes para fornecer transparência de localização. Porém, o Servidor de Nomes de OMNI possui uma implementação distribuída, enquanto o usado em *Port System* é, por simplicidade, centralizado.

3.4 Resumo

Este capítulo ilustra ambientes que dão suporte ao desenvolvimento de aplicações distribuídas. Os sistemas operacionais distribuídos tentam fornecer a transparência de distribuição a nível de micronúcleo, enquanto os ambientes de programação distribuída permitem que o projetista implemente um sistema distribuído sobre um sistema operacional centralizado utilizando recursos do modo usuário.

A importância destes sistemas está na influência que cada um teve sobre o projeto de *Port System*. Alguns sistemas possuem mecanismos fundamentais utilizados por *Port System*, como o ambiente de comunicação em grupo de Amoeba, a definição do conceito de portas de entrada e de saída de Regis/Darwin e a utilidade do Serviço de Nomes em Regis/Darwin e OMNI. Os outros sistemas são descritos para um estudo comparativo e a definição dos recursos essenciais e possíveis extensões a *Port System*.

Capítulo 4

Port System

Neste capítulo, descreve-se a estrutura completa do sistema *Port System*, que implementa os mecanismos de comunicação em grupo e nomes, necessários ao ambiente Xchart. É importante notar que, apesar de ter funcionalidade voltada ao ambiente Xchart, este sistema não é restrito ao mesmo, podendo ser utilizado por outros tipos de aplicações distribuídas.

Inicialmente, é feita uma análise de requisitos e é dada uma visão geral do sistema. Em seguida, a arquitetura do sistema é mostrada, e o projeto e a implementação de cada componente são descritos detalhadamente. No final do capítulo, descreve-se a integração deste sistema no contexto do Gerente de Distribuição.

4.1 Análise de Requisitos

O objetivo do ambiente de programação distribuída *Port System* é dar suporte à implementação de grupos de processos distribuídos. Para isso, é necessário prover um mecanismo de comunicação em grupo. Contudo, existem situações (Seção 4.1.1) onde o ideal é utilizar comunicação ponto-a-ponto, uma especialização da comunicação em grupo.

A comunicação em rede é passível à ocorrência de falhas, sendo necessário um mecanismo de, pelo menos, detecção de falhas.

Além disso, para prover a transparência de localização de recursos compartilhados do ambiente distribuído, é utilizado um serviço de nomes.

O sistema *Port System* possui os seguintes componentes: Sistema de Co-

munição, Servidor de Grupos (coordenador do algoritmo de Kaashoek — Seção 2.2.6-C), Servidor de Nomes e Sistema de Detecção de Falhas.

4.1.1 Sistema de Comunicação

O Sistema de Comunicação é utilizado pelos seguintes módulos do ambiente Xchart:

- Sistema de Processamento de Transações Atômicas: usa o Sistema de Comunicação tanto para o acesso ao Servidor de Nomes quanto para a implementação de transações atômicas distribuídas.
- Núcleo Reativo: troca mensagens com outros Núcleos Reativos, para que eles reajam à ocorrência de eventos.
- Aplicação: o ideal é que a aplicação não use diretamente o Sistema de Comunicação, mas sim uma abstração de mais alto nível como transações ou apenas a interface de programação do cliente, sob o controle do Núcleo Reativo. Contudo, o Sistema de Comunicação está disponível para a utilização em casos especiais.

O Sistema de Processamento de Transações Atômicas opera sobre recursos compartilhados do ambiente distribuído, os quais são cadastrados no Servidor de Nomes. Com isso, o Sistema de Processamento de Transações Atômicas utiliza uma interface de acesso ao Servidor de Nomes, fornecida pelo Sistema de Comunicação, para localizar os recursos usados em uma transação.

Ao requisitar uma operação sobre um recurso remoto, o Sistema de Processamento de Transações Atômicas precisa interagir com seu semelhante remoto, utilizando comunicação ponto-a-ponto do tipo CPR.

O Núcleo Reativo, responsável pelo controle de execução da aplicação, utiliza o Sistema de Comunicação para disseminar um evento local para Núcleos Reativos remotos. Dependendo do evento, é possível enviar uma mensagem ponto-a-ponto do tipo Passagem de Mensagem ou Criação de Processo, ou uma mensagem *multicast* para um grupo de Núcleos Reativos. O Núcleo Reativo analisa o Xchart da aplicação para saber se o evento deve ser consumido por um ou mais membros da aplicação distribuída. Somente os membros habilitados para utilizar o evento precisam ser comunicados. A transmissão assíncrona é utilizada porque a notificação da ocorrência de um evento não

necessita de resposta do receptor. Na recepção, tanto o esquema síncrono quanto o assíncrono podem ser usados, ficando a cargo do Núcleo Reativo optar por um ou outro.

As funcionalidades do Sistema de Comunicação são:

- Comunicação ponto-a-ponto: Passagem de Mensagem, Criação de Processo, CPR e, inclusive, rendez-vous, uma conseqüência da implementação de transmissão síncrona de CPR e recepção síncrona de Passagem de Mensagem.
- Comunicação em grupo: transmissão assíncrona de *multicasts* para um grupo e recepção síncrona ou assíncrona. A transmissão síncrona (por exemplo, CPR para um grupo) não é suportada.
- Interface de acesso ao Servidor de Nomes.

4.1.2 Servidor de Grupos

O Servidor de Grupos é o módulo centralizador das operações de *multicast* requeridas pelos Sistemas de Comunicação. Ele implementa o coordenador do algoritmo de Kaashoek, provendo *multicast* atômico e ordenado. Contudo, o algoritmo original corresponde a um subconjunto das funcionalidades do Servidor de Grupos, que são as seguintes:

- mecanismos de criação e remoção de grupos;
- controle de entrada e de saída de membros de um grupo;
- gerência de *multicast* confiável em rede local.

O algoritmo de Kaashoek assume apenas um grupo formado por todas as máquinas em uma rede local, onde cada máquina possui apenas um membro do grupo. Enquanto isso, o Servidor de Grupos controla vários grupos formados por um ou mais membros em uma rede local.

4.1.3 Servidor de Nomes

O Servidor de Nomes centraliza as informações sobre todos os recursos compartilhados do ambiente distribuído. Suas funcionalidades são:

- criação de um recurso, identificado por um nome e uma localização, para acesso global;
- remoção das informações sobre um recurso;
- localização de um recurso, identificado por um nome.

Com isso, o Servidor de Nomes executa o acoplamento entre dois processos que interagem através do compartilhamento de um recurso. Este acoplamento é simples, ou seja, apenas uma função *f(nome) = localização*.

4.1.4 Sistema de Detecção de Falhas

O Sistema de Detecção de Falhas apresenta-se como um módulo utilizado pelo Sistema de Comunicação e pelo Servidor de Grupos. Pode-se dizer que este módulo está embutido no Sistema de Comunicação e no Servidor de Grupos, sendo, na verdade, um subsistema destes módulos. Porém, conceitualmente ele é colocado como um módulo a parte, para facilitar o entendimento de suas funcionalidades.

Nesta seção, será usado o termo módulo de comunicação para significar Sistema de Comunicação ou Servidor de Grupos.

Este sistema é ativado se e somente se acontecer um problema de comunicação com algum membro remoto do ambiente distribuído, devido à impossibilidade de enviar uma mensagem para um membro remoto ou ao não retorno de mensagem de resposta esperada dentro de um intervalo de tempo (*timeout*). Em seguida, o Sistema de Detecção de Falhas tenta encontrar a causa da falha e retorna um diagnóstico ao módulo de comunicação (detecção da falha). É o módulo de comunicação que toma uma atitude após o diagnóstico (avaliação da falha). Esta atitude pode refletir no processamento de outros módulos do ambiente Xchart, que devem tomar atitudes particulares. Dadas as possíveis atitudes que o módulo de comunicação pode tomar, o Sistema de Detecção de Falhas não garante a transparência de falhas. Caso este sistema fosse tolerante a falhas, o gerenciamento poderia ser

isolado no Sistema de Detecção de Falhas, que faria a detecção da falha e a reorganização do ambiente sem que membros de mais alto nível tivessem a necessidade de se preocupar com isso.

Este sistema analisa falhas do tipo falha-e-parada. A ativação do Sistema de Detecção de Falhas ocorre após falha de envio de mensagem ou *timeout*. O segundo caso é um problema que, à primeira vista, pode ocorrer por falha de temporização. O membro remoto talvez envie a resposta solicitada pelo módulo de comunicação, mas após o intervalo de tempo especificado. Neste caso, o módulo de comunicação sempre descarta tais respostas, de forma a não se preocupar com falhas de temporização.

O Sistema de Detecção de Falhas, quando ativo, preocupa-se em testar alguns componentes (sujeitos à ocorrência de alguma falha) e obter um de dois diagnósticos: componente está ativo e funcionando ou componente falhou e parou. Se todos os componentes analisados estão funcionando, o Sistema de Detecção de Falhas retorna para o módulo de comunicação sem encontrar nada errado e este deve retransmitir a mensagem.

Este sistema detecta dois tipos de falhas:

- Isolamento: o Sistema de Detecção de Falhas verifica que o membro local do ambiente distribuído está isolado dos demais quando ele detecta uma falha local do *hardware* ou *software* de rede. Neste caso, o membro fica impossibilitado de interagir com o ambiente remoto.

- Particionamento: quando uma falha não local em um *hardware* ou *software* de rede ocorre, quebrando a comunicação em algum ponto entre dois membros do ambiente distribuído, então serão geradas duas partições do ambiente [22]. Porém, este ambiente é menos restritivo que o de Isolamento, pois os elementos na mesma partição continuam interagindo.

Portanto, o Sistema de Detecção de Falhas analisa o ambiente de comunicação entre os interlocutores e retorna "Isolamento", "Particionamento" ou "Nada Errado".

4.1.5 Ambiente Computacional Requerido

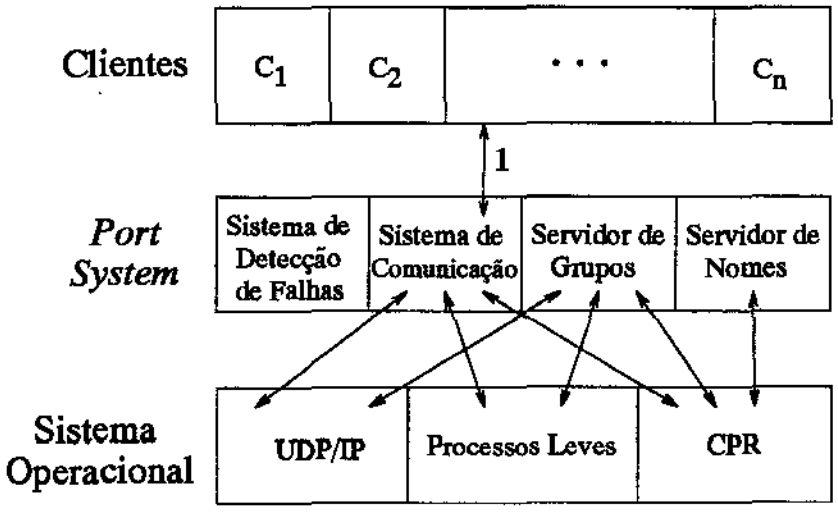
Após a análise dos requisitos funcionais de *Port System*, pode-se notar a necessidade de um sistema operacional com suporte a um protocolo de co-

municação em rede de baixo nível, para permitir a construção do Sistema de Comunicação e do Servidor de Grupos. O protocolo escolhido foi o UDP/IP (transporte de datagramas), por ser um padrão reconhecido na comunidade acadêmica e comercial.

A comunicação entre o Sistema de Comunicação e os servidores (Servidor de Grupos e Servidor de Nomes) é típica para a utilização de CPRs. O Sistema de Comunicação é cliente do Servidor de Grupos para as operações de cadastro de membros em um grupo, e cliente do Servidor de Nomes para as operações de cadastro de nomes.

Se o sistema operacional utilizado provê suporte a processos leves, a implementação do Sistema de Comunicação e do Servidor de Grupos torna-se mais eficiente pois, nos dois casos, pode-se utilizar no mínimo dois processos leves: um para receber mensagens e outro para enviar mensagens.

A Figura 4.1 mostra as relações de dependência entre os módulos de *Port System*, os recursos necessários do sistema operacional e as aplicações clientes.



1. Todos os clientes interagem com o Sistema de Comunicação

Figura 4.1: Ambiente Computacional

4.2 Arquitetura do Sistema

O diagrama da Figura 4.2 representa a arquitetura completa de *Port System*. Cada retângulo representa uma classe. O relacionamento entre classes é denotado por uma linha ligando as classes. A cardinalidade do relacionamento é inserida nos extremos da linha. As linhas tracejadas representam a separação conceitual entre os módulos de *Port System*. O mapeamento entre os módulos e classes é o seguinte:

- Sistema de Comunicação: formado pelas classes **PortIn**, **PortOut**, **PortOutMulticast**, **NameClient** e **Communic**.
- Servidor de Grupos: classe **GroupServer**.
- Servidor de Nomes: classe **NameServer**.
- Sistema de Detecção de Falhas: classe **Failure**.

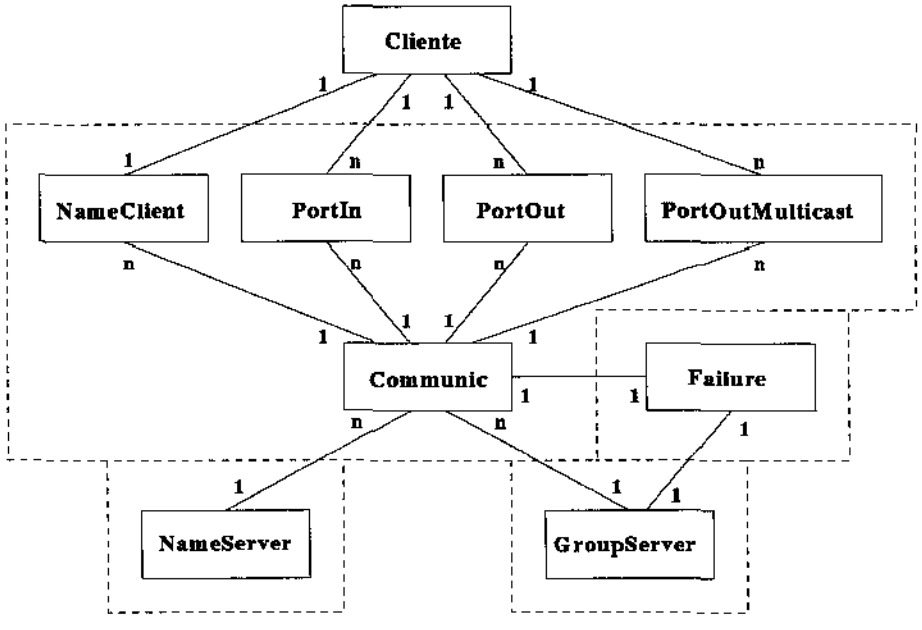


Figura 4.2: Diagrama de Classes de *Port System*

No topo do diagrama encontra-se um **Cliente** de *Port System*, que representa alguns dos subsistemas do ambiente Xchart ou até mesmo um programa que utiliza comunicação. O **Cliente** utiliza *Port System* através de uma interface de programação oferecida pelos objetos **PortIn**, **PortOut**, **PortOutMulticast** e **NameClient**. Um **Cliente** pode estar associado a um ou mais desses objetos, durante sua execução.

Os objetos **PortIn**, **PortOut**, **PortOutMulticast** e **NameClient** usam o objeto **Communic**, que implementa a comunicação em baixo nível (UDP/IP e CPR) e, em conjunto com o objeto **Failure**, realiza a detecção de falhas.

Os servidores (**NameServer** e **GroupServer**) são únicos no ambiente distribuído. Os objetos **Communic** existentes no ambiente concorrem no acesso às suas funcionalidades.

A Figura 4.3 ilustra a distribuição dos objetos por uma rede de computadores. Cada computador que possui um membro de uma aplicação distribuída, um **Cliente**, também contém pelo menos um dos objetos que interagem com o **Cliente** (objetos de interface com cliente), um **Communic** e um **Failure**. Nada impede que todos os objetos integrantes de uma aplicação distribuída estejam localizados em um mesmo nó.

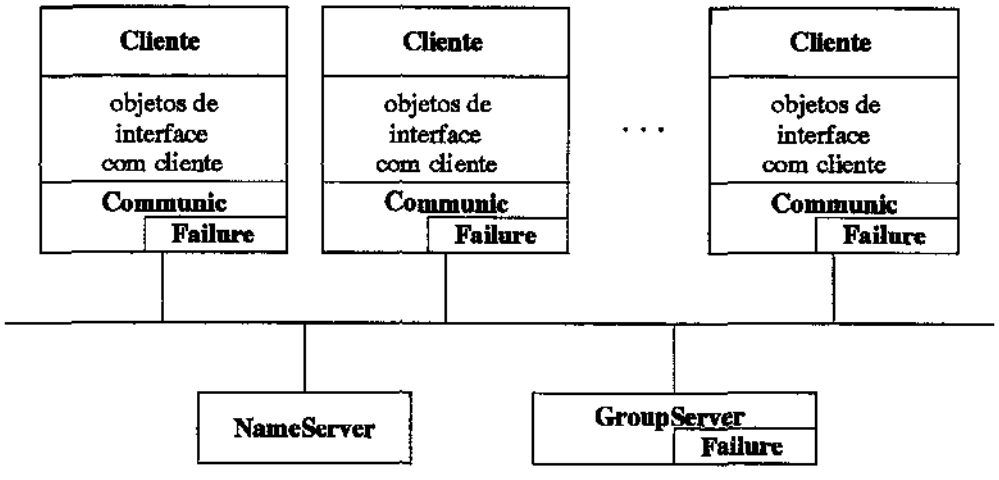


Figura 4.3: Distribuição dos Objetos na Rede

4.3 Projeto dos Componentes do Sistema

A partir do estudo dos requisitos e da arquitetura do sistema, é possível descrever detalhadamente as classes que formam *Port System*. Para cada classe, são mostrados um diagrama de classe, que ilustra os atributos e métodos da classe, e uma descrição informal dos mesmos.

Em primeira instância, são descritas as classes dos objetos de interface com cliente. Em seguida, a classe **Communic**, que fornece a infra-estrutura do Sistema de Comunicação. Então, descreve-se **GroupServer**, **NameServer** e, por último, **Failure**.

As classes **PortIn**, **PortOut** e **PortOutMulticast** são baseadas no mecanismo de portas de comunicação, semelhante ao encontrado em Mach (Seção 3.1.2), OMNI (Seção 3.2.4) e principalmente Regis (Seção 3.2.3). Foram definidos os termos porta de entrada (recebe mensagens), porta de saída (envia mensagens ponto-a-ponto) e porta de saída de *multicast* (envia mensagens *multicast*), detalhados a seguir.

4.3.1 Classe PortIn

Esta classe é utilizada para criar uma porta de entrada associada a um processo (Figura 4.4). Um objeto **PortIn** representa uma abstração de um fila de entrada de mensagens para o processo, sob controle de um objeto **Communic**. Com o uso deste objeto, o processo preocupa-se apenas em consumir mensagens que chegam na sua porta de entrada.

Seus atributos são:

- `portID`: identificador de porta de entrada, inicializado com informação obtida através de consulta ao Servidor de Nomes;
- `ptPort`: apontador para a posição da instância de **PortIn** na tabela de portas de entrada de **Communic**;
- `Receive_EV`: evento relacionado com a chegada de uma mensagem nesta porta.

Entre os métodos providos estão:

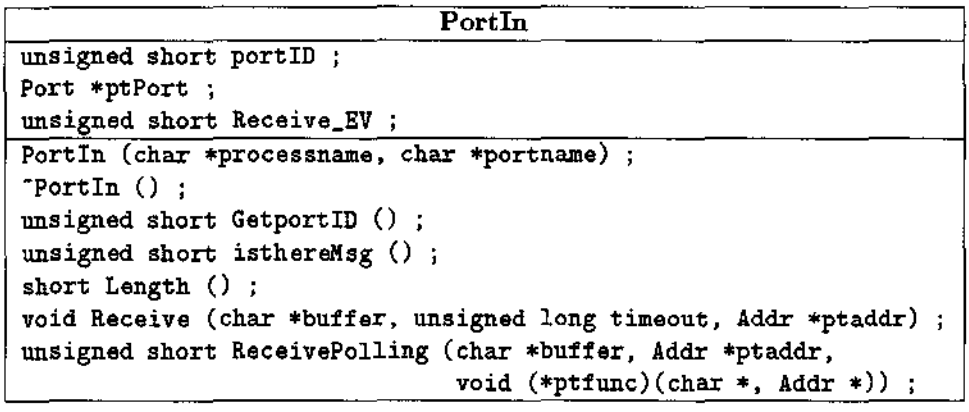


Figura 4.4: Diagrama da Classe **PortIn**

- Construtor de Objeto: toma como entrada duas cadeias de caracteres (nome do processo e nome da porta), as quais devem identificar esta porta univocamente no ambiente distribuído. Então, chama o Servidor de Nomes (através de **Communic**) para criar um recurso porta de entrada e retornar um identificador numérico de recurso (*ID*) e, em seguida, chama **Communic** para inserir esta porta na sua tabela de portas de entrada locais.

- Destrutor de Objeto: efetua o processo contrário ao Construtor, através de uma chamada a **Communic** para remover sua entrada da tabela de portas de entrada (inclusive mensagens pendentes) e uma chamada ao Servidor de Nomes, que remove o recurso associado àquele *ID*.

- `GetportID`: retorna o *ID* associado a esta porta. Pode ser utilizado para confirmar a criação da porta de entrada. Se retornar 0, então a porta de entrada não foi criada adequadamente. A causa do fracasso na criação é a não inserção do recurso no Servidor de Nomes, ou porque já existe um nome de recurso formado pelas cadeias de caracteres citadas no Construtor ou porque a chamada ao Servidor de Nomes falhou. Um retorno diferente de 0 indica o sucesso na criação da porta.

- `isthereMsg`: verifica se existe alguma mensagem para esta porta de entrada, retornando `TRUE` ou `FALSE`.

- `Length`: retorna o número de mensagens enfileiradas nesta porta. Caso a porta não tenha sido corretamente criada, `Length` retorna -1.

- `Receive` (recepção síncrona): verifica se existe mensagem para a porta: em caso positivo, retorna mensagem (no parâmetro `buffer`); caso contrário, bloqueia por um tempo (`timeout`) e faz nova verificação, retornando nulo em `buffer`, se não existir mensagem. Um terceiro parâmetro (opcional) é um apontador para uma estrutura que armazenará o endereço completo de uma porta de entrada do emissor da mensagem, para posterior retorno de resposta para esta porta, através das operações `Bind` (2) e `Send/SendDatagram` de **PortOut**, analisadas na próxima seção.

  Obs: O evento `Receive_EV` é usado por `Receive`, quando ele bloqueia na espera por uma mensagem. Quando o objeto **Communic** recebe uma mensagem, ele sinaliza o evento `Receive_EV` da respectiva porta de entrada, desbloqueando `Receive`.

- `ReceivePolling` (recepção assíncrona): quando chamado, este método cria um processo leve (`Polling`) para ficar esperando a mensagem e retorna. Desta forma, o cliente é liberado para fazer outras coisas. O processo leve `Polling`, que possui o mesmo protótipo de `ReceivePolling`, chama `Receive (buffer, INFINITE, ptaddr)`. O valor de *timeout* igual a `INFINITE` significa que `Receive` não deve retornar enquanto não existir uma mensagem na fila. Quando a mensagem chega, são preenchidos `buffer` e `ptaddr`, e é chamada uma função definida pelo usuário, através de `ptfunc`. Esta função deve ter o protótipo descrito na definição de `ptfunc`, e receberá como parâmetros de entrada `buffer` e `ptaddr`.

  Obs: A função definida pelo usuário é executada no contexto do processo leve `Polling`, de forma que ela deve utilizar mecanismos de sincronização como, por exemplo, região crítica ou semáforo, para interagir com o processo leve principal e outros processos leves pertencentes à aplicação. Além disso, ela deve utilizar os recursos da aplicação como, por exemplo, variáveis, com muito cuidado, para que não ocorram problemas de acesso concorrente a recursos.

Após a instanciação, um objeto **PortIn** pode estar em um estado válido ou inválido. Se ele foi corretamente inserido no Servidor de Nomes e

em **Communic**, então ele está habilitado a fornecer sua funcionalidade. Caso contrário, as chamadas aos métodos `isthereMsg`, `Length`, `Receive` e `ReceivePolling` retornam sem fazer nada.

No Apêndice A são dados exemplos de utilização da classe **PortIn**, tanto em comunicação ponto-a-ponto quanto comunicação em grupo.

4.3.2 Classe PortOut

Esta classe é utilizada para criar uma porta de saída, ou seja, uma referência a uma porta de entrada, para transmissão de mensagens (Figura 4.5).

PortOut
`Addr portaddr ;`

`PortOut () ;`
`PortOut (char *processname, char *portname) ;`
`PortOut (Addr *ptaddr) ;`
`unsigned short Bind (char *processname, char *portname) ;`
`void Bind (Addr *ptaddr) ;`
`void Unbind () ;`
`unsigned short isBound () ;`
`void SendDatagram (char *buffer, PortIn *ptportin) ;`
`void Send (char *buffer, PortIn *ptportin, unsigned long timeout) ;`

Figura 4.5: Diagrama da Classe **PortOut**

Seu único atributo é:

- `portaddr`: estrutura que representa o endereço completo da porta de entrada referenciada.

Entre os métodos providos estão:

- Construtor de Objeto (0): inicializa o atributo endereço com nulo. É usado quando não se conhece, em tempo de compilação, a porta de entrada que será referenciada.
- Construtor de Objeto (1): recebe como parâmetro o nome do processo e o nome da porta, que representam a porta de entrada univocamente no ambiente distribuído, e chama o método `Bind` (1) para ligar-se à porta de entrada.

- Construtor de Objeto (2): em uma situação especial em que o processo associado a esta porta recebeu a informação sobre o endereço completo da porta de entrada[1], este construtor pode ser invocado com este endereço. Desta forma, não há necessidade de se conhecer as duas cadeias de caracteres que representam a porta de entrada. A seguir, o método `Bind` (2) é ativado pelo Construtor.

- `Bind` (1): chama o Servidor de Nomes (através de **Communic**) para capturar o endereço completo da porta de entrada representada pelas duas cadeias de caracteres e atualizar o atributo endereço. No caso da porta de entrada nomeada não estar no Servidor de Nomes ou a chamada ao Servidor de Nomes falhar, o atributo endereço é atualizado com valor nulo, o que indica que a porta de saída não foi ligada à porta de entrada. Este método retorna `TRUE`, se conseguir a referência à porta de entrada, e `FALSE`, caso contrário.

- `Bind` (2): atualiza o atributo endereço com o parâmetro de entrada (endereço completo de uma porta de entrada). É mais eficiente que `Bind` (1), pois não necessita invocar o Servidor de Nomes.

- `Unbind`: insere nulo no atributo endereço, permitindo posteriormente novas chamadas a um dos métodos `Bind`.

  Os métodos `Bind` (1), `Bind` (2) e `Unbind` podem ser invocados várias vezes, para obter referências dinamicamente durante toda a execução do processo associado a esta porta. Com isso, uma porta de saída pode possuir uma referência a diferentes portas de entrada, no decorrer do tempo, sendo que, no máximo, está ligada a uma porta de entrada, em um dado instante.

- `isBound`: retorna `TRUE`, se a porta está ligada, e `FALSE`, caso contrário.

- `SendDatagram` (transmissão assíncrona): verifica se a porta de saída está ligada e, em caso positivo, transfere a mensagem para **Communic**, que a envia ao destino. Possui como parâmetro, além da própria mensagem (`buffer`), um apontador para uma porta de entrada local (opcional), utilizada quando a mensagem enviada necessita de resposta,

[1] Tal informação é capturada pelo terceiro parâmetro de PortIn::Receive, supra citado.

a qual será retornada pelo receptor da mensagem através desta porta, após algum processamento. Desta forma, o método não bloqueia e o processo associado utiliza os métodos `isthereMsg/Receive` de **PortIn** para capturar a resposta futuramente, na sua porta de entrada local.

- `Send` (transmissão síncrona): verifica se a porta de saída está ligada e, em caso positivo, transfere a mensagem para **Communic**, que a envia ao destino. Possui como parâmetros: a mensagem, um apontador para uma porta de entrada local para retorno de resposta (não opcional) e um limite de *timeout* de espera de resposta. Após a transferência da mensagem, este método invoca `Receive` associado à porta de entrada local, passando como parâmetro `buffer` (para receber a resposta) e `timeout`. Se receber uma resposta em tempo, retorna em `buffer`; caso contrário, retorna o `buffer` nulo. O parâmetro `buffer` é usado na transmissão da mensagem e na recepção da resposta.

  O uso de `PortOut::SendDatagram` em conjunto com `PortIn::Receive` fornece uma funcionalidade semelhante a `PortOut::Send`.

  Os métodos `Send/SendDatagram` podem fracassar ao enviar a mensagem requerida para o seu destino. Caso isto ocorra, o método `Unbind` é automaticamente chamado, pois supõe-se que o destino falhou.

Após a instanciação, um objeto **PortOut** pode estar em um estado válido ou inválido. Se ele encontrou o objeto **PortIn** requerido e está referenciando este objeto, então ele está habilitado a fornecer sua funcionalidade. Caso contrário, as chamadas aos métodos `SendDatagram` e `Send` retornam sem fazer nada.

No Apêndice A (exemplos A.1, A.2, A.3, A.4 e A.7) são dados exemplos de utilização da classe **PortOut**.

4.3.3 Classe PortOutMulticast

Esta classe é usada para criar uma porta de saída de *multicast*, ou seja, uma referência a n portas de entrada de um grupo, para transmissão de mensagens ordenadas para n processos no ambiente distribuído em uma operação atômica (Figura 4.6).

Seus atributos são:

PortOutMulticast
Addr portlocaladdr ; unsigned short groupID ;
PortOutMulticast () ; PortOutMulticast (char *groupname, PortIn *ptportlocal) ; ~PortOutMulticast () ; unsigned short Enter (char *groupname, PortIn *ptportlocal) ; void Leave () ; short NumPorts () ; unsigned short GetgroupID () ; void SendDatagram (char *buffer) ; void GetMessages () ;

Figura 4.6: Diagrama da Classe **PortOutMulticast**

- `portlocaladdr`: endereço completo da porta de entrada local, usada para receber mensagens do grupo. Quando o processo que possui um objeto **PortOutMulticast** almeja apenas enviar mensagens para o grupo, ele não precisa inicializar este atributo (opcional);
- `groupID`: identificador de grupo (*ID*), inicializado a partir do Servidor de Grupos.

Os métodos providos são:

- Construtor de Objeto (0): inicializa os atributos com nulo. É usado quando não se conhece, em tempo de compilação, o grupo a entrar.
- Construtor de Objeto (1): recebe como parâmetros o nome do grupo a entrar e um apontador para um objeto porta de entrada local (opcional). A seguir, o método `Enter` é invocado.
- Destrutor de Objeto: chama `Leave` para sair do grupo.
- `Enter`: chama o Servidor de Grupos (através de **Communic**) para inserir esta porta de saída de *multicast* (e sua respectiva porta de entrada, caso exista alguma) no grupo `groupname`. Caso não aconteça

uma falha na chamada ao Servidor de Grupos[2], é retornado um identificador de grupo (*ID*) não nulo do Servidor de Grupos, para posteriores operações sobre o grupo; caso contrário, o identificador de grupo é zerado. Este método retorna `TRUE`, se conseguiu entrar no grupo, e `FALSE`, caso contrário.

- `Leave`: chama o Servidor de Grupos (através de **Communic**) para retirar a porta de saída de *multicast* (e sua respectiva porta de entrada, caso exista alguma) do grupo ao qual pertence, representado por um *ID*.

  Os métodos `Enter` e `Leave` podem ser invocados várias vezes, para efetuar entrada/saída dinâmica de grupos durante toda a execução do processo associado a esta porta. Com isso, uma porta de saída de *multicast* (e sua respectiva porta de entrada, caso exista alguma) pode estar contida em diferentes grupos, ao longo de sua execução, sendo que, no máximo, pertence a um grupo em um dado instante.

  Esta abordagem de entrada/saída de grupos, onde as portas de entrada dependem das portas de saída de *multicast* para participarem de grupos, é usada para facilitar a visão de membro de um grupo, um elemento que pode enviar e, opcionalmente, receber mensagens de um grupo. Neste esquema, não existem membros que potencialmente só recebem mensagens de um grupo, a menos que estes não utilizem suas portas de saída de *multicast* para enviar mensagens. Com isso, para existir uma porta de entrada em um grupo, necessariamente deve haver uma porta de saída de *multicast* associada.

- `NumPorts`: chama o Servidor de Grupos (através de **Communic**) para verificar se o grupo associado a esta porta possui zero ou mais portas de entrada para recepção de mensagens. Retorna o número de portas de entrada do grupo. Pode ser usado no intuito de saber se existe alguma porta de entrada para receber mensagens ou se todos os membros do grupo já estão presentes com suas portas de entrada. Caso esta porta de saída de *multicast* tenha fracassado em `Enter` ou a chamada ao Servidor de Grupos falhou, `NumPorts` retorna -1.

[2] A não entrada no grupo só ocorre se a comunicação com o Servidor de Grupos falhar.

- `GetgroupID`: retorna o *ID* do grupo. Pode ser utilizado para confirmar a entrada desta porta em um grupo. Se retornar 0, então a porta de saída de *multicast* (e sua respectiva porta de entrada, caso exista alguma) não estão no grupo requerido, devido à falha na comunicação com o Servidor de Grupos.

- `SendDatagram` (transmissão assíncrona): verifica se a porta de saída de *multicast* está em um grupo e, em caso positivo, transfere a mensagem para **Communic**, que a envia ao Servidor de Grupos. O Servidor de Grupos responsabiliza-se pelo envio da mensagem aos membros do grupo, ou seja, responsabiliza-se pelo *multicast* da mensagem.

- `GetMessages`: método usado apenas quando a porta de saída de *multicast* possui uma porta de entrada local associada. Quando, durante a execução de `PortIn::Receive`, após esperar por mensagem do grupo ao qual pertence, ocorrer o *timeout*, o processo associado a esta porta pode invocar `PortOutMulticast::GetMessages` para pedir mensagens ao Servidor de Grupos. Mais detalhes da sua funcionalidade estão na Seção 4.3.6.

  Os métodos `SendDatagram/GetMessages` podem não conseguir enviar a mensagem requerida para o Servidor de Grupos. Caso isto ocorra, o método `Leave` é automaticamente chamado para zerar `groupID`, pois supõe-se que o Servidor de Grupos falhou (falha-e-parada).

Após a instanciação, um objeto **PortOutMulticast** pode estar em um estado válido ou inválido. Se ele está inserido em um grupo, então ele está habilitado a fornecer sua funcionalidade. Caso contrário, as chamadas aos métodos `NumPorts`, `SendDatagram` e `GetMessages` retornam sem fazer nada.

No Apêndice A (exemplos A.5 e A.6) são dados exemplos de utilização da classe **PortOutMulticast**.

4.3.4 Classe NameClient

Esta classe é usada para criar uma interface de acesso às funcionalidades do Servidor de Nomes (Figura 4.7). Ela recebe suporte do objeto **Communic**, que implementa a comunicação CPR com o Servidor de Nomes.

Seu único atributo é:

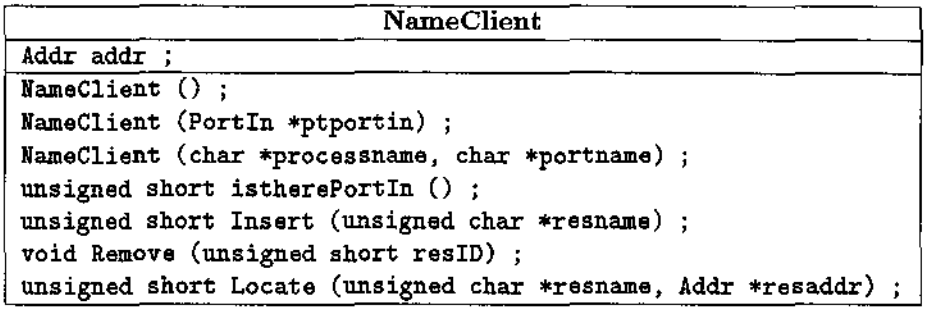


Figura 4.7: Diagrama da Classe **NameClient**

- addr: endereço completo do objeto **PortIn** usado pelo criador (servidor) de um recurso para receber requisições de acesso ao mesmo, ou seja, endereço onde o recurso pode ser encontrado pelos seus clientes. O seu valor é inserido no Servidor de Nomes junto com o nome do recurso, representando sua localização. Quando o objeto **NameClient** é usado por clientes do recurso, este atributo não é usado (nulo).

Seus métodos são:

- Construtor de Objeto (0): inicializa o atributo endereço com nulo. É usado pelos clientes do recurso ou, quando não se conhece em tempo de compilação o objeto **PortIn** que será usado, pelo servidor do recurso.

- Construtor de Objeto (1): usado pelo servidor do recurso para inicializar addr com o endereço do **PortIn** de acesso ao recurso.

- Construtor de Objeto (2): usado quando o servidor do recurso não é o processo que recebe as requisições sobre o recurso. Nesta arquitetura, existe um processo gerente de recursos, que recebe uma requisição e inicializa o correspondente servidor do recurso para processá-la. É necessário que o processo seja único em cada máquina, e que o nome de seu **PortIn** (processname, portname) seja conhecido pelo servidor do recurso, para que este possa chamar o construtor com os parâmetros corretos.

- `istherePortIn`: quando o Construtor de Objeto (2) é usado, este método pode ser invocado em seguida para verificar se o endereço de entrada de requisições (`addr`) está corretamente configurado, retornando `TRUE`, em caso positivo, e `FALSE`, caso contrário. Isto porque este Construtor procura no Servidor de Nomes pelo nome da porta, e se este nome não existir ou a chamada ao Servidor de Nomes falhar, então o atributo `addr` não estará correto.

- `Insert`: interface de acesso ao método `NameServer::InsertResource`, que insere o recurso `resname` e retorna um *ID*. É usado pelo servidor do recurso.

- `Remove`: interface de acesso ao método `NameServer::RemoveResource`, que remove o recurso `resID`. É usado pelo servidor do recurso.

- `Locate`: interface de acesso ao método `NameServer::LocateResource`, que procura pelo recurso `resname` e retorna sua localização em `resaddr` e seu *ID*. É usado pelo cliente do recurso.

No Apêndice A (exemplo A.7) é dado um exemplo de utilização da classe **NameClient**.

4.3.5 Classe Communic

Todas as funcionalidades providas pelo Sistema de Comunicação e utilizadas através de objetos de interface com cliente são gerenciadas pelo objeto **Communic** (Figura 4.8). Entre estas funcionalidades estão:

- controle da tabela de portas de entrada locais;

- controle das filas de mensagens de cada **PortIn**;

- transmissão de mensagens, requisitadas por um **PortOut** ou **PortOutMulticast**;

- comunicação cliente/servidor com o Servidor de Nomes e o Servidor de Grupos.

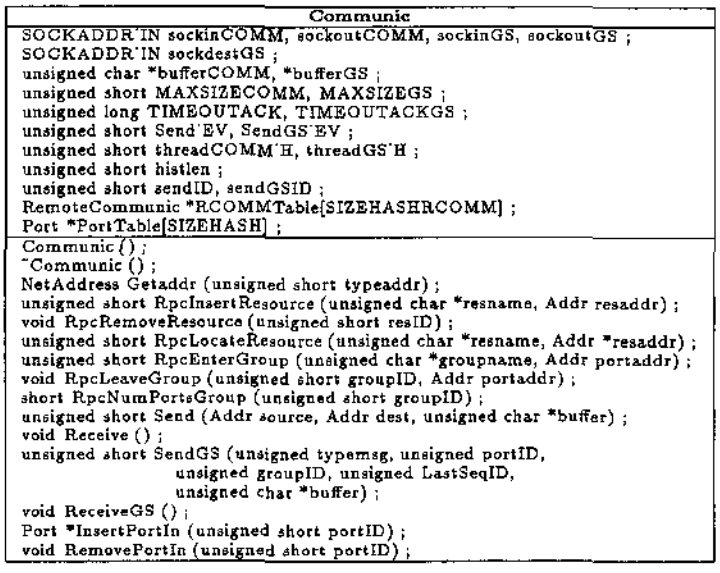


Figura 4.8: Diagrama da Classe **Communic**

O processo de comunicação de baixo nível de **Communic** (entrada e saída de mensagens) é executado através de chamadas a rotinas do protocolo UDP/IP em uma topologia de rede *Ethernet*, presente no ambiente local.

Os atributos principais de **Communic** são:

- *sockets*[3] de entrada de mensagens (`sockinCOMM` e `sockinGS`) e de saída de mensagens (`sockoutCOMM` e `sockoutGS`). Apesar dos *sockets* poderem ser usados para entrada e saída, eles não suportam dois ou mais processos leves com acesso concorrente ao mesmo *socket*. Como este sistema possui vários processos leves entre as funções de envio e de recepção de mensagens, então é necessário que os *sockets* de entrada e de saída sejam independentes.

- `sockdestGS`: *socket* que armazena o endereço de entrada de mensagens do Servidor de Grupos;

[3] *sockets* constitui uma interface para a comunicação em rede usando protocolos como o UDP/IP, entre outros.

- *buffers* que armazenam mensagens que chegam de um **Communic** remoto (`bufferCOMM`) ou do Servidor de Grupos (`bufferGS`);
- tamanhos máximos das mensagens a receber de um **Communic** remoto (`MAXSIZECOMM`) ou do Servidor de Grupos (`MAXSIZEGS`);
- valores de *timeout* de espera de **ACK** de um **Communic** remoto (`TIMEOUTACK`) e de espera de **ACK** do Servidor de Grupos (`TIMEOUTACKGS`), dados em milisegundos;
- eventos relacionados à espera de **ACK** de um **Communic** remoto (`Send_EV`) ou de espera de **ACK** do Servidor de Grupos (`SendGS_EV`);
- identificadores dos processos leves `Receive` (`threadCOMM_H`) e `ReceiveGS` (`threadGS_H`);
- `histlen`: tamanho do *buffer* de histórico do Servidor de Grupos;
- números de seqüência da próxima mensagem a enviar para um **Communic** remoto (`sendID`) ou para o Servidor de Grupos (`sendGSID`);
- `RCOMMTable`: tabela de controle de ordenação de mensagens ponto-a-ponto recebidas de objetos **Communic** remotos, usada para detectar perda ou duplicação de mensagens;
- `PortTable`: tabela de portas de entrada locais.

Os métodos fornecidos por **Communic** são:

- Construtor de Objeto: inicializa o Sistema de Comunicação local. O Construtor executa os seguintes passos: inicializar os *sockets*, inicializar o ambiente de CPR (para a comunicação com o Servidor de Nomes e o Servidor de Grupos), alocar os *buffers* com os respectivos tamanhos máximos, inicializar os valores de *timeouts*, criar os eventos, inicializar `histlen`, inicializar os números de seqüência com 1, inicializar as tabelas dinamicamente, e instanciar um processo leve associado à recepção de mensagens de objetos **Communic** remotos (`Receive`) e outro associado à recepção de mensagens do Servidor de Grupos (`ReceiveGS`).

- Destrutor de Objeto: finaliza a operação do Sistema de Comunicação local. O Destrutor executa os seguintes passos: remover o ambiente de comunicação UDP/IP e CPR, os *buffers* e os eventos, matar os processos leves e remover as tabelas.

- `Getaddr`: retorna o endereço local UDP/IP. Se `typeaddr` é igual a `ADDRCOMM`, é retornado o endereço de `sockinCOMM`; se é igual a `ADDRGS`, é retornado o endereço de `sockinGS`. É utilizado por **PortIn** e por **NameClient** para, junto com o método `PortIn::GetportID`, fornecer um endereço local completo na chamada à primitiva `InsertResource` do Servidor de Nomes.

- `RpcInsertResource`: função CPR de acesso à primitiva `InsertResource` do Servidor de Nomes.

- `RpcRemoveResource`: função CPR de acesso à primitiva `RemoveResource` do Servidor de Nomes.

- `RpcLocateResource`: função CPR de acesso à primitiva `LocateResource` do Servidor de Nomes.

- `RpcEnterGroup`: função CPR de acesso à primitiva `EnterGroup` do Servidor de Grupos.

- `RpcLeaveGroup`: função CPR de acesso à primitiva `LeaveGroup` do Servidor de Grupos.

- `RpcNumPortsGroup`: função CPR de acesso à primitiva `NumPortsGroup` do Servidor de Grupos.

- `Send`: método de transmissão de uma mensagem a um **Communic** remoto e espera de **ACK**. Fica bloqueado até que `Receive` receba sua mensagem de **ACK** e sinalize o evento `Send_EV` ou que `TIMEOUTACK` expire e o Sistema de Detecção de Falhas seja ativado. Retorna `TRUE`, se conseguiu enviar a mensagem, e `FALSE`, caso contrário.

- `Receive`: método (processo leve `threadCOMM_H`) que fica continuamente capturando mensagens provenientes de objetos **Communic** remotos. Se a mensagem for **ACK**, é sinalizado o evento `Send_EV` (caso

TIMEOUTACK não expirou). Caso seja dados, enfileira a mensagem na porta de entrada adequada, se não for uma duplicação.

- `SendGS`: método de transmissão de uma mensagem ao Servidor de Grupos e espera de **ACK**. Fica bloqueado até que `ReceiveGS` receba sua mensagem de **ACK** e sinalize o evento `SendGS_EV` ou que `TIMEOUTACKGS` expire e o Sistema de Detecção de Falhas seja ativado. Transmite mensagens do tipo **DATA** ou **RETRANS**, segundo o algoritmo de Kaashoek. Retorna `TRUE`, se conseguiu enviar a mensagem, e `FALSE`, caso contrário.

- `ReceiveGS`: método (processo leve `threadGS_H`) que fica continuamente capturando mensagens provenientes do Servidor de Grupos. Se a mensagem for **ACK**, é sinalizado o evento `SendGS_EV` (caso `TIMEOUTACKGS` não expirou). Caso seja **DATA**, se o número de seqüência estiver correto, enfileira a mensagem na porta de entrada adequada; caso contrário, retorna mensagem **RETRANS** através de `SendGS`. Trata também mensagens **SYNC** e **COMMIT**, de acordo com o algoritmo de Kaashoek.

- `InsertPortIn`: insere uma nova porta na tabela de portas de entrada locais.

- `RemovePortIn`: remove uma porta da tabela de portas de entrada locais, removendo inclusive as mensagens que estiverem pendentes.

Após a instanciação, o objeto **Communic** possui, no mínimo, dois processos leves executando seu código (`Receive` e `ReceiveGS`). Cada chamada que um objeto de interface com cliente faz a **Communic** implica em um novo processo leve entrando em seu código. A implementação de **Communic** possui vários trechos de código com sincronização e controle de concorrência, de forma a controlar o acesso de múltiplos processos leves aos dados compartilhados. Com isso, é possível que uma aplicação distribuída possua vários processos leves usando um objeto **Communic** simultaneamente, através de objetos de interface com cliente.

Processo de Inicialização e Finalização

Quando o objeto **Communic** é instanciado, alguns parâmetros de execução devem ser inicializados de forma a refletir o comportamento local da aplicação distribuída. Na execução do Construtor, são lidos três arquivos, os quais contêm parâmetros para **Communic**:

- `communic.in`: `TIMEOUTACK`, `TIMEOUTACKGS`, `MAXSIZECOMM`, `MAXSIZEGS`, `histlen`.

  1. `TIMEOUTACK`, `TIMEOUTACKGS`: valores em milisegundos. Os valores de *timeout* devem ser inicializados de acordo com as recomendações abaixo, a fim de minimizar diagnósticos errôneos de falha de um nó remoto:
     - `TIMEOUTACK`: valor máximo de espera de **ACK** de um **Communic** remoto. Tal valor deve levar em consideração a localização do nó na topologia e a distância máxima em relação a qualquer outro nó. Está relacionado ao envio de mensagens a um **Communic** remoto, feito por `Communic::Send`.
     - `TIMEOUTACKGS`: valor máximo de espera de **ACK** do Servidor de Grupos, inicializado de acordo com sua distância ao mesmo. Está relacionado ao envio de mensagens ao Servidor de Grupos, feito por `Communic::SendGS`.

     Estes parâmetros de *timeout* são dinâmicos, baseados na localização de cada nó em relação ao interlocutor.
  2. `MAXSIZECOMM`, `MAXSIZEGS`: indicam o tamanho máximo de uma mensagem ponto-a-ponto e de uma mensagem *multicast*.
  3. `histlen`: deve conter o mesmo valor do parâmetro `histlen` citado na Seção 4.3.6, pois este parâmetro está diretamente relacionado com o algoritmo de Kaashoek, e faz parte do esquema de controle de *multicasts* em cada **Communic**. Se a parte de controle de *multicast* não é usada por uma aplicação distribuída, basta zerar este parâmetro. Com isso, não serão criados `bufferGS`, o evento `SendGS_EV` e o processo leve `ReceiveGS`, além de não ler o arquivo `gs.cfg`.

A atribuição coerente de valores para os referidos parâmetros é de fundamental importância para a execução correta do Sistema de Comunicação.

- `ns.cfg`: arquivo gerado pelo Servidor de Nomes, que contém o seu endereço para a conexão por CPR.

- `gs.cfg`: arquivo gerado pelo Servidor de Grupos, que contém o seu endereço para a conexão por CPR e o endereço do seu *socket* de entrada (usado para preencher o atributo `sockdestGS`).

Quando o nó local da aplicação distribuída termina sua execução, o Destrutor de **Communic** é disparado, de forma a retirar de memória toda a configuração do Sistema de Comunicação. Com isso, quaisquer mensagens endereçadas a este nó serão perdidas, após o término de sua execução, e o remetente irá considerar este nó como falho, a partir deste instante. A ocorrência deste fato provavelmente indicará um erro de sincronização entre os nós da aplicação distribuída, o que deve ser analisado e corrigido. Um exemplo desta pseudo-falha ocorre quando o emissor envia uma mensagem e, enquanto espera o **ACK**, o receptor consome a mensagem, retorna o **ACK** e encerra sua execução. Se o **ACK** é perdido na rede, o emissor achará que o receptor falhou, o que não é verdade.

Ordenação de mensagens

Tanto mensagens ponto-a-ponto quanto *multicast* são enviadas com números de seqüência, que identificam unicamente a mensagem proveniente de um **Communic** ou Servidor de Grupos.

As mensagens ponto-a-ponto possuem um número de seqüência usado no **Communic** destino para detectar duplicações. Quando a mensagem chega ao destino, o método `Communic::Receive` retorna um **ACK** e verifica, na sua tabela de controle ponto-a-ponto (`RCOMMTable`), se o número relativo ao **Communic** origem é menor que o número recebido. Em caso afirmativo, é uma nova mensagem que deve ser enfileirada em uma porta de entrada; caso contrário, é uma duplicação que deve ser descartada. Este último caso pode ocorrer quando o emissor não recebe o **ACK** (sua mensagem ainda não

chegou ao destino, o **ACK** não chegou ao emissor ou o **ACK** chegou após `TIMEOUTACK`[4]) e retransmite a mensagem.

As mensagens *multicast* são enviadas em duas fases. A primeira fase consiste na transmissão ponto-a-ponto do **Communic** emissor para o Servidor de Grupos, quando a mensagem apresenta o campo número de seqüência, usado no Servidor de Grupos para detectar duplicações, da mesma forma que mensagens ponto-a-ponto entre objetos **Communic**. Portanto, o Servidor de Grupos também possui uma tabela de controle ponto-a-ponto e analisa os números de seqüência das mensagens. A segunda fase é a transmissão do *multicast* do Servidor de Grupos para os membros do grupo, baseada no algoritmo de *multicast* (Seção 4.3.6). Nas duas fases há um controle de mensagens duplicadas ou perdidas, garantindo a ordem de recepção.

Tipos/Formatos dos pacotes ponto-a-ponto entre Sistemas de Comunicação

Existem dois tipos de pacotes de mensagens:

- dados: usado por `Communic::Send` para enviar mensagens normais entre processos. O formato do pacote é o seguinte:

Campo	Significado
addrACK	`sockinCOMM`, para retorno de **ACK**
portIDdest	identificador do **PortIn** remoto
portIDsrc	identificador do **PortIn** local, para retorno de resposta (usado por `PortOut::SendDatagram` e `PortOut::Send`)
sendID	número de seqüência da mensagem (controle de duplicação)
buffer	dados

- **ACK**: ao receber uma mensagem, `Communic::Receive` retorna um **ACK** segundo este pacote:

Campo	Significado
addrACK	igual a 0, indicando mensagem **ACK**
sendID	mesmo valor do pacote de dados (controle de duplicação)

[4]Quando as respostas chegam após ocorrer o *timeout*, elas são descartadas pelos emissores.

4.3.6 Servidor de Grupos

Devido ao fato do algoritmo de Kaashoek ser um subconjunto da operação do Servidor de Grupos, existem algumas adições em relação ao original, encontradas durante o projeto do servidor.

No algoritmo de Kaashoek, existe um coordenador para cada grupo criado. Portanto, ele conhece apenas o grupo ao qual pertence. O Servidor de Grupos é único no ambiente distribuído, armazenando informações sobre todos os grupos existentes.

Originalmente, quando um membro do grupo envia uma mensagem ao coordenador, ele recebe informalmente um **ACK** quando a sua própria mensagem chega, a partir de um *broadcast* do coordenador aos membros do grupo. Enquanto isso, no algoritmo modificado, um membro do grupo (um **PortOutMulticast** e, opcionalmente, um **PortIn**), ao enviar uma mensagem para o grupo, espera um **ACK** de resposta do Servidor de Grupos. Isto ocorre pois, como o objeto **PortIn** é opcional, não há como confirmar a chegada da mensagem ao Servidor de Grupos através da espera de sua própria mensagem, transmitida para o grupo de objetos **PortIn**. O mesmo esquema de espera de **ACK** é usado para pedidos de retransmissão.

O algoritmo de Kaashoek, ao entrar na Operação de Sincronização, exige que todos os membros do grupo travem o envio de mensagens até que termine a sincronização. Todavia, o Servidor de Grupos continua aceitando pedidos de envio de *multicast* durante esta operação. Ele enfileira os pedidos e processa-os após a sincronização. Desta forma, `PortOutMulticast::SendDatagram` é executada normalmente (assincronamente) e libera o processo correlato para continuar sua execução, após enviar a mensagem, estando o grupo em sincronização ou não.

Outro aspecto importante para análise é o *buffer* de histórico. Dada uma aplicação distribuída cujo fluxo de *multicasts* entre os membros do grupo é pequeno (constantemente pequeno ou pequeno em um certo período de tempo), tal que o número de *multicasts* não é suficiente para lotar o *buffer*, então pode existir uma situação problemática. Suponha que o *buffer* de histórico tenha capacidade para armazenar 16 mensagens e que, no momento, exista nele apenas uma mensagem. Seja uma aplicação, onde um servidor mestre envia um *multicast* para um conjunto de servidores escravos, que devem efetuar uma certa operação e retornar uma resposta para o mestre. O problema ocorre quando, por algum motivo, um dos escravos não recebe o

multicast. Com isso, ele não executa a operação local e o mestre não recebe uma resposta deste escravo em tempo hábil (por exemplo, ocorre o *timeout* de espera de resposta). Além disso, o escravo não confirma a recepção do *multicast* e nem pede por retransmissão para o Servidor de Grupos, pois está esperando uma requisição do mestre. Como o *buffer* não vai lotar, não haverá sincronização. Neste exemplo, o mestre fica esperando uma resposta do escravo e este fica esperando uma requisição do mestre (*deadlock*).

A partir deste problema, conclui-se que o algoritmo só pode garantir atomicidade e confiabilidade de entrega de *multicasts* após cada Operação de Sincronização ou se, durante a Operação Normal, todos os membros do grupo enviam **DATA** (e **RETRANS**) de forma sistemática.

Para aplicações como acima, uma solução é fornecer o método `PortOutMulticast::GetMessages`. Quando, em um ambiente distribuído com baixo fluxo de *multicasts* no grupo, um membro do grupo não recebe uma mensagem *multicast* em um certo período suficientemente grande de tempo, então pode ter ocorrido uma de duas possibilidades:

- falha no envio da mensagem do emissor para o Servidor de Grupos, que é detectada por `Communic::SendGS` do emissor e pode ser impossível de se resolver no momento (por exemplo, se ocorreu uma ruptura de cabo *Ethernet*);

- falha no envio da mensagem a partir do Servidor de Grupos para os destinatários, não detectada pelo Servidor de Grupos. Neste caso, um receptor (membro do grupo) pode invocar `PortOutMulticast::GetMessages` para pedir por mensagem ao Servidor de Grupos. Este método invoca `Communic::SendGS` com `typemsg` igual a **RETRANS** e, caso o problema tenha sido apenas a perda da mensagem após emissão pelo Servidor de Grupos, então ela será retransmitida ao membro do grupo. Se a falha foi causada por ruptura do cabo, por exemplo, então `Communic::SendGS` irá detectá-la e retornará a condição de impossibilidade de comunicação através da rede.

Com isso, se não ocorrerem falhas que impossibilitem a comunicação entre um membro do grupo e o Servidor de Grupos, então é possível requerer uma ou mais mensagens ao Servidor de Grupos sem que, entretanto, necessite de sincronização ou de envio de *multicasts* para o grupo.

Relação com o Sistema de Comunicação

O Servidor de Grupos apresenta-se como um processo de execução independente das demais partes, sendo ativado em somente uma máquina do domínio da rede e cuja localização é previamente conhecida pelos Sistemas de Comunicação existentes.

Quando um objeto **PortOutMulticast** deseja entrar/sair de um grupo ou obter informações sobre o grupo ao qual pertence, ele efetua uma interação cliente/servidor com o Servidor de Grupos, através de **Communic**, parametrizada por um destes métodos. O Servidor de Grupos efetua a operação na sua tabela de grupos e retorna um resultado. A partir disso, o objeto continua sua execução.

Para enviar uma mensagem (dados ou pedido de retransmissão) para o grupo, o objeto **PortOutMulticast** interage com o Servidor de Grupos por mensagens ponto-a-ponto, através de `Communic::SendGS`.

O objeto `Communic` também faz o papel de núcleo do algoritmo de Kaashoek, através de `Communic::SendGS/Communic::ReceiveGS`, tratando as mensagens do tipo **ACK**, **DATA**, **SYNC** e **COMMIT**, e emitindo as mensagens do tipo **DATA**, **RETRANS**, **ACK-SYNC** e **ACK-COMMIT**.

Classe GroupServer

O processo de execução do Servidor de Grupos resume-se à instanciação de um objeto **GroupServer**, que fica continuamente à espera de requisições de clientes (Figura 4.9).

Seus atributos principais são:

- `sockinGS`: *socket* de entrada de mensagens;
- valores de *timeout* de espera de **ACK-SYNC** (`TIMEOUTACKSYNC`) e de **ACK-COMMIT** (`TIMEOUTACKCOMMIT`) de todos os membros de um grupo, dados em milisegundos;
- `bufferGS`: *buffer* que armazena as mensagens que chegam de um membro de um grupo;
- `MAXSIZEGS`: tamanho máximo das mensagens a receber de um membro de um grupo;

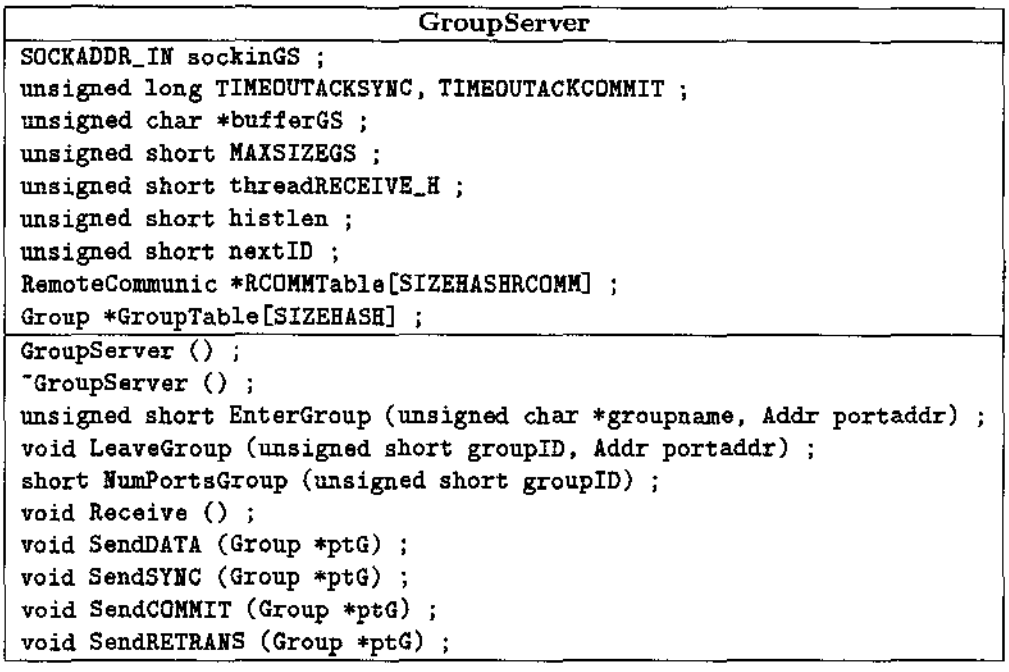


Figura 4.9: Diagrama da Classe **GroupServer**

- `threadRECEIVE_H`: identificador do processo leve `Receive`;
- `histlen`: tamanho máximo do *buffer* de histórico;
- `nextID`: próximo identificador numérico de grupo, usado para localizar um grupo univocamente no ambiente distribuído, cujo valor conceitualmente nunca se repete durante o ciclo de vida do Servidor de Grupos;
- `RCOMMTable`: tabela de controle de ordenação de mensagens ponto-a-ponto recebidas de membros de grupos, usada para detectar perda ou duplicação de mensagens;
- `GroupTable`: tabela de grupos.

Quando um novo grupo é criado, as informações sobre o mesmo são colocadas em uma estrutura (`Group`), inserida na tabela de grupos. Esta estrutura contém as seguintes informações:

- nome do grupo;
- identificador do grupo;
- número de objetos **PortOutMulticast** no grupo;
- lista de endereços completos de objetos **PortIn** no grupo;
- contador de sincronizações;
- apontadores para a fila que representa o *buffer* de histórico do grupo;
- apontadores para a fila de pedidos de retransmissão do grupo;
- número de seqüência da próxima mensagens a enviar para o grupo;
- *sockets* de envio de mensagens *multicast* ou retransmissões para o grupo;
- identificadores dos processos leves `SendDATA` e `SendRETRANS`;
- eventos relacionados à espera de mensagens a enviar para o grupo (`SendDATA_EV`), à espera de pedidos de retransmissão (`SendRETRANS_EV`) e à espera de **ACK-SYNC** ou **ACK-COMMIT** (`SendSYNC_EV`).

Os métodos fornecidos pelo Servidor de Grupos são:

- Construtor de Objeto: inicializa o Servidor de Grupos para utilização pelos clientes. O Construtor executa os seguintes passos: inicializar o *socket* do objeto, inicializar o ambiente de CPR, alocar o *buffer* com o respectivo tamanho máximo, inicializar os valores de *timeouts*, inicializar `histlen`, inicializar *nextID* com 1, inicializar as tabelas dinamicamente, instanciar um processo leve associado à recepção de mensagens (`Receive`), e gravar o endereço local de rede e o endereço de `sockinGS` no arquivo `gs.cfg`.
- Destrutor de Objeto: finaliza a operação do Servidor de Grupos. O Destrutor executa os seguintes passos: remover o ambiente de comunicação UDP/IP e CPR, remover o *buffer*, matar os processos leves e remover as tabelas.

- `EnterGroup`: insere um objeto **PortOutMulticast** (e seu **PortIn**, representado pelo endereço completo de entrada `portaddr`) no grupo `groupname`. Se este grupo não existe, ele deve ser criado na tabela de grupos e é retornado o próximo *ID* válido; caso contrário, é retornado o *ID* do grupo.

- `LeaveGroup`: remove um objeto **PortOutMulticast** (e seu **PortIn**, representado por `portaddr`) do grupo identificado pelo `groupID`. Se este objeto for o único membro atualmente no grupo, o grupo é removido da tabela de grupos.

- `NumPortsGroup`: retorna o número de portas de entrada do grupo `groupID`.

  Através dos métodos `EnterGroup`, `LeaveGroup` e `NumPortsGroup`, o Servidor de Grupos fornece uma interface básica de manutenção de grupos.

- `Receive`: método (processo leve) que fica continuamente capturando mensagens provenientes de Sistemas de Comunicação. Se a mensagem for **DATA**, transfere a mensagem para a fila (*buffer* de histórico) do processo leve `SendDATA` do respectivo grupo, sinalizando o evento `SendDATA_EV`; caso seja **RETRANS**, enfileira o pedido para `SendRETRANS` do respectivo grupo, sinalizando o evento `SendRETRANS_EV`; caso seja **ACK-SYNC** ou **ACK-COMMIT**, sinaliza `SendSYNC_EV` do grupo.

- `SendDATA`: método (processo leve) criado para cada grupo. Fica continuamente esperando mensagens para enviar e, quando existe alguma (evento `SendDATA_EV` foi sinalizado), envia um *multicast* para o grupo. Após enviar a mensagem, verifica se o *buffer* de histórico atingiu o limite `histlen`, chamando `SendSYNC` em caso positivo.

- `SendSYNC`: executa a primeira fase da sincronização, enviando mensagem tipo **SYNC** para todos os membros do grupo e esperando `TIMEOUTACKSYNC` milisegundos pelas confirmações de chegada (**ACK-SYNCs**) de todas as mensagens pendentes por todos os membros do grupo. Se o *timeout* ocorrer, o Sistema de Detecção de Falhas é ativado

para apurar a causa desta falha e habilitar ou não a retransmissão de **SYNC**. Ao final, chama `SendCOMMIT`.

- `SendCOMMIT`: executa a segunda fase da sincronização, enviando mensagem tipo **COMMIT** para todos os membros do grupo e esperando `TIMEOUTACKCOMMIT` milisegundos pelos **ACK-COMMITs** de todos os membros. Se o *timeout* ocorrer, o Sistema de Detecção de Falhas é ativado para apurar a causa desta falha e habilitar ou não a retransmissão de **COMMIT**.

- `SendRETRANS`: método (processo leve) criado para cada grupo. Fica continuamente esperando pedidos de retransmissão de uma fila, enviando uma ou mais mensagens do *buffer* de histórico ponto-a-ponto para o membro do grupo. Fica bloqueado até que `Receive` sinalize o evento `SendRETRANS_EV`.

Após a instanciação, apenas dois processos leves estão ativos em **GroupServer**: um processo leve principal, que espera por CPRs, e o processo leve `Receive`, que recebe as mensagens endereçadas a grupos. Quando um grupo é criado, são iniciados dois processos leves: um para enviar mensagens de dados ou sincronização e o outro para processar pedidos de retransmissão. Ao chegar uma mensagem, `Receive` a despacha para um grupo, que realiza a transmissão *multicast* (dados ou sincronização) ou ponto-a-ponto (retransmissão).

Processo de Inicialização e Finalização

Na execução do Construtor, é lido o arquivo `grouserv.in`, que contém parâmetros para definir a operação de **GroupServer**:

- `TIMEOUTACKSYNC`, `TIMEOUTACKCOMMIT`: valores em milisegundos.

  - `TIMEOUTACKSYNC` é o tempo máximo entre o envio de mensagem tipo **SYNC** e espera de confirmações (**ACK-SYNCs**) por parte de todos os membros de um grupo em sincronização;
  - `TIMEOUTACKCOMMIT` é o tempo máximo entre o envio de mensagem tipo **COMMIT** e espera de **ACK-COMMIT** de cada membro de um grupo em sincronização.

O valor dado para `TIMEOUTACKSYNC` deve ser maior ou igual a `TIMEOUTACKCOMMIT`, pois a primeira fase da sincronização é, normalmente, mais demorada, devido à possível ocorrência de retransmissões de mensagens para alguns membros do grupo antes do retorno do **ACK-SYNC**, enquanto o **ACK-COMMIT** é retornado por todos os membros automaticamente após receberem um **COMMIT**.

Os valores de *timeout* vão depender da localização dos membros de um grupo. O grupo com membros mais distantes do Servidor de Grupos (na média), em comparação com outros grupos, que deve influenciar os valores para os *timeouts*.

- `MAXSIZEGS`: valor entre 1 e 65535 bytes de mensagem.
- `histlen`: tamanho máximo do *buffer* de histórico, ou seja, número máximo de mensagens que são armazenadas neste *buffer* antes de entrar em sincronização. Seu valor não deve ser muito pequeno, a fim de evitar sincronizações precoces; nem deve ser muito grande, para limitar o número de mensagens sem confirmação e garantir a confiabilidade.

Quando o Servidor de Grupos recebe uma chamada de *shutdown*, ele ativa o Destrutor para remover a memória utilizada e terminar a execução, mesmo que hajam mensagens pendentes, grupos em sincronização e pedidos de retransmissão. Com isso, as pendências causarão a detecção de falhas pelos nós remotos envolvidos. A chamada à *shutdown* é feita pelo programa utilitário `shdowngs`. A remoção do processo Servidor de Grupos por sinal do tipo *kill* não permite a chamada do Destrutor e, com isso, a finalização normal, não devendo ser utilizada.

Tabela de Grupos

A tabela de grupos é uma *hash* de apontadores para estruturas `Group`.

A inserção de um novo grupo requer busca por nomes na tabela e a remoção é feita com busca por *ID*. A inserção de um novo membro em um grupo executa uma busca seqüencial por endereço na lista de portas de entrada (se uma porta de entrada é especificada como parâmetro), e o incremento do número de portas de saída de *multicast*. A remoção de um membro do grupo executa a mesma busca por endereço da inserção e o decremento do número de portas de saída de *multicast*. A inserção de um novo membro

em um grupo que ainda não existe implica na automática criação do grupo. A remoção do último membro de um grupo causa a remoção do grupo.

O *ID* gerado para um grupo é igual ao anterior mais 1, variando de 1 a limite superior de inteiro sem sinal da arquitetura da máquina. Eventualmente, esta limitação pode causar problema com o estouro do limite de *IDs*.

Tipos/Formatos dos pacotes entre Sistema de Comunicação e Servidor de Grupos

Existem vários tipos de pacotes para interação entre um Sistema de Comunicação e o Servidor de Grupos:

- **DATA/RETRANS**: utilizado por `Communic::SendGS` para interface com o Servidor de Grupos. Quando um membro de um grupo deseja enviar dados ou `Communic::Receive/PortOutMulticast::GetMessages` emite um pedido de retransmissão, é usado o seguinte pacote:

Campo	Significado
addrACK	`sockinGS` de **Communic**, para retorno de **ACK**
typemsg	tipo de mensagem (**DATA** ou **RETRANS**)
portID	identificador do **PortIn**. Usado no Servidor de Grupos para confirmar mensagens para esta porta de entrada. Se o objeto **PortOutMulticast** não possui um **PortIn** associado, este campo é zerado (opcional).
groupID	identificador do grupo destino da mensagem
LastSeqID	*LastSeqReceived* do algoritmo de Kaashoek. Número de seqüência da última mensagem recebida. Usado no Servidor de Grupos para confirmar mensagens (opcional).
sendID	*MessageID* do algoritmo de Kaashoek. Número de seqüência da mensagem. (controle de duplicação)
buffer	dados (somente **DATA**)

- **ACK**: ao receber mensagem de **DATA** ou **RETRANS**, o Servidor de Grupos retorna um **ACK** segundo este pacote:

Campo	Significado
typemsg	igual a **ACK**
sendID	mesmo valor do pacote **DATA/RETRANS** (controle de duplicação)

- **DATA**: usado por `GroupServer::SendDATA` para envio de mensagens de dados para um grupo de portas de entrada, interagindo com vários `Communic::ReceiveGS`. O método `GroupServer::SendRETRANS` usa o mesmo pacote para mensagens de dados ponto-a-ponto para uma específica porta de entrada:

Campo	Significado
typemsg	igual a **DATA**
portID	identificador da porta de entrada destino (igual a 0 para `GroupServer::SendDATA`)
groupID	identificador do grupo destino (igual a 0 para `GroupServer::SendRETRANS`)
sendID	*NextSeqtoUse* do algoritmo de Kaashoek (`GroupServer::SendDATA`) ou número de seqüência requerido pelo membro do grupo (`GroupServer::SendRETRANS`). Número de seqüência da mensagem. Usado, em conjunto com *portID* ou *groupID*, por `Communic::ReceiveGS`, na análise de ordenação de mensagens provenientes de um grupo
buffer	dados

- **SYNC**: usado por `GroupServer::SendSYNC` para o envio de um aviso da execução da primeira fase de sincronização:

Campo	Significado
typemsg	igual a **SYNC**
groupID	identificador do grupo destino da mensagem
sendID	*NextSeqtoUse* do algoritmo de Kaashoek. Usado por `Communic::ReceiveGS` para verificar se cada porta de entrada presente neste grupo recebeu todas as mensagens e deve retornar **ACK-SYNC** ou se é necessário retornar **RETRANS**.

- **ACKSYNC**: usado pelo Sistema de Comunicação para confirmar a primeira fase de sincronização ao Servidor de Grupos:

Campo	Significado
typemsg	igual a **ACK-SYNC**
portID	identificador da porta de entrada. Usado no Servidor de Grupos para confirmar mensagens para esta porta de entrada.
groupID	identificador do grupo destino da mensagem

- **COMMIT**: usado por `GroupServer::SendCOMMIT` para o envio de um aviso da execução da segunda fase de sincronização:

Campo	Significado
typemsg	igual a **COMMIT**
groupID	identificador do grupo destino da mensagem

- **ACKCOMMIT**: usado pelo Sistema de Comunicação para confirmar a segunda fase de sincronização ao Servidor de Grupos. Sua estrutura é semelhante ao pacote **ACKSYNC**, sendo que *typemsg* é igual a **ACK-COMMIT**.

4.3.7 Servidor de Nomes

Funciona como um servidor que espera requisições de clientes via CPR.

Relação com o Sistema de Comunicação

O Servidor de Nomes apresenta-se como um processo de execução independente das demais partes, sendo ativado em somente uma máquina do domínio da rede e cuja localização é previamente conhecida pelos Sistemas de Comunicação existentes.

Como visto anteriormente, o Sistema de Comunicação constitui-se no módulo de mais baixo nível, sendo utilizado por vários outros módulos que necessitam de interação remota. Porém, para iniciar qualquer tipo de interação, baseada em portas de comunicação ou em recursos compartilhados,

é necessário conhecer a localização do interlocutor (através da localização destes recursos[5]). Por isso, a inicialização do objeto **Communic** de cada nó da aplicação requer o endereço de rede do Servidor de Nomes.

Quando um membro no ambiente distribuído deseja criar/remover um recurso ou localizar um recurso, ele efetua uma interação cliente/servidor com o Servidor de Nomes parametrizada por um destes métodos. O Servidor de Nomes efetua a operação remotamente na sua tabela de nomes e retorna um resultado. A partir disso, o membro continua sua execução.

Classe NameServer

O processo de execução do Servidor de Nomes resume-se à instanciação do objeto **NameServer**, que fica continuamente à espera de requisições de clientes (Figura 4.10).

NameServer
`unsigned short nextID ;` `Name *NameTable[SIZEHASHNS] ;`

`NameServer () ;`
`~NameServer () ;`
`unsigned short InsertResource (unsigned char *resname, Addr resaddr) ;`
`void RemoveResource (unsigned short resID) ;`
`unsigned short LocateResource (unsigned char *resname, Addr *resaddr) ;`

Figura 4.10: Diagrama da Classe **NameServer**

Seus atributos são:

- `nextID`: próximo identificador numérico de recurso, usado para localizar um recurso univocamente no ambiente distribuído, cujo valor conceitualmente nunca se repete durante o ciclo de vida do Servidor de Nomes;
- `NameTable`: tabela de nomes.

Os métodos fornecidos pelo Servidor de Nomes são:

[5]Em termos de Servidor de Nomes, portas de comunicação e recursos compartilhados são considerados recursos.

- Construtor de Objeto: inicializa o Servidor de Nomes para utilização pelos clientes. O Construtor executa os seguintes passos: inicializar o ambiente de CPR, inicializar *nextID* com 1, inicializar a tabela de nomes dinamicamente, e gravar o endereço local no arquivo `ns.cfg`.

- Destrutor de Objeto: finaliza a operação do Servidor de Nomes. O Destrutor executa os seguintes passos: remover o ambiente de CPR e remover a tabela de nomes.

- `InsertResource`: insere um recurso identificado pelo nome `resname` e pelo endereço {origem UDP/IP, *ID* da porta de entrada do **NameClient**} (recursos compartilhados[6]) ou {origem UDP/IP, 0} (portas de entrada) na tabela de nomes. Caso este nome já exista, esta primitiva retorna um *ID* nulo; senão, ela retorna o próximo *ID* válido.

- `RemoveResource`: remove um recurso identificado por seu `resID` da tabela de nomes e retorna para o Sistema de Comunicação remoto.

- `LocateResource`: procura pelo nome `resname` na tabela de nomes e retorna um endereço completo, caso o encontre, e um endereço nulo, caso contrário. No parâmetro `resaddr` é retornado o endereço de acesso ao recurso e o retorno da função corresponde ao *ID* do recurso.

  Através dos métodos `InsertResource`, `RemoveResource` e `LocateResource`, o Servidor de Nomes fornece uma interface básica de manutenção e acesso a recursos dos Sistemas de Comunicação (portas de entrada) e dos servidores de recursos (recursos compartilhados).

Processo de Inicialização e Finalização

Quando o Servidor de Nomes recebe uma chamada de *shutdown*, ele ativa o Destrutor para remover a memória utilizada e terminar a execução, mesmo que hajam mensagens pendentes. A perda de mensagens de CPRs causará a detecção de falha pelos nós remotos envolvidos. A chamada à *shutdown* é feita pelo programa utilitário `shdownns`. A remoção do processo Servidor

[6]Os recursos compartilhados, sob tutela de um servidor de recurso, que utiliza o Servidor de Nomes através de **NameClient**, são alcançados através da porta de entrada do servidor de recurso. Por isso, há a necessidade de *portID*.

de Nomes por sinal do tipo *kill* não permite a chamada ao Destrutor e, com isso, a finalização normal. Por isso, o uso deste esquema de finalização não é recomendado.

Tabela de Nomes

A tabela de nomes é estruturada da seguinte maneira: uma *hash* de estruturas (representações de recursos) com nome, *portID* de acesso ao recurso (para recursos compartilhados), *ID* de recurso e endereço de rede (localização do recurso).

A inserção de nomes requer busca por nomes e a remoção é feita com busca por *ID*.

O *ID* gerado para um recurso é igual ao anterior mais 1, variando de 1 a limite superior de inteiro sem sinal da arquitetura da máquina. Eventualmente, esta limitação pode causar problema com o estouro do limite de *IDs*.

4.3.8 Sistema de Detecção de Falhas

Para tentar detectar a causa de uma falha na comunicação, o Sistema de Detecção de Falhas trabalha em conjunto com o pacote de CPR e com o pacote do protocolo UDP/IP. Basicamente, ele filtra entre os códigos de erro retornados por estes protocolos, aqueles que podem mapear para Isolamento ou Particionamento. Quando o Sistema de Detecção de Falhas é invocado, ele executa os seguintes passos para a detecção de algum tipo de falha:

1. Verifica o ambiente local e a possibilidade de Isolamento. Se não existir, vai para o passo 2; caso contrário, retorna o diagnóstico para o seu invocador (**Communic** ou **GroupServer**), que avalia a falha:

   - **Communic**: o Sistema de Detecção de Falhas pode ter sido ativado por ocasião de falha de uma chamada de CPR para o Servidor de Grupos ou o Servidor de Nomes, por `SendGS` ou por `Send`:
     - CPRs: as CPRs que retornam valor voltam com código de erro e as outras (`RemoveResource` e `LeaveGroup`) não causam conseqüências na execução local.

– `SendGS/Send`: retornam um código de erro para **PortOut-Multicast** (`SendDatagram` e `GetMessages`) ou **PortOut** (`SendDatagram` e `Send`), respectivamente.

Nos casos relacionados acima, após a detecção do Isolamento, o objeto **Communic** desabilita o uso da rede por quaisquer dos mecanismos existentes pois, como a falha é do tipo falha-e-parada, não haverá mais a possibilidade de interação remota.

- **GroupServer**: os métodos `Receive` (envia **ACK**s), `SendDATA` (envia *multicasts*), `SendRETRANS` (envia mensagens ponto-a-ponto), `SendSYNC` e `SendCOMMIT` (enviam *multicasts* e esperam respostas com *timeout*) são os responsáveis pela ativação do Sistema de Detecção de Falhas deste servidor. Todos os casos podem sofrer uma falha no envio da mensagem, enquanto que apenas o último caso sofre tanto no envio quanto na ocorrência de *timeout*. Em todos os casos, como a falha é Isolamento, o Servidor de Grupos passa a receber ou enviar mensagens somente de/para membros de grupos presentes na mesma máquina[7].

2. Verifica o ambiente distribuído entre o membro local e o remoto e a possibilidade de Particionamento. Estas falhas só podem ser detectadas quando da ocorrência de *timeout* de espera de resposta, pois o mecanismo de espera de **ACK**s (**Communic**) ou **ACK-SYNC**s/**ACK-COMMIT**s (**GroupServer**) pode refletir tal falha. Se não ocorrer nenhum dos dois tipos de falhas detectáveis, retorna-se ao invocador que deve repetir a operação anterior; caso contrário, retorna o diagnóstico para avaliação:

   - **Communic**:
     – CPRs: toma a mesma atitude do Isolamento e desabilita uso das CPRs para o referido servidor. Se o servidor for o Servidor de Grupos, também desabilita o uso de `SendGS`.
     – `SendGS`: desabilita o uso deste método e também das CPRs para o Servidor de Grupos, além de retornar um código de erro

[7] O protocolo UDP/IP (e seu *software* que implementa *sockets*) não usa a rede quando a mensagem tem destino local.

para **PortOutMulticast**, indicando que esta porta deve sair do grupo.

- `Send`: retorna um código de erro para **PortOut**, indicando que esta porta deve desligar-se do membro remoto (um objeto **PortIn**, no caso) falho.

- **GroupServer**: somente `SendSYNC` e `SendCOMMIT`, os quais esperam **ACK**s com *timeout*, que podem detectar Particionamento. Para cada membro diagnosticado como em outra partição (pois não retornou **ACK-SYNC** ou **ACK-COMMIT**), deve ser efetuada a operação de `LeaveGroup`, ou seja, só restarão no grupo os membros da mesma partição do Servidor de Grupos.

A detecção da falha pelo Sistema de Detecção de Falhas e a avaliação pelo Sistema de Comunicação (ou Servidor de Grupos) reflete, através dos objetos de interface com cliente, na execução dos clientes deste sistema, pois:

- **PortIn**:

  - Construtor recebe um identificador de porta de entrada nulo, após falha na CPR `NameServer::InsertResource`.
  - Destrutor não consegue executar a CPR `NameServer::RemoveResource`.

- **PortOut**:

  - Construtor (1)/`Bind` (1) recebem como retorno do **NameServer** um endereço nulo de porta de entrada a ligar-se, após falha na CPR `NameServer::LocateResource`.
  - `SendDatagram`/`Send` não enviam suas mensagens para uma porta de entrada remota se `Communic::Send` falha, e devem desligar-se desta porta.

- **PortOutMulticast**:

  - Construtor (1)/`Enter` recebem um identificador de grupo nulo, após falha na CPR `GroupServer::EnterGroup`.

- Destrutor/`Leave` não conseguem executar a CPR `GroupServer::LeaveGroup`.
- `NumPorts` obtém como retorno um código de erro caso a CPR `GroupServer::NumPortsGroup` não tenha sucesso.
- `SendDatagram/GetMessages` não enviam suas mensagens para o Servidor de Grupos se `Communic::SendGS` falha, e devem retirar sua porta de saída de *multicast* do grupo.

- **NameClient:**
  - Construtor (2): recebe um endereço de porta de entrada nulo, após falha na CPR `NameServer::LocateResource`.
  - `Insert`: recebe um identificador de recurso nulo, após falha na CPR `NameServer::InsertResource`.
  - `Remove`: não consegue executar a CPR `NameServer::RemoveResource`.
  - `Locate`: recebe um endereço de recurso nulo, após falha na CPR `NameServer::LocateResource`.

Com isso, os clientes do Sistema de Comunicação devem tomar algumas precauções relativas à ocorrência destes eventos. Por exemplo, um cliente de um recurso deve cancelar operações em certos recursos remotos, caso sua porta de saída seja repentinamente desligada de uma porta de entrada do servidor de recursos remoto.

Classe Failure

Um objeto **Failure** é instanciado automaticamente, quando da criação do Sistema de Comunicação local (ou Servidor de Grupos) (Figura 4.11). É um objeto passivo, ou seja, fica ocioso até que seus métodos sejam requisitados por um invocador.

Possui como atributos:

- `fault`: código de falha `NOFAULT` ou `ISOLATION`. Ele é usado para desabilitar o uso da rede por funções de CPR ou *sockets*;

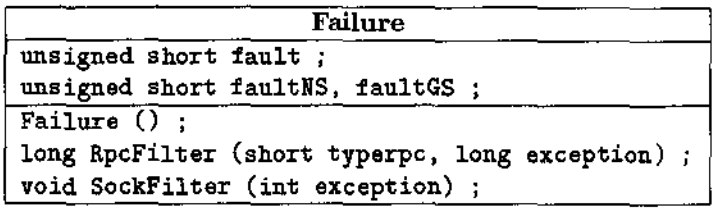


Figura 4.11: Diagrama da Classe **Failure**

- códigos de falha na interação com Servidor de Nomes (`faultNS`) ou com Servidor de Grupos (`faultGS`). Eles são usados para desabilitar a interação com os respectivos servidores. Estes atributos são exclusivos do Sistema de Detecção de Falhas correlato ao Sistema de Comunicação.

Os métodos fornecidos pelo Sistema de Detecção de Falhas são:

- Construtor de Objeto: inicializa os atributos com `NOFAULT`.
- `RpcFilter`: a partir de `typerpc` (`NSFAULT` ou `GSFAULT`) e de um código de erro (`exception`) fornecido pelo pacote de CPR, filtra este código para um mapeamento de `ISOLATION` ou `PARTITION`.
- `SockFilter`: a partir de um código de erro (`exception`) fornecido pelo pacote de *sockets*, filtra este código para um mapeamento de `ISOLATION` ou `PARTITION`.

4.4 Integração com o Gerente de Distribuição

O Sistema de Processamento de Transações Atômicas atualmente utiliza apenas a comunicação ponto-a-ponto, provida pelo Sistema de Comunicação. Ele implementa o acesso aos recursos utilizados em uma transação através de CPRs. Nesta comunicação, estão presentes três componentes: um cliente do recurso, que faz a chamada CPR, um servidor do recurso, que implementa as operações requeridas pelas CPRs, e um gerenciador de CPRs, que

recebe pedidos de inicialização de CPRs dos clientes e inicia a execução dos servidores dos recursos requisitados.

O mecanismo de CPR do Sistema de Processamento de Transações Atômicas é baseado nos recursos fornecidos pelos objetos de interface com cliente do Sistema de Comunicação. Durante o projeto foram implementadas as classes **ClientRPC** (lado cliente) e **ServerRPC** (lado servidor). O cliente do recurso utiliza **ClientRPC** para iniciar o processamento com o gerenciador de CPR e, em seguida, para interagir com o servidor do recurso, que utiliza **ServerRPC**.

As classes **ClientRPC** e **ServerRPC** estão no nível de **Cliente** de *Port System*, de acordo com a Figura 4.2.

4.4.1 Classe ClientRPC

Implementa o lado cliente do sistema de comunicação por CPR (Figura 4.12).

ClientRPC

`unsigned short resID ;`
`PortIn *ClientPI ;`
`PortOut *ClientPO ;`
`NameClient ClientNS ;`

`ClientRPC () ;`
`ClientRPC (char *resname, unsigned long timeout) ;`
`~ClientRPC () ;`
`unsigned short Initiate (char *resname, unsigned long timeout) ;`
`void Terminate () ;`
`unsigned short SendRequest (char *buffer, unsigned long timeout) ;`
`unsigned short isConnectedtoServerRPC () ;`

Figura 4.12: Diagrama da Classe **ClientRPC**

Seus atributos são:

- `resID`: *ID* do recurso utilizado pelo cliente;
- `ClientPI`: **PortIn** utilizado para receber as respostas provenientes do **ServerRPC**;

- `ClientPO`: **PortOut** utilizado para enviar as requisições para **Server-RPC**.

- `ClientNS`: usado pelo cliente para localizar o recurso no **NameServer**.

Os métodos fornecidos por **ClientRPC** são:

- Construtor de Objeto (0): inicializa os atributos com nulo. É usado quando não se conhece, em tempo de compilação, o recurso que será referenciado.

- Construtor de Objeto (1): recebe como parâmetro o nome do recurso a ser referenciado (`resname`) e um valor de *timeout* de espera de conexão ao servidor. Então, chama `Initiate`.

- Destrutor de Objeto: chama `Terminate` para desconectar-se do servidor.

- `Initiate`: através de `ClientNS`, tenta localizar o recurso `resname` no **NameServer**. Se encontrar, instancia seu **PortIn** e seu **PortOut** e, em seguida, envia uma mensagem ao gerenciador de CPR, requerendo uma conexão ao referido recurso. Então, bloqueia por um tempo (`timeout`), esperando pela resposta do servidor, que será chamado pelo gerenciador de CPR. Caso a resposta chegue dentro deste intervalo de tempo, o cliente conecta-se ao servidor, estando pronto para fazer requisições sobre o recurso. Retorna `TRUE` se conseguiu a conexão, e `FALSE`, caso contrário.

- `Terminate`: caso o cliente esteja conectado, é enviada uma mensagem de término ao servidor. Em seguida, são removidos os objetos **PortIn** e **PortOut**.

- `SendRequest`: método usado para enviar uma requisição (`buffer`) ao servidor do recurso, e esperar `timeout` milisegundos pela resposta. Se receber a resposta dentro do intervalo de tempo, retorna a mesma em `buffer` e `TRUE` é a saída do método; caso contrário, retorna um `buffer` nulo e `FALSE`.

- `isConnectedtoServerRPC`: retorna `TRUE` se o cliente está conectado ao servidor, e `FALSE`, caso contrário.

4.4.2 Classe ServerRPC

Implementa o lado servidor do sistema de comunicação por CPR (Figura 4.13).

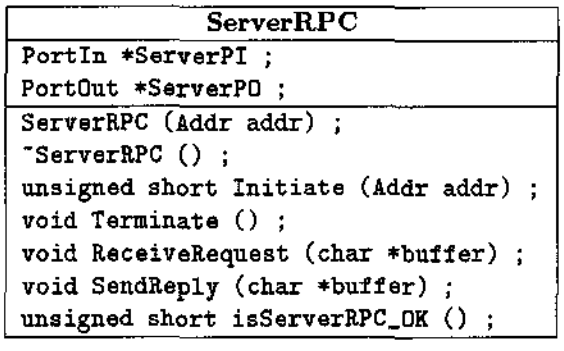


Figura 4.13: Diagrama da Classe **ServerRPC**

Seus atributos são:

- `ServerPI`: **PortIn** utilizado para receber as requisições provenientes do **ClientRPC**;
- `ServerPO`: **PortOut** utilizado para enviar as respostas para **ClientRPC**.

Os métodos fornecidos por **ServerRPC** são:

- Construtor de Objeto: recebe como parâmetro o endereço do objeto **PortIn** usado por **ClientRPC**, quando da instanciação a partir do gerenciador de CPR. Então, chama `Initiate`.
- Destrutor de Objeto: chama `Terminate` para encerrar conexão com cliente.
- `Initiate`: instancia seu **PortIn** e, através de `addr`, faz o acoplamento de seu **PortOut** ao **PortIn** do cliente. Em seguida, envia mensagem para **ClientRPC** indicando que está pronto para receber requisições. Retorna `TRUE`, se a conexão foi estabelecida, e **FALSE**, caso contrário.

- `Terminate`: caso o servidor esteja conectado, desconecta-se do cliente. Em seguida, são removidos os objetos **PortIn** e **PortOut**.
- `ReceiveRequest`: método usado para receber requisições (`buffer`) do cliente do recurso. Fica bloqueado enquanto não chegar uma requisição.
- `SendReply`: método usado para enviar uma resposta (`buffer`) à requisição previamente realizada.
- `isServerRPC_OK`: retorna `TRUE`, se o servidor está conectado ao cliente, e `FALSE`, caso contrário.

4.5 Resumo

O sistema *Port System* implementa mecanismos de comunicação ponto-a-ponto e em grupo e um serviço de nomes, necessários ao ambiente Xchart.

Os mecanismos de comunicação são implementados pelos módulos do Sistema de Comunicação e do Servidor de Grupos. Para implementar uma aplicação distribuída com comunicação ponto-a-ponto apenas, basta utilizar as classes **PortIn** e **PortOut** da interface com cliente, e o Servidor de Grupos não é usado. Uma aplicação com comunicação em grupo necessita as classes **PortIn** e **PortOutMulticast**, e o Servidor de Grupos deve estar em execução.

O serviço de nomes é implementado pelo Servidor de Nomes e utilizado através da classe **NameClient**, de interface com cliente.

Dentro do ambiente Xchart, *Port System* encontra-se abaixo do Sistema de Processamento de Transações Atômicas, que utiliza CPR no controle das transações sobre recursos compartilhados. Sendo assim, foram criadas as classes **ClientRPC** e **ServerRPC** acima de *Port System*, com o objetivo de fornecer uma interface de suporte ao mecanismo de CPR implementado pelo Sistema de Processamento de Transações Atômicas. As classes **ClientRPC** e **ServerRPC** são totalmente implementadas usando as funcionalidades de **PortIn** e **PortOut**, e correspondem a exemplos de **Cliente** de *Port System*.

Capítulo 5

Conclusões

Durante a dissertação, foi apresentado o sistema *Port System*, que implementa um sistema de comunicação em grupo e um serviço de nomes. Este sistema será utilizado pelo Núcleo Reativo e pelo Sistema de Processamento de Transações Atômicas para fornecer recursos básicos de implementação de interfaces concorrentes, onde as sub-interfaces podem estar distribuídas.

Atualmente, o conceito de interfaces concorrentes e o desenvolvimento de um ambiente de suporte à especificação e implementação das mesmas correspondem a uma área de pesquisa ainda pouco explorada. Com isso, este trabalho, em conjunto com outros do Projeto Xchart, pode ser de grande valia, demonstrando que é possível preencher a lacuna existente na área de ambientes de programação de interfaces homem-computador.

A seguir, são apresentados comentários relativos ao trabalho e sugeridas extensões futuras.

5.1 Análise de *Port System*

Durante a especificação dos requisitos do sistema, verificou-se que o mecanismo de comunicação deveria ser mais confiável do que simplesmente enviar e receber mensagens. Deveria existir algum mecanismo que garantisse o envio e o recebimento da mensagem ou que detectasse algum tipo de falha durante a transmissão. Com a confiabilidade de transmissão de mensagens, este sistema forneceria uma abstração de mais alto nível a seus clientes. Ao usar este sistema, o cliente transmitiria a mensagem e, caso tivesse ocorrido alguma

falha na comunicação, ele seria avisado para tomar alguma atitude. Desta análise surgiu a necessidade de um Sistema de Comunicação e de um Sistema de Detecção de Falhas, que controlam o envio e a recepção de mensagens em cada nó da aplicação distribuída.

Para a comunicação *multicast*, seria necessário um algoritmo de comunicação em grupo confiável, atômico e ordenado. A atomicidade seria importante para fornecer uma transparência no envio do *multicast* para um grupo de processos. A ordenação era um pré-requisito para a implementação de comunicação entre um grupo de Núcleos Reativos. Ao ocorrer um evento e_1 em um Núcleo Reativo N_1 e um evento e_2 em um Núcleo Reativo N_2, tal que os dois eventos deveriam ser notificados ao grupo de Núcleos Reativos, todos os membros do grupo deveriam receber os eventos na mesma ordem ((e_1,e_2) ou (e_2,e_1)), para que as reações seguissem a mesma semântica. Se dois Núcleos Reativos recebessem os eventos em diferentes ordens, as reações provavelmente seriam diferentes, causando uma inconsistência no Xchart em execução. Com isso, foram estudados algoritmos de comunicação em grupo confiável, sendo escolhido o algoritmo centralizado de Kaashoek, por simplicidade e eficiência. Com isso, o Sistema de Comunicação passou a ter mais uma funcionalidade, o chamado núcleo do algoritmo de Kaashoek. Para completar o algoritmo, criou-se o Servidor de Grupos, o coordenador do algoritmo. O Servidor de Grupos centraliza as operações sobre grupos, e não é tolerante a falhas. Além disso, sua topologia centralizada implica em baixa escalabilidade, com o incremento do número de requisições do serviço.

A utilização de portas de entrada pelo Sistema de Comunicação e o compartilhamento de recursos pelo Sistema de Processamento de Transações Atômicas criou a necessidade de um serviço de nomes, que fornecesse a transparência de localização no ambiente distribuído. A partir disso, foi definido o Servidor de Nomes, implementado de forma centralizada e que possui as mesmas vantagens (simplicidade e eficiência) e desvantagens (não tolerante a falhas e baixa escalabilidade) do Servidor de Grupos.

A partir deste pequeno histórico do desenvolvimento de *Port System*, pode-se dizer que os requisitos do ambiente Xchart foram atendidos, e que a implementação das funcionalidades priorizou a simplicidade e eficiência, em detrimento de considerações como a tolerância a falhas. Como um primeiro protótipo, *Port System* pode ser utilizado para a análise de execução de interfaces concorrentes.

5.2 Extensões Futuras

A principal extensão a este trabalho consiste na análise, projeto e implementação completa do Sistema de Execução do ambiente Xchart. Os outros subsistemas encontram-se em fases de projeto ou implementação, de forma que ainda não existe um protótipo integrado e operacional do Sistema de Execução.

Dentro do contexto do Gerente de Distribuição, também não existe ainda uma integração completa entre o Sistema de Processamento de Transações Atômicas e o Sistema de Comunicação. Atualmente, o Sistema de Comunicação, através de **ClientRPC** e **ServerRPC**, permite somente a comunicação ponto-a-ponto entre os Sistemas de Processamento de Transações Atômicas. Encontra-se em estudo a utilização de comunicação em grupo entre os processos que controlam transações, para tornar mais eficiente o gerenciamento de transações distribuídas. Normalmente, utiliza-se CPR como o mecanismo de comunicação ponto-a-ponto para os Sistemas de Processamento de Transações Atômicas. Um interessante trabalho seria a implementação de CPRG (chamada de procedimento remoto de grupo), com a mesma facilidade de utilização de CPR e toda a funcionalidade fornecida pela comunicação em grupo.

Outra área que poderia ser aprofundada seria tolerância a falhas. O Sistema de Detecção de Falhas, em conjunto com o Sistema de Comunicação, poderia ser sofisticado para fornecer a transparência de falhas do ambiente de comunicação. Além disso, seria interessante implementar, por exemplo, a replicação dos servidores (Servidor de Grupos e Servidor de Nomes), o que aumentaria, inclusive, a escalabilidade dos serviços, que seriam fornecidos por mais de um servidor.

Pensando na hipótese de estender a abrangência de *Port System* para um ambiente distribuído em uma rede de longa distância, além da necessidade de aumentar a escalabilidade de todas as funções fornecidas, seria interessante implementar um algoritmo de comunicação em grupo que utilizasse o recurso de *multicast* em rede de longa distância. Neste caso, poderia ser utilizado o pacote de *IP-multicast* (*MBone* [10, 12]), o qual já encontra-se disponível nas mais novas implementações do protocolo TCP/IP.

5.3 Considerações Finais

O desenvolvimento do sistema *Port System* foi importante tanto particularmente quanto dentro do contexto do Projeto Xchart. Para o autor, este trabalho permitiu o aprofundamento dos conhecimentos de sistemas distribuídos e redes de computadores e o estudo das funcionalidades e necessidades do componente básico que dá suporte à distribuição: comunicação.

O Projeto Xchart agora possui um subsistema a partir do qual outras camadas de nível mais alto podem ser projetadas e implementadas. A partir deste subsistema, encontra-se em implementação o mecanismo de transações e, em um futuro próximo, o Projeto Xchart contará com um primeiro protótipo do ambiente Xchart, permitindo assim um estudo mais detalhado sobre o comportamento e a funcionalidade de interfaces concorrentes.

Apêndice A

Exemplos de aplicações

A seguir, serão mostrados alguns exemplos da utilização de *Port System*. Os primeiros quatro exemplos demonstram a comunicação ponto-a-ponto. Em seguida, são dados dois exemplos de comunicação em grupo. O sétimo exemplo demonstra o uso de **NameClient** e o último o uso de **ClientRPC** e **ServerRPC**. Todos os exemplos estão em linguagem C++.

A.1 Produtor-Consumidor

O exemplo mais simples é o de um pseudo produtor-consumidor. O processo produtor envia datagramas para o processo consumidor indefinidamente.

```
// produtor
void prod () {
     PortOut portout ("cons", "port1") ;
                     // supoe-se que a porta de entrada
                     // "port1" de "cons" ja' foi criada
     char buffer[256] ;

     while (1) {
          // produzir a cadeia de caracteres
          strcpy (buffer, getprod ()) ;

          // enviar a mensagem
          portout.SendDatagram (buffer, NULL) ;
```

```
                    // segundo parametro indica
                    // que nao existe porta de
                    // entrada para resposta
     }
}

// consumidor
void cons () {
     PortIn portin ("cons", "port1") ;
     char buffer[256] ;
     unsigned long timeout = 0L ; // em milisegundos

     while (1) {
          if (portin.isthereMsg ()) {
               // receber a mensagem
               portin.Receive (buffer, timeout, NULL) ;
                       // terceiro parametro indica que
                       // nao ha' interesse no endereco
                       // de porta de entrada do emissor

               // consumir a cadeia de caracteres
               setcons (buffer) ;
          }
          else {
               DoOtherThings () ;
          }
     }
}
```

A.2 Cliente/Servidor

Um modelo básico cliente/servidor onde as requisições e as respostas são codificadas por cadeias de caracteres.

```
// cliente
void cliente () {
```

```
    PortOut portout ("servidor", "port2") ;
                    // supoe-se que a porta de
                    // entrada ja' foi criada
    PortIn portin ("cliente", "portresp2") ;
    char buffer[256] ;
    unsigned long timeout = 1000 ;

    buffer[0] = NULL ;

    while (1) {
        // obter a requisicao
        strcpy (buffer, getreq ()) ;

        // enviar a requisicao e esperar a resposta
        portout.Send (buffer, &portin, timeout) ;

        /* ou o equivalente:
        portout.SendDatagram (buffer, &portin) ;
        portin,Receive (buffer, timeout, NULL) ;
        */

        if (buffer[0] != NULL) {
            // se servidor retornou resposta, trata-la
            setreply (buffer) ;
            buffer[0] = NULL ;
        }
    }
}

// servidor
void servidor () {
    PortOut portout (NULL) ; // ainda nao conectada
    PortIn portin ("servidor", "port2") ;
    char buffer[256] ;
    unsigned long timeout = 0L ;
    Addr addr ; // endereco de uma porta de entrada
```

```
    while (1) {
        // verificar se existe uma requisicao na fila
        if (portin.isthereMsg ()) {
            portin.Receive (buffer, timeout, &addr) ;

            // processar a requisicao
            setreq (buffer) ;

            // retornar a resposta
            strcpy (buffer, getreply ()) ;
            portout.Bind (&addr) ;
            portout.SendDatagram (buffer, NULL) ;
            portout.Unbind () ;
        }
        else {
            DoOtherThings () ;
        }
    }
}
```

A.3 Cliente/Servidor com Guardas de Portas

Um modelo com dois clientes e um servidor, mostrando o funcionamento das guardas de portas de comunicação, através da confecção de uma função de seleção de porta.

```
// cliente 1
void cliente1 () {
    PortOut portout ("servidor", "port3.1") ;
                    // supoe-se que a porta de
                    // entrada ja' foi criada
    PortIn portin ("cliente1", "portresp3.1") ;
    char buffer[256] ;
    unsigned long timeout = 1000 ;
```

```
     buffer[0] = NULL ;

     while (1) {
          // obter a requisicao
          strcpy (buffer, getreq ()) ;

          // enviar a requisicao e esperar a resposta
          portout.Send (buffer, &portin, timeout) ;

          if (buffer[0] != NULL) {
               // se servidor retornou resposta, trata-la
               setreply (buffer) ;
               buffer[0] = NULL ;
          }
     }
}

// cliente 2
void cliente2 () {
     PortOut portout ("servidor", "port3.2") ;
                    // assume-se que a porta de
                    // entrada ja' foi criada
     PortIn portin ("cliente2", "portresp3.2") ;
     char buffer[256] ;
     unsigned long timeout = 1000 ;

     buffer[0] = NULL ;

     while (1) {
          // obter a requisicao
          strcpy (buffer, getreq ()) ;

          // enviar a requisicao e esperar a resposta
          portout.Send (buffer, &portin, timeout) ;

          if (buffer[0] != NULL) {
               // se servidor retornou resposta, trata-la
```

```
                setreply (buffer) ;
                buffer[0] = NULL ;
            }
        }
}

// servidor
void servidor () {
#define LIMIT 100

        PortOut portout (NULL) ; // ainda nao conectada
        PortIn portin1 ("servidor", "port3.1"),
               portin2 ("servidor", "port3.2") ;
        PortIn *ptportin[2] = {&portin1, &portin2} ;
        char buffer[256] ;
        unsigned long timeout = 100 ;
        Addr addr ; // endereco de uma porta de entrada

        while (1) {
            // verificar se existe uma requisicao na fila
            // de alguma das portas de entrada do servidor
            switch (SelPortIn (ptportin, 2, TRUE, timeout)) {
                case 0:
                    portin1.Receive (buffer, timeout, &addr) ;

                    // processar a requisicao
                    setreq (buffer) ;

                    // retornar a resposta
                    strcpy (buffer, getreply ()) ;
                    portout.Bind (&addr) ;
                    portout.SendDatagram (buffer, NULL) ;
                    portout.Unbind () ;
                    break ;

                case 1:
                    portin2.Receive (buffer, timeout, &addr) ;
```

```
                    // processar a requisicao
                    setreq (buffer) ;

                    // retornar a resposta
                    strcpy (buffer, getreply ()) ;
                    portout.Bind (&addr) ;
                    portout.SendDatagram (buffer, NULL) ;
                    portout.Unbind () ;

               default:
                    // case -1:
                    // SelPortIn retorna -1 se nao existe
                    //        mensagem para nenhuma porta
                    ct++ ;
                    DoOtherThings () ;
          }

          if (ct == LIMIT)
               break ;
     }
}

// Funcao de selecao de uma das portas de entrada do servidor.
// A selecao baseia-se apenas na verificacao de qual porta de
// entrada possui mensagem enfileirada.
//
// Importante: qualquer tipo de funcao de selecao pode ser
//       criada, de acordo com as necessidades, ordem de
//       prioridade das verificacoes, quantidade de mensagens
//       em cada porta, etc; depende do tipo de implementacao.
unsigned short SelPortIn
               (PortIn **ptportin, unsigned short nportin,
                unsigned short block, unsigned long timeout)
{
     unsigned short ct ;
```

```
    do {
        ct = 0 ;
        while (ct < nportin) {
            // verificar se existe mensagem para alguma
            // das portas de entrada do vetor
            if (ptportin[ct]->isthereMsg ())
                return (ct) ;
            ct++ ;
        }
        // nao existe mensagem no momento:
        // bloquear ou nao para esperar mensagem
        if (block == TRUE) {
            wait (timeout) ;
            block = FALSE ; // bloquear somente uma vez
        }
        else
            return (-1) ;
    } while (TRUE) ;
}
```

A.4 Produtor-Consumidor Assíncrono

O exemplo a seguir mostra um pseudo produtor-consumidor usando `PortIn::ReceivePolling`, sendo diferente do Exemplo A.1. Este exemplo utiliza funções de manipulação de eventos do sistema operacional.

```
// produtor
void prod () {
    PortOut portout ("cons", "porti") ;
    char buffer[256] ;

    // verifica se a porta de saida esta ligada
    if (!portout.isBound ()) {
        wait (5000) ;
        if (!portout.Bind ("cons", "port1")) {
            return ;
```

```
        }
    }

    while (1) {
        getprod (buffer) ;
        portout.SendDatagram (buffer, NULL) ;
    }
}

// consumidor
unsigned short Received_EV ;

void setcons (char *text, Addr *nulladdr) {
    printf ("\n Texto: %s", text) ;
    // quando a mensagem chega, e' atualizado o evento
    setevent (Received_EV) ;
}

void DoOtherThings () {
    unsigned long timeout = 10000L ;
    unsigned long WaitResult ;
    // esperando por evento
    while (TRUE) {
        WaitResult = waitevent (Received_EV, timeout) ;
        switch (WaitResult) {
            case WAIT_TIMEOUT: // ocorreu timeout
                break ;
            case WAIT_EVENT: // evento sinalizado
                resetevent (Received_EV) ;
                return ;
        }
    }
}

void cons () {
    PortIn portin ("cons", "port1") ;
    char buffer[256] ;
```

```
    // verificar se a porta de entrada foi criada
    if (!portin.GetportID ())
        return ;

    if ((Received_EV = createevent ()) == NULL)
        return ;

    while (1) {
        portin.ReceivePolling (buffer, NULL, setcons) ;
        DoOtherThings () ;
    }
}
```

A.5 Produtor-Consumidores

O exemplo mais simples de um produtor e n consumidores ($n = 2$). O processo produtor envia *multicast*-datagramas para os consumidores indefinidamente.

```
// produtor
void prod () {
    PortOutMulticast portout ("prodcons", NULL) ;
                              // nao tem porta de entrada
    char buffer[256] ;

    // verificar se o grupo tem duas portas de entrada
    if (portout.NumPorts () < 2) {
        wait (100) ; // esperar pela criacao das portas
        if (portout.NumPorts () < 2)
            return ;
    }

    while (1) {
        // produzir a cadeia de caracteres
        strcpy (buffer, getprod ()) ;
```

```
          // enviar para o grupo
          portout.SendDatagram (buffer) ;
     }
}

// consumidor 1
void cons1 () {
     PortIn portin ("cons1", "port1") ;
     PortOutMulticast portout ("prodcons", &portin) ;
     char buffer[256] ;
     unsigned long timeout = 0L ; // em milisegundos

     while (1) {
          if (portin.isthereMsg ()) {
               portin.Receive (buffer, timeout, NULL) ;
                    // terceiro parametro indica que
                    // nao ha' interesse no endereco
                    // de porta de entrada do emissor

               // consumir a cadeia de caracteres
               setcons (buffer) ;
          }
          else {
               DoOtherThings () ;
          }
     }
}

// consumidor 2
void cons2 () {
     PortIn portin ("cons2", "port2") ;
     PortOutMulticast portout ("prodcons", &portin) ;
     char buffer[256] ;
     unsigned long timeout = 0L ; // em milisegundos

     while (1) {
```

```
        if (portin.isthereMsg ()) {
            portin.Receive (buffer, timeout, NULL) ;
                    // terceiro parametro indica que
                    // nao ha' interesse no endereco
                    // de porta de entrada do emissor

            // consumir a cadeia de caracteres
            setcons (buffer) ;
        }
        else {
            DoOtherThings () ;
        }
    }
}
```

A.6 Trabalho Cooperativo

Um modelo de trabalho cooperativo simples, com dois processos formando um grupo de trabalho onde cada um complementa a tarefa do outro.

```
// processo 1
void processo1 () {
    PortIn portin1 ("proc1", "port1") ;
    PortOutMulticast portout ("proc_1<->2", &portin1) ;
    char *buffer[256] ;
    unsigned long timeout = 1000 ;

    while (1) {
        // gerar nova tarefa para o membro remoto
        strcpy (buffer, getnexttask ()) ;

        // enviar a tarefa
        portout.SendDatagram (buffer) ;

        // esperar a tarefa do membro remoto
        portin1.Receive (buffer, timeout, NULL) ;
```

```
          if (tasklocal (buffer))
               // descarta a tarefa, se e' a propria
               // mensagem local enviada ao grupo, e
               // espera por tarefa remota
               portin1.Receive (buffer, timeout, NULL) ;
          else
               // processa tarefa do membro remoto
               processtask (buffer) ;
     }
}

// processo 2
void processo2 () {
     PortIn portin2 ("proc2", "port2") ;
     PortOutMulticast portout ("proc_1<->2", &portin2) ;
     char *buffer[256] ;
     unsigned long timeout = 1000 ;

     while (1) {
          // gerar nova tarefa para o membro remoto
          strcpy (buffer, getnexttask ()) ;

          // enviar a tarefa
          portout.SendDatagram (buffer) ;

          // esperar a tarefa do membro remoto
          portin2.Receive (buffer, timeout, NULL) ;
          if (tasklocal (buffer))
               // descarta a tarefa, se e' a propria
               // mensagem local enviada ao grupo, e
               // espera por tarefa remota
               portin2.Receive (buffer, timeout, NULL) ;
          else
               // processa tarefa do membro remoto
               processtask (buffer) ;
     }
}
```

A.7 Cliente/Servidor de Recurso

Um exemplo simples de como utilizar **NameClient** para ter acesso à interface do Servidor de Nomes. É implementado um cliente e um servidor relativos ao recurso `Resource`. Então, o cliente realiza requisições para o servidor, através de comunicação ponto-a-ponto entre **PortOut** do cliente e **PortIn** do servidor.

```
// estrutura que representa um recurso
struct Resource {
     unsigned short resID ; // identificador de recurso
     unsigned char sval[256] ; // dado do recurso
} ;

// cliente
void resclient (char *resname) {
     Addr resaddr ; // endereco de acesso ao recurso
     PortOut *portout ;
     char buffer[256] ;
     NameClient *nameclient ;

     if (!(nameclient = new NameClient ()))
          return ;
     if (!(nameclient->Locate (resname, &resaddr)))
          // recurso nao encontrado no Servidor de Nomes
          return ;
     if (!(portout = new PortOut (&resaddr)))
          // cliente nao conseguiu conectar-se ao servidor
          return ;

     while (1) {
          getresourcevalue (buffer) ;
          portout->SendDatagram (buffer, NULL) ;
     }

     delete nameclient ;
     delete portout ;
```

```
}

// servidor
void resserver (char *resname) {
     Resource res ; // recurso gerenciado
     PortIn portin ("server", resname) ;
     char buffer[256] ;
     unsigned long timeout = INFINITE ; // em milisegundos
     NameClient *nameclient ;

     if (!portin.GetportID ())
          return ;
     if (!(nameclient = new NameClient (&portin)))
          return ;
     if (!(res.resID = nameclient->Insert (resname)))
          // recurso nao inserido no Servidor de Nomes
          return ;

     while (1) {
          portin.Receive (buffer, timeout, NULL) ;
          setresourcevalue (buffer, res.sval) ;
     }

     nameclient->Remove(res.resID) ;
     delete nameclient ;
}
```

A.8 Cliente/Servidor de CPR

Semelhante ao exemplo A.7. Porém, ao invés de utilizar diretamente **PortIn** e **PortOut**, usa-se a interface de mais alto nível de **ClientRPC** e **ServerRPC**. Pode ser encarado como uma simulação da interação entre cliente do recurso, gerenciador de CPR e servidor de recurso, implementada pelo Sistema de Processamento de Transações Atômicas.

```
struct Resource {
```

```
     unsigned short resID ;
     unsigned char sval[256] ;
} ;

// cliente
void resclient () {
     ClientRPC *clientrpc ;
     char buffer[256] ;

     clientrpc = new ClientRPC ("resource", TIMEOUTINIT) ;
     if (!(clientrpc->isConnectedtoServerRPC ()))
          return ;

     while (1) {
          getresourcevalue (buffer) ;
          if (clientrpc->SendRequest (buffer, TIMEOUTREPLY))
               printf ("Resposta do servidor: %s", buffer) ;
     }

     delete clientrpc ;
}

// servidor
void resserver (char *resname) {
     Resource res ; // recurso gerenciado
     PortIn portin ("manager", resname) ;
     ServerRPC *serverrpc ;
     char buffer[256] ;
     NameClient *nameclient ;
     Addr addrclient ;

     // trecho referente ao gerenciador de CPR:
     // cria uma porta de entrada para receber
     // uma requisicao de acesso ao servidor
     // de um recurso
     if (!portin.GetportID ())
          return ;
```

```
    if (!(nameclient = new NameClient (&portin)))
        return ;
    if (!(res.resID = nameclient->Insert (resname)))
        return ;

    // esperando pela primeira requisicao para o recurso
    portin.Receive (buffer, INFINITE, &addrclient) ;

    // gerenciador de CPR invoca servidor do recurso
    serverrpc = new ServerRPC (addrclient) ;
    if (!(serverrpc->isServerRPC_OK ()))
        return ;

    while (1) {
        serverrpc->ReceiveRequest (buffer) ;
        setresourcevalue (buffer, res.sval) ;
        printf ("Novo valor para o recurso: %s", res.sval) ;

        // retornar o numero de bytes de res.sval
        // como resposta para o cliente
        itoa (strlen (res.sval), buffer, 10) ;
        serverrpc->SendReply (buffer) ;
    }

    nameclient->Remove(res.resID) ;
    delete nameclient ;
    delete serverrpc ;
}
```

Bibliografia

[1] G. Berry and G. Gonthier. The ESTEREL synchronous programming languague: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, November 1992.

[2] K. P. Birman. THE PROCESS GROUP APPROACH TO RELIABLE DISTRIBUTED COMPUTING. *Communications of the ACM*, 36(12):36–53, December 1993.

[3] K. P. Birman and T. Joseph. Reliable Communication in the Presence of Failures. *ACM Transactions on Computer Systems*, 5(1):47–76, February 1987.

[4] L. Brito. Suporte de Distribuição para Sistemas Desenvolvidos em Xchart. Proposta de Tese de Mestrado, DCC-IMECC-UNICAMP, October 1994.

[5] R. G. Chandras. Distributed Message Passing Operating Systems. *Operating Systems Review*, 24(1):7–17, January 1990.

[6] J. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[7] R. Davis. *Windows Networking And Connectivity Guide: how to survive in a world of Windows, Netware and DOS.* Addison-Wesley Publishing Company, 1993.

[8] F. N. de Lucena and H. Liesenberg. Reflections on Using Statecharts to Capture Human-Computer Interface Behaviour. In *Proceedings of XIV Int. Conf. of the Chilean Computer Science Society*, November 1994.

[9] F. N. de Lucena et al. Xchart-based Complex Dialog Development. In *Simpósio Nipo-Brasileiro de Ciência e Tecnologia*, pages 387–396, Campos do Jordão, SP, Brazil, August 1995.

[10] S. E. Deering and D. R. Cheriton. Multicast Routing in Datagram Internetworks and Extended LANs. *ACM Transactions on Computer Systems*, 8(2):85–110, May 1990.

[11] R. Drummond e C. D. Cianni. OMNI - Sistema de suporte a aplicações distribuídas. In *Anais do VI Simpósio Brasileiro de Engenharia de Software*, pages 309–324, November 1992.

[12] H. Eriksson. Mbone: The Multicast Backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[13] F. Cristian et al. Atomic Broadcast: From Simple Message Diffusion to Bizantine Agreement. In *The 7th International Conference on Distributed Computing Systems*, pages 322–339, Berlin, Germany, September 1987.

[14] F. Hermann et al. Multi-threaded UNIX Processes in CHORUS/MIX. *Chorus systèmes*, June 1989.

[15] H. E. Bal et al. Programming Languages for Distributed Computing Systems. *ACM Computing Surveys*, 21(3):261–322, September 1989.

[16] J. Kramer et al. Configuration Support for System Description, Construction and Evolution. In *Proceedings of the IEEE Fifth International Workshop on Software Specification and Design*, pages 28–33, Pittisburgh, PA, USA, May 1989.

[17] K. P. Eswaran et al. The Notions of Consistency and Predicate Locks in a Database System. *Communications of the ACM*, 19(11):626–633, November 1976.

[18] L. Liang et al. Process Groups and Group Communications. *IEEE Computer*, 23:56–66, February 1990.

[19] M. F. Kaashoek et al. An Efficient Reliable Broadcast Protocol. *Operating Systems Review*, 23(10):5–19, October 1989.

[20] R. Lea et al. COOL: System Support for Distributed Programming. *Communications of the ACM*, 36(9):37–46, September 1993.

[21] S. J. Mullender et al. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23:44–53, May 1990.

[22] H. Garcia-Molina. Elections in a Distributed Computing System. *IEEE Transactions on Computers*, C-31(1):48–59, January 1982.

[23] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[24] M. F. Kaashoek and A. S. Tanenbaum. FAULT TOLERANCE USING GROUP COMMUNICATION. *Operating Systems Review*, 25(4):71–74, April 1991.

[25] J. Kramer and J. Magee. Dynamic Configuration of Distributed Systems. *IEEE Transaction on Software Engineering*, 11(4):424–436, April 1985.

[26] L. Lamport. The Bizantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[27] P. A. Lee and T. Anderson. *Fault Tolerance - Principles and Practice.* Spring-Verlag/Wien - Austria, 2nd edition, 1990.

[28] S. J. Mullender. *Distributed Systems*, chapter 1. ACM Press — Addison-Wesley Publishing Company, 1989.

[29] S. J. Mullender. *Distributed Systems*, chapter 10. ACM Press — Addison-Wesley Publishing Company, 1989.

[30] F. Schneider. Bizantine generals in action: Implementing fail-stop processors. *ACM Transactions on Computer Systems*, 2(2):145–154, May 1984.

[31] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley Publishing Company, 1986.

[32] A. S. Tanenbaum. *Modern Operating Systems*, chapter 9. Prentice-Hall International Editions, 1992.

[33] A. S. Tanenbaum. *Modern Operating Systems*, chapter 15. Prentice-Hall International Editions, 1992.

[34] A. S. Tanenbaum. *Modern Operating Systems*, chapter 14. Prentice-Hall International Editions, 1992.

[35] D. B. Terry. Distributed Name Servers: Naming and Caching in Large Distributed Computing Environments. Technical Report CSL-85-1, Xerox Palo Alto Research Center, February 1985.