UNIVERSIDADE ESTADUAL DE CAMPINAS

FACULDADE DE ENGENHARIA ELÉTRICA

DEPARTAMENTO DE ENGENHARIA DE COMPUTAÇÃO E AUTOMAÇÃO INDUSTRIAL

Serviços de Comunicação para a Plataforma *Multiware*

Autor: **Ladislau Conceição**

Orientador: **Prof. Dr. Eleri Cardozo**

Dissertação submetida à Faculdade de Engenharia Elétrica da Universidade Estadual de Campinas, como parte dos requisitos exigidos para obtenção do título de Mestre em Engenharia Elétrica.

Outubro 1995

Conceição, Ladislau

C744s Serviços de comunicação para a plataforma multiware / Ladislau Conceição. – Campinas: [s.n.], 1995

Orientador: Eleri Cardozo.

Dissertação (mestrado) - Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica.

1. Redes de computação - Protocolos. 2. Tolerância a falha (Computação). 3. Redes de computação. 4. *Comunicação multimídia. I. Cardozo, Eleri. II. Universidade Estadual de Campinas, Faculdade de Engenharia Elétrica. III. Título.

Sumário

Lista de Figuras

Lista de Tabelas

Resumo

A Plataforma *Multiware* é um modelo de referência para o suporte ao processamento distribuído aberto (ODP). Esta plataforma é dividida nas camadas de processamento local, *middleware*, *groupware* e de aplicação. A camada *middleware* tem a responsabilidade de prover as funcionalidades ODP e é composta por serviços de comunicação, de núcleo, de objetos e interface de programação (API). Os serviços de comunicação se constituem na base para o processamento distribuído. Os serviços providos são: comunicação com reserva de banda, comunicação em grupo e comunicação sem-conexão confiável.

Este trabalho apresenta o projeto e implementação destes serviços de comunicação. O serviço de comunicação com reserva de banda provê comunicação para transmissão de dados multimídia, tais como áudio e vídeo. Neste trabalho foi projetado um protocolo para operar sobre redes de tráfego assíncrono, tais como *Ethernet* e FDDI (sem tráfego síncrono). Um esquema de *token-passing* foi utilizado para disciplinar o acesso ao meio. O serviço de comunicação em grupo (*reliable multicast*) é um aperfeiçoamento de um protocolo já existente. Nele foi incluído controle de fluxo e recuperação total em caso de falha do nodo sequenciador. O serviço de comunicação sem-conexão confiável é um serviço de datagramas com reconhecimento. Normalmente, comunicação confiável é somente oferecida em serviços com conexão e serviços sem conexão não oferecem confiabilidade. Neste trabalho um novo protocolo foi projetado com base num protocolo de envio-confirmação. Estes três serviços foram acomodados numa interface de programação uniforme, com primitivas *open*, *close*, *send*, *receive* e *select*. Foi implementado um esquema de portas, similar ao *sockets* do UNIX. Este esquema conta com um *dispatcher* para chamar as funções de cada serviço apropriadamente. Exemplos de como utilizar a interface de programação em C++ e a avaliação desempenho dos serviços concluem o trabalho.

Palavras-chave: Protocolos, *Reliable Multicast*, Comunicação Multimídia, Comunicação Confiável.

Abstract

The Multiware Platform is a reference model to support Open Distributed Processing (ODP). This platform is divided into four layers, namely local processing, middleware, groupware, and application. The middleware layer is responsible for providing ODP functions, comprising communication services, kernel services, object service, and programming interface (API). The communication services – namely bandwidth reservation communication, group communication, and reliable connectionless communication — are the basis for the distributed processing.

In this work these communication services were designed and implemented. The bandwidth reservation service provides support for multimedia communication, such as audio and video. Its protocol was designed to operate in asynchronous traffic networks, such as Ethernet and FDDI (without synchronous traffic). A token passing scheme was used to control the medium access. The group communication service is an improvement from a reliable multicast protocol. Flow control and full recovery sequencer crash were added to the protocol. The reliable connectionless communication service is an acknowledged datagram service. Commonly, realiable communication is provided only in connection-oriented services, and connectionless service does not provide reliability. In this work a new protocol was designed based on a send-acknowledge algorithm. These three services were integrated into an uniform programming interface, which has open, close, send, receive and select primitives. A port scheme such as UNIX sockets was implemented. This scheme has a dispatcher to call the appropriated service functions. The programming interface with utilization examples in C++ and a performance evaluation concludes this work.

Keywords: Protocols, Reliable Multicast, Multimedia Communication, Reliable Communication.

Terminologia

API (*Application Programming Interface*) Interface de programação de aplicação.

Banda passante Capacidade de transmissão de dados no meio físico de uma rede.

B-ISDN (*Broadband Integrated Service Digital Network*) Rede digital de serviços integrados de banda larga.

broadcast Difusão para todos receptores.

buffer Espaço em memória volátil de um computador, onde são armazenadas mensagens temporariamente.

CCITT (*Comité Consultatif Internacional Télégraphique et Téléphonique*) Comitê consultivo internacional de telégrafos e telefonia.

deadlock Estado em que se encontra um sistema onde dois ou mais processos ficam aguardando indefinidamente por recursos alocados entre eles (dependência cíclica).

DQDB (*Distributed Queue Dual Bus*) Barramento dual de fila distribuída.

Ethernet Rede local de computadores, desenvolvida pela Xerox.

FDDI (*Fiber Distributed Data Interface*) Interface de dados distribuídos por fibra.

IP (*Internet Protocol*) Protocolo de rede/transporte do DARPA (Departamento de Defesa Norte-Americano).

ISO (*International Organization for Standardization*) Organização Internacional para Padronização.

multicast Difusão para vários receptores.

OSI (*Open Systems Interconection*) Interconexão de sistemas abertos.

overhead Tempo de processamento ou espaço em memória utilizado por funções do sistema operacional, protocolo, etc, para se autogerir.

TCP (*Transport Control Protocol*) Protocolo de transporte com conexão que utililza o IP.

UDP (*User Datagram Protocol*) Protocolo de datagrama que utiliza o IP.

Capítulo 1

Introdução

Atualmente, um dos maiores desafios na área de processamento distribuído é o suporte às aplicações multimídia [1]. Tais aplicações impõem fortes restrições de temporização e sincronização, capazes de serem atendidas apenas com o emprego de tecnologias de ponta no campo de redes de computadores e programação de sistemas de tempo real.

O projeto temático **Implementação de Modelos para Aplicações Distribuídas Abertas** foi criado para prover o desenvolvimento científico e tecnológico na área de processamento distribuído aberto [1]. Este projeto é composto de três etapas principais. A primeira delas consiste na análise e definição de modelos para processamento distribuído aberto (ODP); a segunda etapa corresponde à implementação de um núcleo de processamento distribuído que suporte os modelos ODP [2] e, por último, a validação dos modelos implementados através de aplicações multimídia. Este trabalho está inserido na segunda etapa do projeto.

1.1 O Projeto de Pesquisa Temático

Os sistemas de processamento distribuído da informação têm sua evolução ditada, basicamente, por três fatores. Primeiro, o constante decréscimo de preço acompanhado do aumento de capacidade dos computadores pessoais e estações de trabalho, vem motivando o emprego de processadores menores e em maior número, em oposição aos chamados

[1] Processamento simultâneo e em tempo real de vídeo, áudio, dados e outros tipos de informação

mainframes. Segundo, a tecnologia de comunicação, também em constante evolução (em particular na última década com o advento da fibra ótica) permite a eliminação do problema colocado pela distância física. O terceiro, a padronização, como os modelos de interconexão OSI, o sistema operacional POSIX e o *X Window System*, constitui peça fundamental na interconexão dos sistemas heterogêneos, através da eliminação das diferenças estruturais.

Em vista disto, é de se esperar que os sistemas de processamento distribuído passem a atender uma gama cada vez maior de aplicações. Aplicações convencionais, tais como compartilhamento, acesso remoto e integração de recursos, já fazem parte do *software* básico do processador. Outras aplicações distribuídas, tais como sistemas de banco de dados e de gerenciamento de redes, são amplamente disponíveis comercialmente.

Entretanto, observando as aplicações não convencionais, por exemplo, aquelas onde requisitos temporais e de sincronização necessitam ser garantidos, os padrões e modelos atualmente existentes são totalmente inadequados. Exemplos de tais aplicações são:

- Sistemas multimídia, onde dados de vídeo e áudio devem ser gerados, transmitidos e regenerados de forma síncrona e periódica. Aplicações neste campo incluem CAD cooperativo, telemedicina, telepresença, etc.
- Sistemas de supervisão e controle de processos, onde determinadas tarefas (leitura de sensores, estimação de estado, atuação, etc.) devem ser processadas respeitando-se prioridades e *deadlines*.
- Sistemas flexíveis de manufatura, onde processos automatizados devem ser executados segundo uma precisa seqüência temporizada.
- Simuladores para treinamento (de vôo, de processos, etc.), onde a ocorrência de eventos no tempo deve obedecer àquela verificada no sistema real.

O grande desenvolvimento tecnológico que vem ocorrendo nestas áreas tem criado uma enorme expectativa em relação à uma nova gama de serviços e, em conseqüência, novas aplicações que poderão ser atendidas com estas tecnologias. Os recentes avanços em redes de computadores, redes públicas e sistemas multimídia introduziram novas dimensões ao projeto e realização de sistemas distribuídos. A instalação de redes interligadas por fibra ótica e o desenvolvimento correspondente de técnicas de transmissão e comutação (B-ISDN, DQDB, FDDI, etc) são pontos fundamentais na oferta de recursos quase ilimitados de comunicação. A integração de voz, dados e imagem vem exigindo o desenvolvimento de

novos protocolos, modelos e metodologias.

Em um mercado aberto, muitos serviços serão oferecidos através de redes de computadores. Os fornecedores destes serviços (provedores) localizar-se-ão em regiões distintas, apresentando custos e qualidade diferentes. Será tarefa do usuário estruturar as suas aplicações, baseando-se nos serviços disponíveis e levando em conta as restrições organizacionais e operacionais.

A tendência atual indica que o trabalho cooperativo (que inclua a comunicação multimídia por um lado, e as plataformas necessárias ao desenvolvimento das aplicações distribuídas por outro) deve tornar-se a ênfase básica dos sistemas distribuídos futuros. Em contraste com os sistemas tradicionais, o projeto de aplicações distribuídas nos próximos anos significará a integração de entidades e seus serviços heterogêneos existentes, mais do que a simples distribuição de funcionalidade a componentes dedicados, que caracteriza os projetos atuais de sistemas distribuídos.

Vários esforços de pesquisa e desenvolvimento, bem como inúmeras atividades de padronização, estão em curso no mundo inteiro. Como exemplo, os esfoços do CCITT (projeto DAF: *Distributed Application Framework*), da ISO (projeto ODP: *Open Distributed Processing* [2]) e da OMG (arquitetura CORBA: *Common Object Request Broken Architecture*). Os padrões deverão, como mencionado anteriormente, ser de importância decisiva para a implantação, com sucesso, desses novos ambientes abertos, heterogêneos e cooperativos.

O projeto temático tem como meta central o desenvolvimento de modelos, padrões, técnicas e procedimentos que suportem aplicações distribuídas com restrições de temporização e sincronização.

Os tópicos principais são:

- Estudo de modelos de processamento distribuído, com ênfase especial nos aspectos de sincronização de aplicações cooperativas;
- Análise desses modelos, tendo em conta os esforços de padronização hoje em curso (por ex: ODP/OSI, DAF/CCITT) e o uso de padrões já estabelecidos, como a arquitetura CORBA e padrões IMA (*Interactive Multimedia Architecture*);
- Projeto e implementação de um núcleo de suporte de processamento aberto, distribuído

e cooperativo em um ambiente *testbed*, que viabilize a especificação, prototipagem e avaliação dos modelos propostos.

Os resultados deste desenvolvimento deverão ser testados em aplicações com ênfase na área multimídia como, por exemplo, CAD cooperativo e simulação distribuída.

O projeto e implementação do núcleo de suporte de processamento distribuído foi denominado de **Plataforma *Multiware***. Este subprojeto é discutido a seguir.

1.2 A Plataforma *Multiware*

Denominou-se de Plataforma *Multiware* um modelo de referência para o suporte ao processamento distribuído aberto [1]. Este modelo de referência foi estabelecido na primeira fase do projeto e se presta a identificar os vários componentes e serviços de uma plataforma de suporte ao processamento distribuído aberto. Uma plataforma deve executar sobre múltiplas famílias de *hardware*; sobre diversos sistemas operacionais; e sobre diversas infraestruturas de comunicação. Em termos funcionais, uma plataforma é dividida em camadas. A plataforma *Multiware* é dividida em quatro camadas (figura 1.1).

A **camada de processamento local** é constituída pelo nodo processador com o seu *software* básico como sistema operacional (eventualmente baseado em *microkernel*), protocolos de comunicação, recursos de apresentação, etc. Esta camada se limita ao nodo processador e não provê nenhuma facilidade de processamento distribuído.

A **camada *middleware*** provê facilidades de processamento distribuído para as camadas superiores. Alguns serviços oferecidos por esta camada são: serviço de nomes, gerenciamento de recursos distribuídos e replicados, processamento de transações, transparência de acesso e localização, dentre outros. Esta camada, idealmente, deve implementar suas funcionalidades segundo o paradigma de orientação a objetos.

A **camada *groupware*** provê as funcionalidades requeridas por diferentes classes de aplicações, como trabalho cooperativo suportado por computador (CSCW), inteligência artificial distribuída (IAD), automação de escritórios, etc. Serviços típicos oferecidos por esta camada são: gerenciamento de diálogo, protocolos de interação, manipulação de documentos multimídia, dentre outros.

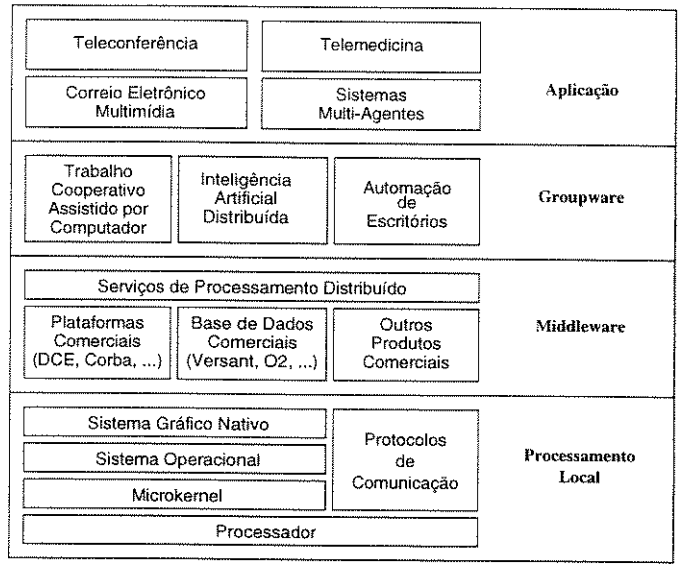

Figura 1.1: Estruturação de uma plataforma *Multiware*

A **camada aplicação** reúne aplicações específicas que se utilizam dos serviços da camada *groupware* oferecidos para a sua respectiva classe. Exemplos típicos: teleconferência, telemedicina, CASE (da classe CSCW); sistemas multi-agentes e sistemas especialistas distribuídos (da classe IAD); correio eletrônico de voz e editoração multimídia (da classe automação de escritórios).

Este subprojeto descreve a implementação de uma plataforma aderente ao modelo ODP da ISO denominada **plataforma ODP**. A plataforma implementa na camada *middleware* as funções de gerenciamento e ligação (*binding*) estabelecidas no modelo ODP. A camada de processamento local é composta do sistema operacional UNIX, da pilha de protocolos TCP/IP e de protocolos de comunicação implementados acima do TCP/IP.

1.3 A Camada *Middleware*

A camada *middleware* foi implementada com determinados componentes que têm a responsabilidade de prover as funcionalidades ODP. A seguir são descritos os componentes desta camada:

- serviços de comunicação;
- serviços de núcleo;
- serviços de objetos;
- interface de programação.

Os **serviços de comunicação** constituem-se da base para o processamento distribuído. Nesta implementação, os serviços de comunicação provêem a comunicação em grupo, a comunicação sem-conexão ponto-a-ponto confiável e a comunicação com reserva de banda (utilizada na comunicação multimídia).

O **núcleo** provê os serviços básicos de processamento distribuído. Nesta implementação, o núcleo implementa serviços de identificação (*naming*), processamento de transações, persistência e mapeamento de *ports* (*portmapping*).

Os **serviços de objetos** provêem a infraestrutura necessária à existência de objetos. Esta infraestrutura estabelece as propriedades fundamentais de objetos (tais como instanciação, classificação e encapsulamento), além de prover mecanismos para o gerenciamento de objetos. No modelo ODP as atividades de gerenciamento ocorrem em três diferentes níveis:

- nível de objeto: criação, remoção, *checkpointing* e recuperação de objetos;
- nível de *cluster*[2]: criação, desativação, reativação, destruição e migração de *cluster*;
- nível de cápsula[3]: criação e destruição de cápsulas.

Nesta implementação, tais serviços estão concentrados numa **base de objetos**. A base de objetos provê facilidades de agrupamento de objetos em *clusters*, e de *clusters* em cápsulas. Outra funcionalidade importante da base de objetos é prover a interação entre objetos onde um canal liga uma interface de um objeto à uma interface de um objeto servidor. Em [3] encontra-se a descrição detalhada destes serviços.

A **interface de programação** (API – *Aplication Programming Interface*) fornece um acesso uniforme aos serviços da camada *middleware*. A camada *groupware* acessa estes serviços via API. Nesta implementação, a API consiste de uma hierarquia de classes

[2] Agrupamento de objetos
[3] Agrupamento de *cluster*

especificadas segundo a metodologia OMT (*Object Modeling Technique* – Técnica de Modelagem de Objeto) e disponíveis na linguagem de programação C++. Em [4] encontra-se a descrição detalhada da interface de programação.

1.4 Os Serviços de Comunicação

São três os serviços de comunicação providos:

- comunicação com reserva de banda;
- comunicação em grupo;
- comunicação sem-conexão ponto-a-ponto confiável.

Estes serviços são necessários para atender a comunicação de dados multimídia, funções de comunicação do núcleo e outras camadas.

O serviço de **comunicação com reserva de banda** provê comunicação com conexão, com difusão e garantia de banda passante mínima e um retardo máximo. A banda mínima é garantida por um mecanismo de reserva de banda distribuído. Este serviço, entretanto, não garante a entrega de mensagens. Este serviço é necessário para a comunicação de dados multimídia, tais como vídeo e áudio.

O serviço de **comunicação em grupo** provê comunicação com conexão a um grupo com difusão entre todas as portas conectadas ao grupo e com garantia de entrega. Este serviço é implementado utilizando um algoritmo de difusão confiável (*reliable broadcast*). Este serviço é necessário para as funções no núcleo da plataforma *multiware*, tais como geração de identificadores globais e mapeamento de porta.

O serviço de **comunicação sem-conexão ponto-a-ponto confiável** provê comunicação sem-conexão (datagramas) com garantia de entrega dos dados. Este serviço é implementado utilizando um protocolo de envio e reconhecimento e é necessário para muitas funções do núcleo, onde não é conveniente utilizar um serviço com conexão e que requerem confiabilidade na comunicação.

1.5 Objetivo

Este trabalho tem como objetivo: implementar os serviços de comunicação com reserva de banda, de grupo, sem-conexão ponto-a-ponto confiável, necessários para a camada *middleware*.

Para atingir este objetivo, foram estabelecidas as seguintes metas:

- definir, projetar e implementar um protocolo de comunicação para multimídia;
- definir, projetar e implementar um protocolo de comunicação em grupo;
- definir, projetar e implementar um protocolo de comunicação sem-conexão ponto-a-ponto confiável;
- integrar a interface e funções dos três serviços para acesso dos três protocolos.

1.6 Ambiente de Desenvolvimento

Segundo a visão do Modelo de Referência OSI (RM-OSI) [5], os serviços de comunicação deste trabalho localizam-se na camada de transporte. A **entidade** é o elemento em cada camada que provê os serviços para a camada superior. Para prover estes serviços, ela utiliza os serviços da camada inferior. A entidade que provê um serviço é chamada de **provedora de serviço** e a que utiliza os serviços é chamada de **usuária de serviço**. A pilha de protocolos TCP/IP[4] é utilizada como provedora de serviços, sendo o núcleo e outros serviços da camada *middleware* usuários dos serviços desta camada (figura 1.2). Neste trabalho, todos os usuários de serviço desta camada, os demais serviços da camada *middleware*, bem como outras camadas da plataforma *multiware*, serão tratados genericamente por aplicação ou processo-aplicação.

[4] Mais especificamento, utiliza o UDP para a comunicação de mensagens

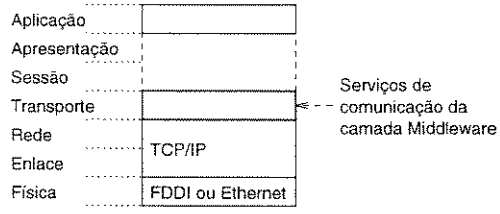

Figura 1.2: Modelo de Referência OSI

Os serviços de comunicação são acessados através de ponto de acesso de serviço (SAP). A **porta** é tipo de ponto de acesso dos três serviços de comunicação. O serviço é acessado através da porta por uma primitiva. A porta possui as informações necessárias para determinar quais serviços pode prover e que protocolo utilizar (figura 1.3).

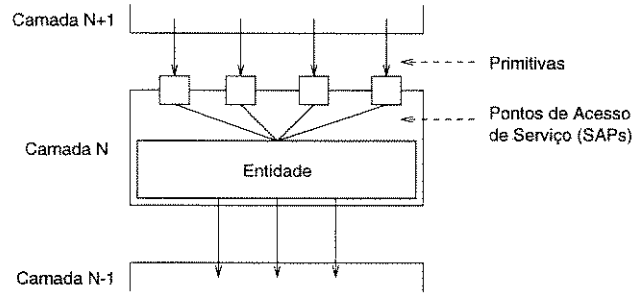

Figura 1.3: Serviço, Primitiva e Protocolo no RM-OSI

O ambiente de desenvolvimento de *software* determina os recursos que podem ou não ser utilizados. Um ambiente de desenvolvimento de *software* normalmente associa uma ambiente de programação, um sistema operacional e, neste projeto, uma pilha de protocolos. Sistemas operacionais atualmente disponíveis incluem DOS, UNIX, OS/2, VMS, entre outros. Neste projeto, o ambiente escolhido foi o UNIX devido a sua familiaridade e disponibilidade. Linguagens de programação disponíveis num ambiente UNIX incluem C, C++, Pascal, Smalltalk, etc. A linguagem C é nativa dos sistemas UNIX e, provavelmente, a mais utilizada no desenvolvimento de *software* no mundo. A linguagem de programação C++ é uma extensão da linguagem C para suportar programação orientada a objetos. Por ser uma linguagem padronizada e ter muitos fornecedores de compiladores e outras ferramentas de desenvolvimento, ela foi escolhida para a implementação do *software*.

1.7 Descrição dos Capítulos

Cada serviço de comunicação é apresentado em um capítulo próprio. A pesquisa bibliográfica, as questões de projeto, a descrição do serviço, o protocolo e a descrição da implementação compõem cada um destes capítulos.

O capítulo 2 apresenta o serviço de comunicação com reserva de banda.

O capítulo 3 apresenta o serviço de comunicação em grupo.

O capítulo 4 apresenta o serviço de comunicação sem-conexão ponto-a-ponto confiável.

O capítulo 5 apresenta as questões de projeto e a implementação da integração dos três serviços através de portas e de um *dispatcher*. A porta é o acesso das aplicações aos serviços desta camada. As primitivas manipulam a porta para acessar estes serviços. Como cada protocolo oferece um serviço específico, as primitivas diferem nos seus parâmetros para cada protocolo. Entretanto, muitas funções são idênticas entre os três serviços. Estas funções foram reunidas num módulo, o *dispatcher*, que acessa as portas de forma uniforme. Quando uma função específica de um serviço é chamada, o *dispatcher* chama a função específica de um protocolo.

A conclusão apresenta os resultados do desenvolvimento do *software*, tais como desempenho dos protocolos e a qualidade do *software*. Os objetivos traçados são confrontados com os resultados obtidos. Ao final, são apresentadas sugestões para a continuidade do trabalho.

Capítulo 2

Comunicação com Reserva de Banda

2.1 Introdução

Aplicações multimídia são aquelas que interagem com o usuário fazendo uso simultâneo de diversos meios, como áudio, imagens estáticas, vídeo em movimento, gráficos e texto [6]. Aplicações multimídia podem variar muito. Desde uma enciclopédia animada, onde o usuário pode ouvir sons de animais ou cenas de um filme (que envolve apenas a reprodução de dados), até teleconferências (que envolve comunicação em redes), onde cada participante em sua estação pode ouvir e observar os demais como se estivessem numa mesma sala.

As mídias de vídeo (imagem em movimento) e áudio são também chamadas de mídias contínuas. Isto porque a percepção humana as vê como tais.

A comunicação de mídias tipo textos, gráficos, imagens (estáticas), desenhos, etc., podem ser feitas por protocolos existentes. Mas as mídias contínuas não são atendidas adequadamente por eles.

A seção a seguir apresenta as características de mídias contínuas relevantes para realizar a sua comunicação.

2.1.1 Características das Mídias de Vídeo e Áudio

Vídeo

Uma imagem de vídeo, para ser percebida como contínua pelo olho humano, deve ser atualizada a uma taxa mínima de 25 quadros por segundo. Quando digitalizada, uma imagem com resolução de cores e pontos de um televisor comum (700x525 pontos x 24 bits de cores) ocupa aproximadamente 1M bytes. Com a exibição de 30 quadros por segundo, temos uma taxa de transferência de 30M bytes por segundo. Mesmo para os mais rápidos computadores, esta taxa de transferência de dados não é possível nem mesmo nos barramentos CPU-memória.

Existem várias formas de reduzir este volume de dados. Algumas delas são:

- diminuir o número de **bits de cor**. Para aplicações que não exigem alta fidelidade de cor, 8 bits por *pixel* é aceitável. Com 256 tons de cor, pode-se perfeitamente exibir imagens de TV.
- diminuir a **resolução** em pixels da imagem. Para a transmissão de imagens de TV se utilizam aproximandamente 360K (700x525) pixels. Mas outras aplicações podem aceitar uma resolução bem menor. Pode-se ver nitidamente o rosto de uma pessoa com apenas 40K (200x200) pixels.
- diminuir o número de **quadros por segundo**. Em aplicações onde as imagens se alteram lentamente (animação por computador), ou quando é aceitável uma perda de fidelidade (videofone), uma taxa menor de quadros por segundo pode ser utilizada.
- utilizar **compressores e descompressores** de dados. Imagens sempre ocupam grandes volumes de dados. Mas, normalmente, permitem grandes taxas de compressão. Em aplicações de mídias contínuas, compressores e descompressores devem trabalhar em tempo real para viabilizar aplicações interativas, e normalmente são implementados em *hardware*. Existem várias técnicas para realizar a compressão de dados. As técnicas sem perdas permitem que os dados descomprimidos sejam idênticos àqueles que foram comprimidos. Estas técnicas são ideais para preservar o máximo da fidelidade. As técnicas com perdas implicam em perda da fidelidade em prol de um aumento significativo na taxa de compressão. Normalmente, estas técnicas são utilizadas quando não há muita banda passante disponível na rede de computadores para transmitir uma imagem de me-

lhor qualidade ou então quando a perda de fidelidade não prejudica a compreensão das imagens.

- **tipo de imagem.** Com o emprego de compressores de dados, o tipo de imagem influencia muito a taxa de compressão e, consequentemente, o volume dos dados. Quanto menos nuances de cores ou menos detalhes tiver uma imagem, maior será a taxa de compressão obtida. Por exemplo, numa teleconferência, é recomendável que os participantes fiquem de costas para um fundo que não se modifique.

Áudio

O volume de dados para a transmissão de áudio é bem menor que aquele para a transmissão de vídeo. A tabela 2.1 apresenta valores de banda passante para alguns tipos de qualidade de áudio.

Qualidade	Frequência de Amostragem	Bits por amostra	Taxa transferência
Telefonia	8KHz	12	12KB/s
Música AM	10KHz	14	17KB/s
Música disco	16KHz	16	32KB/s
Música FM	35KHz	16+16 (estéreo)	140KB/s
Música CD	44KHz	16+16 (estéreo)	176KB/s

Tabela 2.1: Qualidade de áudio

Para serem transmitidas, as amostras de áudio capturadas normalmente são enviadas em blocos. Os blocos devem ter um número de amostras suficientemente grande para manter a exibição contínua. Quanto maior o bloco maior a economia de chaveamentos de contexto de processos (menor *overhead* do sistema operacional) e menor o número de pacotes na rede. Por outro lado, quanto maior for o tamanho do bloco, maior será o retardo na chegada do áudio ao seu destino, que não é desejável. Existe um tamanho de bloco ótimo conforme o desempenho da rede utilizada para a sua transmissão.

Outra característica particular do áudio é o silêncio. Quando for detetado que a aquisição de áudio corresponde a um intervalo de silêncio, os dados não necessitam ser transmitidos. Assim, CPU e banda passante podem ser aproveitadas para outras aplicações.

O *hardware* de aquisição deve ser capaz de prover esta informação (silêncio) ao processo que o controla.

Assim como o vídeo, o áudio também pode ser comprimido. As mesmas técnicas de compressão de vídeo podem ser utilizadas para comprimir áudio. Entretanto, se opta por técnicas diferenciais em tempo. Um exemplo deste tipo de técnica consiste em se ter um valor absoluto no inicío de cada bloco com todos os bits do espectro (por exemplo 16 bits), e os valores diferenciais a partir deste valor absoluto com um número bem menor de bits para cada amostra (por exemplo 4 bits). Assim se economiza 12 bits por amostra. Este tipo de técnica pode introduzir perdas quando a variação de uma amostra para outra for maior que a quantidade que 4 bits podem representar.

Compressão

A aquisição, processamento e exibição de dados de mídia contínua são realizados sempre numa taxa constante. Isto facilita na determinação da periodicidade de tarefas para o escalonamento. Entretanto, com o uso de compressão, a variância do tamanho destes dados é muito alta. Mesmo existindo um valor máximo para o tamanho destes dados (que seria uma taxa de compressão 1:1, isto é, nenhuma compressão), ele é muito distante do valor médio. Isto causa uma dificuldade no cômputo da banda passante da rede necessária para a sua transmissão. Por exemplo, numa imagem com tamanho de 500KB sem compressão e 25KB em média com compressão, há uma variação 20 vezes maior para o pior caso.

2.1.2 Redes de Computadores

Quando dados de mídia contínua são transmitidos em redes de computadores, é necessário que a rede tenha velocidade suficiente para transmitir o volume de dados e que a rede garanta as restrições de tempo destes dados sejam satisfeitas. Existe uma grande variedade de redes que para fins de transmissão de mídia contínua, elas podem ser classificadas conforme o tipo de serviço que oferecem (tabela 2.2).

O ideal para a transmissão de mídia contínua são as redes que oferecem serviço

[1]Apesar de redes como TOKEN-RING não estarem sujeitas à contenção e terem filas de mensagens priorizadas, o tráfego de mensagens não pode ser considerado síncrono.

Serviço	Retardo	Variância do retardo	Exemplos
Assíncrono	Indefinido	Grande	*Ethernet*, TOKEN-RING[1]
Síncrono	Definido	Pequena	FDDI
Isócrono	Definido	Nenhuma	ATM, DQDB, FDDI-II

Tabela 2.2: Tipos de tráfego de dados em redes

isócrono (retardo constante) [7]. Estas redes, entretando, são ainda caras e de baixa disponibilidade.

Redes de tráfego síncrono permitem o envio de dados numa taxa constante, apesar de sofrerem variações no retardo. FDDI é um exemplo de rede que oferece serviços de tráfego síncrono. Entretanto, muitos fabricantes implementam redes como FDDI sem a opção do tráfego síncrono, oferecendo apenas o tráfego assíncrono. Consequentemente, a implementação completa do FDDI é difícil de se obter comercialmente.

Redes convencionais como a *Ethernet* são largamente utilizadas, pois são de baixo custo e operação simples. O problema básico é o não determinismo no retardo de transmissão de suas mensagens. Não existe limite do retardo máximo para uma mensagem chegar ao outro nodo, mesmo em redes de alta-velocidade. Nestas redes oferecem somente serviço assíncrono.

Neste trabalho, apenas redes de serviço assíncrono (*Ethernet* e FDDI) estavam disponíveis para implementação dos protocolos. Mesmo a rede FDDI utilizada não oferecia serviço síncrono, devido à opção do fabricante quanto a implementação do padrão FDDI.

2.1.3 Transmissão de Mídia Contínua em Redes de Tráfego Assíncrono

Em aplicações não interativas é possível armazenar toda a mídia contínua no disco local e então passar a exibí-la.

Entretanto, em aplicações interativas, não há como realizar a transmissão completa da mídia antes da sua exibição (pois assim não haveria interação). Portanto, é necessário a transmissão da mídia em tempo real. Isto é, a velocidade de comunicação deve ser a mesma da velocidade da captura da mídia.

Para compensar a variação de retardo, podem ser utilizados *buffers*. Neles são armazenados um número inicial de dados para serem consumidos. Se a taxa de transmissão dos dados for a mesma que a de consumo (isto é, se não há muita variação de retardo), a quantidade de dados no *buffer* não irá variar. Quando houver sobrecargas na rede, e ocorrerem atrasos na transmissão dos dados, o consumo dos dados pode manter-se constante durante algum tempo. Isto porque ainda haverá dados no *buffer* para consumo. Quando a situação da rede se normalizar, possivelmente haverá uma rajada (uma série de dados numa taxa muito mais rápida que a velocidade de consumo) e estes dados serão armazenados no *buffer* para compensar a variação de retardo [8]. Na figura 2.1 é apresentado o funcionamento do *buffer* quando ocorrem atrasos na transmissão dos dados e posteriormente uma rajada.

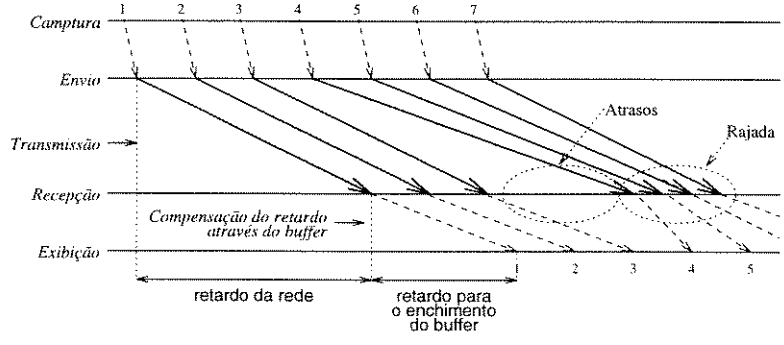

Figura 2.1: Diagrama de tempo do funcionamento do *buffer*.

Quando o retardo das mensagens for muito longo, pode ocorrer um *buffer underflow*, e a aplicação perder o seu *deadline* por não ter dados para consumir. O *buffer underflow* pode ser evitado, aumentando o tamanho do *buffer* e, consequentemente, o tempo para enchê-lo.

Entretanto, para redes de tráfego assíncrono, não adianta ter um *buffer* muito grande, pois podem ocorrer atrasos tão longos que não sejam possíveis de se compensar. Além disso, quanto maior o *buffer*, maior será o tempo para enchê-lo e, consequentemente, maior será o retardo total da interação. Numa aplicação tipo videofone, com uma rede que oferece um retardo de 100ms e um *buffer* dimensionado para um tamanho que exija 400ms de retardo para enchê-lo, tem-se um total de 500ms de atraso. Este atraso tira toda a naturalidade da conversação e pode até inviabilizá-la.

Para eliminar ou pelo menos minimizar o efeito do retardo imprevisível, pode-se

priorizar o tráfego de mídia contínua. Esta pode ser uma solução simples e rápida se nos nodos da rede raramente ocorre a utilização de aplicações interativas que empreguem mídia contínua. Se for comum o uso de mídia contínua, poderá haver momentos onde o tráfego destes dados congestiona a rede, ocasionando um maior retardo das mensagens e com consequente perda dos *deadlines* das aplicações.

Para garantir que as mensagens de mídia contínua não sofram retardos muito longos, é necessário uma disciplina de envio de todas as mensagens na rede. Uma solução é utilizar um esquema de prioridade de mensagens combinado com um esquema de reserva de recursos. No caso, o recurso é a banda passante da rede, isto é, a capacidade de transmissão de dados no tempo, normalmente medida em bits por segundo (bps). O termo **reserva de banda** será utilizado daqui por diante para expressar o mecanismo que permite reserva de banda passante e o termo **reserva** para expressar a reserva de uma parte da banda passante da rede para um nodo ou uma aplicação do nodo, conforme o contexto.

A reserva de banda garante que a taxa de transmissão de dados será, em média, sempre a mesma. O esquema de prioridade garante que o retardo das mensagens será limitado. Neste trabalho, o esquema de reserva de banda e o esquema de prioridades de mensagens será tratado daqui por diante, como um todo, por **protocolo de reserva de banda.**

2.1.4 Protocolos de Reserva de Banda

O objetivo de um protocolo de reserva de banda é garantir uma banda passante à uma aplicação, de forma que ela possa transmitir seus dados constantemente,sem sofrer com a sobrecarga da rede [9, 10]. A banda passante fica distribuída entre os nodos e cada nodo atribui parte de sua banda às aplicações (figura 2.2).

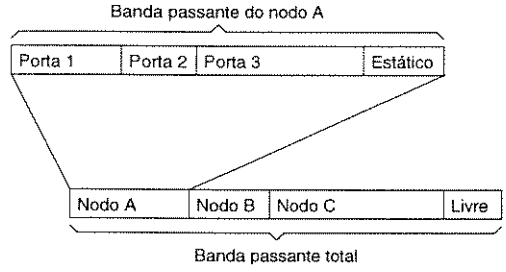

Figura 2.2: Reserva de banda entre os nodos e entre as portas

Para obter uma reserva, uma aplicação deve solicitar banda passante suficiente para transmitir seus dados. A aplicação deve informar qual a taxa de dados e o atraso máximo necessários. Se for possível atender a reserva solicitada, o protocolo a permite e irá garanti-la durante a sua existência. Se não for possível atender o pedido de reserva, a aplicação simplesmente não pode transmitir.

Por exemplo, seja uma rede que provê 100Mbps e destes 80Mbps já estão reservados para várias aplicações. Uma aplicação solicita uma nova reserva de 12Mbps: a rede tem condições de atendê-la e ainda continuar assegurando as demais reservas. Uma outra aplicação solicita uma reserva de 10Mbps: este pedido será negado, porque há somente 8Mbps de banda passante restante.

A banda passante não totalmente utilizada pela reserva é aproveitada para transmissão de tráfego assíncrono.

Para o esquema de reserva de banda, existem duas abordagens básicas: *hard real-time* e *soft real-time*.

Na abordagem *hard real-time*, a utilização da banda nunca pode extrapolar a banda reservada. Isto é, se um nodo reservou 15Mbps, ele não pode num dado instante utilizar 16Mbps. Assim, as aplicações devem pedir uma reserva de banda para o pior caso. É uma abordagem pessimista, pois considera que num dado instante todas as aplicações podem ter a necessidade de transmitir o máximo de dados. Assim se garante que nenhuma aplicação irá perder o seu *deadline*. Entretanto, quando a transmissão for em torno de um valor médio, parte da banda alocada ficará ociosa e não poderá ser utilizada para outras aplicações.

Na abordagem *soft real-time*, a utilização da banda pode, por breves momentos, extrapolar a banda reservada. As aplicações devem requisitar banda para o caso médio. Assim, é possível que um maior número de reservas possam ser feitas. É uma abordagem otimista, pois considera que nunca (ou muito raramente) todas as aplicações irão utilizar, num mesmo instante, mais banda passante do que lhes fora reservado.

Pode-se fazer variações a partir da abordagem *soft real-time*. Uma delas é impor um limite de utilização de banda de cada reserva. Isto ajuda a minimizar o efeito de sobrecarga da rede, que leva algumas aplicações a perder seus *deadlines*.

2.1.5 Operação Inter-Rede

Para se estabelecer uma conexão que percorre por várias sub-redes, é necessário realizar a reserva de banda em cada uma destas sub-redes e realizar a retransmissão dos dados em cada nodo repetidor. Para se obter esta reserva de banda ao longo de várias sub-redes é necessário um **protocolo de reserva de banda inter-redes**, que opere acima do protocolo de reserva de banda. Este protocolo deve reservar toda a banda passante necessária em cada uma das sub-redes e realizar a retransmissão adequadamente.

A reserva de banda inter-redes deve levar em consideração os seguintes pontos:

- conexão iniciada/terminada pelo receptor;
- alocações ocorrem concorrentemente;
- múltiplas conexões para um mesmo emissor em diferentes sub-redes;
- conexões dinâmicas.

A figura 2.3 mostra um exemplo de como deve atuar o protocolo. O nodo com um circulo (nodo F) é o nodo emissor e o nodo com um quadrado (nodo J) é o nodo receptor. Para estabelecer uma conexão de 10 KBps, por exemplo, é necessário que sejam alocados 10 KBps nas redes a, b e c.

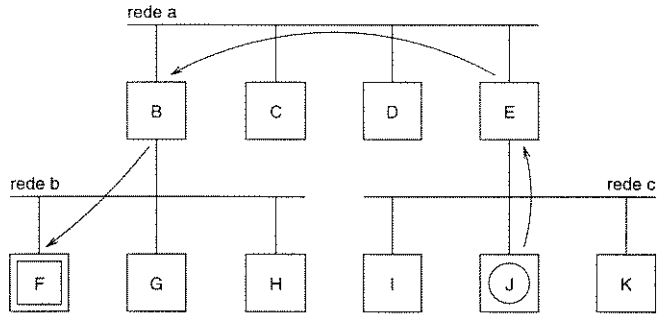

Figura 2.3: Conexão entre duas portas atrevés de várias redes

Os nodos B e E irão atuar como nodos retransmissores. O nodo E estabelece uma conexão com J, de tal forma que ele é o receptor de J e emissor para B. O nodo B estabelece uma conexão com E da mesma forma. Se um nodo receptor deseja receber os dados de J também, por exemplo, o nodo C, não é necessária nenhuma alocação de banda adicional. O nodo E passa a realizar *multicast* na sub-rede a fim da mensagem retransmitida atingir os nodos C e B.

Para realizar a alocação inter-redes, pode-se consultar em cada sub-rede onde é necessário a alocação, se existe banda suficiente para a transmissão dos dados. Se em todas as sub-redes houver banda passante suficiente, a conexão é efetivada com o pedido de alocação em cada uma delas. Se não houver, retorna-se um erro para a aplicação receptora. Entretanto, durante a consulta, a banda passante restante em uma das sub-redes pode se alterar e a consulta já não ter mais validade.

Pode-se contornar este problema, alocando diretamente em cada sub-rede a banda passante. Se em alguma sub-rede não houver banda passante suficiente e a requisição for negada, as alocações obtidas com sucesso nas outras sub-redes são liberadas e um erro é enviado a aplicação receptora.

Uma aplicação receptora, que não consegue conexão com a emissora numa primeira tentativa, pode realizar outras tentativas até que alguma banda passante seja liberada no caminho e ela consega a conexão. Se este procedimento for adotado, pode ocorrer na rede um grande fluxo de pedidos de reserva e liberação sem que haja um efetivo estabelecimento de conexão. Podem ocorrer, inclusive, casos de postergação indefinida, apesar de existir banda suficiente em todas as sub-redes para estabelecer uma conexão, mas devido às contínuas

alocações e liberações, nenhuma conexão é estabelecida.

Pode-se, ao invés de liberar imediatamente as reservas obtidas com sucesso, esperar até se obter banda passante suficiente nas sub-redes onde se obteve fracasso e completar a conexão. Entretanto, este procedimento pode levar um à *deadlock*.

Existem várias formas de tratar este problema, já que é um problema clássico de programação concorrente: vários processos disputando vários recursos. Uma solução simples é estabelecer uma ordem de reserva de banda passante entre as sub-redes. A alocação inter-redes sempre deve realizar a alocação na mesma ordem. Por exemplo, no caso da alocação simultânea de F-J e K-G, têm-se as mesmas sub-redes em comum. Se a ordem de alocação de banda nas sub-redes for a, b e c, então os dois nodos receptores, F e K, devem solicitar a reserva de banda primeiro à rede a, depois à rede b e, por fim, à rede c. Se em alguma destas sub-redes houver banda passante suficiente somente para uma das conexões, então somente uma requisição dos nodos obterá sucesso, sem o perigo de *deadlock* ou postergação indefinida.

O problema desta abordagem é a necessidade de obedecer-se a ordem de alocação nas sub-redes. Podem ocorrer casos em que a sub-rede do nodo receptor não dispõe da banda passante necessária, mas uma alocação está sendo feita numa sub-rede distante. Isto é, com este método gasta-se mais mensagens. Para realizar o estabelecimento de conexões com menor número de mensagens, pode-se adaptar um algoritmo de exclusão mútua distribuído, tal como apresentado em [11], para realizar a alocação de banda nas sub-redes.

Soluções para este problema são propostas em [12, 13, 14]. Neste trabalho a alocação inter-redes não foi implementada devido à impossibilidade de estabelecer o roteamente desejado sobre o TCP/IP e também a falta de tempo hábil e complexidade dos algoritmos.

2.2 Projeto

Este protocolo de transporte tem como protocolo de rede o IP. Apesar do IP não oferecer nenhum recurso de tempo real, nem prioridades para mensagens, foi escolhido, pois não havia nenhum outro disponível para as redes locais que seriam utilizadas (*Ethernet* e FDDI), nem seria possível implementar em tempo hábil um protocolo de rede.

Uma alternativa atraente seria o protocolo XTP [15]. O XTP já realiza comunicação em tempo real e comunicação em grupo. Mas este ainda não é disponível comercialmente.

Foram utilizados *sockets* de UDP (*User Datagram Protocol*) como interface para o IP (pois não existe acesso direto ao IP[2]). O UDP mapeia quase todas as suas funções diretamente para o IP, com baixo *overhead*[3].

Este protocolo provê reserva de banda para aplicações com uma interface bem conhecida, semelhante à interface de *socket*. No lugar de um *socket*, é utilizado uma porta. As operações que podem ser realizadas numa porta são apresentadas na seção 2.3.

As aplicações utilizam a porta para requisitar banda passante e conexão com outras portas.

Como existem outros protocolos, que irão operar em paralelo, é necessário também disciplinar o envio de mensagens destes para não interferir na reserva de banda. O protocolo de reserva de banda foi então dividido em dois módulos funcionalmente distintos:

- **protocolo de controle de envio** faz o gerenciamento e acesso de reserva de banda passante (seção 2.4);
- **protocolo de conexão de portas** realiza conexão de portas e transferência de dados (seção 2.5).

Todo o envio de mensagem normal (não crítica de tempo) e de tempo real é feito através do protocolo de controle de envio (figura 2.4).

[2]Poderiam ter sido utilizados *raw-sockets*, entretanto somente o super-usuário tem acesso à este recursos
[3]O *overhead* do UDP advém do fato do mesmo computar o *checksum* por *software*

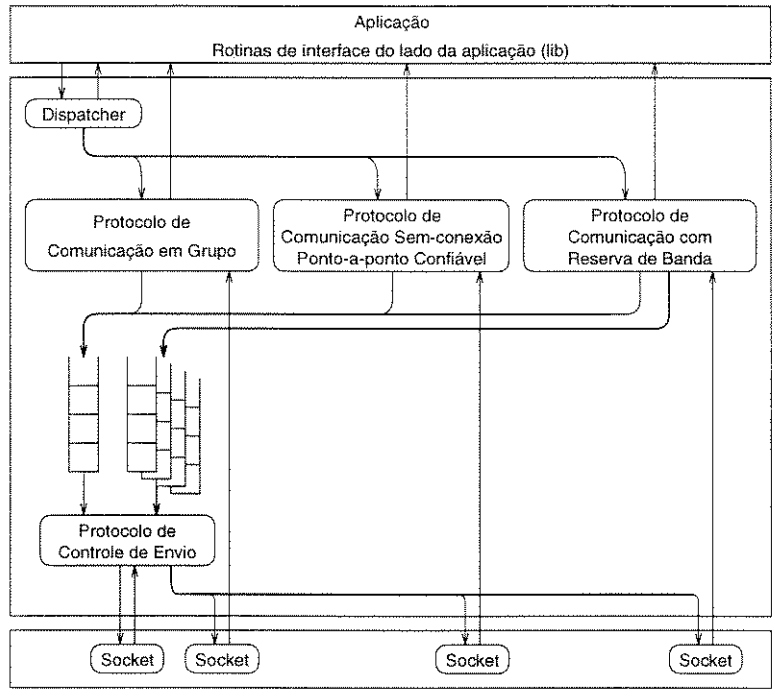

Figura 2.4: Arquitetura da plataforma de comunicação

O protocolo de reserva de banda provê um serviço "aproximadamente" síncrono. Existem vários aspectos no ambiente de implementação (rede, sistema operacional, programação da plataforma) que impede o prococolo de oferecer um serviço realmente síncrono. No capítulo 5 isto é discutido em detalhes.

Definições

A seguir são dadas algumas definições de constantes e variáveis que serão utilizadas na descrição do protocolo:

L (bytes) comprimento máximo das mensagens;
M_{setup} (s) tempo de processamento de cada mensagem antes de ser enviada;
H (s) tempo máximo de posse do *token* por um nodo;
N número máximo de nodos na sub-rede;
TR_{max} tempo máximo para o *token* circular por todos os nodos ($TR = H \times N$);
TR_{min} tempo mínimo para o *token* circular por todos os nodos (obtido por experimentação);
B (Bps) banda passante efetiva da sub-rede;
S (Bps) banda passante estaticamente alocada à cada nodo;
D (Bps) banda passante que pode ser dinamicamente alocada pelos nodos ($D = B - N \times S$);
P_i (s) periodicidade do envio de dados pela porta i;
C_i (bytes) tamanho dos dados enviados a cada período pela porta i;
M número de portas com reserva de banda no nodo h;
Q (Bps) soma de banda passante das portas do nodo ($Q = \sum_{i=1}^{M} \frac{C_i}{P_i}$);
R (Bps) reserva de banda feita ao nodo ($R = \frac{Q}{U(M)}$);
$U(n)$ taxa de utilização do escalonamento de taxa monotônica ($U(n) = n \times (2^{\frac{1}{n}} - 1)$).

O *Token*

O *token* cumpre duas funções neste protocolo. Ele é o elemento de informação que circula entre os nodos da sub-rede para realizar a alocação de banda. O nodo que possui o *token* pode verificar quanto de banda passante resta na sub-rede para alocação e, então, atender a requisição de banda de suas aplicações. O *token* também determina o acesso dos nodos ao meio de comunicação. O nodo que possui o *token* pode transmitir as suas mensagens.

A posse do *token* e a sua transferência de um nodo para outro formam um anel lógico (figura 2.5).

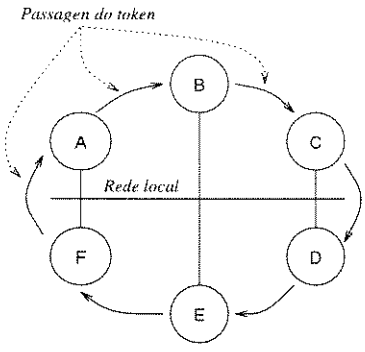

Figura 2.5: Anel lógico entre os nodos numa rede em barramento.

Um nodo A pode reter o *token* por até H segundos. Antes de expirar este tempo, ele deve enviar o *token* para o nodo seguinte B. A confirmação da recepção do *token* pelo nodo sucessor é verificada com a passagem do *token* para o nodo C, seguinte ao nodo B. A figura 2.6 apresenta o diagrama de tempo da passagem do *token* entre os nodos. Os retângulos correspondem ao envio de mensagens durante a posse do *token*. A parte hachuriada do retângulo é a passagem do *token* para o nodo especificado.

O nodo A deve aguardar pelo menos H segundos antes de considerar falha a passagem do *token*. Depois deste tempo (mais uma fração como tolerência), o nodo considera falha a passagem do *token*. Em vez de tentar passar novamente o *token* para B, o nodo vai passar o *token* diretamente para C (figura 2.7). Isto permite que o tempo de rotação do *token* não sofra atraso muito grande e, caso o nodo B tenha falhado indefinidamente, o nodo A não perderá tempo tentando passar o *token* para um nodo que não irá responder.

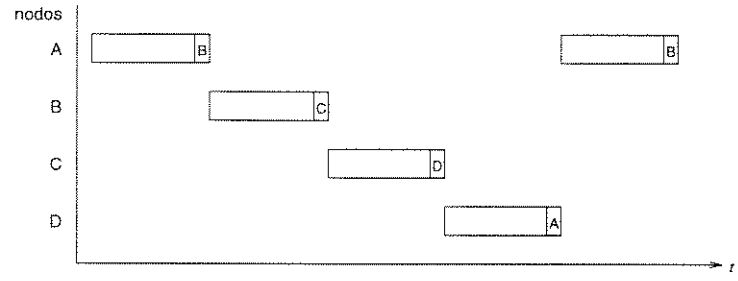

Figura 2.6: Diagrama de tempo da passagem e posse de *token* entre os nodos

Quando dois nodos consecutivos do anel B e C falham[4] e o nodo C tinha a posse do *token*, ocorre a perda do *token*. O mecanismo de falha de passagem de *token* descrito não deteta este tipo de erro pois o nodo A teve o reconhecimento da sua passagem de *token* com a passagem de *token* do nodo B para o C.

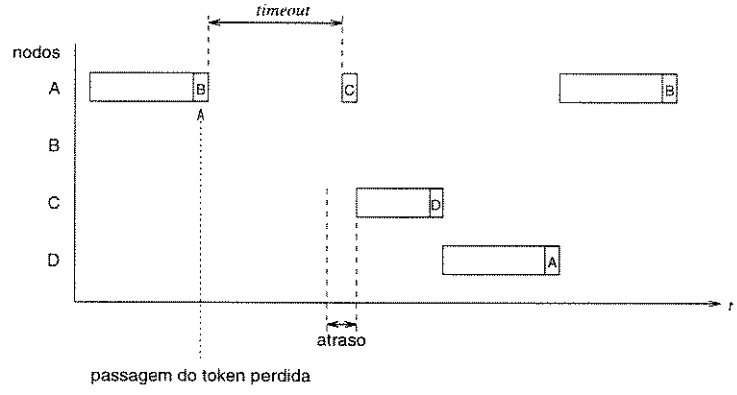

Figura 2.7: Falha na passagem do *token* para o nodo sucessor

Para contornar este caso, o mecanismo de deteção de falha de passagem do *token* foi estendido. O nodo que passa o *token* deve não apenas verificar a passagem do *token* para o seu sucessor como, também, verificar a passagem do *token* dos nodos seguintes. Ao detetar um silêncio na rede por um determinado período, ele conclui que o *token* foi perdido e reenvia a sua última versão do *token* para o nodo da última tentativa de passagem do *token*. O tempo para esta espera é diretamente proporcional à "distância" deste nodo para

[4]falha tipo *fail-stop*: do ponto de vista dos demais nodos da rede o nodo pára de funcionar, isto é, pára de receber e enviar mensagens.

o nodo que deveria receber o *token*. Este distância é o número de passagens necessárias para o *token* chegar ao nodo, sem a ocorrência de falhas. Por exemplo, seja um anel lógico composto pelos nodos A, B, C, D e E nesta ordem. Se o nodo A deteta a perda do *token* após C ter tentado o envio para o nodo D, e os nodos B, D e E falharam, ele espera $H + 3 \times tol$ (*tol* é um valor de tolerância para cobrir atrasos de envio de mensagens) após ter percebido a última passagem de *token* (figura 2.8).

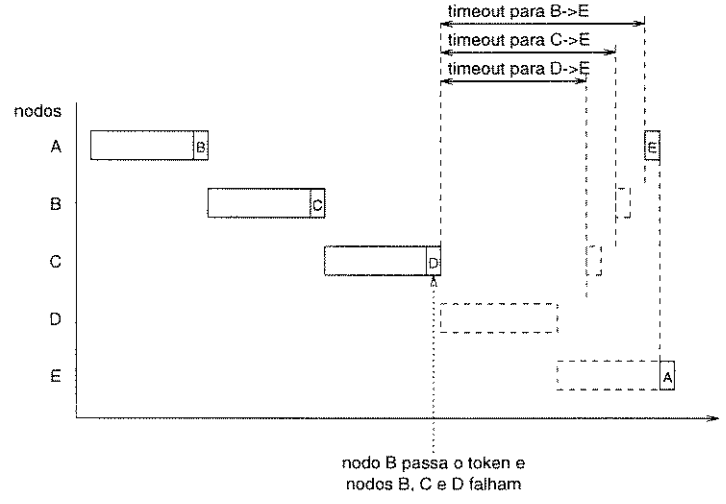

Figura 2.8: Perda do *token* e mecanismo de recuperação

Com este procedimento, pode ocorrer o atraso no giro do *token* pela rede. Se o nodo que recebeu o *token* perceber isto, ele deve reduzir o tempo de posse do *token* para passar o *token* no tempo esperado para o nodo seguinte. Estes dois procedimentos certamente irão causar o atraso no envio de mensagens das portas com reserva de banda, e assim causar a perda de seus *deadines*. Entretanto esta é uma degradação inevitável, uma vez que este controle é feito por *software*.

2.2.1 Retardo e Variação do Retardo

O tempo de retardo está diretamente relacionado com o tamanho dos dados da reserva (C). Se for possível transmitir todos os dados durante a posse do *token* (H) e os demais nodos não estiverem utilizando a rede (exceto passando o *token* adiante), o retardo será $G_{min} = G_M + TR_{min}$, onde $G_M = \frac{C}{B}$ que é o tempo de transmissão dos dados.

Por exemplo, numa rede com banda passante de 1 MBps (B=1MBps) e um nodo com uma reserva de 200 KBps (C=10KB e P=50ms), supor que o tempo mínimo para o *token* rodar o anel lógico seja de 10ms, o tempo mínimo para transmitir os dados da reserva será de 20ms ($G_M = \frac{10KB}{1MBps} = 10ms$, $G_{min} = 10ms + 10ms = 20ms$).

Se o tempo para transmitir os dados for maior que o tempo de posse do *token*, então é necessário dividir o envio dos dados em várias passagens do *token*. O tempo mínimo para o envio passa a ser então $G_{min} = \lceil \frac{G_M}{H} \rceil \times TR_{min} + G_M$.

Continuando o exemplo anterior, se tempo de posse do *token* for 10ms e a quantidade da reserva for aumentada para 20KB (400 KBps), o tempo mínimo será de 40ms ($G_M = \frac{20KB}{1MBps} = 20ms$, $G_{min} = \lceil \frac{20ms}{10ms} \rceil \times 10ms + 20ms = 40ms$).

Para saber o tempo máximo de retardo, deve-se considerar que os demais nodos irão utilizar a banda passante para o envio de tráfego assíncrono e que o restante da banda passante esteja alocado para reservas de maior prioridade.

O tempo máximo de retardo será o número mínimo de rotação do *token* para transmitir os dados vezes o tempo mínimo de rotação do *token*: $G_{max} = \frac{G_M + G_D}{H} \times TR_{min} + G_M + GD$, onde $G_D = \frac{D \times P - C}{B}$. Considera-se que toda a banda que não foi alocada pela reserva está alocada para reservas de maior prioridade. Simplificando a equação obtem-se: $G_{max} = \frac{D \times P \times H}{B} \times TR_{min} + \frac{D \times P}{B}$.

Supondo que a parte da banda total da rede alocada para reservas dinâmicas seja de 900KBps (D=900KBps), então o tempo máximo de retardo que a reserva do exemplo anterior pode sofrer é de 130ms ($G_D = \frac{900Kbps \times 50ms - 20ms}{1MBps} = 25ms$, $G_{max} = \frac{40ms + 25ms}{10ms} + 40ms + 25ms = 130ms$).

O tempo máximo de retardo é comparado com aquele fornecido com a requisição da reserva. Se o retardo exigido pela reserva for menor, então não será possível realizar a reserva.

O valor de retardo máximo será utilizado posteriormente para calcular o tamanho do *buffer* das mensagens recebidas.

2.2.2 Reserva

Quando uma aplicação solicita uma porta para transmissão de dados contínuos, ela informa a banda passante desejada e o retardo máximo admissível. Esta requisição é colocada numa fila de pedidos de banda passante. Quando o *token* chegar ao nodo, cada pedido desta fila é comparado com a banda passante restante para a alocação na rede. Se houver banda passante suficiente para o pedido, o *token* é atualizado e uma resposta de sucesso é enviada a aplicação. Se não houver banda passante suficiente, o pedido é postergado, isto é, ele será testado nas próximas passagens do *token*. Este teste não se faz indefinidamente. Um *timeout* é especificado para se obter a banda passante. Se o *timeout* for atingido, o pedido de reserva é retirado da fila.

Quando a aplicação não necessitar mais da reserva, ela solicita a sua retirada. Quando o *token* chegar ao nodo, ele é atualizado com a banda passante liberada da reserva.

2.2.3 Transferência de Dados

Para o envio de tráfego síncrono, isto é, as mensagens das reservas, é utilizado o escalonamento de taxa monotônica. O escalonamento é realizado a nível de reserva de porta, isto é, o protocolo decide qual a ordem de envio de suas mensagens. O tempo de envio de mensagens é o tempo de processamento (C) e o período de envio das mensagens é a periodicidade de execução das tarefas (P).

Quando uma reserva é feita, a banda passante é especificada em dois parâmetros: C e P. C é a quantidade de bytes que serão transmitidos periodicamente com intervalos de P segundos. Por exemplo, seja uma reserva para transmissão de vídeo. Cada quadro tem um máximo de 100KBytes e deve ser apresendado 20 vezes por segundo. Para esta reserva, os valores de C e P são, respectivamente, 100KBytes e 50ms. A relação C/P fornece a taxa de transmissão, neste caso de 200KBps.

Quando o nodo recebe o *token*, ele passa a utilizar a reserva de banda alocada. Entre as várias reservas feitas no nodo, o nodo começa transmitindo as de menor período (maior prioridade), conforme o escalonamento de taxa monotônica.

Para a transmissão de tráfego assíncrono, é utilizado o tempo remanescente da posse do *token* após a transmissão das mensagens síncronas. O envio de mensagens assíncronas é feito através de uma reserva especial, cujo período é igual a TR (a fim de que a cada giro do *token* sempre haja oportunidade de enviar dados) e quantidade de bytes igual a S/TR. Mesmo que o período da reserva para envio de mensagens assíncronas seja menor que as reservas normais, ela sempre será a de menor prioridade, isto é, a última a ser transmitida na posse do *token*.

2.2.4 Inicialização de um Nodo

Quando um nodo inicia o protocolo, ele precisa entrar no anel lógico formado pelos demais nodos. Para tanto, envia um pedido de inclusão na lista de nodos que participam da passagem de *token*. Este pedido é feito com uma mensagem de *broadcast* para que todos os nodos a recebam. O nodo com posse do *token* inclui o novo nodo no fim da lista. O novo nodo então espera até receber o *token*, configurando assim sua entrada no anel lógico.

2.2.5 Inicialização do *token*

Se o nodo enviou o pedido de inclusão do anel lógico e não recebeu o *token* depois de um determinado tempo, ele torna a re-enviar a mensagem até um tempo máximo para receber o *token*. Se neste tempo, após várias requisições do *token*, ele pode concluir que o anel ainda não foi formado e não existe nenhum *token* circulando.

Para criar o anel lógico, uma mensagem de inicialização de *token* é difundida (*broadcast*) e o nodo fica monitorando a rede por um determinado período de tempo. Isto é feito para garantir que nenhum outro nodo irá tentar inicializar o *token* também. Se o nodo receber uma outra mensagem de inicialização de *token* durante este período, ele cancela seu procedimento de inicialização e volta à fase de inicialização do nodo.

Ao chegar ao final do período e não tendo recebido nenhuma mensagem de inicialização do *token*, o nodo se considera de posse do *token* e como o único elemento do anel lógico.

2.3 Serviço

Este serviço opera sobre o protocolo de reserva de banda e provê a conexão entre portas. O serviço provido tem as seguintes características:

- com conexão;
- conexão iniciada pelo receptor [16, 17];
- transferência de dados por datagramas;
- multiponto;
- unidirecional;
- não-confiável;
- tempo real;
- assíncrono.

A porta é o ponto de comunicação fim[5]. Cada porta é identificada unicamente num nodo por um número inteiro, e na rede pelo par identificador de porta e endereço do nodo. Uma porta pode se conectar com outra porta para estabelecer um fluxo de dados, formando uma **conexão**. Existem dois tipos de portas: emissoras e receptoras.

A porta de envio de mensagens é responsável por solicitar a reserva de banda passante para enviar suas mensagens. Uma porta emissora pode existir mesmo sem que haja portas receptoras para ela. A porta emissora não tem conhecimento da conexão de portas receptoras. O controle de segurança deve ser feito pela aplicação (ou outro protocolo sobre este).

A porta de recepção de mensagens é responsável por estabelecer a conexão com a porta emissora (conexão iniciada pelo receptor). Uma porta receptora pode somente existir se a porta emissora, à qual quer se conectar, já existir. Para realizar a conexão, a aplicação deve ter conhecimento prévio do endereço do nodo e do identificador da porta emissora. Quando a porta receptora é aberta, este identificador é utilizado para proceder a conexão à porta emissora. Uma porta receptora não tem conhecimento de quais outras portas receptoras estão conectadas à porta emissora. Uma porta receptora pode se conectar a somente uma porta emissora.

[5] *end-point communication*

As operações que podem ser realizadas numa porta são:

- `open`: torna a porta disponível para transferência de dados;
- `close`: encerra a utilização da porta;
- `send`: envia uma mensagem (somente porta emissora);
- `receive`: recebe uma mensagem (somente porta receptora);
- `has_data`: verifica se uma porta tem mensagem para ser recebida (somente porta receptora);
- `flush`: retira todas as mensagens recebidas do *buffer* (somente porta receptora);
- `select`: permite aguardar até uma ou mais portas fiquem prontas para receber dados dentre uma lista de portas (somente porta receptora).

2.4 Protocolo de Controle de Envio

2.4.1 Estrutura de Dados

A figura 2.9 apresenta as estruturas de dados utilizadas no protocolo de reserva de banda.

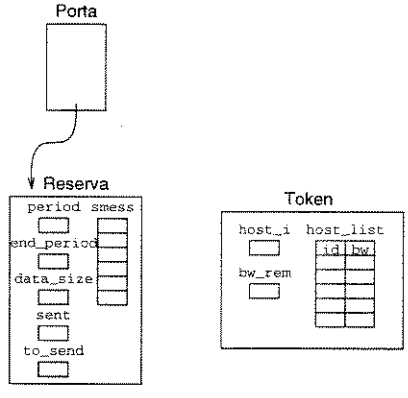

Figura 2.9: Estrutura de dados utilizada no protocolo de reserva de banda

Os campos da estrutura de reserva é composta pelos seguintes campos:

- `period`: duração de um período de envio de dados;

- `end_period`: fim do período corrente;
- `data_size`: quantidade de dados que podem ser enviados num período;
- `sent`: quantidade de dados enviados no corrente período;
- `to_send`: quantidade de dados a serem enviados (soma do tamanho das mensagems em `smess`);
- `smess`: fila de mensagens a serem enviadas.

Uma cópia do *token* é mantida no nodo a fim de possibilitar o seu reenvio. Os campos desta estrutura de dados são:

- `bw_rem`: banda passante restante para a alocação dinâmica;
- `host_i`: índice que aponta o atual detentor do *token* na lista de nodos `host_list`;
- `host_list`: lista de nodos que participam do anel lógico, onde cada elemento da lista é composto pelo identificador do nodo (`id`) e pela quantidade de banda passante reservada para ele (`bw`).

2.4.2 Mensagens

A figura 2.10 apresenta as mensagens utilizadas no protocolo. No apêndice B cada mensagem é descrita em detalhes.

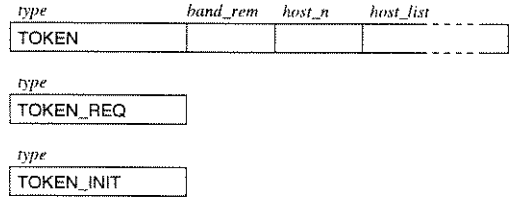

Figura 2.10: Mensagens do protocolo de reserva de banda passante.

A mensagem `TOKEN` é utilizada para a passagem do *token*. A mensagemm`TOKEN_REQ` é utilizada pelo nodo que ainda não faz parte do anel lógico para receber o *token*. A mensagem `TOKEN_INIT` é utilizada pelo primeiro nodo que inicia o anel lógico para garantir a criação de um único *token*.

2.4.3 Protocolo

Os nodos formam entre si um anel lógico com a passagem do *token*. Na figura 2.11 é apresentado o envio da mensagem `TOKEN` entre os nodos A, B e C.

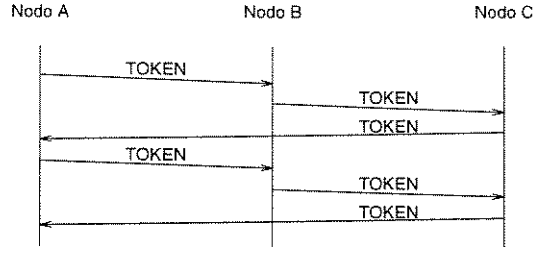

Figura 2.11: Passagem do *token*

Quando um nodo não faz parte do anel lógico, ao iniciar o protocolo, ele envia a mensagem `TOKEN_REQ` para todos os nodos da rede. O nodo que possui o *token* irá acrescentar este novo nodo na lista de nodos do *token*. A partir deste momento, o *token* pode ser passado ao novo nodo. Na figura 2.12 é apresentado a troca de mensagens para a requisição do *token*.

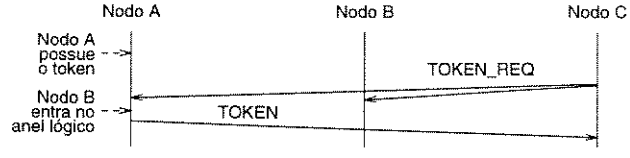

Figura 2.12: Requisição do *token*

Quando o anel lógico ainda não foi formado, isto é, nenhum nodo possui o *token*. O procedimento de requisição de *token* irá terminar por *timeout*. Este *timeout* indica que nenhum nodo na rede possui o *token*. Entretanto, mais de um nodo pode estar verificando a mesma situação. Para garantir que o *token* vai ser criado em apenas um nodo, todo nodo que pretende criar o *token* envia a mensagem `TOKEN_INIT`. Se depois de um tempo não houver nenhuma outra mensagem `TOKEN_INIT`, então o nodo assume que ele é o primeiro do anel lógico e portanto tem a posse do *token*. Se durante este tempo de espera o nodo receber outra mensagem `TOKEN_INIT`, ele verifica se o nodo emissor tem maior prioridade para inicialização do *token*. A prioridade, no caso, é o endereço de nodo mais alto. Se a prioridade for maior, o nodo volta ao procedimento de requisição do *token*. Senão, ele

reenvia sua mensagem `TOKEN_INIT` para ter certeza que o outro nodo vai abandonar a tentativa de iniciar o *token*.Na figura 2.13 é apresentado a inicialização do *token*.

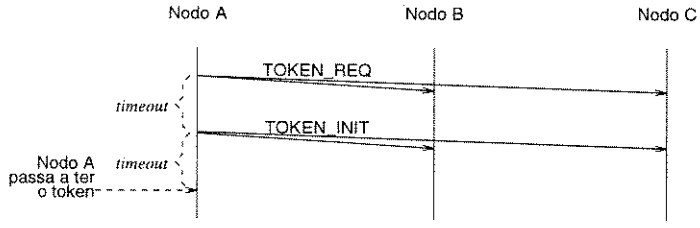

Figura 2.13: Inicialização do *token*

A figura 2.14 apresenta o diagrama de estados de operação de um nodo para protocolo descrito. No diagrama, uma transição é representada por um arco e uma reta. Sobre a reta está o evento que causou a transição e sob a reta está a ação a ser tomada.

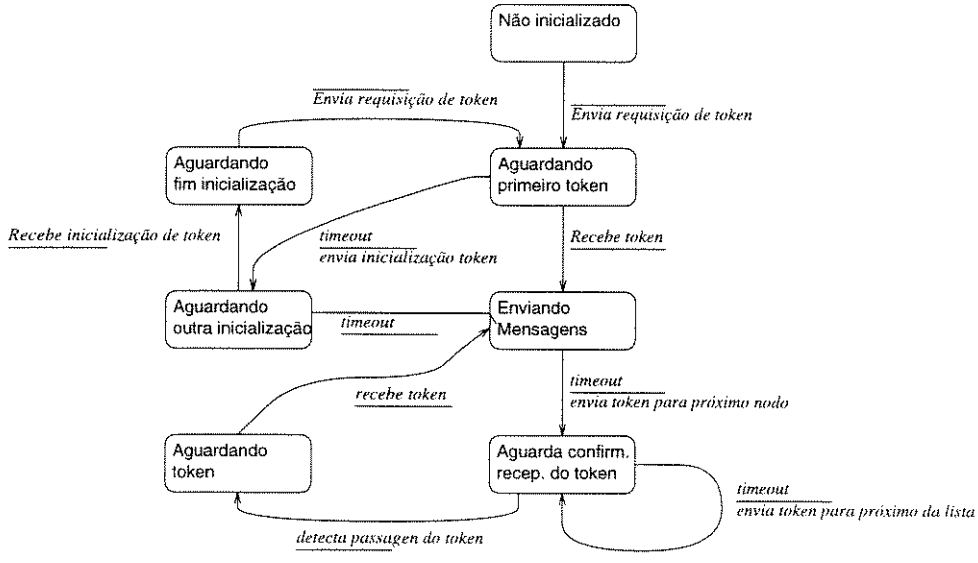

Figura 2.14: Diagrama de estados do protocolo

2.5 Protocolo de Conexão de Portas

2.5.1 Estrutura de Dados

Na figura 2.15 são apresentadas as estruturas de dados utilizadas no protocolo.

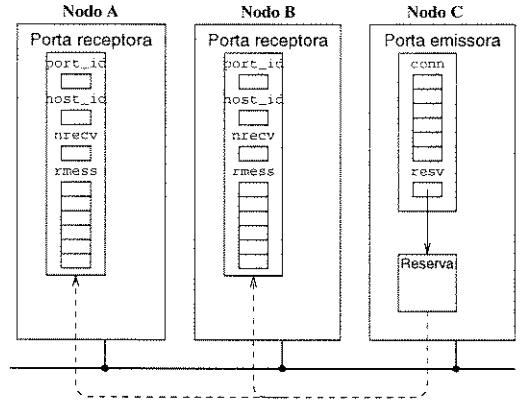

Figura 2.15: Estrutura de dados das portas receptoras e emissoras

Os campos de uma porta receptora são:

- `host_id`: identificador do nodo da porta emissora;
- `port_id`: identificador da porta emissora;
- `nrecv`: identificador da próxima mensagem de dados esperada;
- `rmess`: fila de mensagens recebidas.

Os campos de uma porta emissora são:

- `conn`: lista das portas receptoras conectadas (cada item da lista contém um par identificador de nodo e identificador de porta);
- `resv`: "ponteiro" para a estrutura de reserva de banda (seção 2.4.1).

2.5.2 Mensagens

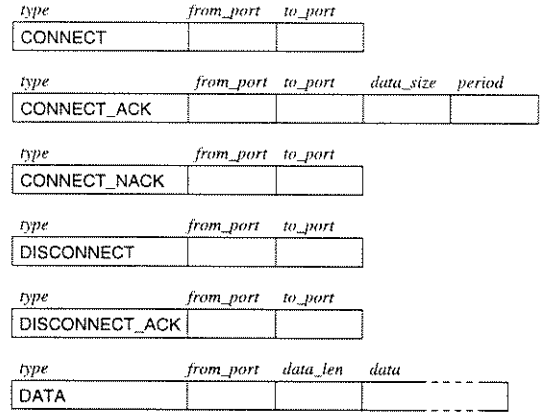

Figura 2.16: Mensagens do protocolo de conexão de portas

A figura 2.16 apresenta as mensagens utilizadas no protocolo. No apêndice B encontra-se a descrição detalhada de cada mensagem.

2.5.3 Protocolo

Alocação de *buffer*

É função de cada porta receptora a alocação do *buffer* de mensagens recebidas. O tamanho do *buffer* é calculado com base no retardo máximo, período e tamanho dos dados da porta emissora. Estes dados chegam à porta receptora através da mensagem `CONNECT_ACK`.

Como o tamanho dos dados (C) pode ser maior que o comprimento máximo de uma mensagem (L), deve haver a fragmentação dos dados em várias mensagens (função da aplicação). Considera-se que todas as mensagens, exceto a última, num período terão o tamanho máximo de mensagem.

Durante o tempo de retardo máximo (E_{max}) podem ocorrer vários ciclos de envio de mensagens[6]. Portanto, são no máximo $\frac{E_{max}}{P}$ ciclos. O tamanho do *buffer* é então o

[6] um ciclo é o período de envio de mensagens

número de mensagens recebidas no intervalo equivalente à duração máxima do retardo: $F = \frac{C}{L} \times \frac{E_{max}}{P}$.

Após a conexão ser estabelecida, e antes da aplicação começar a receber os dados, é necessário encher este *buffer*. Para determinar o quanto o *buffer* deve ser cheio, é necessário saber-se o tempo mínimo de retardo. Como valor deste tempo não pode ser obtido com precisão, adota-se como medida prática o preencimento total do *buffer*.

Assim, a aplicação somente começará a receber dados quando o *buffer* ficar completamente cheio. Consequentemente, pode vir a ocorrer *buffer overflow* nos instantes iniciais da existência da porta receptora. Mas este efeito, a perda das primeiras mensagens, pode ser rapidamente absorvido pela aplicação no inicío da recepção dos dados.

Conexão de Portas

Quando uma porta receptora é aberta, uma tentativa de conexão é feita. Para o estabelecimento uma conexão, a porta receptora envia uma mensagem `CONNECT_REQ` à porta emissora. A porta emissora responde com uma mensagem `CONNECT_ACK` e registra a porta receptora (identificador de porta e endereço do nodo). Se houver falhas no envio ou recepção da mensagem `CONNECT_REQ`, ou `CONNECT_ACK`, outras tentativas são feitas até o *timeout* especificado na abertura da porta. Se o *timeout* for atingido, um erro é retornado.

Quando a porta receptora for fechada, ela envia uma mensagem `DISCONNECT_REQ` para a porta emissora. A porta emissora responde com uma mensagem `DISCONNECT_ACK` e desregistra a porta receptora. Se houver falhas no envio ou recepção da mensagem `DISCONNECT_REQ` ou `DISCONNECT_ACK`, outras tentativas são feitas até o *timeout* de fechamento de porta. Se o *timeout* for atingido, a desconexão é encerrada e a porta fechada.

Na figura 2.17 são apresentados os estados e transições de uma porta receptora.

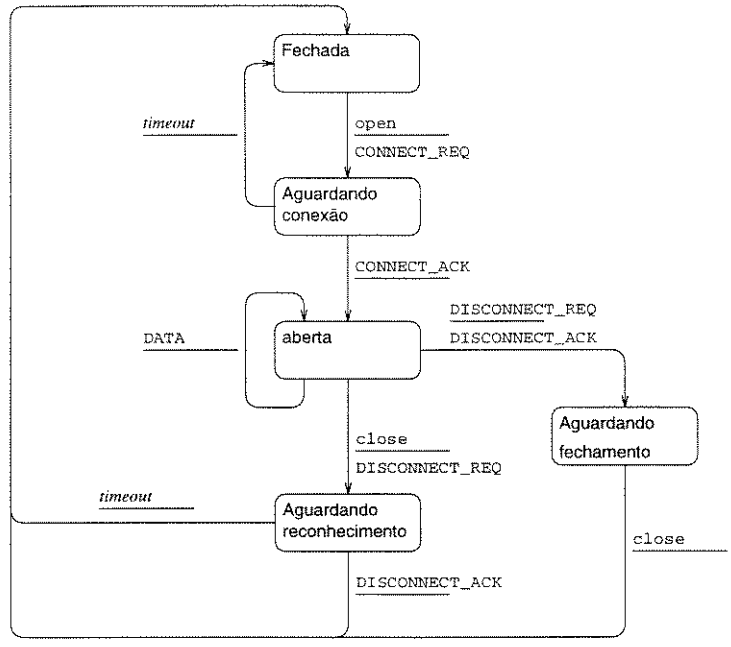

Figura 2.17: Diagrama de estados/transição de uma porta receptora

Quando a porta emissora for fechada, ela envia uma mensagem `DISCONNECT_REQ` para todas as portas receptoras registradas. Cada uma das portas receptoras responde com `DISCONNECT_ACK` e muda para um estado que só permite o fechamento da porta. Se houver falhas no envio ou recepção da mensagem `DISCONNECT_REQ` ou das mensagens `DISCONNECT_ACK`, outras tentativas são feitas até o *timeout* de fechamento de porta. Se o *timeout* for atingido, a desconexão é encerrada e a porta emissora fechada. As portas receptoras são mantidas abertas, mas qualquer operação diferente de fechar, retorma erro.

Na figura 2.18 são apresentados os estados e transições de uma porta emissora.

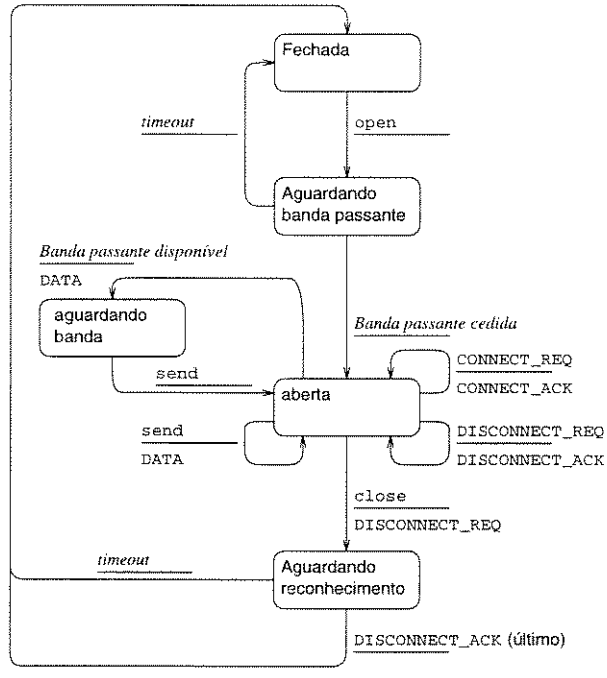

Figura 2.18: Diagrama de estados/transição de uma porta emissora

A figura 2.19 apresenta um exemplo de fluxo de mensagens para estabelecer conexão e transferência de dados. A figura 2.20 apresenta o encerramento de conexão por parte da porta emissora (a) e por parte da porta receptora (b).

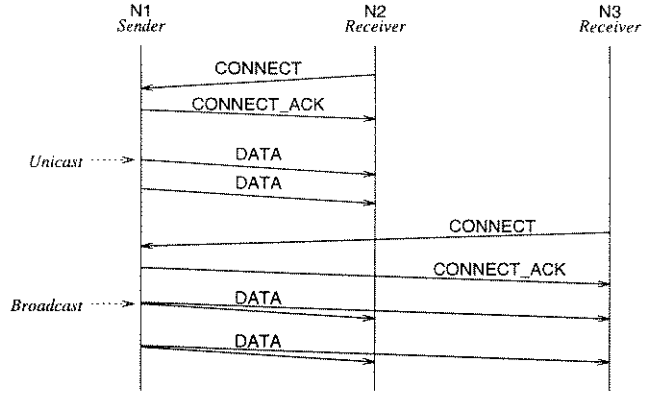

Figura 2.19: Troca de mensagens para estabelecimento de conexão e transferência de dados

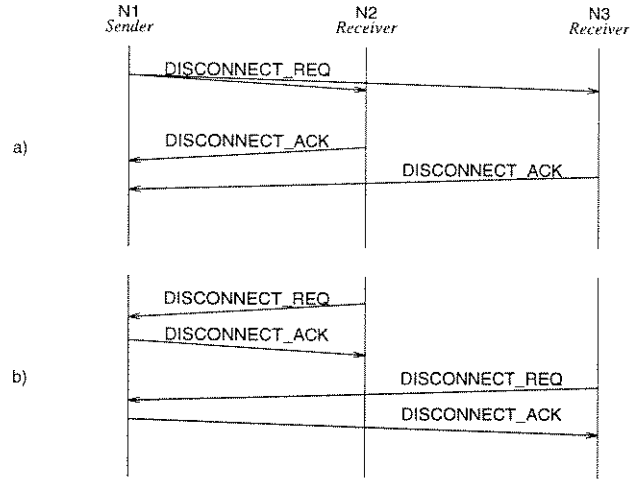

Figura 2.20: Troca de mensagens para encerramento de conexão. Iniciada pelo emissor a) e iniciada pelo receptor b)

Transferência de Dados

Uma vez estabelecida a conexão, pode-se passar a transmitir os dados. A mensagem `DATA` é utilizada para o envio de dados para a porta receptora (ou portas, se mais de uma estiver conectada à porta emissora).

Não é utilizado nenhum mecanismo de controle de erro. Se alguma mensagem for perdida, a aplicação deve ser capaz de detetar (o protocolo não informa a falta de alguma mensagem) e recuperar o erro. O protocolo também não realiza a fragmentação/remontagem de mensagens. Normalmente dados de mídia de vídeo são longos (p.e., 100K por quadro) e devem ser transmitidos como uma unidade. A aplicação deve fragmentar e remontar suas mensagens longas.

Tão logo uma mensagem `DATA` for recebida por um nodo, ela é passada à aplicação, tal como proposto em [18], para simplificar a implementação. A sincronização das mídias fica, portanto, a cargo da aplicação.

Capítulo 3

Comunicação de Grupo

3.1 Introdução

Vários protocolos de *reliable multicast* (e *broadcast*) têm sido apresentados na literatura. Cada um possui diferentes considerações sobre a rede, tipos de falhas, conceito de confiabilidade, etc. Em [19] são apresentados vários protocolos de *reliable broadcast*. Alguns destes protocolos são apenas teóricos. Outros foram realmente implementados, como os descritos em [20, 21, 22]. Entretanto, nestas publicações, nenhum destes protocolos é descrito em detalhes. Muitas questões relevantes são abstraídas, tal como a reabilitação do estado do protocolo em caso de falha de nodo.

Este trabalho não vem apresentar um protocolo inovador, mas uma abordagem mais pragmática, levando em conta aspectos particulares de redes existentes. O protocolo é basicamente o mesmo apresentado por Kaashoek em [22]. Os seguintes aspectos foram considerados no projeto deste protocolo:

- taxas de erros muito baixas em redes locais: isto permite que o protocolo funcione de uma maneira otimista, isto é, considera que raramente haja perda de pacotes e ocorram retransmissões, ou que o protocolo entre em fase de reforma;
- grande quantidade de memória disponível: permite altas taxas de "bufferização" e que não ocorra perda de mensagens;
- sub-redes com tecnologia que permite *broadcasting* (anel ou barramento);

- implementação sobre um sistema operacional com recursos de processos ou *threads* (UNIX ou OS/2) e protocolo IP (internet).

3.1.1 Definição de *Reliable Multicast*

Existem na literatura diversas definições para o termo *reliable*. As definições variam, principalmente, no modelo do ambiente em que o protocolo deve operar. Entretanto, o seguinte princípio está presente em todas as definições:

> A difusão de uma mensagem k para um grupo X de nodos é considerada confiável (*reliable*) se todos (ou um subconjunto de X) recebem k corretamente e a passem às suas aplicações ou, havendo alguma falha, nenhum nodo passe k às aplicações, dentro de um limite de tempo pré-estabelecido.

Isto significa que deve existir um consenso para a recepção da mensagem nos nodos destinatários. Em não havendo consenso (isto é, algum nodo não acusar a recepção da mensagem), nenhuma aplicação pode receber a mensagem.

Quanto a *multicast*, existe um consenso quanto ao termo que pode ser definido como:

> Seja X o conjunto de todos os nodos (dentro de uma mesma subrede ou entre várias subredes interconectadas), um *multicast* é a difusão de uma mensagem para um subconjunto X_a de X, tal que somente os nodos em X_a recebam a mensagem e a passem às suas aplicações. Cada um destes subconjuntos deve ser identificado de alguma forma e normalmente são designados por **grupos**. Pode haver vários subconjuntos de X e estes podem se intersectar, logo cada nodo pode participar de vários grupos (ou nenhum).

Assim, podemos definir que um *reliable multicast* é a difusão confiável de mensagens para um grupo de nodos, tal que ou todos (ou parte dos) nodos do grupo passem a mensagem para as suas aplicações ou nenhum o faça.

Também muito utilizado é o conceito de ordenação, como agregado da definição de *reliable*:

> Todos as aplicações que recebem mensagens de um grupo, devem recebê-las na mesma ordem em todos os nodos.

Entretanto, conforme é apresentado a seguir, este conceito não implica em difusão confiável e pode ser relaxado conforme a semântica das aplicações.

3.2 Questões de Projeto

Devido a diferentes definições de *reliable multicast* existentes, os protocolos que a implementam podem ser completamente diferentes entre si. A seguir são apresentadas 6 questões de projeto em comunicação de grupo que estes protocolos optaram, de uma forma ou outra, no seu projeto. Estas questões foram definidas em [22] e são:

- endereçamento;
- confiabilidade;
- ordenação;
- semântica de entrega;
- semântica de resposta;
- estrutura de grupos.

3.2.1 Endereçamento

Para se ter uma difusão de mensagens para determinados nodos, é necessário que se especifique, de alguma forma, os destinatários. O método mais simples é especificar o endereço de cada nodo, ou seja, enviar uma mensagem para cada um. Entretanto, este método requer um número de mensagens proporcional ao número de destinatários. Um segundo método, mais econômico, é enviar mensagens com endereços de grupo como destino. Os nodos devem estar aptos a reconhecer um endereço de grupo. Existem ainda dois outros métodos, que são pouco utilizados: o endereçamento pela fonte e endereçamento funcional. Utilizando endereçamento pela fonte, o nodo que recebeu a mensagem verifica se o emissor pertence ao seu grupo. No endereçamento funcional, o nodo verifica se uma função retorna verdadeiro quando submetida à mensagem.

Neste trabalho a forma de endereçamento de grupos escolhida foi **identificador de grupo**. Não se trata nem de um endereço de rede fictício nem de um identificador de porta como em [22]. A razão para esta escolha é simplicidade de implementação, e a utilização de

apenas um identificador na camada de aplicação é mais simples. A difusão da mensagem na subrede é feita por *broadcast* disponível no protocolo IP.

3.2.2 Confiabilidade

Este critério diz respeito à capacidade de recuperação do protocolo frente a falhas. Há vários níveis de confiabilidade que um protocolo pode oferecer. A seguir são apresentados alguns tipos tolerância a falhas.

Nenhuma confiabilidade, é o protocolo mais simples que não se recupera de nenhum tipo de falha.

Tolerância a falhas de mensagens, quando há perda de mensagens (por *buffer overflow*, mensagem corrompida, mensagem perdida, etc.). A implementação é simples, pois exige apenas a retransmissão das mensagens faltantes.

Tolerância a falhas de nodos, quando algum nodo deixa de atender as mensagens do protocolo (desconexão da rede, mal funcionamento da placa de rede, nodo desligado, etc.). A implementação é mais complexa, pois exige que os demais nodos sobreviventes assumam as funções do nodo falho, mesmo com alguma degradação no desempenho.

Tolerância a falhas maliciosas, quando mensagens de controle do protocolo incorretas são enviadas por algum nodo (erro de programação do protocolo, falha no sistema operacional, etc.). A implementação é extremamente complexa, pois os demais nodos devem detectar e ignorar a mensagem e, se possível, restaurar o funcionamento correto no nodo falho. Mesmo assim, pode-se não atender a todos os casos de falhas maliciosas.

Tolerância a falhas de partição, quando a rede é interrompida em algum ponto, formando duas partições de nodos, tal que não haja comunicação entre as duas partições, ainda que os nodos continuem funcionando corretamente. Uma forma de tratar o problema é eleger uma partição para continuar operando normalmente e bloquear a outra até que a partição seja desfeita. A eleição da partição pode ser feita por votação e maioria simples, mas é necessário que o protocolo mantenha uma lista de nodos participante para determinar o quorum para a votação. A implementação é complexa, pois impõe a manutenção de listas de atividade dos nodos participantes do grupo em cada nodo durante toda a operação do

protocolo.

Neste trabalho, foi escolhida a tolerância a falhas de mensagens e falhas de nodos. Em caso de falha de um nodo, este não se recupera no estado em que se encontrava exatamente quando falhou, mas passa a funcionar como um nodo recém chegado ao grupo. A falha de um nodo é transparente ao nodos sobreviventes, que não podem verificar se uma mensagem realmente chegou ou não ao nodo falho.

Tolerância a falhas maliciosas e tolerância a falhas de partição não são suportadas, devido à grande dificuldade de implementação.

3.2.3 Ordenação de Mensagens

A semântica para ordenação de mensagens pode ser uma das seguintes:

- nenhuma ordenação;
- ordenação FIFO;
- ordenação causal;
- ordenação total.

Na semântica de **nenhuma ordenação**, mensagens enviadas pelos nodos podem chegar em ordens diferentes nos nodos receptores. Esta forma é extremamente simples de implementar.

Na semântica de **ordenação FIFO**, mensagens enviadas de um nodo são recebidas na mesma ordem em todos os nodos receptores. Mas a seqüência de mensagens enviadas de vários nodos pode ser intercalada, de tal forma que a seqüência de recepção pode diferir nos nodos receptores. Esta forma é simples de implementar, pois a ordenação de mensagens é controlada por cada nodo receptor, não necessitando de uma decisão global para o sequenciamento das mensagens, isto é, o controle pode ser totalmente descentralizado.

A semântica de **ordenação causal** é a ordenação FIFO mais a restrição de causalidade entre mensagens recebidas e depois enviadas pelo nodo. Por exemplo, se um nodo recebe uma mensagem A, depois envia B, todos os nodos deverão receber A e depois B. A seqüência de chegada das mensagens ainda pode diferir nos nodos receptores. Esta forma é um pouco mais complexa que a ordenação FIFO, pois exige que os nodos tenham uma

tabela de dependência entre as mensagens. Ainda assim o controle pode ser totalmente descentralizado.

Na semântica de **ordenação total**, todos os nodos recebem todas as mensagens na mesma ordem. É a forma mais restritiva de ordenação e também a mais complexa. Esta forma exige que, em algum instante, algum nodo decida qual será a ordem das mensagens. Nem mesmo em protocolos considerados totalmente distribuídos, a decisão pode ser tomada distribuída, como é o caso do protocolo em [20].

Neste trabalho, foi escolhida a ordenação de mensagens total em cada grupo. Para implementar esta forma de ordenação, foi escolhida a arquitetura de nodo seqüenciador para cada grupo, tal como é descrito na seção 3.3.1.

3.2.4 Semântica de Entrega

Um nodo participante do grupo não pode passar uma mensagem recebida imediatamente para suas aplicações por desconhecer se os demais participantes também a receberam. É necessário, então, que todos os nodos receptores notifiquem ao nodo emissor a recepção da mensagem. Baseado nestas notificações, o nodo emissor determina se houve ou não sucesso no envio da mensagem. Em havendo sucesso, o nodo emissor comunica que a mensagem pode ser entregue às aplicações[1]. Para se determinar o sucesso do envio da mensagem, as seguintes formas podem ser empregadas:

- entregue a k nodos;
- entregue a um quorum;
- entrega atômica.

Na semântica de **entregue a k nodos**, uma mensagem é considerada entregue com sucesso quando pelo menos k nodos acusam o seu recebimento. Esta forma é simples de implementar, pois exige apenas um contador de reconhecimento que, quando atinge o valor k, acusa o sucesso do envio da mensagem.

Na semântica **entregue a um quorum**, uma mensagem é considerada entregue com sucesso quando a maioria simples dos nodos participantes de um grupo recebem a

[1] O envio dos reconhecimentos de mensagens e confirmação de sucesso podem ser feitos "de carona" (*piggybacking*) em mensagens trocadas entre os nodos.

mensagem. A implementação é mais complexa, pois exige que o nodo emissor saiba quais nodos estão participando do grupo, para determinar o número mínimo de nodos para a formação do quorum.

Na semântica **entrega atômica**, uma mensagem é considerada entregue quando **todos** os nodos participantes do grupo a recebem. Esta é a semântica considerada ideal. Entretanto, a implementação é difícil, pois é necessário que se determine o estado de cada nodo participante, isto é, se o nodo falhou, se a mensagem com o reconhecimento foi perdida, etc.

Estas formas de garantir atomicidade na entrega de mensagens implicam numa inevitável degradação de desempenho. De certa forma, o protocolo funciona como uma confirmação em duas fases. Na primeira fase o nodo emissor envia a mensagem a todos os nodos. Todos os nodos receptores acusam recebimento da mensagem. Se o nodo emissor recebeu todas os reconhecimentos (ou um número k ou um quorum), ele envia uma confirmação para todos os nodos receptores, que então passam a mensagem para as aplicações. Durante todo o período em que é realizado esta comunicação, a aplicação que está enviando a mensagem deve ficar bloqueada, caso necessite saber o resultado do seu *multicast*.

Uma forma de melhorar o desempenho do *multicast*, tanto diminuindo o atraso entre o momento da solicitação do *multicast* até a recepção da mensagem pelas aplicações, é alterando o conceito de *reliable*.

Considera-se que toda mensagem enviada é recebida por todos os nodos corretamente. Sendo assim, não é necessário que os nodos receptores esperem pela confirmação do nodo emissor da recepção de todos os nodos, pois se o nodo receptor a recebeu é porque todos os demais também a receberam. O nodo emissor por sua vez, sabendo que todos os nodos irão receber a mensagem corretamente, não necessita esperar a confirmação da recepção e também reter a aplicação emissora para confirmar o envio da mensagem.

Se ocorrer uma falha de mensagem, e algum nodo não receber o *multicast*, o nodo irá perceber isto pela próxima mensagem recebida e solicitar retransmissão da mensagem perdida.

Se o nodo receber a mensagem e depois falhar, o protocolo operou corretamente. Se o nodo falhar antes de receber a mensagem e depois dela ser enviada pelo emissor, considera-

se que o nodo falhou antes do envio da mensagem. Assim pode-se dizer que o protocolo funcionou corretamente e obedeceu à semântica de atomicidade.

Desta forma, acelera-se muito a comunicação de mensagens, pois o atraso na entrega da mensagem à aplicação é mínimo. E não há necessidade de bloquear-se inutilmente a aplicação emissora.

Entretanto, desta forma não é possível determinar se algum nodo falhou durante o envio de uma mensagem e informar à aplicação emissora. Nas aplicações, críticas onde é necessário saber se todos os nodos (falhos ou não) receberam a mensagem, esta verificação pode ser feita na aplicação, com a implementação de um protocolo simplificado de terminação.

3.2.5 Semântica de Resposta

Semântica de resposta diz respeito ao tipo de resposta esperada pelo nodo emissor de uma mensagem. Há quatro formas:

- nenhuma resposta;
- uma única resposta;
- várias respostas;
- todas respostas.

Nenhuma resposta é a mais empregada, quando o envio de uma mensagem não implica numa resposta à cada mensagem.

Uma única resposta é empregada quando a mensagem contém a requisição de um serviço que tem uma resposta única e mesmo que todos os nodos do grupo respondam, a primeira a chegar é suficiente.

Várias respostas é empregado quando mais que uma resposta é necessária para atender uma mensagem, mas não se pode esperar por todas as respostas (restrição de tempo, por exemplo.

Todas respostas é empregado quando um serviço crítico está sendo efetuado e é necessário ter certeza que todos receberam a mensagem.

A forma com nenhuma resposta é a mais simples de implementar. As demais formas que empregam uma ou mais respostas podem ser vistas como semântica RPC (*Remote Procedure Call*) e, consequentemente são mais complexas de implementar, pois exigem a associação da(s) resposta(s) com a mensagem original.

Neste trabalho foi escolhido a semântica de nenhuma resposta, por ser a mais fácil de implementar. Qualquer aplicação que necessitar de resposta pode implementar um mecanismo que forneça as formas de resposta única, várias respostas e todas as respostas com alguma facilidade.

3.2.6 Estrutura de Grupos

Grupos podem ser:

- abertos ou fechados;
- dinâmicos ou estáticos.

Grupos abertos permitem que nodos que não são membros do grupo possam também enviar mensagens ao grupo. Em **grupos fechados**, somente os membros podem trocar mensagens entre si. Grupos fechados são mais simples de implementar que os abertos, pois não necessitam controlar as mensagens vindas de outros nodos que não são membros do grupo. O controle de falhas de nodos também é mais fácil de ser efetuado, pois todos os nodos são conhecidos e constantes.

Grupos dinâmicos permitem que novos membros se unam e abandonem o grupo durante a sua existência. Em **grupos estáticos**, os mesmos membros pertencem ao grupo, durante toda a existência deste. Grupos estáticos são mais simples de implementar, pois as adesões e abandonos do grupo ocorrem apenas na criação e destruição do grupo, respectivamente, e todos os membros recebem todas mensagens enviadas ao grupo. Com grupos dinâmicos há maior complexidade envolvida, pois é necessário controlar cada nodo quanto as mensagens que ele necessita receber. O controle de falhas de nodos torna-se complexo com grupos dinâmicos, pois é difícil estabelecer quais nodos devem participar da reestruturação do grupo.

Neste trabalho, foi escolhido grupos fechados e dinâmicos. A razão para escolher grupos fechados foi a maior simplicidade de implementação. Como a adesão a um grupo é simples e rápida, qualquer aplicação que deseje enviar uma mensagem ao grupo, mas não participar mais ativamente, pode juntar-se ao grupo por um curto intervalo de tempo para o envio de uma mensagem.

A razão para escolha de grupos dinâmicos foi viabilizar o funcionamento de aplicações assíncronas, que não podem se unir ao grupo quando este for criado nem permanecer funcionando até a destruição do grupo.

3.3 Arquitetura

Com as decisões de projeto feitas para o protocolo, configurou-se a arquitetura descrita a seguir.

3.3.1 Nodos Sequenciadores e Clientes

Para realizar um *multicast*, todo nodo deve enviar primeiro a mensagem para um **nodo sequenciador**. Este nodo, então, envia a mensagem para os **nodos clientes**, que são aqueles que enviam e recebem mensagens (inclusive o próprio nodo sequenciador pode ser um nodo cliente).

Há somente um nodo sequenciador por grupo. A escolha do nodo sequenciador é arbitrária e qualquer nodo está apto para assumir esta função. Um nodo pode ser sequenciador para vários grupos.

A criação de grupos é feita por demanda. Quando um nodo necessitar enviar mensagem para um grupo, e ele não existir, o grupo é automaticamente criado por algum nodo que assuma a função de sequenciador. O nodo sequenciador mantém uma lista de nodos clientes do grupo. Cada nodo que estiver interessado em receber/enviar mensagens do/para o grupo, se increve nesta lista. Quando esta lista permanecer vazia por um determinado período, o nodo sequenciador destrói as informações sobre o grupo.

3.3.2 Portas, Conexões e *Buffers*

Cada nodo pode ter mais de uma aplicação enviando e recebendo mensagens de um grupo. Para uma aplicação realizar a comunicação com um determinado grupo, ela utiliza uma **porta**. A porta deve estabelecer comunicação com o grupo através de uma conexão. Num mesmo nodo pode haver várias portas conectadas a um mesmo grupo.

Quando uma aplicação envia uma mensagem através de uma porta, todas as portas conectadas ao grupo no nodo e em outros nodos receberão a mensagem, exceto a porta pela qual foi enviada a mensagem.

Todas as portas conectadas ao mesmo grupo recebem as mesmas mensagens e na mesma ordem. Entretando, as aplicações não recebem necessariamente as mensagens através da porta no mesmo instante. Devido à natureza assíncrona das aplicações, é necessário armazenar as mensagens recebidas, para que cada aplicação possa fazer a recepção da mensagem em tempos diferentes. Um *buffer* é compartilhado entre todas as portas conectadas. Cada porta tem um apontador para a sua próxima mensagem a ser recebida dentro do *buffer*. As mensagens recebidas pelo grupo são armazenadas até todas as portas a receberem. Na figura 3.1 as portas P_1 e P_2 têm mensagens disponíveis no *buffer* para serem lidas, ao passo que a porta P_3 deve esperar até a mensagem 12 chegar ao nodo. As mensagens 1 a 4 já podem ser descartadas pois todas as portas conectas já as receberam.

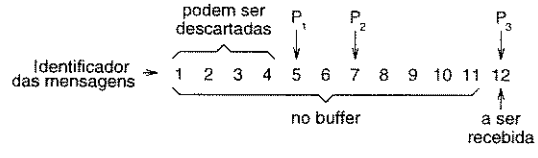

Figura 3.1: Mensagens que serão recebidas pelos processos

3.3.3 Conexões Dinâmicas

Durante a existência de um grupo, as portas podem se conectar e desconectar do grupo. Quando uma porta se conecta a um grupo já em andamento, a porta terá como primeira mensagem a ser recebida a próxima mensagem que o grupo vier a receber.

Não é possível estabelecer conexões dinâmicas que recebam todas as mensagens do

grupo desde a sua criação. Se assim fosse, o sequenciador necessitaria armazenar todas as mensagens do grupo desde a sua criação para que, quando uma eventual conexão surgisse, a mesma pudesse receber todas as mensagens enviadas ao grupo. Como memória é um recurso limitado e o número de mensagens enviadas a um grupo pode ser elevado, não seria possível armazená-las em sua totalidade.

Mesmo havendo mensagens no *buffer* de mensagens recebidas, elas também não serão passadas ao processo recém-conectado ao grupo. A razão para este comportamento é que, se a conexão realizada for conseqüência de um mensagem recebida do grupo, a nova conexão deve somente receber as mensagens posteriores a recepção da mensagem. Na figura 3.2 a porta recém-conectada P_4 terá como primeira mensagem a número 13, que o nodo está para receber.

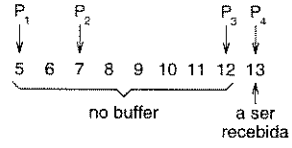

Figura 3.2: Nova conexão ao grupo

3.4 Estrutura de dados

Cada nodo cliente mantém uma estrutura de dados para o controle de envio e recepção de mensagens de cada grupo. O nodo sequenciador mantém uma estrutura de dados para cada nodo cliente conectado a ele e outras informações para controle de mensagens enviadas. A figura 3.3 apresenta a estrutura de dados completa para um grupo, com 3 nodos clientes e um sequenciador.

A estrutura de dados para o nodo cliente é composta pelos seguintes campos:

- `smess`: fila de mensagens locais enviadas ao nodo sequenciador;
- `nlocal`: identificador da próxima mensagem local que o nodo irá enviar ao sequenciador;
- `llocal`: identificador da próxima mensagm local a ser reconhecida pelo sequenciador;
- `clocal`: identificador da próxima mensagem local a ser liberada pelo sequenciador;
- `rmess`: fila de mensagens globais recebidas do nodo sequenciador;

- `nglobal`: identificador da próxima mensagem global que o nodo espera receber do sequenciador;
- `cglobal`: identificador da próxima mensagem global a ser liberada pelas portas locais;
- `lglobal`: identificador da próxima mensagem global confirmada por todos os nodos clientes;
- `hglobal`: identificador da mensagem mais recente recebida pelo grupo.

A estrutura de dados para o nodo sequenciador é composta pelos seguintes campos:

- `smess`: fila de mensagens enviadas ao nodos clientes;
- `nglobal`: identificador da próxima mensagem global a ser enviada;
- `lglobal`: identificador da próxima mensagem global a ser reconhecida por todos os nodos clientes;
- `cglobal`: identificador da próxima mensagem global a ser liberada por todos os nodos clientes

Para cada nodo cliente conectado ao grupo existe a seguinte estrutura de dados:

- `rmess`: fila de mensagens locais recebidas do nodo cliente;
- `lglobal`: identificador da próxima mensagem global a ser reconhecida pelo nodo cliente;
- `cglobal`: identificador da próxima mensagem global a ser liberada pelo nodo cliente;
- `nlocal`: identificador da próxima mensagem local a ser recebida do nodo cliente;
- `hlocal`: identificador da mensagem mais recente recebida do nodo cliente.

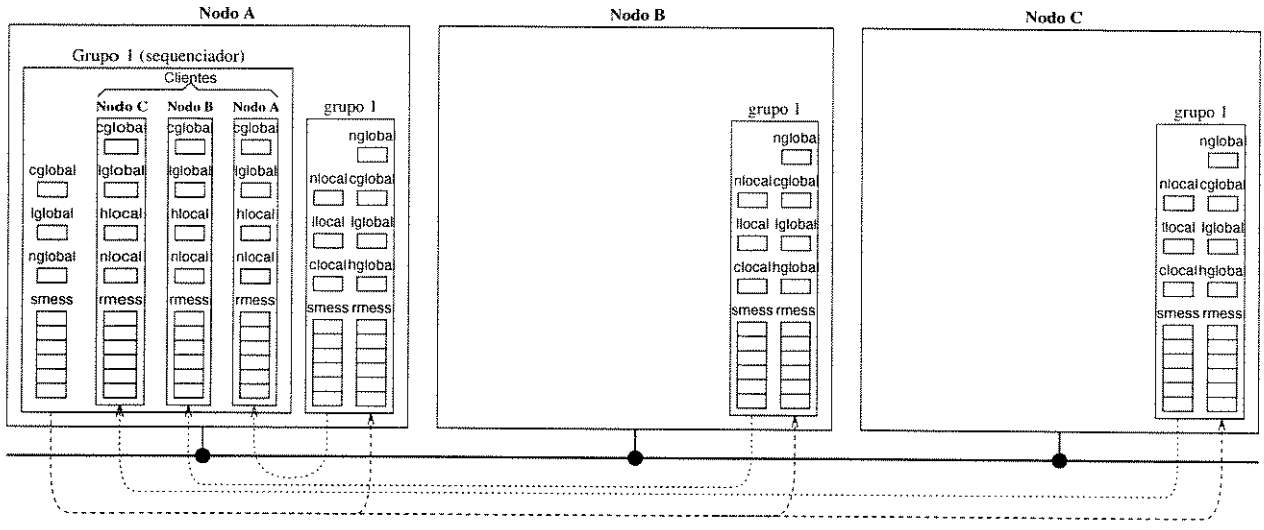

Figura 3.3: Estrutura de dados para controle das mensagens

3.5 Serviço

O serviço provido tem as seguintes características:

- conexão;
- transferência de dados por datagramas;
- multiponto;
- confiável;
- assíncrono.

O serviço é acessado através do ponto de acesso chamado **porta**. Cada porta pode ser associada a um grupo para envio e recepção de mensagens. Quando a porta é criada, um **identificador de porta** é associado automaticamente. Uma porta pode conectar-se somente a um grupo.

Cada porta pode ser conectada de três modos:

- recepção e envio de mensagens;
- somente recepção de mensagens;

- somente envio de mensagens.

Quando uma porta for conectada para o envio de dados, o envio de uma mensagem pela porta não é recebido de volta. Isto é, o envio de uma mensagem é recebido por todas as portas conectadas ao grupo, exceto pela porta emissora.

Através da porta podem ser acessados os serviços de associação a grupos, transferência de dados e manipulação do *buffer* de mensagens recebidas. As primitivas para acessar estes serviços são:

- `open` associa a porta a um grupo, tornando-a disponível para envio e recepção de dados;
- `close` encerra a utilização da porta;
- `send` envia dados para o grupo;
- `receive` recebe dados do grupo;
- `select` permite aguardar até que uma ou mais portas fiquem prontas para receber dados dentre uma lista de portas;
- `has_data` verifica se a porta tem dados para receber;
- `flush` retira todas os dados recebidos do *buffer*.

As primitivas `send`, `receive`, e `select` permitem que seja estipulado um tempo máximo de espera (*timeout*). A utilização das primitivas é descrita em detalhes no apêndice A.

3.6 Mensagens

A figura 3.4 apresenta os formatos de mensagens do protocolo. No apêndice B cada mensagem é descrita em detalhes.

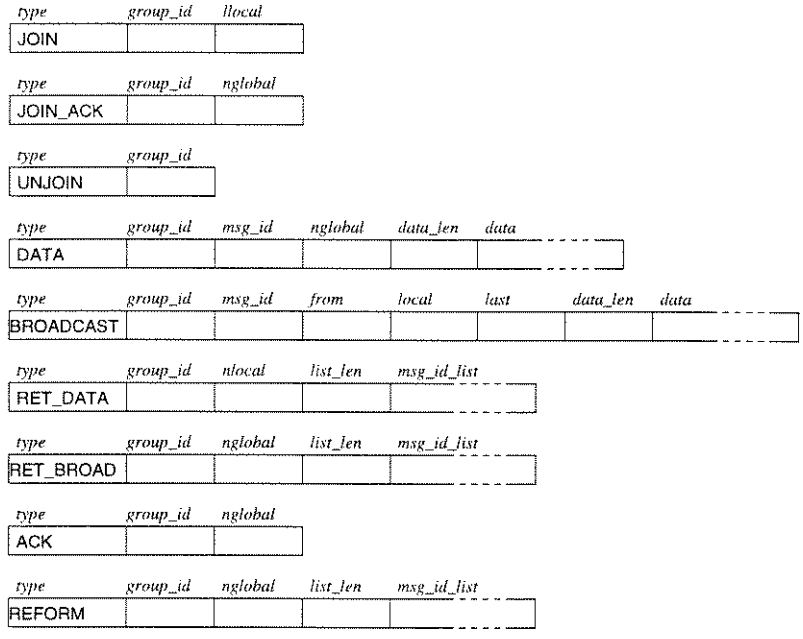

Figura 3.4: Formato das mensagens

3.7 Protocolo

As mensagens utilizadas no protocolo estão definidas na seção 3.6. O tipo da mensagem sempre aparece em letras maiúsculas de formato diferente do texto.

Toda mensagem enviada tem um identificador de grupo ao qual é destinada. Todo grupo tem um estado próprio e não interfere em nenhum outro grupo. A troca de mensagens descrita no protocolo sempre se refere a um grupo determinado.

Uma solicitação de *multicast* (`DATA`) enviada por um nodo cliente é unicamente identificada através do seu endereço de rede (endereço IP) e um número sequencialmente gerado a cada mensagem. Os identificadores iniciam a partir do número 1. Uma mensagem ou um identificador **local** refere-se à uma mensagem do cliente para o sequenciador.

Um *multicast* (`BROADCAST`) enviado pelo nodo sequenciador é identificado através de um número sequencialmente gerado a cada mensagem global. O identificador inicia a partir do número 1. Uma mensagem ou um identificador **global** refere-se à uma mensagem

do sequenciador para todos os membros do grupo.

3.7.1 Operação Normal

Para realizar um *multicast*, um nodo cliente primeiro envia uma mensagem (`DATA`) para o nodo sequenciador. O nodo sequenciador define um identificador global para a mensagem e a envia a todos os nodos clientes através de um *broadcast* (`BROADCAST`) (figura 3.5). Todos os nodos clientes do grupo (inclusive o nodo que enviou a mensagem) recebem a mensagem.

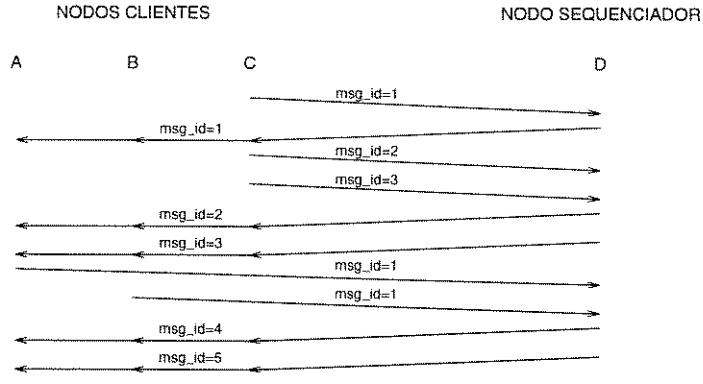

Figura 3.5: Exemplo de uma seqüência de *multicasts*

Toda mensagem enviada por um nodo cliente é armazenada num *buffer* (`c_smess`), pois caso a mensagem seja perdida, o nodo cliente pode retransmiti-la. A mensagem fica neste *buffer* até que o nodo receba a mensagem de volta do nodo sequenciador (figura 3.6).

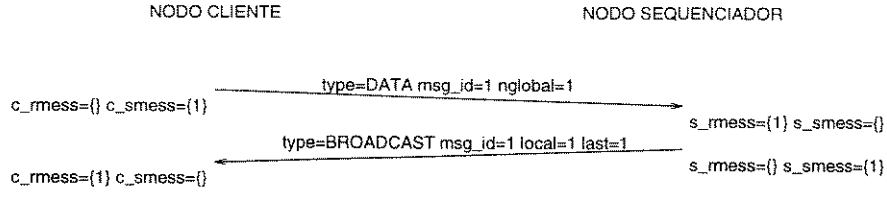

Figura 3.6: Troca de mensagens entre um determinado nodo cliente e o nodo sequenciador

Toda mensagem enviada pelo nodo sequenciador é armazenado num *buffer* (`s_smess`), pois caso algum nodo cliente perca a mensagem, o nodo sequenciador pode retransmiti-la. A

mensagem fica neste *buffer* até **todos** os nodos clientes confirmarem sua recepção. Quando todos os nodos confirmarem a recepção da mensagem, o sequenciador informa a todos os nodos clientes que podem descartar a mensagem (figura 3.7).

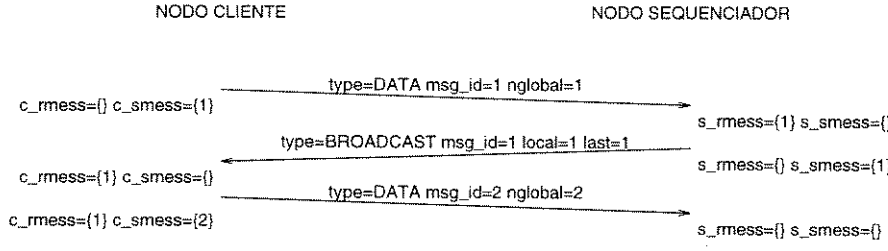

Figura 3.7: O nodo cliente envia a sua segunda mensagem e confirma a recepção da mensagem global 1

Toda mensagem recebida por um nodo cliente é armazenada num *buffer* (`c_rmess`). Todos os processos que estiverem esperando pela mensagem são desbloqueados e podem receber a mensagem. Quando todos as conexões já tiverem passado pela mensagem e o nodo sequenciador confirmou a recepção da mensagem por todos os clientes, ela é retirada do *buffer* (figura 3.8).

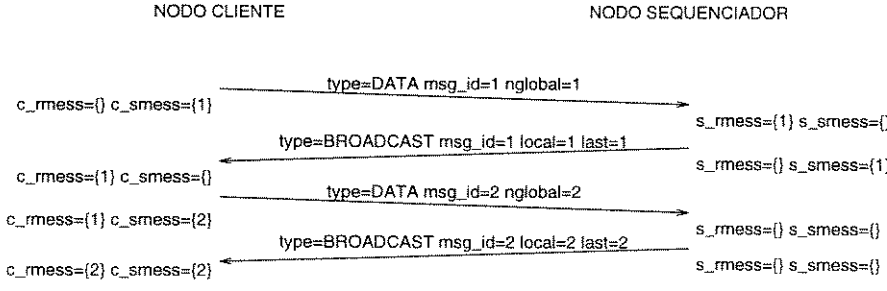

Figura 3.8: O nodo sequenciador envia a sua segunda mensagem e avisa que a mensagem global 1 foi confirmada por todos os nodos clientes (campo `last`)

Toda mensagem recebida pelo nodo sequenciador é imediatamente enviada para todos os nodos clientes, exceto se não houver espaço em `s_smess`. Neste caso, a mensagem é armazenada num *buffer* (`s_rmess`) até que haja espaço em `s_smess`. Se `r_rmess` também estiver com sua capacidade exaurida, a mensagem é descartada.

3.7.2 Entrada e Saída de Clientes

Um nodo que deseje participar de um grupo deve solicitar ao nodo sequenciador a sua adesão. Como o nodo cliente não sabe qual nodo na rede é o sequenciador, a mensagem é enviado por *broadcast*. Como resposta, o nodo sequenciador informa ao nodo cliente o próximo número de mensagem global do grupo. A primeira porta a conectar ao grupo evoca este procedimento (figura 3.9).

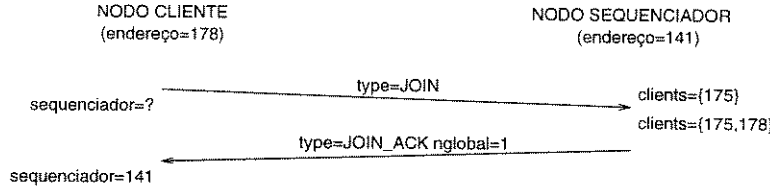

Figura 3.9: Adesão de um nodo cliente ao grupo

Quando o nodo cliente não quiser mais participar do grupo, ele envia uma mensagem de abandono do grupo. O nodo sequenciador simplesmente retira o cliente das tabelas de dados sobre clientes. Somente a última porta que se desconecta do grupo evoca este procedimento num nodo cliente (figura 3.10). Não há necessidade de confirmação da mensagem pois, mesmo que o nodo sequenciador não receba a mensagem `UNJOIN`, após um determinado tempo sem atividade do nodo cliente, o nodo sequenciador irá remove-lo.

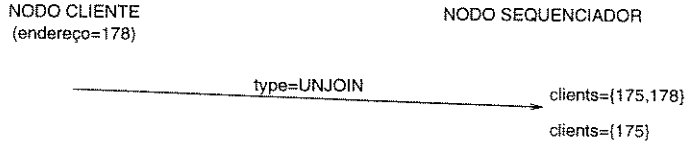

Figura 3.10: Abandono de um nodo cliente do grupo

3.7.3 Criação do Grupo e Nodo Sequenciador

Quando um nodo quiser se unir a um grupo e este ainda não existir, o nodo irá detectar a ausência do nodo sequenciador por *timeout*. A falha de um nodo é detectada quando após várias tentativas a mensagem de adesão não é respondida (figura 3.11).

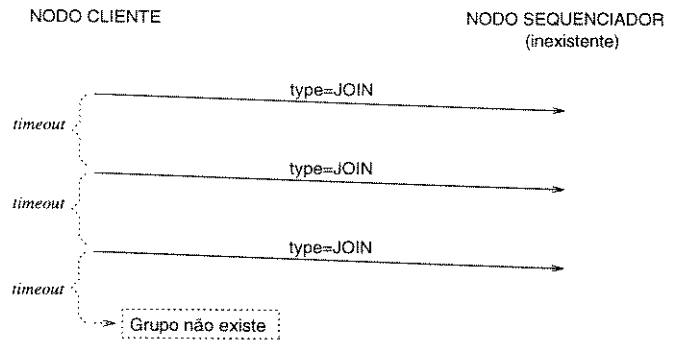

Figura 3.11: Nodo cliente detecta que sequenciador não existe e a etapa de reforma é iniciada

O nodo então inicia a etapa de reforma do grupo (veja seção 3.7.7). Ele envia um *broadcast* informando uma sugestão de próximo número global de identificação de mensagens para o grupo. Como o nodo está querendo criar o grupo, o número sugerido será 1. Caso dois nodos realizem o mesmo procedimento ao mesmo tempo, ambos irão receber a mensagem de reforma. Através de um critério de desempate, um dos dois nodos irá se autodesignar sequenciador e o outro irá novamente tentar adesão ao grupo.

Este protocolo de criação de grupo está sujeito também a falhas de mensagens. O tratamento dado neste caso é visto na seção 3.7.7.

3.7.4 Falhas de Mensagens

Quando o nodo sequenciador recebe uma mensagem fora de ordem de um nodo cliente, ele detecta a perda de uma ou mais mensagens e solicita a retransmissão ao nodo cliente. A mensagem recebida fora de ordem é armazenada em `s_rmess` se houver espaço (figura 3.12).

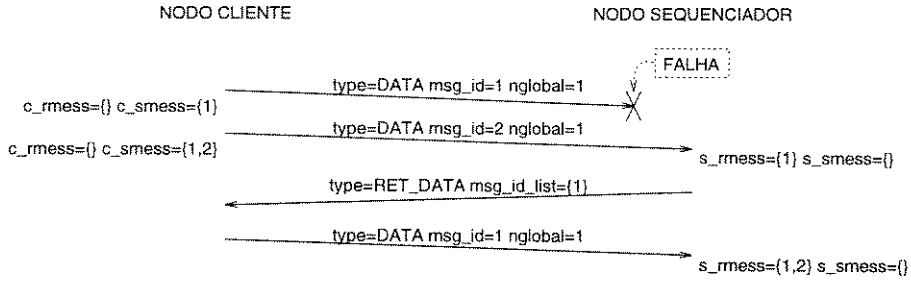

Figura 3.12: Nodo sequenciador detecta falta de uma mensagem do nodo cliente e solicita a retransmissão

Quando o nodo cliente receber uma mensagem fora de ordem do sequenciador, ele detecta a perda de uma ou mais mensagens e solicita retransmissão ao sequenciador. A mensagem recebida fora de ordem é armazenada em `c_rmess`, se houver espaço (figura 3.13).

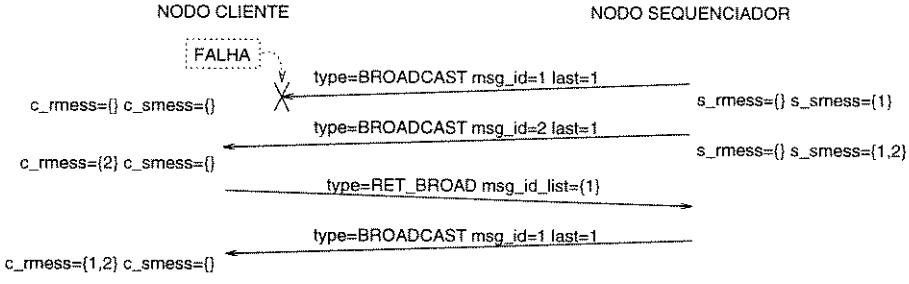

Figura 3.13: Nodo cliente detecta falta de uma mensagem do nodo sequenciador. A retransmissão da mensagem é então solicitada

3.7.5 Monitoramento do Nodo Cliente

Enquanto um nodo cliente é membro de um grupo, ele deve informar ao nodo sequenciador quais mensagens ele já recebeu. Mesmo quando não houver mensagens para enviar ao nodo sequenciador, o cliente deve, dentro um período máximo de tempo, enviar uma mensagem de reconhecimento explícito de *multicast* (`ACK`). Assim, o nodo sequenciador pode ir liberando as mensagens já confirmadas por todos os nodos, e também saber quais nodos estão em correto funcionamento (figura 3.14).

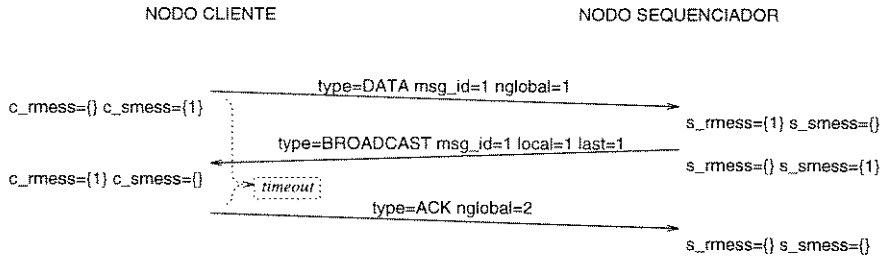

Figura 3.14: Nodo cliente envia reconhecimento explícito de mensagem após ficar um período sem enviar nenhuma mensagem ao sequenciador

3.7.6 Falha de Nodos Clientes

O nodo sequenciador mantém uma tabela com *timeouts* de cada nodo cliente para determinar quando foi recebida a última mensagem do nodo cliente. Quando não for recebida nenhuma mensagem do cliente após um período de tempo, o sequenciador detecta a falha do nodo e o remove da tabela de nodos clientes.

3.7.7 Falha do Nodo Sequenciador (Fase de Reforma)

A fase de reforma é iniciada por um nodo participante do grupo quando ocorrer um dos seguintes casos:

- *timeout* de resposta à uma mensagem de adesão no grupo (`JOIN`): o grupo ainda não existe e não há sequenciador para responder à requisição ou havia um sequenciador, mas este falhou;
- *timeout* de resposta à mensagem de solicitação de *multicast* (`DATA`): o sequenciador falhou;
- múltiplos sequenciadores: foi detectado mais de um sequenciador enviando mensagens;
- recebimento da mensagem de reforma (`REFORM`): algum outro nodo detectou uma das falhas anteriores.

Nos três primeiros casos, o nodo deve, incondicionalmente, enviar uma mensagem de reforma a todos os nodos. Nesta mensagem está o número global de mensagem mais recente que o nodo conhece. No quarto caso, a mensagem de reforma somente é enviada se o nodo conhecer uma mensagem global que seja mais recente que aquela recebida com

a mensagem de reforma. A medida que os nodos forem se informando de até que ponto chegaram as mensagens globais, eles fazem requisições de retransmissão aos nodos que as conhecem para completar as suas filas de mensagens.

Se o nodo cliente também for sequenciador, então o nodo deixa de sê-lo e as informações sobre o controle do grupo são descartadas. Isto pode ocorrer quando forem detectado múltiplos sequenciadores. Os dados descartados não podem ser aproveitados, pois não é possível afirmar se eles são corretos ou não.

Quando um nodo recebe a mensagem `REFORM`, ele verifica se possui um identificador de mensagem global mais recente que o da mensagem. Se houver, o nodo então envia uma mensagem de reforma imediatamente (figura 3.15). Caso a lista de mensagens globais não esteja completa, junto com o `REFORM` ele coloca os identificadores das mensagens faltantes. Se o identificador da mensagem `REFORM` recebida for o mesmo que o mais recente que o nodo conhece, o desempate é feito utilizando-se o identificador do nodo (o endereço IP).

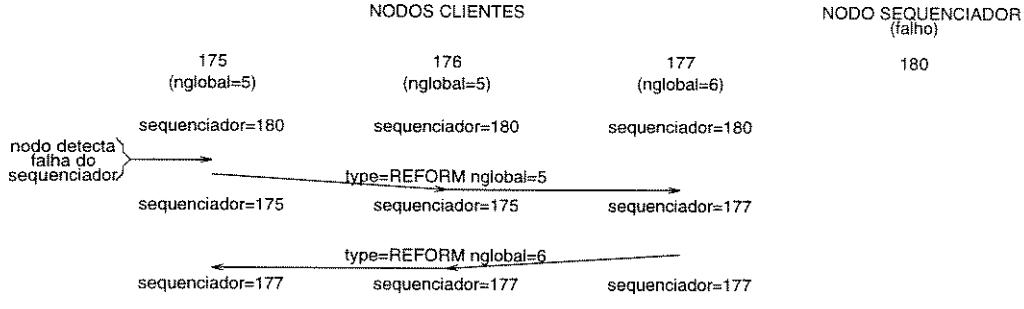

Figura 3.15: Fase de reforma em andamento

Sempre que um nodo recebe uma mensagem `REFORM` e verifica que possui alguma mensagem solicitada na lista de mensagens faltantes, ele envia as mensagens para o nodo que enviou o `REFORM`.

Depois de um certo tempo, após o último nodo se manifestar com a mensagem de reforma, dá-se por encerrada a etapa de reforma. O nodo que passa a ser o sequenciador é o nodo que se manifestou por último, ou seja, aquele que conhece a mensagem global mais recente. Todos os nodos participantes do grupo devem então solicitar ao sequenciador a sua adesão ao grupo.

O caso de múltiplos sequenciadores pode ocorrer quando dois nodos, após detectarem que o sequenciador não existe (ou falhou), enviam suas mensagens de reforma e a mensagem de reforma do nodo, que tem o mais recente número de mensagem global, é perdida. Assim os dois nodos decidem se tornar sequenciadores. Para resolver este problema, assim que o primeiro nodo tentar adesão ao grupo vai receber duas mensagens de aceitação (Mesmo sabendo o endereço do novo nodo sequenciador, o nodo cliente faz um *broadcast*, assim se houver mais de um sequenciador, há como se detectar). Neste momento é detectado a existência de mais de um sequenciador e uma nova etapa de reforma é iniciada (figura 3.16).

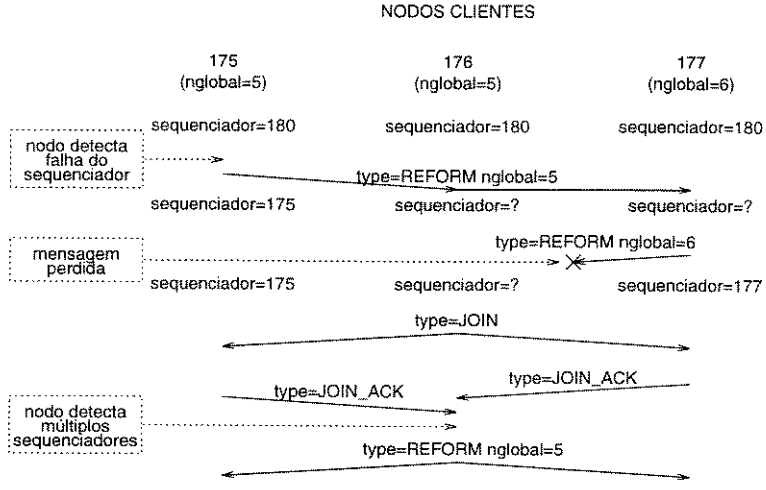

Figura 3.16: Detecção de múltiplos sequenciadores

Pode ocorrer ainda que um nodo tenha conhecimento da mensagem global mais recente, mas tenha perdido alguma mensagem global anterior. Na mensagem de reforma, ele informa quais mensagens ele não dispõe, pois ele necessita delas para tornar-se sequenciador. Os nodos clientes que dispuserem das mensagens enviam-nas para este nodo. Se ao final do período de reforma o nodo não recebeu todas as mensages solicitadas, ele reinicia a etapa de reforma (figura 3.17).

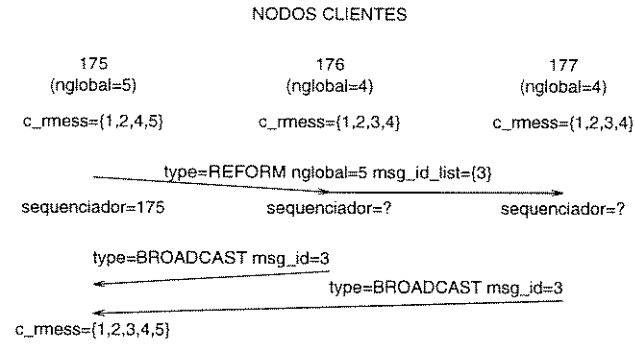

Figura 3.17: Reforma com pedido de retransmissão de mensagens

Capítulo 4

Comunicação Sem-conexão Ponto-a-ponto Confiável

4.1 Introdução

Um serviço sem conexão permite que se envie dados sem a necessidade de se estabelecer conexões. Cada mensagem é autocontida, com destinatário, origem e outras informações de controle. Normalmente protocolos que prevêem serviço sem conexão não oferecem nenhuma garantia de entrega da mensagem ao destinatário, isto é, não são confiáveis. O protocolo UDP da pilha TCP/IP é um exemplo. Ele envia datagramas ponto-a-ponto ou por *broadcast*, mas não garante a chegada ao destino.

Na prática, somente serviços com conexão oferecem confiabilidade, tal como provê o TCP/IP. Para aplicações que exijam confiabilidade e necessitam comunicar-se com muitos destinos ou enviam poucas mensagens, ou ainda outras razões de desempenho e complexidade, o serviço com conexão não é adequado. É, portanto, necessário um protocolo que ofereça um serviço sem conexão e confiável.

Este capítulo apresenta um protocolo sem conexão confiável. Ele é baseado no protocolo datagrama-reconhecido[1], apresentado em [5] com propósitos didáticos. Por ser um serviço raramente implementado em sistemas de comunicação comercialmente disponíveis (apesar do padrão IEEE 802.2 o recomendar com o tipo 3), existe pouca literatura que

[1] *acknowledged datagram*

trate do assunto. Assim, este protocolo foi projetado e implementado baseado apenas no datagrama-reconhecido.

A seção seguinte apresenta as questões de projeto e as próximas seções descrevem a implementação.

4.2 Questões de Projeto

As seguintes questões de projeto para este protocolo foram consideradas:

- confiabilidade;
- sincronismo;
- sequenciamento de mensagens;
- semântica de recepção;
- controle de fluxo.

4.2.1 Confiabilidade

Este critério trata da capacidade de recuperação do protocolo frente a falhas. Há vários níveis de confiabilidade que um protocolo pode oferecer. A seguir são apresentados alguns tipos tolerância a falhas. Este critério foi adaptado de [22] e foi apresentado na seção 3.2.2.

Neste trabalho foram implementado tolerância a falhas de mensagens e tolerância a falhas de nodos. Em caso de falha de um nodo, este não se recupera no estado em que se encontrava exatamente quando falhou, mas passa a funcionar como um nodo recém iniciado.

Tolerância a falhas maliciosas não são suportadas devido à grande dificuldade de implementação.

4.2.2 Sincronismo

Uma vez que a comunicação se dá ponto a ponto, o sincronismo refere-se à aplicação emissora e à aplicação receptora. Uma comunição **síncrona** requer que ambas as aplicações

estejam engajadas na comunicação no mesmo instante. A aplicação emissora, ao solicitar o envio da mensagem, é bloqueada até que a aplicação receptora receba a mensagem. O mesmo ocorre caso a aplicação receptora requeira a recepção da mensagem: ela deve ficar bloqueada até que a aplicação emissora envie a mensagem. Na comunicação **assíncrona**, a aplicação emissora não fica bloqueada até a recepção da mensagem.

A vantagem da comunicação síncrona é a possibilidade de se verificar o sucesso ou não do envio da mensagem. Entretanto, isso implica numa espera que restringe o paralelismo natural das aplicações.

Neste projeto foi implentada a comunicação síncrona, mas com o recurso de *timeout* para o envio e a recepção de mensagens. Isto se deve ao fato de não existir garantia total de entrega da mensagem ao destino (pode ocorrer falha de um nodo, por exemplo), sendo necessária uma forma de conferir o sucesso da comunicação.

4.2.3 Ordenação de Mensagens

A ordenação de mensagens numa comunicação sem-conexão é verificada entre dois pontos-fim de comunicação, isto é, entre duas portas. Somente as mensagens trocadas entre elas são consideradas.

Deste ponto de vista, pode haver ou não ordenação de mensagens. Para uma comunicação sem ordenação, qualquer seqüência de chegada é válida. Para uma comunicação com ordenação, a mesma seqüência de mensagens enviadas deve ser observada na recepção.

Neste trabalho, a ordenação de mensagens está implícita no funcionamento do protocolo.

4.2.4 Semântica de Entrega

A recepção de mensagens pode ser **seletiva** ou **indiscriminada**. Numa recepção seletiva, uma aplicação pode solicitar a recepção de uma mensagem de uma aplicação específica. Se a mensagem ainda não chegou ao nodo, a aplicação fica bloqueada, mesmo havendo mensagens de outras aplicações disponíveis para recepção. Na recepção indiscriminada a aplicação recebe a primeira mensagem disponível para recepção, não determinando

de qual aplicação receber.

Neste projeto, a recepção indiscriminada foi escolhida pela simplicidade de implementação.

4.2.5 Controle de Fluxo

O excesso de mensagens enviadas para um nodo pode implicar em perdas de mensagens, simplesmente por não haver espaço para armazená-las. Como o serviço deve ser confiável, estas mensagens perdidas possívelmente serão retransmitidas[2]. Entretanto, o envio das mensagens poderá ser um desperdício de banda passante, uma vez que não serão aproveitadas.

Para evitar esta situação, o nodo receptor deve informar, de alguma forma, ao nodo emissor, sobre suas condições de recepção.

Neste projeto, o controle de fluxo foi implementado somente do lado do nodo emissor. O nodo emissor possui um *buffer* para o envio de mensagens dedicado a cada um dos nodos com o qual mantém comunicação. Este *buffer* é limitado em número de mensagens. Quando uma mensagem é enviada, ela ocupa um *slot* neste *buffer*. Quando a mensagem for confirmada ou então ocorrer *timeout*, a mensagem é retirada do *buffer*.

O nodo receptor possui um *buffer* de recepção de mensagens dedicado a cada um dos nodos com o qual mantém comunicação. Este *buffer* possui o mesmo tamanho que o *buffer* do nodo emissor.

Quando o nodo emissor encontrar o seu *buffer* cheio, saberá que o *buffer* do nodo receptor também poderá estar cheio. A aplicação que solicitou o envio será bloqueada até que o *buffer* tenha espaço para enviar a mensagem.

4.3 Estruturas de Dados

Cada nodo mantém uma estrutura de dados para cada outro nodo com o qual mantém comunicação (figura 4.1). A estrutura de dados contém:

[2] Não serão retransmitidas caso o *timeout* da primitiva send seja muito curto

- `host_id`: endereço do nodo remoto com o nodo mantém comunicação;
- `nsend`: identificador da próxima mensagem a ser enviada ao nodo remoto;
- `nrecv`: identificador da próxima mensagem a ser recebida do nodo remoto;
- `smess`: *buffer* estático para as mensagens enviadas;
- `rmess`: *buffer* estático para as mensagens recebidas.

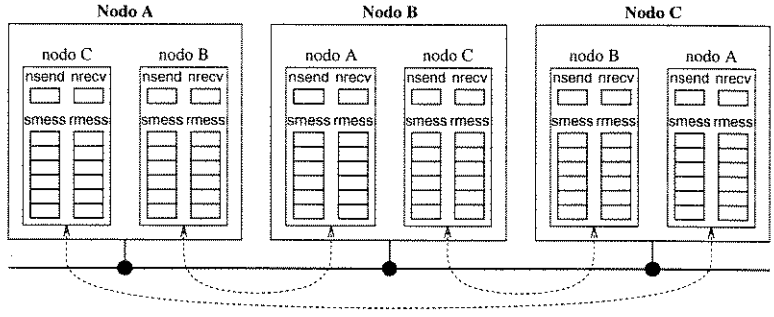

Figura 4.1: Estrutura de dados para controle de sequenciamento dos nodos.

Após ser criada, a estrutura para cada nodo é mantida indefinidamente. Ela somente será destruída quando o nodo falhar e todo o sistema reiniciar[3].

4.4 O Serviço

O serviço provido tem as seguintes características:

- sem conexão;
- transferência de dados por datagramas;
- ponto-a-ponto;
- confiável;
- síncrono.

O serviço é acessado através do ponto de acesso chamado **porta**. Cada porta deve ser associada a um **identificador de porta**. A associação porta-identificador é um-para-

[3] *boot*

um. Não pode haver duas ou mais portas com o mesmo identificador nem uma porta associada a mais de um identificador.

Através da porta podem ser acessados os serviços de associação de identificador de porta, transferência de dados e manipulação do *buffer* de mensagens recebidas. As primitivas para acessar estes serviços são:

- `open`: associa a porta a um identificador e torna-a disponível para envio e recepção de dados;
- `close`: encerra a utilização da porta e libera o identificador de porta;
- `send`: envia dados para outra porta;
- `receive`: recebe dados de outra porta;
- `select`: permite aguardar até que uma ou mais portas fiquem prontas para receber dados dentre uma lista de portas;
- `has_data`: verifica se a porta tem dados para receber;
- `flush`: retira todas os dados recebidos do *buffer*.

As primitivas `send`, `receive`, e `select` permitem que seja estipulado um tempo máximo de espera (*timeout*). A utilização das primitivas é descrita em detalhes no apêndice A.

4.5 O Protocolo

O protocolo utiliza duas mensagens para o envio e confirmação de dados e uma para controle de sequenciamento de mensagens. Estas mensagens são apresentadas na figura 4.2. A descrição detalhada das mensagens encontra-se no apêndice B.

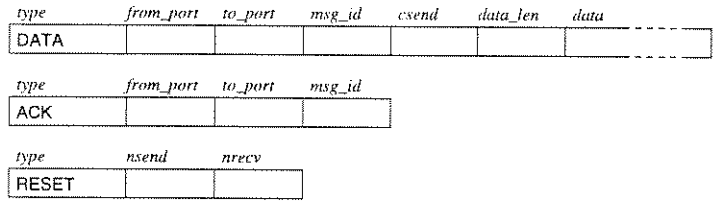

Figura 4.2: Formato das mensagens.

Quando a aplicação evoca a primitiva **send**, uma mensagem DATA é enviada ao nodo destino e a aplicação permanece bloqueada. A mensagem também é armazenada em **smess** para ser retransmitida, se necessário. Quando o nodo receber a mensagem ACK, a aplicação é liberada e a mensagem armazenada em **smess** é retirada. No nodo B, quando a mensagem for recebida, ela é armazenada em **rmess**. Quando a aplicação receber a mensagem da porta, o nodo B envia a mensagem ACK ao nodo A. A figura 4.3 ilustra este caso.

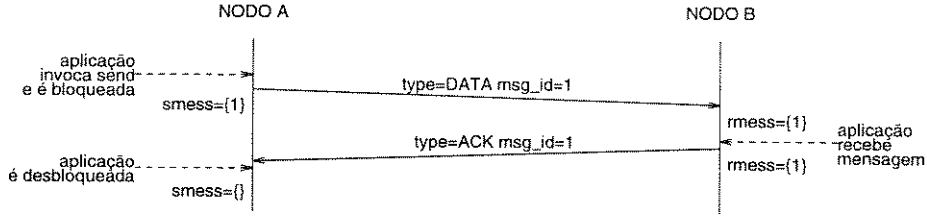

Figura 4.3: Funcionamento normal do protocolo

Na figura 4.4 a mensagem DATA é perdida na primeira tentativa. No segundo envio a mensagem chega ao nodo destino. Quando a aplicação recebe a mensagem DATA, a mensagem ACK é enviada. O número de repetições no envio da mensagem DATA depende do *timeout* especificado na primitiva **send**. Quanto mais longo, maior o número de retransmissões. Se o *timeout* for muito curto, pode não haver tempo para retransmissão e a primitiva **send** retorna erro por *timeout*.

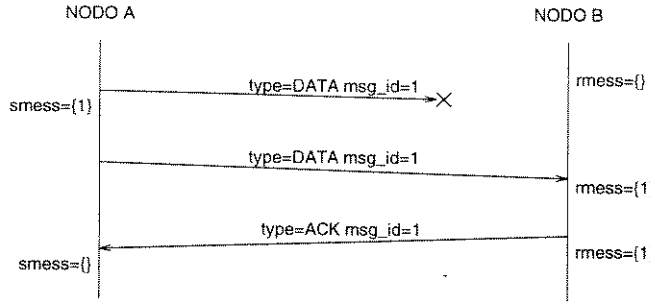

Figura 4.4: Falha da mensagem `DATA`

Se a mensagem `DATA` chegar corretamente ao destino, mas a mensagem `ACK` demorar para retornar (a aplicação pode atrasar a recepção da mensagem), o nodo emissor pode considerar falha a tentativa de envio dos dados por *timeout* e retornar erro à aplicação. A figura 4.5 ilustra este caso.

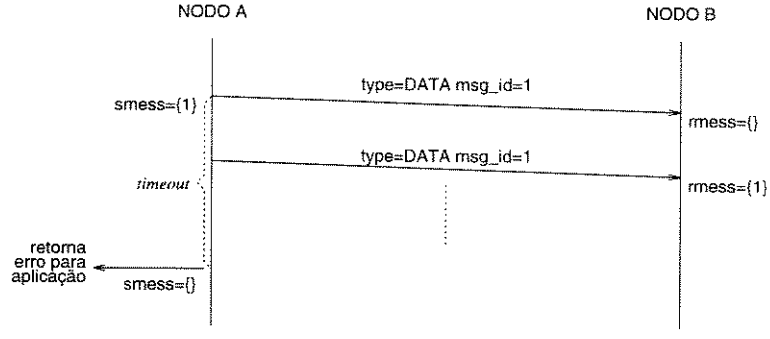

Figura 4.5: Erro de envio por *timeout*

Pode ocorrer também a falha da mensagem `ACK`. Porém, do ponto de vista do nodo emissor não é possível definir-se se o que se perdeu foi a mensagem `DATA` ou o seu reconhecimento, `ACK`. Uma vez que o reconhecimento de mensagens ocorre somente quando a aplicação recebe a mensagem, é necessário que se registre este fato. Se uma segunda mensagem `DATA` com mesmo identificador de mensagem for recebida, é verificada se aquela mensagem em `rmess` foi marcada como recebida pela aplicação. Se foi recebida, o nodo pode responder com `ACK`. Caso contrário a mensagem duplicada é descartada. Na figura 4.6 a mensagem `ACK` é perdida. Na segunda tentativa de `DATA`, a mensagem já foi recebida pela aplicação (e marcada como tal) e o nodo pode responder com `ACK`.

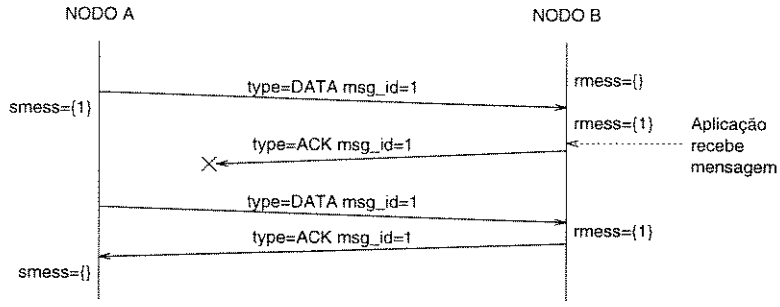

Figura 4.6: Falha da mensagem `ACK`

Para evitar que se armazenem indefinidamente mensagens em `rmess` no nodo receptor, o nodo emissor informa até qual mensagem ele ainda espera confirmação. Esta informação está no campo `csend` da mensagem `DATA`. Desta forma o nodo receptor sabe quando pode descartar as mensagens antigas. O valor do campo `csend` é o identificador de mensagem mais antigo em `smess`, isto é, aquele do qual o nodo espera uma confirmação. Desta forma também se realiza o controle de fluxo de dados. O tamanho de `smess` é o mesmo de `rmess`. Se `smess` não estiver cheio, `rmess` também não estará, e o nodo emissor poderá enviar suas mensagens sem o problema de *buffer overflow*. Na figura 4.7 a mensagem 1 fica retida em `rmess` do nodo B até chegar a mensagem 2, quando `csend` indica que a mensagens anteriores a 2 já foram descartadas.

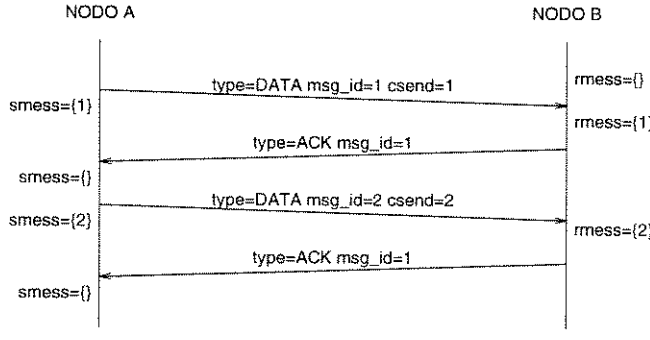

Figura 4.7: Controle das mensagens em `rmess` e `smess`

4.5.1 Controle de Sequenciamento

Cada nodo é responsável pelo controle do sequenciamento das suas mensagens enviadas e acompanha o sequenciamento das mensagens recebidas de outros nodos.

Um nodo A ao enviar sua primeira mensagem `DATA` a um nodo B, cria a estrutura de dados de controle de mensagens trocadas entre A e B. Antes de enviar `DATA`, uma mensagem `RESET` é enviada para B criar a estrutura de dados correspondente (caso já não a tenha criado). A mensagem `RESET` leva duas informações de sequenciamento:

- o próximo identificador de mensagem que o nodo A irá enviar a B, `nsend`. O nodo B sabe então qual mensagem deve ficar esperando;
- o próximo identificador de mensagem que o nodo A espera de B, `nrecv`. O nodo B verifica se o nodo A tem o sequenciamento correto de mensagens enviadas de B para A.

O nodo B, ao receber a mensagem `RESET`, verifica se a estrutura de dados para a comunicação com A já existe. Se não existir, a estrutura é criada com `nsend` sendo igual 1, `nrecv` sendo igual ao da mensagem `RESET` e os *buffers* de mensagens vazios. Com a estrutura de dados criada ou utilizando a já existente, B compara `nsend` da mensagem `RESET` com o da estrutura de dados. Se diferirem é porque o nodo A sofreu alguma falha e não tem o sequenciamento correto de mensagens. O nodo B deve corrigir esta diferença enviando uma mensagem `RESET` para o nodo A. O nodo A ao receber este segundo `RESET` atualiza sua estrutura de dados com o valor corrento para `nrecv`.

Se nodo B falhar (*fail-stop*), do ponto de vista do nodo A isto não será perceptível. As portas que mantinham comunicação com portas do nodo B continuarão a tentar enviar suas mensagens normalmente (se o *timeout* for longo o suficiente). O nodo B ao recuperar-se e reiniciar o protocolo não terá a estrutura de dados para comunicação com A. Entretanto, apesar de não ter recebido nenhuma mensagem `RESET`, ele irá receber mensagens `DATA` (e até mesmo `ACK`). Ao receber estas mensagens, B deve criar a estrutura de dados para se comunicar com A e enviar a mensagem `RESET`. `nrecv` deve receber o valor do identificador da mensagem `DATA` recebida. Se o nodo A verificar que `nrecv` não está correto, envia outro `RESET` para corrigi-lo (figura 4.8).

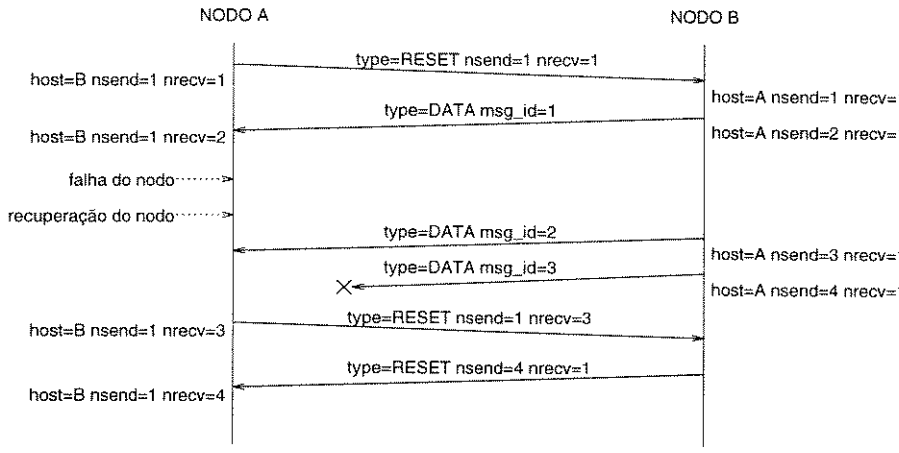

Figura 4.8: Falha de nodo e recuperação do sequenciamento

Capítulo 5

Plataforma de Comunicação

5.1 Introdução

As primitivas `open`, `close`, `send`, `receive`, `has_data` e `flush` estão presentes nos três serviços descritos anteriormente. A porta é o ponto-final de comunicação por onde são evocadas estas primitivas. Existem duas razões para que a plataforma tenha a porta como elemento comum para acessar os três protocolos.

A primeira, é a economia de código. Os protocolos têm várias partes de tratamento de *buffers*, testes e *timeouts* que são comuns. Reunidos em um ponto comum, reduzem a quantidade de código fonte total e também facilitam a depuração.

A segunda razão é a funcionalidade da primitiva `select`. Se os três protocolos possuissem uma porta diferente, não seria possível reunir num único `select` o teste de portas de serviços diferentes.

As portas são comuns até o ponto onde são exigidos parâmetros específicos para cada protocolo. A tabela 5.1 apresenta a diferença dos parâmentros em cada protocolo.

Primitivas	**Protocolos**		
	Reserva de Banda	**Grupo**	**Sem-conexão**
`open`	Porta emissora: • Identificador de porta • Tamanho dos dados • Período • Retardo máximo Porta receptora: • Identificador do nodo remoto • Identificador da porta emissora • *timeout*	• Identificador do grupo	• Identificador da porta
`close`	–	–	–
`send`	Porta emissora: • Ponteiro para os dados • Tamanho dos dados Porta receptora não realiza envio	• Ponteiro para os dados • Tamanho dos dados	• Endereço do nodo destino • Identificador da porta destino • Ponteiro para os dados • Tamanho dos dados • *timeout*
`receive`	Porta receptora: • Ponteiro para os dados • Tamanho dos dados • *timeout* Porta emissora não realiza recepção	• Ponteiro para os dados • Tamanho dos dados • Endereço do nodo origem • Identificador da porta origem • *timeout*	• Ponteiro para os dados • Tamanho dos dados • Endereço do nodo origem • Identificador da porta origem • *timeout*
`select`	Porta receptora: • Número de portas • Lista de portas • *timeout* Porta emissora não realiza `select`	• Número de portas • Lista de portas • *timeout*	• Número de portas • Lista de portas • *timeout*
`has_data`	–	–	–
`flush`	–	–	–

Tabela 5.1: Comparação dos parâmetros das primitivas entre os três protocolos

As primitivas `close`, `has_data` e `flush` não possuem parâmetros e são completamente iguais entre os protocolos. A primitiva `select` recebe na lista de portas o identificador de qualquer tipo de porta (exceto uma porta do protocolo de reserva de banda no modo emissora). A primitiva `receive` difere em alguns parâmetros. No protocolo de reserva de banda, a porta receptora não tem os parâmetros para endereço do nodo e o identificador da porta origem. A primitiva `open` é completamente diferente em cada protocolo. A primitiva `send` tem em comum os dados a serem enviados, mas difere no protocolo sem-conexão, onde o destino dos dados é especificado.

Para enviar as mensagens que evocam as primitivas, foi criada uma **interface de programação** orientada a objetos em C++ (apêndice A). Existe uma classe base, `conn`, e três classes especializadas, `conn_br`, `conn_rm` e `conn_ru`, uma para cada protocolo.

As primitivas semelhantes são chamadas por métodos (funções) da classe base. Os métodos da classe base são: `close`, `receive`, `has_data` e `flush`. A primitiva `receive` foi incluida para permitir que, depois de um `select`, se receba dados mesmo sem saber qual tipo de porta tem dados diponíveis. No caso das portas do protocolo reserva de banda o endereço e porta origem podem ser desprezados, pois sempre serão aqueles especificados na primitiva `open`.

As primitivas `send` e `open` são chamadas por métodos implementados nas classes especializadas para cada protocolo.

5.2 Processos

A implementação da plataforma de comunicação no sistema UNIX pode ser feita de duas formas: como um **processo** ou como um módulo **integrado ao núcleo**.

A implementação integrada ao núcleo permite que as primitivas de comunicação possam ser chamadas diretamente dos processos aplicação através de chamadas do sistema (*system calls*). Este é o método de melhor desempenho e permite maior controle sobre os processos de aplicação. Entretanto, o método implica numa alteração do código do núcleo para comportar novas chamadas do sistema, o que é inevitavelmente complexo e sujeito a falhas, além do problema prático da fase de testes que necessita, a cada versão do código,

de uma reiniciação do sistema operacional.

A implementação como um processo é a forma mais acessível e menos traumática para o sistema operacional. Cada versão do executável pode ser testada sem reinicializar o sistema operacional, tal como é feito com os *deamons* do UNIX. Entretanto, já existe uma necessidade de uma comunicação entre processos, que implica num *overhead* significativo. Também há, na maioria dos sistemas UNIX, total falta de controle sobre o escalonamento do processo, que pode causar inevitáveis perdas de *deadlines* do processo que fazem uso dos serviços de comunicação.

A comunicação entre processos (transferência de dados e sincronização) no UNIX pode ser realizada de duas formas: por **memória compartilhada e semáforos** ou por **troca de mensagens**.

A comunicação por troca de mensagens permite uma melhor estruturação do código fonte (programação "elegante") e facilita a depuração. Entretanto, cada envio (ou recepção) de mensagem implica em, pelo menos, mais duas cópias de dados (do processo origem para a área do sistema e desta para o processo destino), aumentando ainda mais o *overhead* da plataforma.

A comunicação por memória compartilhada e semáforos permite o controle direto sobre os *buffers* de mensagens dos procotolos. A transferência de dados não implica em chamadas do sistema operacional e por isso é mais eficiente que a troca de mensagens. Entretanto, a sincronização através de semáforos é complexa e induz a erros de programação, que são difíceis de depurar.

A implementação da plataforma de comunicação foi feita com um processo de usuário e utilizando troca de mensagens para realizar a comunicação aplicação/protocolos.

Controle dos processos

Para um processo aplicação evocar uma primitiva, ele envia uma mensagem ao processo protocolo e aguarda uma resposta. Enquanto o protocolo não responder, a aplicação fica bloqueada. Portanto, para obter o efeito de bloquear uma aplicação até um recurso ficar disponível (por exemplo, uma mensagem), basta o protocolo postergar a resposta.

O processo protocolo possui uma fila de mensagens para receber a evocação de primitivas. Esta fila possui um identificador que é conhecido de todos os processos aplicação. Cada processo aplicação possui uma fila para receber as respostas do processo protocolo (figura 5.1).

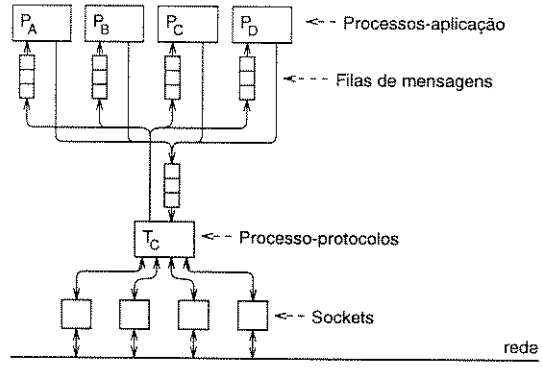

Figura 5.1: Comunicação entre processo-aplicação e o processo-protocolo.

Cada processo-aplicação deve, antes de evocar qualquer primitiva, informar ao processo-protocolo o identificador de sua fila mensagens. Este procedimento é a iniciação local para utilização dos protocolos. O processo protocolo ao receber o identificador da fila da aplicação cria uma estrutura de dados para armazená-la juntamente com outras informações de controle do processo.

Quando o processo aplicação terminar, deve liberar a fila de mensagens local e avisar o processo protocolo do ocorrido. Pode haver, eventualmente, portas ainda abertas ou outros estados pendentes que devem ser desfeitos. Este procedimento é necessário porque o protocolo não tem como verificar se um determinado processo ainda existe ou não para liberar as estruturas dedicadas a ele.

5.3 Portas

Um processo acessa os serviços do protocolo através de primitivas. Estas primitivas são divididas em bloqueantes e não bloqueantes:

- **bloqueantes:** `open`, `close`, `receive`, `send` e `select`;

- **não-bloqueantes**: `has_data` e `flush`.

As primitivas não bloqueantes não dependem de troca de mensagens e podem ter resposta imediata. A primitiva `has_data` realiza apenas uma consulta no *buffer* de mensagens recebidas da porta. A primitiva `flush` esvazia este `buffer`. A primitiva `close`, no protocolo de reserva de banda, implica na troca de mensagens para terminação correta da conexão.

As primitivas bloqueantes eventualmente podem não completar toda a sua atividade imediatamente. A primitiva `open` no protocolo de reserva de banda, ao abrir uma porta para recepção de dados, desencadeia o processo de conexão de porta, que implica numa espera de mensagem de confirmação. A primitiva `close`, no protocolo de reserva de banda, implica na troca de mensagens para terminação correta da conexão. A primitiva `send` pode bloquear quando não houver espaço no *buffer* do protocolo para envio de mensagens, ou necessitar de alguma confirmação, como é o caso do protocolo sem-conexão. A primitiva `receive` bloqueia sempre que não houver mensagens no *buffer* de mensagens recebidas da porta. A primitiva `select` bloqueia quando todas as portas que estão sendo testadas na lista de portas não dispuserem de mensagens nos seus *buffers*. Em todas estas primitivas bloqueantes é possível especificar-se um *timeout*.

Como estas primitivas bloqueantes comportam-se de uma forma homogênea entre os protocolos, foi criado um protocolo de comunicação entre a porta e o protocolo que a porta está utilizando. Este protocolo é implementado com chamada de funções. A comunicação é realizada entre o módulo *dispatcher* e os módulos dos protocolos.

Este protocolo é composto basicamente das seguintes funções:

- *primitiva*`_request` é o tratamento da primitiva em si. Se o resultado para a primitiva estiver disponível, esta função chama *primitiva*`_answer` e termina. Senão, altera o estado da porta e espera a condição necessária.
- *primitiva*`_request_cancel` é o cancelamento de uma primitiva que foi bloqueada e terminou por *timeout*. As ações tomadas para o bloqueio devem ser desfeitas pelo protocolo.
- *primitiva*`_answer` devolve ao processo o resultado do serviço que solicitou através de *primitiva*. Esta função é implementada no *dispatcher* e chamada a partir do protocolo.

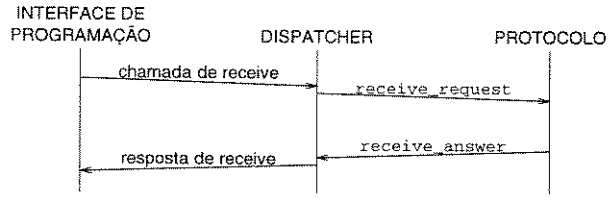

Figura 5.2: Exemplo de requisição-resposta com a primitiva `receive`

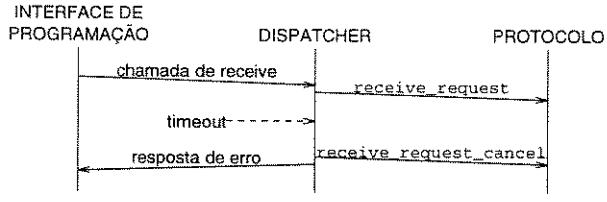

Figura 5.3: Exemplo de requisição-*timeout* com a primitiva `receive`

Na figura 5.2 acontece uma chamada da primitiva `receive` com uma resposta normal, isto é, sem *timeout*. A comunicação da interface de programação (processo-aplicação) para o *dispatcher* é realizada por troca de mensagem, ao passo que a comunicação do *dispatcher* com o protocolo é realizada com chamada de função (em programação C++). Na figura 5.3 acontece um erro por *timeout*.

5.3.1 Criação de Porta

A figura 5.4 apresenta (parcialmente) a estrutura interna da plataforma de comunicação.

Quando uma aplicação evoca a primitiva `open`, o *dispatcher* chama o protocolo correspondente para criar a estrutura de dados da porta. Cada protocolo tem particularidades para o controle de uma porta, portanto somente ele sabe como criar a estrutura de dados. Com a porta criada, o *dispatcher* pode então chamar a função `open_request` para aquela porta.

Cada porta criada é registrada na estrutura de dados do processo. Isto é necessário para destruir todas as portas daquele processo, caso ele termine sem fechar as portas.

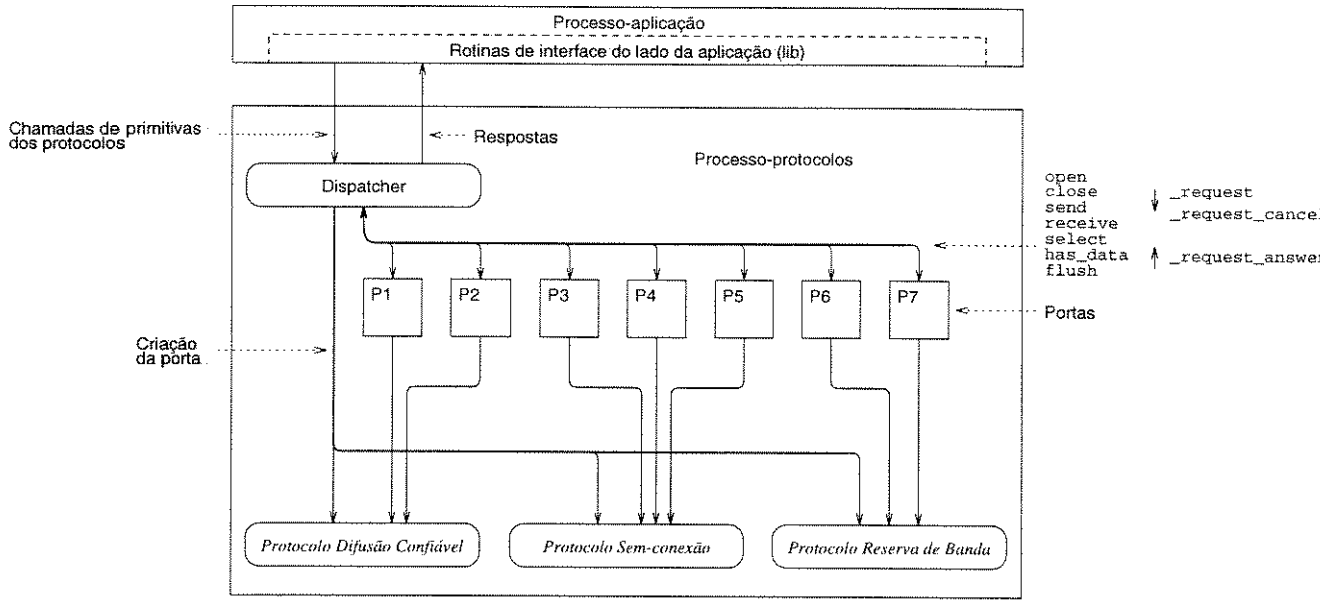

Figura 5.4: Estrutura de comunicação entre processo-aplicação, portas e protocolos

5.3.2 Destruição da Porta

Quando o processo-aplicação termina e avisa ao processo-protocolos, as portas pertencentes à aplicação devem ser fechadas e destruídas. A terminação de um processo UNIX pode ocorrer assíncronamente, isto é, através de um *signal*. Neste caso, possivelmente a aplicação pode estar esperando alguma resposta do processo-protocolos. Isto significa que pode haver algum estado transitório nas portas do processo. Do lado do processo-protocolos é necessário desfazer esta situação transitória em cada porta, fechar todas as portas e destruir as estruturas de dados destas portas.

5.4 Estrutura de *Software*

A plataforma é contituída de vários módulos. Os módulos foram divididos em duas categorias: dependentes da aplicação e independentes da aplicação.

O módulos independentes da aplicação, que podem ser utilizados em outras aplicações, são divididos em módulos dependentes do sistema e independentes do sistema. Os módulos dependentes do sistema necessitam uma implementação específica em cada sistema operacional onde operam. Os módulos independentes do sistema são perfeitamente portáveis. Todos os módulos dependentes da aplicação também são portáveis.

Dependentes do sistema:

- Classe `sys_time` (manipulação de hora): `sys_time.h` e `sys_time.cxx`
- Classe `sys_addr` (manipulação de endereços IP): `sys_addr.h` e `sys_addr.cxx`
- Classe `sys_sock` (manipulação de *sockets*): `sys_sock.h` e `sys_sock.cxx`
- Classe `sys_msg` (manipulação de mensagens): `sys_msg.h` e `sys_msg.cxx`

Independentes do sistema:

- Classe `list` (manipulação de listas): `list.h` e `list.cxx`
- Classe `squeue` (manipulação de filas estáticas): `squeue.h` `esqueue.cxx`
- Classe `timer` (manipulação de temporizadores para *timeouts*): `timer.h` e `timer.cxx`

Os módulos dependentes da aplicação implementam funções dos protocolos divididos da seguinte forma:

- Módulo principal e de inicialização: `ltp.cxx`
- Configuração dos protocolos: `config.h` e `config.cxx`
- Interface das aplicações com o protocolo:
  - Interface de programação da aplicação: `conn.h` e `conn.cxx`
  - Mensagens da interface: `conn_mess.h`, `conn_mess.cxx`
  - Tratador de mensagens da interface: `conn_ctl.h` e `conn_ctl.cxx`
- Protocolo controle de envio:
  - Controle: `bwc.h` e `bwc.cxx`
  - Mensagens: `bwc_mess.h` e `bwc_mess.cxx`
- Protocolo de conexão de portas:
  - Controle: `br.h` e `br.cxx`
  - Mensagens: `br_mess.h` e `br_mess.cxx`

- Protocolo comunicação ponto-a-ponto sem-conexão confiável:
  - Controle: `ru.h` e `ru.cxx`
  - Mensagens: `ru_mess.h` e `ru_mess.cxx`
- Protocolo comunicação em grupo:
  - Controle: `rm.cxx` e `rm.h`
  - Definição de mensagens: `rm_mess.cxx` e `rm_mess.h`
  - Controle do lado do cliente: `rm_cli.cxx` e `rm_cli.h`
  - Controle do lado do sequenciador: `rm_seq.cxx` e `rm_seq.h`

5.5 Medida de Desempenho

A medida de desempenho dos serviços de comunicação foram realizadas com o objetivo de se ter uma idéia da capacidade de transmissão de dados dos protocolos implementados. Os resultados obtidos nesta avaliação foram modestos, pois existem fatores que deterioram o desempenho. Estes fatores são:

- processo-protocolo executa em prioridade de usuário, portanto ele disputa CPU com todos os demais processos ordinários de cada nodo;
- competição com outros serviços de rede, tais como NFS, telnet, ftp, mail, etc;
- o UDP apresenta um *overhead* significativo, pricipalmente no cômputo do *checksum* por *software*.

Os testes foram realizados em 4 estações IBM POWERserver 360 numa rede FDDI[1] 100 Mbps e em horários onde a carga dos processadores e da rede fosse mínima.

O teste de cada serviço foi realizado separadamente. Por exemplo, o serviço de comunicação em grupo foi testado quando não havia nenhum tráfego dos protocolos de reserva de banda e do protocolo sem-conexão. Desta forma, o desempenho apresentado é o melhor possível (nas condições de teste atuais). Com a utilização conjunta de todos os serviços, o desempenho certamente não será o mesmo.

O desempenho do serviço de comunicação sem-conexão ponto-a-ponto confiável foi medido através do tempo de confirmação do envio da mensagens (1000 mensagens).

[1] sem tráfego síncrono

Tamanho da mensagem	Tempo para confirmação
10 bytes	40 ms
100 bytes	41 ms
500 bytes	42 ms
1000 bytes	43 ms

Tabela 5.2: Desempenho do serviço de comunicação sem-conexão ponto-a-ponto confiável

Este tempo corresponde aproximadamente ao dobro do tempo para a mensagem chegar ao processo-aplicação destino (considerando que o processo já está esperando a mensagem). A tabela 5.2 apresenta o resultado dos testes. Observando os valores da tabela, conclui-se que o tamanho da mensagem não varia muito o tempo de transmissão de uma mensagem.

Tamanho da mensagem	Tempo para envio
10 bytes	53 ms
100 bytes	54 ms
500 bytes	56 ms
1000 bytes	58 ms

Tabela 5.3: Desempenho do serviço de comunicação em grupo com 2 nodos participantes

Número de nodos	Tempo para envio
2	58 ms
3	65 ms
4	77 ms

Tabela 5.4: Desempenho do serviço de comunicação em grupo com variação no número de nodos participantes e mensagens de 1000 bytes

O desempenho do serviço de comunicação em grupo foi medido através do tempo para o envio de mensagens (1000 mensagens). Este tempo corresponde ao tempo para o processamento e difusão de um *multicast*. O tempo de retardo para a mensagem chegar ao processo-aplicação corresponde aproximadamente a este tempo (eventuais perdas de mensagens e atrasos na recepção dos processos-aplicação podem aumentar este retardo). A tabela 5.3 apresenta o resultado dos testes com variação de tamanho de mensagem e

número fixo de nodos participantes (2 nodos). Verifica-se neste caso também que o tamanho da mensagem influencia pouco o tempo total para o envio de uma mensagem. A tabela 5.4 apresenta o resultado dos testes com variação no número de nodos participantes em tamanho de mensagem fixo (1000 bytes). Verifica-se que o número de nodos participantes influi significativamente no tempo médio para a difusão de mensagens.

Quantidade de dados por período	Período	Banda passante solicitada	Banda passante observada
10K bytes	1 s	10 KBps	10 KBps
1K bytes	100 ms	10 KBps	9.4 KBps
200 bytes	50 ms	10 KBps	8.9 KBps
100 bytes	10 ms	10 KBps	8.6 KBps
10 bytes	1 ms	10 KBps	0.6 KBps
1K bytes	1 s	1 KBps	1 KBps
1K bytes	100 ms	10 KBps	9.4 KBps
1K bytes	10 ms	100 KBps	80 KBps
1K bytes	5 ms	200 KBps	136 KBps

Tabela 5.5: Desempenho do serviço de comunicação com reserva de banda

O desempenho do serviço de comunicação com reserva de banda foi medido através de tentativas de reserva de banda. Estas tentativas foram feitas apenas com variação nos parâmetros de banda passante. Os valores de banda são especificados em quantidade de dados e periodicidade de envio. Foram feitas duas séries de testes. A primeira com uma banda passante fixa (10 KBps) e variando o período de envio dos dados de 1s até 1ms. A segunda com uma quantidade de dados fixo (1 Kbytes) e variando a banda passante de 1 KBps até 200 KBps.

Verificou-se uma degradação acentuada para valores de período abaixo de 10 ms. Isto deve-se ao fato da arquitetura de comunicação processo-aplicação com o processo-protocolos (e outros fatores mencionados anteriormente) introduzir um *overhead* excessivo. Entretanto, pode-se transmitir adequadamente canais típicos de 64 Kbps (aproximadamente 8 KBps).

Capítulo 6

Conclusão

Neste trabalho foram apresentados o projeto e a implementação dos serviços de comunicação para a plataforma *multiware*. Os protocolos para estes serviços já foram fartamente explorados em outros trabalhos, mas sempre enfocando a correção[1] dos mesmos. Estes trabalhos deixam de lado questões como controle de fluxo e manipulação de *buffers*, que numa implementação são tão importantes quanto a correção do protocolo.

O serviço de comunicação com reserva de banda demandou maior esforço, pois foi necessário implementar um controle de acesso ao meio para aumentar o determinismo de redes como *Ethernet* e FDDI[2]. O esquema de reserva de banda é explorado em muitos trabalhos atuais na área de comunicação multimídia. Praticamente todos atestam a necessidade de uma rede com tráfego isócrono, ou pelo menos síncrono, para se ter um desempenho aceitável.

6.1 Contribuições

Ao final de dois anos e meio de pesquisa (um ano e meio dedicados exclusivamente ao projeto e implementação dos serviços de comunicação), as principais contribuições foram:

- proposta para comunicação de mídia contínua sobre uma rede de tráfego assíncrono;

[1] no sentido de estar correto
[2] sem tráfego síncrono

- extensão do algoritmo de difusão confiável para atender controle de fluxo;
- projeto de um algoritmo de datagrama confirmado com controle de fluxo;
- encapsulamento de serviços de troca de mensagens, *sockets*, semáforos e memória compartilhada[3] em classes C++, portável entre os sistemas operacionais SunOS (versão 4.1), AIX (versão 3), e OS/2 (versão 2.1).
- projeto de um esquema de *dispatching* genérico para os serviços de comunicação a partir de uma única interface de programação, que permite, inclusive, a inclusão de novos serviços de comunicação (por exemplo, serviço com-conexão confiável) com alterações mínimas no *dispatcher*.

6.2 Trabalhos Futuros

Em termos de continuidade do trabalho, são sugeridos os seguintes desdobramentos:

- implementação da fragmentação/remontagem de mensagens longas;
- ajuste do tamanho de *buffers* dos serviços de comunicação em grupo e comunicação sem-conexão ponto-a-ponto confiável para se obter um melhor desempenho;
- utilização de *sockets raw* para evitar o *overhead* do UDP e para se obter um melhor desempenho;
- estender os serviços para que possam atuar sobre múltiplas redes, realizando funções de roteamento [12, 13, 14];
- estender o serviço de comunicação sem-conexão ponto-a-ponto confiável para realizar confirmação em duas fases, a fim de tornar o resultado do envio de mensagens completamente confiável[4];
- aperfeiçoar o serviço de comunicação em grupo utilizando algoritmos totalmente distribuídos.

[3] os serviços de memória compartilhada e semáforos não foram utilizados na versão final da implementação

[4] por completamente confiável entende-se que a primitiva de envio retorne sucesso à aplicação se a mensagem for recebida pela aplicação receptora, e retorne falha somente se a aplicação receptora não receber a mensagem

6.3 Considerações Finais

Sabe-se que nenhum *software* é livre de erros, especialmente quando este não foi desenvolvido utilizando-se uma especificação formal. Mesmo neste trabalho onde cada serviço de comunicação foi testado exaustivamente, podem existir erros. Apenas a sua utilização num ambiente de trabalho real, isto é, no caso específico, na plataforma *multiware*, é que estes serviços poderão ser validados. Ao término deste trabalho, ainda não foram feitos testes de integração com todos os serviços da camada *middleware*, e os serviços de comunicação se restringiram a testes fictícios.

Entretanto, a programação dos módulos foi feita da forma mais clara possível, a fim de possibilitar sua manutenção, mesmo para quem nunca teve contato com o mesmo. Existem, é claro, pontos inerentemente complexos, como o controle de erro e o fluxo de dados do serviço de comunicação em grupo, mas a descrição dos serviços neste trabalho mais a documentação nos próprios módulos são suficientes para o seu entendimento.

Acredita-se que o objetivo estabelecido no início deste trabalho foi alcançado e que os resultados obtidos atendem as necessidades da plataforma *multiware*.

Apêndice A

Manual da Plataforma de Comunicação

A.1 Introdução

Este guia descreve como utilizar os serviços de comunicação da plataforma *multi-ware*. Os serviços providos são:

- comunicação com reserva de banda;
- comunicação em grupo;
- comunicação sem-conexão ponto-a-ponto confiável.

O acesso aos arquivos de *include*, bibliotecas e executáveis, bem como a compilação e link-edição são apresentados na seção A.2.

Para acessar as funções dos protocolos, é necessário utilizar uma interface de programação baseada em classes C++. Esta interface é descrita na seção a A.3.

Exemplos simples de como utilizar a interface de programação são apresentados na seção A.5.

A.2 Compilação, Link-edição e Execução

Para compilar um módulo em C++ que utiliza o protocolo, siga os seguintes passos:

1. Inclua o arquivo `conn.h`.
   Os arquivos de *include* se encontram em `/proj/multimedia/LTP/include`. Coloque no comando de compilação do seu programa a opção `-I/proj/multimedia/LTP/include`.
2. Compile e depois faca a link-edição com a biblioteca `ltp`.
   O arquivo está no diretorio `/proj/multimedia/LTP/lib/`*OS*, onde *OS* pode ser `AIX` ou `SUNOS` conforme for compilar nas IBMs ou nas SUNS. Coloque no comando de link-edição a opção `-L/proj/multimedia/LTP/lib/`*OS* e `-lltp`.
3. Execute o programa `ltp` em cada estação.
   O arquivo executável esta em `/proj/multimedia/LTP/bin/`*OS*. Se for utilizar freqüentemente o protocolo, inclua a seguinte no ao fim do seu arquivo .cshrc:
   `set path = ( $path /proj/multimedia/LTP/bin/‘uname | dd conv=ucase‘ )`
4. Execute o seu programa(s) nas estações onde esta rodando `ltp`.

Os protocolos executam somente numa sub-rede, portanto eles não fazem roteamento. Por problemas de configuração que serão solucionados num futuro próximo, o protocolo não funciona nas estações que possuem duas interfaces de rede (as estações roteadoras).

Este protocolo está em fase de testes, portanto podem haver alterações na implementação e (pouco provavelmente) na interface, e novos executáveis e biblioteca de funções.

O programa `ltp` emite várias mensagens utilizadas para sua depuração. Redirecione a saída do programa para `/dev/null` se não quiser acompanhar as mensagens.

A.3 Interface de Programação

A interface de programação para a utilização dos servimcos são feitas com classes C++. A classe `conn` (figura A.1) é base para as classes `conn_ru`, `conn_rm` e `conn_br_send` e `conn_br_recv` (serviços de comunicação em grupo, sem-conexão ponto-a-ponto confiável e com reserva de banda, respectivamente). Nela estão definidos os métodos `receive`, `close`, `has_data`, `flush` e `port_id` que realizam funções idênticas nos três serviços (figura A.1). Os métodos `open` e `send` são específicos para cada serviço.

Também nesta classe estão definidas as variáveis estáticas `error` e `error_str` e as

funções `init` e `select`. A seção A.3.1 descreve os tipos de erros que podem ocorrer e as variáveis `error` e `error_str`. A seçao A.3.2 descreve a inicialização para acesso aos serviços e a função `init`. E a seção A.3.3 descreve como utilizar a função `select`.

```
class conn {
public:
  conn();
  ~conn();

  int receive(sys_addr& from_host,int& from_port,int& len,char* data,
              const sys_time timeout=sys_time::MAX);
  int close();
  int has_data();
  int port_id();
  int flush();

  static int select(int& len,class _conn* conn_list[],
                    const sys_time timeout=sys_time::MAX);

  static int init();

  static int error;
  static char* error_str[];
};
```

Figura A.1: Classe `conn`.

A.3.1 Erros

Todas as funções retornam zero em caso de sucesso ou um valor diferente de zero em caso de erro (salvo algumas exceções). Em caso de erro, a variável global `error` indica o tipo de erro ocorrido com a última operação da interface. A variável `error_str` é uma matriz com a descrição dos erros para o programa apresentar uma mensagem de erro. A variável `error` serve como indexador para acessar a descrição do último erro.

A forma para acessar as variáveis é `conn::error` e `conn::error_str`.

A.3.2 Inicialização

Antes de utilizar qualquer classe ou função dos serviços, deve-se inicializar o módulo chamando a função `init` na forma `conn::init();`

A função `init` estabelece a comunicação entre o processo-aplicação e o processo-serviços. Se a comunicação for estabelecida, a função retorna valor 0. Em caso de erro retorna um valor diferente de zero e a variável `error` assume os seguintes valores:

`E_NOPROTO` o processo-serviços não está rodando neste nodo;

`E_NMSGQ` não foi possível criar uma fila de mensagens para se comunicar com o processo protocolo;

`E_MPROC` não é possível incluir mais este processo.

No caso de erro, nenhuma outra função que utilize o protocolo pode ser chamada. Um exemplo para realizar a inicialização é apresentado na figura A.2.

```
if(conn::init()){
  cout << "conn::init falhou:" << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.2: Função `conn_init`.

A.3.3 Função `select`

A função `select` permite aguardar entre várias conexões a primeira que dispor de dados para receber. O número máximo de conexões que podem ser testadas numa chamada de `select` é `MAX_PORT_SEL`. O parâmetro `len` deve ser inicializado com o número de conexões a serem testadas. A matriz `port_id_list` deve ser inicializada com o ponteiro do objeto de cada conexão a ser testada. No retorno da função, `len` é alterado para o número de conexões prontas para realizar recepção de mensagens e a matriz recebe somente os ponteiros das conexões prontas para receber mensagens. É possível especificar um *timeout* para a função.

Podem ocorrer os seguintes valores de erro para a função `select`:

`E_TIMEOUT` se o *timeout* for atingido;

`E_MPORT` se o número de conexoes especificadas em `len` for maior que o permitido;

`E_NOPEN` se alguma conexão não estiver aberta;

`E_NOWN` se alguma conexão não pertencer ao processo;

`E_IPORT` se alguma conexão não pode ser utilizada para recepção de mensagens.

A.3.4 Métodos

O método `open` cria uma porta e permite o acesso ao serviço desejado. Cada serviço possuí uma forma específica para realizar a conexão. O método implementado nos três serviços pode retornar os seguintes erros:

`E_AOPEN` se a conexão já estiver aberta;

`E_UPORT` se a porta requisitada não estiver disponível;

`E_IPORT` se o identificador de porta for inválido (quando informado).

Todos os métodos (exceto `open`) podem ocorrer os seguintes erros:

`E_NOPEN` se a conexão ainda não foi aberta;

`E_IPORT` se a conexão não existe mais (provavelmente foi encerrada assincronamente);

`E_NOWN` se a conexão pertence à outro processo.

O método `send` também é específico de cada classe é descrito nas seções seguintes.

Os métodos descritos a seguir são comuns à todas as classes.

O método `has_data` informa se a conexão possui dados para receber. Em caso de sucesso retorna 1 se há mensagens para serem recebidas, zero se não há mensagens para serem recebidas ou -1 no caso de erro. A variável `error` recebe os seguintes valores:

`E_IPORT` se a conexão não existe (a conexão foi desfeita assicronamente);

`E_NOWN` se o processo não é o dono da conexão.

O método `port_id` retorna o identificador da porta utilizada pela conexão. Em caso de sucesso retorna o identificador da porta (um número maior que zero) ou -1 no caso de erro.

O método `flush` retira da fila de recepção de mensagens da conexão todas as mensagens disponíveis para recepção. A variável `error` recebe o valor `E_NORECV` se a conexão não permite recepção de mensagens.

O método `close` encerra a utilização da conexão e permite que a mesma possa novamente ser aberta.

O método `receive` recebe uma mensagem da conexão. Pode-se especificar um tempo de *timeout* para a espera por uma mensagem. Este método não é aplicável para conexões das classes `conn_br_send` e `conn_rm` no modo `RM_SENDER`. Neste caso o método retorna erro e `error` recebe `E_NORECV`.

A.3.5 Serviço de Comunicação Sem-conexão Ponto-a-ponto Confiável

A classe `conn_ru` (Figura A.3) implementa características específicas para o serviço nos métodos `open` e `send`.

```
class ru_conn : public _conn {
public:
  int open(const int port_id=0);
  int send(const sys_addr to_host,const int to_port,const int len,
           const char *data,const sys_time& timeout=sys_time::MAX);
};
```

Figura A.3: Classe `conn_ru`.

O método `open` permite a conexão ser utilizada e a associa à um identificador de porta. Se a aplicação não necessitar de uma porta específica, pode ser omitido e o protocolo irá escolher uma porta livre.

O método `send` envia dados pela conexão para outro nodo especificado em `to_host` e porta `to_port`. Este método somente retorna quando quando a aplicação receber a mensagem através de `receive` ou quando alcançar o *timeout* (se for especificado). Este método pode retornar erro com os seguintes valores para `error`:

`E_IPORT` se o identificador da porta destino não for válido

`E_TIMEOUT` se o *timeout* foi atingido

A.3.6 Serviço de Comunicação em Grupo

A classe `conn_rm` (Figura A.4) implementa características específicas para o serviços nos métodos `open` e `send` e fornece o método `group_id`.

```
enum rm_port_mode { RM_RECEIVER, RM_SENDER, RM_SENDER_RECEIVER };
class rm_conn : public _conn {
public:
  int open(const int group_id,const rm_port_mode mode=RM_SENDER_RECEIVER);
  int send(const int len,const char *data);
  int group_id();
};
```

Figura A.4: Classe `conn_rm`.

O método `open` permite a conexão ser utilizada e a associa à um grupo. O parâmetro `mode` permite definir o modo de operação da conexão. É recomendável especificar o modo, pois se uma conexão for do tipo receptora e suas mensagens não forem recebidas, a fila de recepção ficará lotada e o protocolo irá bloquear até serem liberadas. Se o identificador de grupo especificado for inválido, `error` recebe `E_IGROUP` e é retornado erro.

O método `send` serve para o envio de mensagens pela conexão para todo as demais conexões do grupo (a mensagem não retorna para a própria conexão).

O método `group_id` informa a qual grupo pertence a conexão.

A.3.7 Serviço de Comunicação com Reserva de Banda

A interface de programação são duas classes C++, uma para envio e outra para recepção de mensagens. Um objeto da classe `conn_br_send` deve ser instanciado para realizar a reserva de banda e envio de dados. Um objeto da classe `conn_br_recv` deve ser instanciado para estabelecer uma conexão e receber dados.

```
class br_send_conn : public _conn {
public:
  int open(const int data_size,const sys_time period,
           const sys_time delay,const int port_id=0);
  int send(const int len,const char *data);
};
```

Figura A.5: Classe `conn_br_send`

Na classe `conn_br_send` (figura A.5), o método `open` realiza a reserva de banda e associa um identificador de porta à conexão. Se a aplicação não necessitar de uma porta específica, pode ser omitido e o serviço irá escolher uma porta livre. Caso não seja possível alocar a banda passante desejada, o método retorna em `error` o valor `E_NOBWA`.

A banda passante é liberada quando o método `close` é invocado para a conexão.

O método `send` envia uma mensagem para os nodos conectados à porta. Se nenhum outra porta está conectada à esta porta, nenhuma mensagem é transmitida efetivamente.

Os métodos `flush` e `has_data` não tem efeito nesta classe. Uma conexão desta classe não pode ser utilizada na função `select`.

```
class br_recv_conn : public _conn {
public:
  int open(const sys_addr remote_host,const int remote_port,
           const sys_time& timeout=sys_time::MAX);
};
```

Figura A.6: Classe `conn_br_recv`

Na classe `conn_br_recv` (figura A.6), o método `open` realiza a conexão à uma porta remota e passa a receber as mensagens. O parâmetro de `timeout` para conexão é opcional. O método pode retornar os seguintes valores em `error`:

`E_TIMEOUT` se não foi possível estabelecer a conexão no período desejado

`E_NCONN` se a conexão transmissora não existe

No método `receive` pode ser retornado o erro `E_CPORT` se a conexão emissora for fechada. Este mesmo erro pode ser retornado se esta conexão for a única restante na função `select`.

A.4 Exemplos de Utilização da Plataforma

Em cada **módulo** de programa em C++ (arquivo fonte) deve ser incluído o arquivo com a definição das classes de acesso aos protocolos (Figura A.7).

```
#include <conn.h>
```

Figura A.7: Inclusão do arquivo de definições

Em cada **processo**, antes de utilizar qualquer classe ou função dos protocolos, deve-se inicializar o módulo chamando a função `init` na forma `conn::init();`.

A função `init` estabelece a comunicação entre o processo da aplicação e o processo do protocolo. Se a comunicação for estabelecida, a função retorna valor 0. Em caso de erro, retorna um valor diferente de zero e nenhuma outra função que utilize o protocolo pode ser chamada. Um exemplo para realizar a inicialização é apresentado na figura A.8.

```
if(conn::init()){
  cout << "conn::init falhou:" << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.8: Exemplo de inicialização do protocolo

Uma vez inicializado o protocolo, pode-se instanciar e chamar os métodos das classes que implementam as portas. O método `open` é o que deve ser chamado primeiro. O método `open` permite que a conexão seja utilizada e a associa a um identificador de porta. Se a aplicação não necessitar de uma porta específica, pode ser omitido o parâmetro de

identificador de porta e o protocolo irá escolher uma porta livre. Para cada protocolo, o método open varia os parâmetros.

Um exemplo para cada protocolo do método open é apresentado nas figuras A.9, A.10, A.11 e A.12.

Na figura A.9 é aberta a porta 123 onde é realizada reserva de banda para 10KB a cada 100ms, isto é, 100KBps, com um retardo máximo de 200ms. Se a reserva de banda não for atendida, isto é, a função retornar um valor diferente de zero, é apresentada uma mensagem de erro e o programa termina.

```
conn_br_send c;
if(c.open(10000,sys_time(.1),sys_time(0.2),123)){
  cerr << "port: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.9: Exemplo para abrir uma porta no modo emissor do protocolo de reserva de banda

Na figura A.10 é aberta uma porta para se conectar à porta 123, com reserva de banda no nodo "ubatuba". Um *timeout* de 2 segundos foi determinado para esperar pela conexão se estabelecer. Se a conexão com a porta emissora não for possível, isto é, se ocorrer o *timeout* e a função retornar um valor diferente de zero, é apresentado uma mensagem de erro e o programa termina.

```
sys_addr from_host("ubatuba");

conn_br_recv c;
if(c.open(from_host,123,sys_time(2.0))){
  cerr << "port: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.10: Exemplo para abrir uma porta no modo receptor do protocolo de reserva de banda

Na figura A.11 é aberta uma porta para se comunicar com o grupo 1. Esta porta pode tanto enviar como receber mensagens do grupo. Se a conexão com o grupo não for possível, isto é, se a função retornar um valor diferente de zero, é apresentado uma mensagem de erro e o programa termina.

```
conn_rm c;
if(c.open(1,RM_SEND_RECEIVER)){
  cerr << "group 1: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.11: Exemplo para abrir uma porta do protocolo de Difusão confiável

Na figura A.11 é aberta a porta 1 para comunicação sem-conexão. A porta está pronta para enviar e receber dados de qualquer outra porta deste protocolo. Se a abertura da porta não for possível (por exemplo, se o identificador de porta já estiver em uso), isto é, se a função retornar um valor diferente de zero, é apresentado uma mensagem de erro e o programa termina.

```
conn_ru c;
if(c.open(12)){
  cerr << "port 12: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.12: Exemplo para abrir uma porta do protocolo Sem-conexão

Para enviar dados pelas portas, é utilizada o método `send`. Uma porta no modo receptor do serviço de reserva de banda ou do serviço de Difusão Confiável não pode realizar o envio de mensagens.

Cada protocolo exige parâmetros diferentes para o envio de mensagens. As figuras A.13, A.14 e A.15 apresentam exemplos de programação para cada protocolo. Em todos os exemplos, caso a função retorne um valor diferente de zero, significa que ocorreu um erro.

Na figura A.13 é realizado o envio de dados para uma porta do protocolo de reserva de banda. No caso de erro, uma mensagem é apresentada e o programa termina.

```
char data[1000];

if(c.send(1000,data)){
  cerr << "port: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.13: Exemplo de envio de dados de uma porta do protocolo reserva de banda.

Na figura A.14 é realizado o envio de dados para uma porta do protocolo de Difusão Confiável. No caso de erro, uma mensagem é apresentada e o programa termina.

```
char data[30];

if(c.send(20,data)){
  cerr << "group 1: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
```

Figura A.14: Exemplo de envio de dados de uma porta do protocolo Difusão Confiável.

No exemplo da figura A.15 é realizado o envio de dados para uma porta do protocolo sem-conexão. No caso de erro, é verificado se houve *timeout* através da variável `conn::error`. Se o valor desta variável for `E_TIMEOUT`, significa que a porta destino (porta 15 no nodo "parati") não confirmou a recepção dos dados dentro dos 2 segundos de *timeout* dado. Neste caso, é chamado a função `port_error()` para tratar o erro. Caso o valor não seja `E_TIMEOUT`, uma mensagem de erro é apresentada e o programa termina.

```
sys_addr dest("parati");
int data=40;

if(c.send(dest,15,sizeof(data),&data,sys_time::now()+sys_time(2.0))){
  if(conn::error==E_TIMEOUT){
    port_error();
  }else{
    cerr << "port: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
}
```

Figura A.15: Exemplo de envio de dados de uma porta do protocolo sem-conexão Confiável.

O método `receive` é comum para todos os serviços. Na figura A.16 é apresentado um exemplo de recepção de dados. Se dentro de 10 segundos a porta não receber uma mensagem (`conn::error` igual a `E_TIMEOUT`), a função `receive_timeout()` é chamada. Se ocorrer outro tipo de erro, uma mensagem é apresentada e o programa termina. Se a recepção ocorrer normalmente, o identificador da porta e endereço do nodo estarão armazenados em `from_port` e `from_host`, respectivamente. Os dados recebidos estarão armazenados em `msg` com o tamanho dos dados em `len`. Sendo que o tamanho máximo que a recepção comporta é 100 bytes. Se o tamanho da mensagem recebida for maior que aquele especificado em `len` (100 bytes), a mensagem é truncada no centésimo byte.

```
char msg[100];
int len=sizeof(msg);
int from_port;
sys_addr from_host;
sys_time timeout=now()+sys_time(10.0);

if(c.receive(from_host,from_port,len,msg,timeout)){
  if(conn::error==E_TIMEOUT){
    receive_timeout();
  }else{
    cerr << "receive error: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
}
```

Figura A.16: Exemplo de recepção de dados de uma porta

Na figura A.17 é apresentado um exemplo da função `select` com portas de todos os protocolos. As portas a serem testadas são especificadas no vetor `s`. As variáveis `a`, `b`, `c` e `d` são portas de qualquer um dos protocolos. No vetor é armazenado o ponteiro de cada uma destas portas. A variável `n` possui o número inicial de portas no vetor. Depois da chamada da função `select` é alterada para o número de portas restantes no vetor. A variável `timeout` especifica o tempo máximo que a função `select` ficará bloqueada. Se a função `select` terminar por *timeout*, a função `select_timeout()` é chamada. Se ocorrer um erro diferente de *timeout*, uma mensagem de erro é apresentada e o programa termina.

Se a função `select` retornar com sucesso, no vetor `s` estará o ponteiro para as portas com dados para receber, e número de portas no vetor estará em `n`. De cada porta podem ser recebidos os dados e tratados pela função `handle_msg()`. Se a função `receive` retornar um erro, uma mensagem é apresentada e o programa termina.

```
int n=4;
conn* s[2]={&a,&b,&c,&d};
sys_time timeout=now()+sys_time(10.0);

if(conn::select(n,s)){
  if(conn::error==E_TIMEOUT){
    select_timeout();
  }else{
    cerr << "select error: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
}else{

  for(int i=0;i<n;i++){

    char msg[100];
    int len=100;
    int from_port;
    sys_addr from_host;

    if(s[i]->receive(from_host,from_port,len,msg)){
      cerr << "port : " << s[i]->port_id()
           << conn::error_str[conn::error] << "\n";
      exit(1);
    }
    handle_msg(msg,len);
  }
}
```

Figura A.17: Exemplo da função `select`

Na figura A.18 é apresentado um exemplo da função `has_data`. Esta função se aplica à uma porta de qualquer um dos serviços, que possa receber dados. A variável `a` é a instanciação de uma porta de um dos serviços. O método `has_data` é chamado e o resultado é armazendo na variável `has`. Se ocorrer algum erro, uma mensagem de erro é apresentada e o programa termina. Se houver uma mensagem para ser recebida por `a`, o método `receive` é chamado e os dados recebidos são tratados pela função `handle_data()`. Neste caso, não é testado nenhum erro do método `receive`. Se ocorrer algum erro, o programa terá os dados na variável `msg` incorretos.

```
int has=a.has_data();
if(has==-1){
  cerr << "has_data error: " << conn::error_str[conn::error] << "\n";
  exit(1);
}
if(has){
  char msg[100];
  int len=100;
  int from_port;
  sys_addr from_host;

  a.receive(from_host,from_port,len,msg);
  handle_data(msg);
}
```

Figura A.18: Exemplo de teste de mensagem recebida em uma porta

Na figura A.19 está um exemplo do método `flush`. O objetivo, neste caso, é limpar o *buffer* de mensagens recebidas, para então receber a próxima mensagem. Neste exemplo, o método `receive` é chamado sem o parâmetro de *timeout*. Isto significa que o processo ficará esperando indefinidamente por uma mensagem na porta. Também não é considerado nenhum caso de erro.

```
a.flush();

char msg[100];
int len=sizeof(msg);
int from_port;
sys_addr from_host;

c.receive(from_host,from_port,len,msg);
```

Figura A.19: Exemplo do método `flush`

A.5 Programas Exemplos

A.5.1 Classe conn_ru

Na figura A.20 é apresentado um programa que recebe mensagens *string* e na figura A.21 um programa que envia uma mensagem *string*.

```
#include <conn.h>
#include <sys_addr.h>
#include <iostream.h>
#include <stdlib.h>

main(){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_ru c;
  if(c.open(123)){
    cerr << "port 123: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  while(1){
    char msg[100];
    int len=100;
    int from_port;
    sys_addr from_host;
    if(c.receive(from_host,from_port,len,msg)){
      cerr << "port 123: " << conn::error_str[conn::error] << "\n";
      exit(1);
    }
    cout << "from " << from_host.name() << ", port" << from_port
         << ":" << msg << "\n" << flush;
  }
}
```

Figura A.20: Programa exemplo da classe conn_ru

```
#include <conn.h>
#include <sys_addr.h>
#include <sys_time.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

main(int argc,char** argv){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_ru c;
  if(c.open()){
    cerr << "port: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  sys_addr to_host(argv[1]);
  if(c.send(to_host,123,strlen(argv[2])+1,argv[2],
            sys_time::now()+sys_time(2.0))){
    cerr << "port: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
}
```

Figura A.21: Programa exemplo da classe `conn_ru`

A.5.2 Classe `conn_rm`

Na figura A.22 é apresentado um programa que recebe mensagens *string* e na figura A.23 um programa que envia uma mensagem *string*.

```
#include <conn.h>
#include <sys_addr.h>
#include <iostream.h>
#include <stdlib.h>

main(){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_rm c;
  if(c.open(1,RM_RECEIVER)){
    cerr << "group 1: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  while(1){
    char msg[100];
    int len=100;
    int from_port;
    sys_addr from_host;
    if(c.receive(from_host,from_port,len,msg)){
      cerr << "group 1: " << conn::error_str[conn::error] << "\n";
      exit(1);
    }
    cout << "from " << from_host.name() << ", port" << from_port
         << ":" << msg << "\n";
  }
}
```

Figura A.22: Programa exemplo da classe conn_rm

```
#include <conn.h>
#include <sys_addr.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>

main(int argc,char** argv){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_rm c;
  if(c.open(1,RM_SENDER)){
    cerr << "group 1: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  if(c.send(strlen(argv[1])+1,argv[1])){
    cerr << "group 1: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
}
```

Figura A.23: Programa exemplo da classe `conn_rm`

A.5.3 Classes `conn_br_send` e `conn_br_recv`

Na figura A.24 é apresentado um programa que recebe mensagens *string* e na figura A.25 um programa que envia uma mensagem *string*.

```
#include <conn.h>
#include <sys_addr.h>
#include <sys_time.h>
#include <iostream.h>
#include <stdlib.h>

main(int argc,char** argv){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  sys_addr from_host(argv[1]);

  conn_br_recv c;
  if(c.open(from_host,123,sys_time(2.0))){
    cerr << "port: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  while(1){
    char msg[100];
    int len=100;
    int from_port;
    sys_addr from_host;
    if(c.receive(from_host,from_port,len,msg)){
      cerr << "port: " << conn::error_str[conn::error] << "\n";
      exit(1);
    }
    cout << msg << "\n";
  }
}
```

Figura A.24: Programa exemplo da classe `conn_br_recv`

```
#include <conn.h>
#include <sys_addr.h>
#include <sys_time.h>
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

main(){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_br_send c;
  if(c.open(10000,sys_time(.1),sys_time(1.0))){
    cerr << "port: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  for(int n=0;1;n++){
    char msg[100];
    sprintf(msg,"mensagem %1",n);
    if(c.send(strlen(msg)+1,msg)){
      cerr << "port: " << conn::error_str[conn::error] << "\n";
      exit(1);
    }
  }
}
```

Figura A.25: Programa exemplo da classe `conn_br_send`

A.5.4 Função `select`

Na figura A.26 é apresentado um programa que recebe mensagens *string* de 4 conexões.

```
#include <conn.h>
#include <sys_addr.h>
#include <sys_time.h>
#include <iostream.h>
#include <stdlib.h>

main(){
  if(conn::init()){
    cerr << "cannot init conn: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  conn_ru c;
  if(c.open(123)){
    cerr << "port 123: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }
  conn_rm d;
  if(d.open(1)){
    cerr << "group 1: " << conn::error_str[conn::error] << "\n";
    exit(1);
  }

  while(1){
    int n=2;
    conn* s[2]={&c,&d};

    conn::select(n,s);
    for(int i=0;i<n;i++){
      char msg[100];
      int len=100;
      int from_port;
      sys_addr from_host;
      if(s[i]->receive(from_host,from_port,len,msg)){
        cerr << "port : " << s[i]->port_id()
             << conn::error_str[conn::error] << "\n";
        exit(1);
      }
      cout << "in port " << s[i]->port_id() << ", from host "
           << from_host.name() << ", port" << from_port << ":" << msg
           << "\n";
    }
  }
}
```

Figura A.26: Programa exemplo da função `select`

Apêndice B

Descrição do Formato das Mensagens

B.1 Protocolo de Reserva de Banda

- TOKEN: utilizada para um nodo passar o direito de transmissão de dados para outro nodo.

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser TOKEN.
band_rem	2	inteiro	Banda passante remanescente para alocação dinâmica (em Kbytes/s).
host_n	2	inteiro	Número de nodos na lista de nodos do anel lógico.
host_list { host_id bwidth }	host_n × 8	vetor { inteiro inteiro }	Lista dos nodos no anel lógico. Em cada item há um identificador de nodo e a quantidade de banda passante reservada para o nodo.

- TOKEN_REQ: utilizada por um nodo para entrar no anel lógico.

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser TOKEN_REQ.

- TOKEN_INIT: utilizada pelo primeiro nodo que detectou a falta do anel lógico.

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser TOKEN_INIT

B.2 Protocolo de Conexão de Portas

- `CONNECT`: utilizada por uma conexão para se conectar à uma porta e passar a receber seus dados.

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `CONNECT`.
`to_port`	2	inteiro	Identificador da porta emissora.
`from_port`	2	inteiro	Identificador da porta receptora.

- `CONNECT_ACK`: reconhecimento a mensagem `CONNECT`. O nodo que envia os dados informa quais os parâmetros utilizados na reserva de banda.

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `CONNECT_ACK`.
`to_port`	2	inteiro	Identificador da porta emissora.
`from_port`	2	inteiro	Identificador da porta receptora.
`data_size`	2	inteiro	Tamanho máximo das mensagens enviadas.
`period`	4	inteiro	Período de envio de mensagens em microsegundos.
`delay`	4	inteiro	Atraso máximo de mensagens em microsegundos.

- `DISCONNECT`: utilizada por uma conexão para se desconectar de uma porta.

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `DISCONNECT`.
`to_port`	2	inteiro	Identificador da porta emissora.
`from_port`	2	inteiro	Identificador da porta receptora.

- `DISCONNECT_ACK`: reconhecimento à mensagem `DISCONNECT`.

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `DISCONNECT_ACK`.
`to_port`	2	inteiro	Identificador da porta emissora.
`from_port`	2	inteiro	Identificador da porta receptora.

- `DATA`: utilizada para transferência de dados.

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `DATA`.
`from_port`	2	inteiro	Identificador da porta emissora.
`data_len`	2	inteiro	Comprimento do campo `data`.
`data`	máx. 1024	byte	Dados da mensagem.

B.3 Protocolo de Comunicação em Grupo

- `JOIN`: utilizada para solicitar a adesão ao grupo (cli→seq)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `JOIN`.
`group_id`	2	inteiro	Identificador do grupo.
`llocal`	4	inteiro	Identificador da próxima mensagem local que será enviada.

- `JOIN_ACK`: utilizada para confirmar a adesão ao grupo (seq→cli)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `JOIN_ACK`.
`group_id`	2	inteiro	Identificador do grupo.
`nglobal`	4	inteiro	Número da próxima mensagem global do grupo que o sequenciador irá enviar.

- `UNJOIN`: utilizada para solicitar a saída do grupo (cli→seq)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `UNJOIN`.
`group_id`	2	inteiro	Identificador do grupo.

- `DATA`: utilizada para solicitar um *multicast* (cli→seq)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `DATA`.
`group_id`	2	inteiro	Identificador do grupo.
`msg_id`	2	inteiro	Identificador local da mensagem.
`nglobal`	4	inteiro	Número da próxima mensagem global do grupo que o sequenciador irá enviar.
`data_len`	2	inteiro	Comprimento do campo de dados.
`data`	`data_len`	byte	Dados da mensagem.

- BROADCAST: utilizada para realizar o *multicast* (seq→cli)

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser BROADCAST.
group_id	2	inteiro	Identificador do grupo.
msg_id	2	inteiro	Identificador local da mensagem.
from	4	inteiro	Identificador do nodo cliente que solicitou o *multicast*.
local	4	inteiro	Identificador da mensagem local à qual corresponde esta mensagem global (serve para o cliente identificar as mensagens que enviou).
last	4	inteiro	Identificador da próxima mensagem global que o sequenciador espera confirmação de todos os nodos.
data_len	2	inteiro	Comprimento do campo de dados.
data	data_len	byte	Dados da mensagem.

- RET_DATA: utilizada para solicitar a retransmissão de mensagens DATA (seq→cli)

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser RET_DATA.
group_id	2	inteiro	Identificador do grupo.
nlocal	4	inteiro	Identificador da próxima mensagem local a ser recebida.
list_len	2	inteiro	Número de identificadores de mensagens na lista.
msg_id_list	list_len × 4	vetor de inteiros	Lista com o identificador da mensagens a serem retransmitidas.

- RET_BROAD: utilizada para solicitar a retransmissão de mensagens BROADCAST (cli→seq)

Campo	Tamanho	Tipo	Descrição
type	1	inteiro	tipo deve ser RET_BROAD.
group_id	2	inteiro	Identificador do grupo.
nglobal	4	inteiro	Identificador da próxima mensagem global que o cliente espera receber (*piggyback ack*).
list_len	2	inteiro	Número de identificadores de mensagens na lista.
msg_id_list	list_len × 4	vetor de inteiros	Lista com o identificador da mensagens a serem retransmitidas.

- `ACK`: utilizada para o reconhecimento explícito de mensagens `BROADCAST` (cli→seq)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `RET_BROAD`.
`group_id`	2	inteiro	Identificador do grupo.
`nglobal`	4	inteiro	Identificador da próxima mensagem global que o cliente espera receber (*piggyback ack*).

- `REFORM`: utilizada para iniciar a fase de reforma (cli→cli)

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `RET_BROAD`.
`group_id`	2	inteiro	Identificador do grupo.
`nglobal`	4	inteiro	Identificador da próxima mensagem global que o cliente espera receber (*piggyback ack*).
`list_len`	2	inteiro	Número de identificadores de mensagens na lista.
`msg_id_list`	`list_len` × 4	vetor de inteiros	Lista com o identificador da mensagens a serem retransmitidas.

B.4 Protocolo Sem-conexão Ponto-a-ponto Confiável

- `DATA`: utilizada para enviar dados

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `DATA`.
`from_port`	2	inteiro	Identificador da porta origem.
`to_port`	2	inteiro	Identificador da porta destino.
`msg_id`	4	inteiro	Identificador da mensagem.
`csend`	4	inteiro	Próxima mensagem que nodo emissor espera alguma confirmação (mensagens anteriores já foram confirmadas ou descartadas).
`data_len`	2	inteiro	Tamanho do campo `data`.
`data`	`data_len`	byte	Campo de dados.

- `ACK`: utilizada para confirmar o recebimento dos dados

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `ACK`.
`from_port`	2	inteiro	Identificador da porta origem.
`to_port`	2	inteiro	Identificador da porta destino.
`msg_id`	4	inteiro	Identificador da mensagem.

- `RESET`: utilizada para reinicializar os contadores de mensagens entre dois nodos

Campo	Tamanho	Tipo	Descrição
`type`	1	inteiro	tipo deve ser `RESET`.
`nsend`	4	inteiro	Próximo identificador que o nodo irá utilizar para enviar uma mensagem `DATA`.
`nrecv`	4	inteiro	Próximo identificador que o nodo espera receber do nodo remoto.

Referências Bibliográficas

[1] Manuel Jesus Mendes, Eleri Cardizo, and Maurício Magalh aes. Plataforma multiware: Serviços ODP – relatório parcial de atividades. Relatório técnico, UNICAMP, FEE, DCA, Abril 1995.

[2] ISO. *RM-ODP - Part 1: Overview and Guide to Use.*

[3] Tetsu Gunji. Plataforma multiware: Suporte à objetos em tempo de execução. Dissertação de Mestrado, UNICAMP, FEE, Agosto 1995.

[4] Márcio Maezi. Plataforma multiware: Interface de programação. Dissertação de Mestrado, UNICAMP, FEE, Agosto 1995.

[5] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, 2nd edition, 1989.

[6] Luiz Fernando Gomes Soares, Luiz Tucherma, Marco Antonio Casanova, and Paulo Roberto Rosa Lopes Nunes. *Fundamentos de Sistemas Multimídia.* VIII Escola de Computação – Gramado, RS. Instituto de Informática/UFRGS, 1992.

[7] Doung Shepherd, David Hutchison, Francisco Garcia, and Geoff Coulson. Protocol support for distributed multiemdia applications. *Computer Communications*, 15(6), Julho 1992.

[8] Bernd Wolfinger and Mark Moran. A continuos media data transport scervice and protocol for real-time communication in high speed networks. Relatório técnico, University of California, Berkeley.

[9] T. D. C. Little and A. Ghafor. Sheduling of bandwidth-constrained multimedia traffic. *Computer COmmunications*, 15(6):381–387, Agosto 1992.

[10] David Anderson, Ralf Guido Herrtwich, and Carl Schaefer. Srp: A resource reservation protocolo for guaranteed performance communication in the internet. Relatório técnico.

[11] MICHEL RAYNAL. *Distributed Algorithms and Protocols.* John Wiley, 1988.

[12] VACHASATHI P. KOMPLEZLA, JOSEPH C. PASQUALE, AND GEORGE C. POLYZOS. Multicast routing for multimedia communication. *Computer COmmunications*, 1(3):286–292, Junho 1993.

[13] ISRAEL CIDON, INDER S. GOPAL, AND ADRIAN SEGALL. Connection establishment in high-speed networks. *IEEE/ACM Transactions on Networking*, 1(4):469–481, Agosto 1993.

[14] DOMENICO FERRARI AND DINESH C. VERMA. A scheme for real-time channel establischment in wide-area networks. *IEEE Journal on Selected Areas in Communications*, 8(3):368–379, Abril 1990.

[15] ALFRED C. WEAVER. Xpress Transport Protocol version 4. Relatório técnico, University of Virginia – Department of Computer Science.

[16] CLEMENS SZYPERSKI AND GIORGIO VENTRE. Effecient multicasting for interactive multimedia applications. Relatório técnico, Março 1993.

[17] LUCA DELGROSSI, RALF GUIDO HERRWICH, FRANK OLIVER HOFFMANN, AND SIBILLE SCHALLER. Receiver-initiated communication with st-ii. *Multimedia Systems*, (2):141–149, 1994.

[18] K. W. TINDELL, A. BURNS, AND A. J. WELLINGS. Guaranteeing hard real time end-to-end communications deadlines. Relatório técnico, University of York – Departament of Computer Science, 1993.

[19] EMMANUELLE ANCEAUME. A comparison of fault-tolerant atomic broadcast protocols. In *Proceedings of the 4th Workshop on Future Trends of Distributed Computing Systems*, pages 166–172, Lisbon, PT, Setembro 1993.

[20] JO-MEI CHANG AND N. F. MAXEMCHUK. Reliable broadcast protocols. *ACM Transactions on COmputer Systems*, 5(1):47–76, Fevereiro 1987.

[21] KENNETH P. BIRMAN AND THOMAS A. JOSEPH. Reliable communication in the presence of failures. *ACM Transactions on COmputer Systems*, 2(3):251–273, Agosto 1984.

[22] M. FRANS KAASHOEK AND ANDREW S. TANENBAUM. Efficient reliable group communication for distributed systems. Relatório técnico, 1992.