UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Vladimir Soares Catão

Um Serviço de Comunicação Síncrono para Nodos com Semântica de Falha Controlada Utilizando Redes Assíncronas

Dissertação apresentada ao curso de Mestrado em Informática da Universidade Federal da Paraíba, em cumprimento às exigências parciais para obtenção do grau de Mestre

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Sistemas Distribuídos

Orientador: Francisco Vilar Brasileiro

Campina Grande, janeiro de 1998

C357s Catão, Vladimir Soares.
Um serviço de comunicação síncrono para nodos com semântica de falha controlada utilizando redes assíncronas / Vladimir Soares Catão. - Campina Grande, 1998.
85 f.

Dissertação (Mestrado em Informática) - Universidade Federal da Paraíba, Centro de Ciências e Tecnologia, 1998.
"Orientação : Prof. Dr. Francisco Vilar Brasileiro".
Referências.

1. Sistemas Distribuídos. 2. Sistema Operacional - Software. 3. Tolerância à Falhas. 4. Transmissão de Mensagens. 5. Dissertação - Informática. I. Brasileiro, Francisco Vilar. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.75(043)

UM SERVIÇO DE COMUNICAÇÃO SÍNCRONO PARA NODOS COM SEMÂNTICA DE FALHA CONTROLADA UTILIZANDO REDES ASSÍNCRONAS

VLADIMIR SOARES CATÃO

DISSERTAÇÃO APROVADA EM 31 .12.1997

Francisco Vilar Brasileiro

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Presidente

PROF. JACQUES PHILIPPE SAUVÉ, Ph.D
Examinador

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Examinador

CAMPINA GRANDE - PB

Agradecimentos

Em primeiro lugar, agradeço à minha família pelo apoio e amor sempre presentes, em todas as situações. Meus pais, Álvaro e Cidinha, e meus irmãos Érika, Renata e Emerson. Os dois últimos, apesar da distância Campina-Brasília, sempre me deram a maior força para que levasse esse trabalho adiante.

Agradeço também ao meu orientador, Fubica, pela confiança e pragmatismo durante as produtivas discussões que tivemos. Tinha a impressão de que diante de qualquer impasse, ele sempre conseguia uma resposta convincente dentro dos próximos 10 segundos.

Ao pessoal da **Light-Infocon** (Alessandro, Adriano Lemos, Adriano Sérgio, Marcos e Katyusco), que participou indiretamente da evolução deste trabalho, discutindo, ou simplesmente ouvindo as idéias aqui presentes; especialmente a Marcos, pelo incentivo para que conseguisse terminar este trabalho o quanto antes. Devo agradecer ainda à diretoria da empresa, por me liberar sempre que necessário (e obviamente, sempre que possível) para me dedicar ao mestrado.

À turma do LSD (atenção: LSD = **L**aboratório de **S**istemas **D**istribuídos), que sempre esteve presente nas discussões deste trabalho, especialmente durante o *sprint* final para a escrita da dissertação. Particularmente agradeço a Érica, que compartilhou de praticamente todos os instantes desde que tudo começou.

Por último, mas não menos importante, agradeço a Ana, pelo apoio contínuo e pela compreensão durante os momentos roubados para que o trabalho seguisse adiante. Sem seu amor e carinho, tudo teria sido bem mais árduo e bem mais difícil de ser feito.

Resumo

A replicação de processos em processadores que falham de forma independente é uma abordagem bastante utilizada para tolerar faltas de processadores em um sistema distribuído. Em particular, essa abordagem é seguida pelo serviço de processamento confiável disponível no ambiente operacional **Seljuk-Amoeba**. No modelo de replicação, a ordenação das mensagens recebidas por cada réplica é normalmente implementada através de protocolos para disseminação atômica de mensagens, os quais supõem um serviço de comunicação síncrono, ou seja, um serviço que permite que se conheça a priori o atraso máximo para transmissões de mensagens entre quaisquer dois processos do sistema que estão executando segundo suas especificações. Neste trabalho, estudamos as dificuldades envolvidas no desenvolvimento de um serviço de comunicação síncrono implementado em *software*, utilizando redes locais convencionais (i.e. assíncronas). Para contornar tais dificuldades, propomos a simulação em *software*, dentro do núcleo do sistema operacional, de um método de acesso TDMA (*Time Division Multiple Access*) ao meio de comunicação, o qual permite que processos reservem para si parte do *slot* TDMA do processador em que executam. A proposta detalha ainda considerações sobre o escalonamento das tarefas dentro do sistema operacional de forma a garantir não só o atraso na transmissão das mensagens propriamente dito, mas também o atraso fim-a-fim para os protocolos de ordenação de mensagens do **Seljuk-Amoeba**, compreendendo desde o pedido de envio da mensagem até sua efetiva entrega no destino. Além disso, com o objetivo de evitar que os processos transmitam mais do que a reserva no *slot* TDMA pode comportar, sugerimos também uma forma de controlar o fluxo de mensagens desses processos.

Abstract

Replicated processing on independent processors is a common way to achieve fault-tolerant processing. It is the basis for the reliable processing service offered by the **Seljuk-Amoeba** operating environment. In order to assure that the replicated processes will achieve order and agreement on the input messages received by the replicas, replicated processing normally uses atomic broadcast protocols. These protocols assume that communication between any two operational processes is synchronous, i.e. there is a known finite time bound for message transmission between any two processes that are executing in accordance with their specification. In this work, we study the dificulties involved in the development of a synchronous communication service implemented in software, built on top of conventional asynchronous networks. In order to overcome these dificulties, we propose a simulation of a TDMA (*Time Division Multiple Access*) access method to the communication media, by which a process can reserve a portion of a TDMA slot exclusively for its use. The proposal also details the scheduling of the tasks in the operating system, necessary to assure a time limit not only for the transmission delay itself, but also to the end-to-end transmission delay for the message ordering protocols under **Seljuk-Amoeba**, accounting for the transmission request at the sending side until the proper delivery of the message at the destination. Besides, we also suggest a flow control mechanism, in order to guarantee that the process' transmission requests are compatible with the reservation made on the TDMA slot.

Sumário

Lista de Figuras

1.

Introdução

Ambientes de computação baseados em sistemas distribuídos vêm se tornando um paradigma cada vez mais popular atualmente. Paralelamente, à medida em que sua popularidade vem crescendo, vem aumentando também a nossa dependência com relação a sistemas desse tipo. Apesar de esses sistemas trazerem uma série de benefícios ao nosso dia-a-dia, agilizando processos, disseminando informações mais rápida e eficientemente, eles podem nos trazer também graves prejuízos. Para isso, basta que um serviço essencial não esteja disponível na hora em que for necessário, ou ainda que seu tempo de resposta seja excessivamente alto, o que pode determinar o mesmo efeito de uma indisponibilidade do serviço. Exemplos de serviços essenciais prestados por sistemas distribuídos temos os mais diversos; só para citar alguns: aplicações de controle de tráfego aéreo, controle de processos industriais, e mesmo aplicações de bancos de dados corporativos. Portanto, faz-se cada vez mais necessário que esses sistemas se previnam da ocorrência de faltas[1], sejam elas físicas (de equipamento, por exemplo) ou mesmo faltas da própria aplicação (faltas de projeto).

Assim, é preciso prover mecanismos que consigam lidar com o comportamento indesejado dos componentes do sistema, sejam eles de *hardware*, ou de *software*. Dessa forma, além de se preocupar com a funcionalidade dos sistemas que desenvolvem, os projetistas de tais sistemas teriam ainda a responsabilidade de adicionar a eles mecanismos para tolerância a faltas. O problema é que a inserção desses mecanismos aumenta consideravelmente a complexidade dos sistemas. Na verdade, a implementação desses mecanismos é diretamente afetada pela a semântica de falha que se supõe para os componentes do sistema: quanto menos restritiva a semântica de falha maior a complexidade para a implementação dos mecanismos para tolerância a faltas [Crist91].

[1] Faltas são as **causas** da falha. Na verdade, uma falta pode gerar um estado inconsistente no sistema (um erro), o qual pode determinar o comportamento incorreto do sistema como um todo (a falha). Essa notação, proposta em [LV91], é uma tradução para o português dos termos definidos por Laprie em seu artigo clássico [Lapri89].

Para diminuir essa complexidade, uma abordagem adotada por um grande número de sistemas distribuídos apresentados na literatura (ex. [Bartl81, KM85, BJ87, SDP91, Powel92]) é desenvolver os sistemas supondo que os componentes que compõem a infra-estrutura sobre a qual tais sistemas serão construídos, principalmente as unidades de processamento, possuem uma semântica de falha controlada (*fail-controlled* [Lapri89]), isto é, falham de uma forma previsível e bem conhecida, facilitando assim a construção de mecanismos para tolerância a faltas.

Alguns sistemas reportados na literatura foram construídos supondo que executam em unidades de processamento que possuem semântica de falha silenciosa (*fail-silent nodes*). Essas unidades asseguram que quando ocorre uma falha em algum componente, seu processamento pára, evitando que se realizem transições não especificadas. Outros sistemas, por sua vez, supõem que executam em unidades de processamento que nunca falham (*failure masking nodes*); essas unidades devem, portanto, apresentar esse comportamento desde que apenas um número máximo de seus componentes possa falhar durante todo o tempo em que a aplicação executará (tempo de missão).

Quando as unidades de processamento disponíveis não garantem, com probabilidade suficientemente alta, a semântica de falha necessária ao sistema, faz-se necessário construir unidades de processamento que através do uso de redundância apresentem a semântica requerida. Nodos replicados com semântica de falha controlada [SESTT92, BESST96] (ou simplesmente nodos replicados) são capazes de prover este serviço.

Um nodo replicado é formado por um número de processadores que podem falhar de forma independente, sendo capaz de tolerar um número finito de falhas de seus componentes. Para conseguir isso, cada processador executa uma réplica da aplicação em paralelo com os outros processadores componentes do nodo; cada réplica compara os resultados produzidos localmente com os que foram gerados pelas outras réplicas. Por fim, através de uma função de validação conveniente, é possível impedir que saídas incorretas possam ser geradas pelo nodo.

Nodos replicados podem ser implementados tanto em *hardware* (*hard nodes*) quanto em *software* (*soft nodes*) [Brasi95]. A principal vantagem da implementação em *software* é sua flexibilidade; é mais fácil portar para arquiteturas diferentes, conseguindo-se assim independência de fornecedor. Além disso, evita-se a complexidade de se

desenvolver circuitos especiais de *hardware*; a complexidade desses circuitos tende a aumentar devido ao fato de que os componentes do nodo têm que estar sincronizados a nível de micro-instrução, isto é, têm que executar as mesmas instruções a cada ciclo do relógio.

O modelo Voltan de *soft-nodes* é apresentado em [SESTT92]. Esse modelo permite a construção de nodos com diferentes tipos de semântica de falha. Maiores detalhes com respeito à implementação de nodos Voltan podem ser encontrados em [Brasi95] e [BESST96]. Os nodos Voltan são o modelo de nodos utilizado no ambiente **Seljuk-Amoeba**, que é apresentado na seção a seguir, e no qual este trabalho se insere.

1.1. O AMBIENTE SELJUK-AMOEBA

Em [Brasi97] é apresentada a especificação de um ambiente - o **Seljuk** - para facilitar o desenvolvimento e a execução de aplicações tolerantes a faltas. Uma versão do ambiente - o **Seljuk-Amoeba** - está sendo desenvolvida sobre o sistema operacional Amoeba [MRTRS90], utilizando uma rede local *off-the-shelf*, isto é, disponível livremente no mercado. O princípio básico que norteia o ambiente **Seljuk-Amoeba** é minimizar o esforço necessário para a construção de sistemas que possuem requisitos de tolerância a faltas. Como vimos, ao se adicionar mecanismos para tolerância a faltas, ocorre um aumento da complexidade geral do sistema. Assim, o objetivo do **Seljuk-Amoeba** é fornecer o suporte para a construção e execução de aplicações distribuídas tolerantes a faltas, retirando das mãos do programador a tarefa de adicionar os mecanismos para tolerância a faltas necessários [Vasco97]. Esse suporte é oferecido pelo ambiente através de duas classes de serviços:

(i) serviços que permitem restringir a semântica de falha das unidades de processamento do sistema; e

(ii) serviços que implementam mecanismos para tolerância a faltas.

A primeira classe de serviços possibilita especificar, em tempo de ativação, diferentes tipos de semântica de falha para a aplicação; a semântica de falha é assegurada através da implementação de um nodo Voltan que consiga prover a semântica requerida. A segunda classe, por sua vez, inclui serviços de mais alto nível como replicação de processamento, comunicação em grupo, diagnóstico de faltas e reconfiguração do sistema. Esses serviços usam o suporte oferecido pela arquitetura de nodos com semântica

de falha controlada. Maiores detalhes a respeito dessa classe de serviços podem ser encontrados em [Vasco97]. Daqui em diante estaremos concentrados apenas na primeira classe de serviços, que são baseados em nodos com semântica de falha controlada e nos quais se situa a motivação para este trabalho.

O modelo de nodos replicados utilizado no ambiente **Seljuk-Amoeba** supõe um sistema síncrono, ou seja, um sistema em que o tempo de transmissão de mensagens é limitado e conhecido [Crist95]. No entanto, o suporte de comunicação usado no **Seljuk-Amoeba** é baseado em uma rede tipo *Ethernet*[1], que é inerentemente assíncrona. Dessa forma, não é possível satisfazer a necessidade de um sistema síncrono sem que se forneçam mecanismos adicionais que dêem suporte à comunicação síncrona, construídos sobre redes assíncronas.

Nosso objetivo neste trabalho é estudar as dificuldades que devem ser tratadas ao se estender o serviço de comunicação disponível no ambiente **Seljuk-Amoeba**, de forma a fornecer um suporte síncrono para o ambiente, mas fazendo uso de uma rede assíncrona (no nosso caso, *Ethernet*). Essas dificuldades se relacionam basicamente com o fato de que ambientes assíncronos não são determinísticos, ou seja, não é possível saber a priori o tempo de execução de suas tarefas, especialmente a transmissão de mensagens.

Após o levantamento dos problemas a serem tratados, sugerimos uma solução que contorne esses problemas numa futura implementação no sistema operacional Amoeba. Apesar de não se ter implementado a proposta aqui apresentada, mostramos os pontos que devem ser modificados no sistema operacional de forma a facilitar uma posterior implementação.

A solução proposta se baseia na simulação sobre uma rede *Ethernet*, de um método TDMA (*Time Division Multiple Access*) de acesso ao meio físico; a proposta permite ainda que alguns processos reservem para si uma porção do *slot* TDMA do processador em que executam. O serviço de comunicação síncrono proposto se baseia assim na união do método de acesso TDMA com a possibilidade de se fazer reservas em *slots* TDMA.

Para complementar a proposta, tecemos considerações adicionais de forma a se possibilitar que alguns processos que compõem a implementação do modelo de nodos

[1] Na verdade o tipo de rede utilizado é baseado no padrão IEEE 802.3. O termo *Ethernet* identifica um produto criado pela Xerox e que serviu de base para a especificação do padrão. Apesar de não ser a mesma coisa, utilizamos o termo *Ethernet* para referenciar uma rede 802.3 por ser comumente usado assim.

Voltan (já que esses serão os usuários da proposta apresentada aqui) possuam escalonamentos especiais, para que a comunicação fim-a-fim entre eles possa ser efetivamente síncrona. É necessário ainda que se disponha de mecanismos para controle de fluxo das mensagens, para impedir que um processo que fez uma reserva transmita mais mensagens do que sua reserva pode comportar.

Outros aspectos positivos da proposta:

- uso the componentes off-the-shelf (COTS - **C**omponents **O**f **T**he **S**helf); ao se evitar o uso de componentes de hardware específicos, aumentamos o número de aplicações que podem usufruir de um serviço de processamento confiável, sem que tenham sido projetadas para isso;

- transparência para o software legado; aplicações em uso no sistema, e que não necessitam utilizar o serviço síncrono aqui detalhado, não terão que ser modificadas para se adequarem às alterações propostas para o Amoeba.

1.2. ESTRUTURA DA DISSERTAÇÃO

O restante deste trabalho está organizado da seguinte forma. No Capítulo 2, discutiremos o modelo de nodos com semântica de falha controlada utilizado no ambiente **Seljuk-Amoeba** - os nodos Voltan. Em particular, iremos detalhar por que o modelo necessita de um meio de comunicação síncrono, e em que pontos essa restrição é importante.

No Capítulo 3, abordaremos os problemas que devem ser tratados ao se dar um suporte síncrono a um ambiente assíncrono, e as tecnologias que forneceram subsídio para a nossa proposta. Os problemas encontram-se divididos em dois tipos: problemas decorrentes do uso de uma rede tipo *Ethernet*; e os problemas relacionados diretamente com o uso de um sistema operacional de propósito geral (o Amoeba, no caso). No mesmo capítulo, trataremos também das redes de tempo real, que têm o objetivo de fornecer o suporte de comunicação necessário para aplicações distribuídas de tempo real. Uma vez que essas aplicações possuem requisitos de tempo para a execução de suas tarefas, as idéias utilizadas forneceram alternativas para a nossa proposta de solução para o problema.

No Capítulo 4, apresentaremos o projeto da nossa solução para o problema, tratando as dificuldades mostradas no Capítulo 3, e enfocando a maneira como utilizamos as principais idéias da área de redes de tempo real também abordadas no Capítulo 3. Dessa forma, especificaremos o que pode ser usado para resolver o nosso problema, detalhando como resolver situações que não são tratadas pelas alternativas já discutidas.

A seguir, no Capítulo 5, vamos abordar os detalhes de implementação da proposta apresentada, particularizando para o sistema operacional Amoeba. Iniciaremos com uma breve explicação da estrutura de comunicação do Amoeba do ponto de vista do programador, ou seja, quais as estruturas e qual a API (*Application Programming Interface*) que pode ser usada para comunicação no Amoeba. Mais adiante, vamos apresentar como implementar o projeto proposto no Capítulo 4, levando em consideração a estrutura de comunicação apresentada inicialmente; mostraremos o que deve ser modificado no Amoeba e em que pontos essas modificações devem ser inseridas no núcleo do sistema.

Por fim, no Capítulo 6, apresentamos as discussões finais deste trabalho, salientando sua relevância, e indicando os trabalhos que podem ser desenvolvidos como continuação do que foi proposto aqui.

2.

O Modelo de Nodos com Semântica de Falha Controlada do Seljuk-Amoeba

2.1. INTRODUÇÃO

O ambiente **Seljuk-Amoeba** foi concebido com o objetivo de permitir às aplicações: (i) restringir a semântica de falha dos componentes do sistema; e (ii) usar mecanismos de mais alto nível para tolerância a faltas. Neste capítulo vamos abordar a estrutura responsável por assegurar a primeira parte dos objetivos do ambiente: o modelo de nodos com semântica de falha controlada. Esse modelo, baseado na família de nodos Voltan, permite oferecer dois tipos de semântica de falha: semântica de falha silenciosa e semântica de falha mascarada.

O modelo de aplicações que se supõe no modelo de nodos Voltan são aplicações distribuídas formadas por processos cooperantes, os quais se comunicam apenas através de troca de mensagens. A tolerância a faltas é suportada através da execução desses processos sobre nodos Voltan. Os nodos podem se utilizar de outros nodos para poderem prestar o serviço desejado; neste caso, é como se um nodo funcionasse como cliente de outro nodo. Apesar de se supor que um nodo executa num ambiente de rede local, é possível que nodos se comuniquem com outros nodos através de uma rede a longa distância, WAN (*Wide Area Network*).

O capítulo está divido da seguinte forma: na seção 2.2 abordamos o modelo de nodos Voltan, mostrando suas estruturas e uma breve explicação dos principais processos que o compõem. Na seção 2.3, tratamos com mais detalhe o protocolo de ordenação de mensagens usado no ambiente **Seljuk-Amoeba**; é neste protocolo que a suposição de comunicação síncrona se faz necessária. Por fim, a seção 2.4 faz uma análise das restrições encontradas no modelo, fornecendo assim a motivação para o restante deste trabalho.

2.2. O MODELO VOLTAN DE NODOS REPLICADOS COM SEMÂNTICA DE FALHA CONTROLADA

A família de nodos Voltan encontra-se apresentada em [SESST92]. Essa família permite a construção de nodos com diferentes tipos de semântica de falha, dentre os quais destacamos nodos com semântica de falha mascarada (*failure masking nodes*) e nodos com semântica de falha silenciosa (*fail silent nodes*), que serão utilizados para prover o serviço de processamento confiável do **Seljuk-Amoeba** [GBC97].

Nas sub-seções a seguir veremos os detalhes que compõem a arquitetura dos nodos Voltan, suas principais estruturas e protocolos.

2.2.1. Modelo do Sistema e Pressupostos

O modelo de nodos Voltan é baseado na abordagem da máquina de estado, apresentada em [Schne90]. A principal característica que define uma máquina de estado é a realização de uma computação determinística que recebe uma seqüência de pedidos, processa cada um, e eventualmente produz alguma saída. Nesta abordagem, supõem-se que as máquinas de estado não compartilham memória e se comunicam apenas por troca de mensagens. Em [Schne90] também é proposta uma versão da máquina de estado tolerante a faltas, na qual se replica uma máquina de estado em vários processadores de um sistema distribuído. Assim, pela suposição de determinismo, se cada réplica correta, i.e., uma réplica que executa segundo sua especificação, inicia a partir do mesmo estado inicial, e executa os mesmos pedidos e na mesma ordem, cada uma produzirá as mesmas saídas. Dessa forma, para se implementar uma máquina de estado tolerante a faltas deve-se garantir:

acordo: todas as réplicas corretas recebem as mesmas mensagens; **e**

ordem: todas as réplicas corretas processam as mensagens na mesma ordem.

Além disso, para garantir saídas corretas, deve-se utilizar algum mecanismo de validação das saídas produzidas pelas réplicas.

Uma vez que o modelo de nodos Voltan segue a abordagem da máquina de estado descrita acima, concluímos que as aplicações que podem ser replicadas em nodos Voltan devem possuir as mesmas características de uma máquina de estado: têm que ser determinísticas e comunicar-se apenas através de troca de mensagens. É possível ainda

que um nodo utilize os serviços de outros nodos para poder processar os pedidos que lhe são submetidos. Assim, podemos replicar tanto aplicações que servem como servidores de um serviço qualquer, bem como os próprios clientes desse serviço.

O serviço oferecido por um nodo Voltan pode ser caracterizado por sua semântica operacional e sua semântica de falha. A semântica operacional corresponde à especificação padrão do serviço a ser fornecido pelo nodo, enquanto que a semântica de falha descreve o comportamento do nodo quando até um número limitado de seus componentes falham. Qualquer comportamento que o nodo possa apresentar que não seja especificado por sua semântica operacional nem sua semântica de falha, é considerado um comportamento excepcional. O nodo só apresenta um comportamento excepcional se o número de falhas de seus componentes exceder o limite tolerado por aquele nodo, especificado na sua semântica de falha.

No caso de nodos com semântica de falha mascarada, por exemplo, a semântica de falha é equivalente à semântica operacional; o nodo consegue seguir suas especificações desde que não mais que um número limitado de falhas ocorra. No caso de nodos com semântica de falha silenciosa, a semântica de falha é dita segura [Laprie89], ou seja, após a detecção da falha de algum componente, o nodo nem apresenta seu comportamento padrão segundo suas especificações, nem realiza transições não especificadas, e simplesmente pára.

No modelo Voltan considera-se que um nodo é composto por N processadores e que destes no máximo π podem falhar ($\pi > 0$); ou seja, devemos ter no mínimo N-π processadores corretos no nodo. Para garantir a semântica de falha controlada tratada acima, é necessário que $N \geq 2\pi+1$, para nodos com semântica de falha mascarada, e $N \geq \pi+1$ para nodos com semântica de falha silenciosa. O valor de π deve ser escolhido de forma que a semântica de falha controlada possa ser atingida com a probabilidade requerida pela aplicação. Como não há um limite para o valor de N, essa probabilidade pode ser arbitrariamente alta.

O modelo faz ainda uma suposição com relação ao tempo de transmissão de mensagens entre processadores dentro de um nodo. Assim, a comunicação intra-nodo deve ser síncrona, ou seja, o atraso para o envio de mensagens deve possuir um limite máximo conhecido.

Consideremos então que o tempo total de transmissão de uma mensagem (T_t) é composto de: o tempo de processamento na origem (T_o), equivalente ao tempo decorrido desde a solicitação de envio da mensagem pelo processo emissor até a entrega ao adaptador (placa) de rede para sua transmissão propriamente dita; o tempo de envio da mensagem (T_e), que é o tempo transcorrido no percurso da mensagem do processador origem até o processador destino; e o tempo de processamento no destino (T_d), correspondente ao intervalo compreendido entre a recepção da mensagem no processador destino até ela ser efetivamente entregue ao processo destino. Ou seja, $T_t = T_o + T_e + T_d$. A Figura 2.1 ilustra melhor estes intervalos; nela vemos os tempos relacionados com a transmissão de mensagens desde um processo A1 numa máquina A até o processo B1 numa máquina B. Vemos assim que os tempos T_o e T_d estão relacionados com o processamento da mensagem dentro do núcleo do sistema operacional.

O modelo Voltan necessita conhecer o atraso máximo para transmissão de mensagens (ou seja, o valor máximo de T_t). Definimos então que esse atraso máximo é representado pela constante *d* (também representada na Figura 2.1). Assim, *d* é utilizado por todos os processadores corretos do nodo para medir, através de seus relógios internos, o atraso máximo para transmissões de mensagens na rede, ou seja, o valor máximo de T_t. Apesar de não se supor que o atraso na transmissão de mensagens destinadas ao nodo seja limitado (o que inclui mensagens originadas de outros nodos, ou mesmo de uma aplicação cliente qualquer), o atraso para a comunicação entre os processadores dentro do nodo deve possuir o limite máximo *d*.

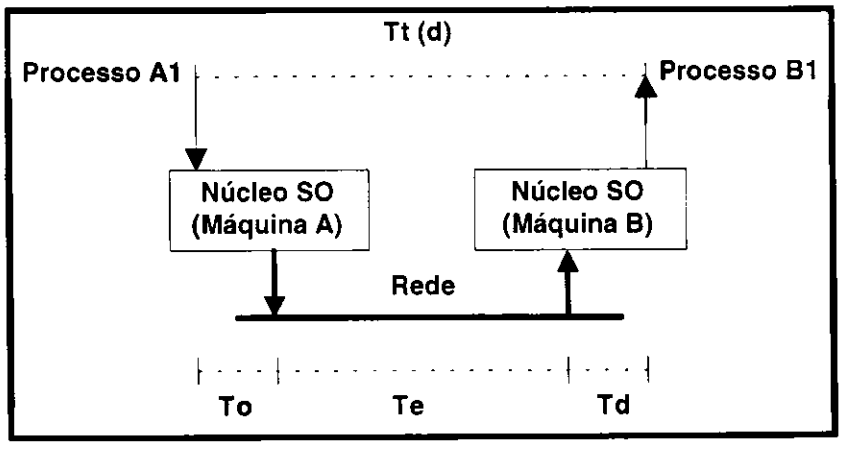

Figura 2.1 - Tempos associados com a transmissão de mensagens entre dois processos

No modelo Voltan, para se efetuar a validação das saídas produzidas por cada réplica, supõe-se a existência de mecanismos de geração e verificação de assinaturas digitais. Isso permite que as saídas produzidas por cada réplica pertencentes a um nodo

replicado possam ser validadas antes de serem entregues ao nodo destino. Dessa forma, cada processador correto inclui em cada mensagem enviada uma assinatura única, impossível de se falsificar e dependente da mensagem enviada. Essas mensagens podem ser validadas através de uma função de autenticação que verifica a autenticidade de uma assinatura contida numa mensagem. Técnicas de criptografia de chave pública podem ser usadas para implementar essa funcionalidade com probabilidade arbitrariamente alta [RSA78]. Uma vez que só podemos ter no máximo π processadores incorretos, uma mensagem contendo π+1 assinaturas determina que pelo menos 1 processador correto assinou essa mensagem, permitindo considerá-la válida.

2.2.2. Arquitetura

A Figura 2.2 mostra a arquitetura genérica dos nodos Voltan; ela é baseada na existência das seguintes fases. Supondo a existência de n réplicas (representados como R_1 até R_n na Figura 2.2), as mensagens destinadas a um processo replicado devem ser ordenadas antes de efetivamente serem entregues à aplicação; essa primeira fase garantirá os pressupostos de ordem e acordo mencionados na seção anterior. Após terminada a ordenação, as mensagens são entregues ao processo replicado. Por fim, as mensagens geradas por cada processo são validadas e só então, se for o caso, a saída (resultado da computação executada) é realmente enviada ao processo destino. A validação em um nodo com semântica de falha mascarada é baseada num esquema de votação, enquanto que a validação num nodo com semântica de falha silenciosa é determinada por comparação de mensagens.

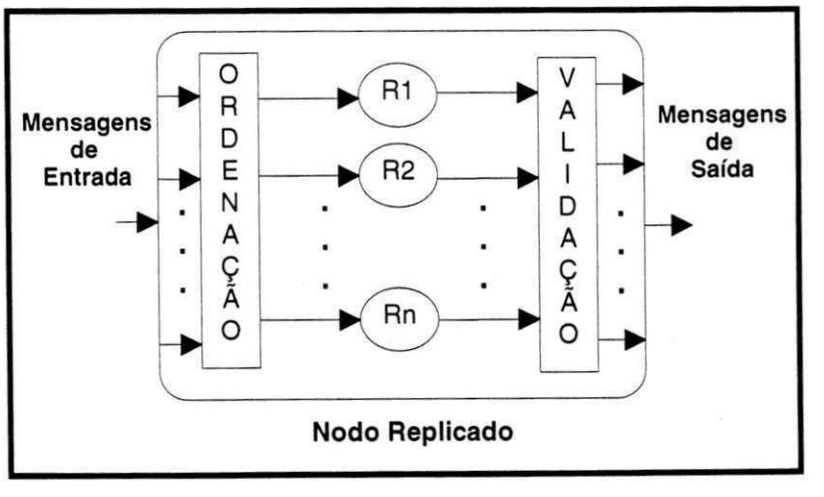

Figura 2.2 - Funcionamento de um nodo com n réplicas

Cada nodo no sistema, cada processador em um nodo, e cada grupo de processos replicados recebe um identificador único. Além disso, cada processador mantém um contador interno que serve como número de seqüência das mensagens por ele geradas. Esses identificadores e contadores têm a função de distinguir unicamente cada mensagem gerada por um processo replicado qualquer. Assim, toda mensagem gerada por um processador leva consigo essas informações de controle além de informação de autenticação (assinatura digital do processador). Essas informações são usadas pelos protocolos de gerência de redundância com o objetivo de selecionar mensagens corretas, e detectar e remover mensagens duplicadas ou corrompidas.

Na Figura 2.3 mostra-se o fluxo de informações entre os processos que compõem um nodo Voltan, ilustrando a visão de um processador. Note que as mensagens que se destinam ao nodo podem ser originadas tanto dos processadores do próprio nodo (que correspondem aos *links* na Figura 2.3), bem como de uma aplicação cliente qualquer, que também pode estar replicada (essas mensagens correspondem às mensagens inter-nodo na Figura 2.3). Num mesmo processador pode haver mais de um processo replicado em execução (o item Réplica_i na Figura 2.3); além deles, cada processador do nodo executa ainda os seguintes processos:

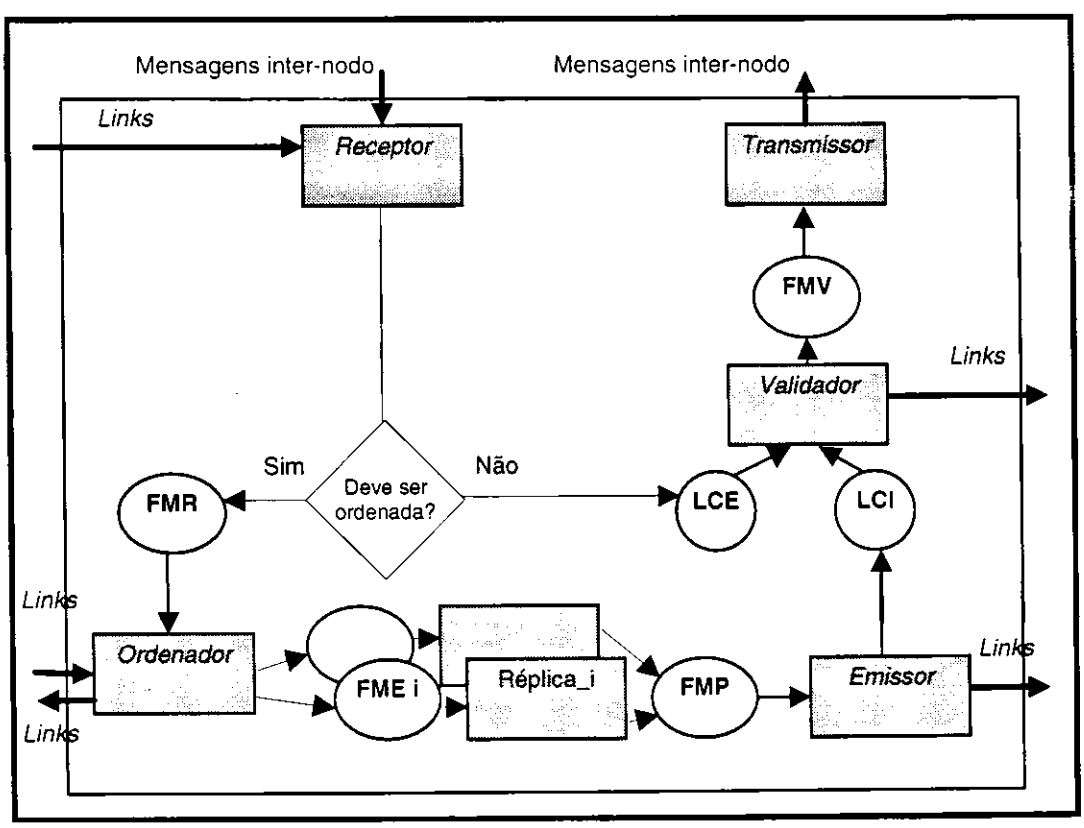

Figura 2.3 - Fluxo de mensagens de um nodo Voltan (visão de um processador)

Receptor: autentica mensagens recebidas de outros processadores do nodo ou de outros nodos, descartando qualquer mensagem cuja autenticação não tenha sucesso, ou que seja uma duplicata (já tenha sido recebida). Mensagens autênticas vindas de outros nodos têm π+1 assinaturas e são entregues ao processo *Ordenador* local; mensagens autênticas vindas do próprio nodo têm menos que π+1 assinaturas e são entregues para o processo *Validador* local.

Ordenador: executa um protocolo de ordenação de mensagens em conjunto com os outros processos *Ordenadores* do nodo. Sua função é construir filas de mensagens para serem tratadas pelos processos replicados em todos os processadores corretos do nodo; essas filas devem conter as mesmas mensagens e na mesma ordem. Uma vez que uma mensagem chegar para ordenação em qualquer processador correto, ela será enviada também para todos os outros processadores corretos do nodo.

Emissor: obtém mensagens produzidas pelo processo replicado (Réplica_i na Figura 2.3), assina-as e as envia para os outros processadores do nodo para validação.

Validador: a função deste processo depende do tipo do nodo executando os processos replicados. Em nodos com semântica de falha mascarada, comporta-se como um processo *Votante*: compara mensagens autênticas assinadas e enviadas por outros processadores com as que foram geradas localmente. Se a comparação falha, a mensagem enviada é descartada; caso contrário, a mensagem é contra-assinada, e se na mensagem existirem π+1 assinaturas, ela é considerada uma mensagem válida. É então repassada ao processo *Transmissor* para ser enviada ao processo destino (comunicação inter-nodo). Se existirem menos que π+1 assinaturas na mensagem, ela é enviada para os outros processadores do nodo que ainda não a assinaram. No caso de nodos com semântica de falha silenciosa o processo *Validador* age como um *Comparador*: comporta-se da mesma forma que o *Votante* descrito acima, com a diferença de que ao se detectar uma falha, ao invés de simplesmente descartar a mensagem recebida, o processo *Validador* pára por completo, assim como o processo *Emissor*. Neste caso, considera-se uma falha quando as mensagens comparadas não forem iguais ou quando nenhuma mensagem contendo π assinaturas chegar no processador para se efetuar a comparação dentro de intervalo de tempo máximo específico do nodo. Dessa forma, quando uma falha é detectada, nenhuma mensagem contendo π+1 assinaturas será gerada, refletindo a parada do nodo. Em ambos os tipos de nodos, só se consideram mensagens válidas aquelas com π+1 assinaturas.

Transmissor: responsável por enviar as mensagens válidas (aquelas com π+1 assinaturas) para o processo destino.

A comunicação entre processos num mesmo processador é feita através de várias listas e filas (como aparecem na Figura 2.3). O processo de retirada de mensagens das listas apesar de permitir acesso aleatório, garante uma retirada justa, isto é, qualquer mensagem que entre na lista com certeza será retirada em algum momento. As seguintes filas e listas são usadas:

Fila de Mensagens Recebidas (FMR): contém mensagens autenticadas recebidas de outros nodos, prontas para serem ordenadas.

Fila de Mensagens Entregues (FME): contém mensagens ordenadas prontas para serem processadas pelo processo replicado.

Fila de Mensagens Processadas (FMP): contém mensagens ainda não assinadas produzidas pelos processos replicados locais. Essas mensagens têm que ser validadas pelo processo *Validador*, antes de serem transmitidas para seu destino final.

Lista de Mensagens Candidatas - Externas (LCE): contém mensagens assinadas autenticadas que foram recebidas de outros processadores para validação.

Lista de Mensagens Candidatas - Internas (LCI): contém mensagens não assinadas, à espera de mensagens na lista LCE que coincidam com elas.

Fila de Mensagens Válidas (FMV): contém mensagens válidas com π+1 assinaturas, prontas para serem transmitidas para o processo destino.

Os processos *Receptor*, *Emissor*, *Transmissor*, e *Validador* são implementados de forma trivial, e são discutidos com maior detalhe em [Brasi95]; o principal processo do nodo Voltan (em termos de complexidade), é aquele encarregado de ordenar as mensagens, o processo *Ordenador*. Veremos maiores detalhes sobre ele na próxima seção.

2.3. CONSIDERAÇÕES SOBRE A NECESSIDADE DE COMUNICAÇÃO SÍNCRONA

Nosso interesse é implementar esse modelo num ambiente aberto, utilizando um sistema operacional de propósito geral (Amoeba). Neste tipo de ambiente, não é possível

prever a carga que será imposta aos processos replicados no nodo. Uma vez que não se consegue prever a carga de processamento a que se pode submeter um processo replicado, não podemos prever também o tempo de resposta necessário ao processamento de pedidos feitos ao processo replicado. Dessa forma, aplicações que possuam restrições de tempo de execução não podem ser replicadas utilizando a implementação do modelo de nodos Voltan em sistemas operacionais de propósito geral, a menos que se imponham restrições na carga máxima a qual a aplicação será exposta.

Portanto, para o tipo de implementação que pretendemos, vemos que é necessário restringir a semântica de falha que se pode supor para os processadores no nodo. Devido ao que foi exposto acima, não é possível tolerar faltas no domínio do tempo, e portanto, também não é possível tolerar faltas arbitrárias dos processadores. No entanto, podemos tolerar faltas no domínio do valor, ou seja, aplicações que necessitam que as saídas produzidas sejam corretas, mesmo que demorem um intervalo de tempo indeterminado para serem geradas, podem ser replicadas segundo o modelo de nodos apresentado anteriormente, implementado num sistema operacional de propósito geral.

A partir das considerações acima, podemos agora identificar em que pontos se faz necessária a restrição do limite para a transmissão de mensagens nos nodos Voltan a serem implementadas no ambiente **Seljuk-Amoeba**.

2.3.1. O protocolo de Ordenação do modelo de nodos replicados

O processo *Ordenador* possui dois comportamentos diferentes dependendo do tipo de nodo do qual faz parte: nodo com semântica de falha silenciosa ou nodo com semântica de falha mascarada. Num nodo com semântica de falha silenciosa, o protocolo de ordenação de mensagens é bem mais simples; na verdade, o protocolo nem sequer precisa tolerar faltas que possam ocorrer durante a ordenação de mensagens. Isso é verdade porque uma falha na ordenação de mensagens pode ser detectada no momento da validação, e detectar a ocorrência de uma falha é suficiente num nodo com semântica de falha silenciosa: após a detecção da falha o nodo pára. Já num nodo com semântica de falha mascarada, a ordenação desempenha papel fundamental, pois apenas saídas corretas podem ser geradas pelo nodo, e o mecanismo de validação não tem como corrigir uma falha gerada pelo processamento de mensagens na ordem incorreta. Dessa forma, nesta seção só iremos tratar do protocolo de ordenação necessário para implementar nodos com semântica de falha mascarada. Maiores detalhes sobre o protocolo de ordenação usado no

ambiente **Seljuk-Amoeba** para nodos com semântica de falha silenciosa podem ser encontrados em [Brasi95], [BESST96] e [GBC97].

Num nodo com semântica de falha mascarada, o processo *Ordenador* efetua a ordenação das mensagens através de uma série de rodadas de disseminação de mensagens que implementam *broadcasts* atômicos [BE95]. A implementação do protocolo envolve (i) difundir as mensagens, para garantir que todos os processadores corretos recebam o mesmo conjunto de mensagens e (ii) verificar se as mensagens chegaram dentro de um intervalo de tempo aceitável (*timeliness checks*); isso permitirá que os processadores corretos possam descartar mensagens inválidas, isto é, mensagens recebidas tarde demais. O protocolo de ordenação que vamos abordar aqui é discutido em [Brasi95].

2.3.2. Descrição do protocolo

Para efetuar as duas fases de que é composto o protocolo (difusão de mensagens e verificação de aceitação de mensagens), o processo *Ordenador* é divido em três outros processos: *Broadcast*, *Difusor* e *Entregador*. A Figura 2.4 descreve as relações entre cada um eles. Esta figura corresponde a uma expansão do processo *Ordenador* que aparece na Figura 2.3.

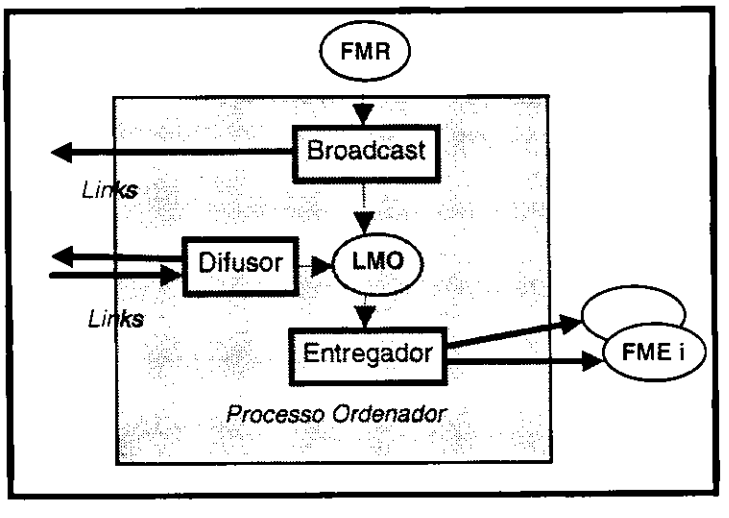

Figura 2.4 - Estrutura do processo Ordenador

O processo *Broadcast* lida com as mensagens de entrada para o *Ordenador* recebidas a partir da fila FMR; esta fila é alimentada pelo processo *Receptor* do nodo. Mensagens recebidas de outros processos *Ordenadores* são tratadas no processo *Difusor*, enquanto que a ordenação e posterior entrega de mensagens aos processos replicados fica sob a responsabilidade do processo *Entregador*.

Para entendermos melhor o funcionamento desses processos, vamos supor que um nodo com semântica de falha mascarada seja composto por N processadores, identificados por P_1, P_2, ..., P_n. Cada processador P_i mantém um contador de mensagens cujo valor é um número inteiro que nunca é decrementado e que é inicializado com 1. Esse contador, denotado por MC_i, servirá para identificar as mensagens a serem ordenadas e é mantido de maneira análoga ao esquema de relógios lógicos descrito por Lamport [Lampo78]. Neste esquema, cada vez que uma mensagem é enviada, ela é identificada com o valor corrente do contador local, que é o relógio lógico (no nosso caso, MC_i), o qual é concatenado à mensagem; logo em seguida, o valor do contador é incrementado. Quando uma mensagem é recebida, se o valor do contador presente na mensagem for maior ou igual ao valor do contador mantido localmente, este deve ser atualizado para o valor do contador na mensagem adicionado de 1.

Toda mensagem *m* que circula no protocolo de ordenação é interpretada como a tupla <μ, *TS, O, S*>, na qual *m*.μ é o conteúdo da mensagem propriamente dito; m.*TS* é o número de seqüência de *m* (*timestamp*); *m.O* é a identificação do processador que originou *m*; e por fim, *m.S* é a lista de assinaturas (informação de autenticação) de *m*. Denota-se por |*m.S*| o número de assinaturas de *m*.

Feitas as suposições acima, vamos ver agora com maior detalhe o funcionamento do protocolo de ordenação para um processador correto P_i. Cada mensagem retirada da fila FMR deve ser repassada a todos os outros processadores do nodo para ordenação; é isso o que o processo *Broadcast* faz. Para tanto, atribui à mensagem no campo *m.TS* o valor do seu relógio lógico local MC_i, no campo *m.O* a identificação do processador P_i e concatena no campo *m.S* sua assinatura digital. Esta mensagem é então enviada para os outros processadores do nodo e copiada para a lista LMO (Lista de Mensagens para Ordenação), de onde está apta a ser ordenada localmente. A lista LMO contém todas as mensagens recebidas para ordenação pelo processador, sejam elas recebidas diretamente de outros nodos, ou mesmo difundidas por outros processadores do próprio nodo.

O processo *Difusor* recebe mensagens enviadas pelos processos *Broadcast* e *Difusor* dos outros processadores que formam o nodo, e verifica a validade da mensagem com relação ao instante em que ela chegou; isto é, se tiver chegado tarde demais (na seção 2.3.2 trataremos como determinar isso) a mensagem é inválida, e deve ser descartada. Se

for válida e se o número de assinaturas presentes na mensagem for menor que $\pi+1$, ela é assinada e disseminada para os outros processadores que ainda não a assinaram.

A função do processo *Entregador*, por sua vez, é identificar as mensagens estáveis na lista LMO, e entregá-las ao processo replicado de destino. Uma mensagem é dita estável, quando for garantido que todos os processadores corretos do nodo já a receberam, e que não é possível a chegada de nenhuma outra mensagem válida que deveria ser ordenada antes desta mensagem. Uma vez que se tem uma lista de mensagens estáveis, essas mensagens são ordenadas localmente segundo o mesmo critério em todos os processadores corretos, e só depois são entregues ao processo replicado correspondente. Dessa forma, garante-se que todas as réplicas recebem as mesmas mensagens e na mesma ordem.

2.3.3. Testes de validade de mensagens

Um processador qualquer só pode detectar o envio de uma mensagem quando esta mensagem chegar até ele. Por outro lado, a ordem de chegada de mensagens não determina necessariamente a ordem em que elas foram enviadas. Por exemplo, o envio de uma mensagem m_1 antes de uma outra mensagem m_2 de um processador P_i até um outro processador P_j, não determina necessariamente o recebimento de m_1 antes de m_2 em P_j. Essa situação encontra-se ilustrada na Figura 2.5.

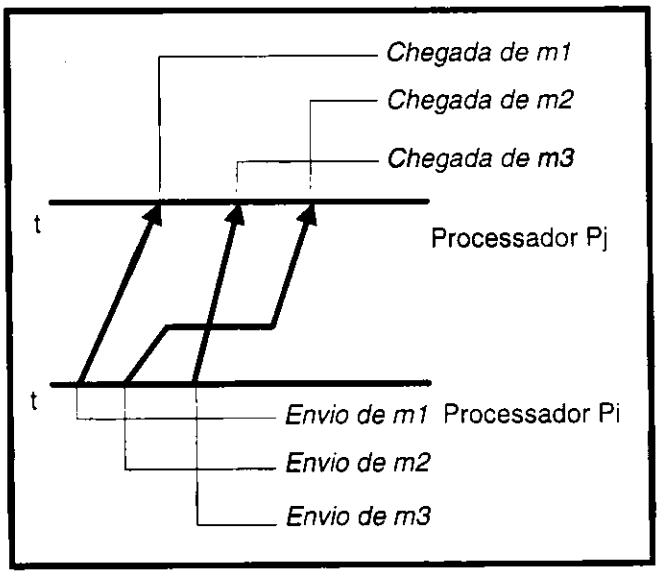

Figura 2.5 - Atrasos na transmissão de mensagens entre dois processadores

Desta forma, se a ordem de chegada não determina a ordem de envio, quando se recebe uma mensagem m qualquer, quanto tempo se deve esperar até a chegada de uma

outra mensagem *m'* que deveria ser ordenada antes de *m*? Veremos que esse intervalo depende do tempo máximo de transmissão de mensagens *d* definido anteriormente.

Pelo protocolo de ordenação brevemente exposto na seção anterior, cada mensagem que deve ser ordenada passa por no máximo π+1 rodadas em que é transmitida de processador a processador, de forma a garantir que todos os processadores corretos do nodo replicado recebam essa mensagem. Na primeira rodada, ela é recebida pelos processos *Ordenadores* (entregue pelo processo *Receptor* local), e conseqüentemente assinada e difundida para os outros processos *Ordenadores* do nodo; esse é o trabalho do processo *Broadcast*. Assim, ao receber pela primeira vez uma mensagem *m* qualquer, cada processador P_i concatena à mensagem o valor do seu relógio lógico local, MC_i, assina a mensagem, e a envia para todos os outros processadores do nodo.

Isso implica que no máximo após o intervalo de tempo *d* (medido pelo relógio local de qualquer processador correto), essa mensagem certamente terá sido recebida em todos os outros processadores corretos do nodo. Assim, os valores dos relógios lógicos locais de cada um deles certamente serão maiores que MC_i dentro de um intervalo de tempo máximo *d*.

Portanto, uma mensagem *m'* originada de um outro processador correto P_j qualquer, cujo número de seqüência *m'.TS* seja menor que MC_i, só pode ter sido enviada antes de P_j ter recebido a mensagem *m* enviada por P_i. Assim, considerando que o tempo máximo (medido por P_i) para a transmissão de *m* e de *m'* é *d*, P_i precisa esperar no máximo o intervalo de tempo *2d* (contado a partir do envio de *m*), até garantir que nenhuma outra mensagem *m'* válida, enviada diretamente por *m'*.O, e que deveria ser ordenada antes que *m* vai ser recebida.

O raciocínio acima é válido para mensagens contendo 1 assinatura; no entanto, podemos generalizá-lo para as outras rodadas do protocolo, ou seja, para mensagens recebidas pelos processos Ordenadores contendo um número qualquer de assinaturas. Assim, de forma análoga ao que foi exposto acima, após receber uma mensagem *m* contendo |*m.S*| assinaturas, cada processador P_i deve esperar um intervalo de tempo de no máximo 2|*m.S*|*d*, para poder garantir que nenhuma outra mensagem válida com número de seqüência menor que o de *m*, e que contenha até |m.S| assinaturas, será recebida. Essa condição é exposta mais formalmente a seguir:

Condição 1: uma mensagem m, recebida por um processador P_i, cujo número de seqüência $m.TS$ é menor ou igual ao valor do relógio lógico local MC_i, é considerada válida apenas se foi recebida antes do instante $t+2|m.S|d$, onde t é o menor instante no qual uma outra mensagem m' foi recebida ou enviada e tornou MC_i maior que $m.TS$. Caso contrário, m é garantidamente uma mensagem enviada por um processador incorreto e deve ser descartada.

Existe um caso especial que ainda deve ser tratado e que não está contemplado na Condição 1 acima. Pelo que foi exposto, o processo *Difusor* de um processador P_i ao receber uma mensagem m de um outro processador P_j, irá repassar essa mensagem para os outros processadores do nodo que ainda não assinaram a mensagem. Assim, deve-se esperar o intervalo de tempo $2|m.S|d$, até se garantir que nenhuma outra mensagem m' que deveria ser ordenada antes que m possa chegar. Esse intervalo é o mesmo para todos os outros processadores do nodo, exceto para o processador que enviou m, no caso, P_j. Veremos por que a seguir.

Para o caso específico de P_j, sabemos que após o envio de m, o relógio lógico local de P_j, MC_j, já é atualizado para um valor maior que $m.TS$. Dessa forma, uma mensagem m' enviada por P_j, diretamente para P_i, cujo número de seqüência $m'.TS$ seja menor ou igual ao valor do relógio lógico local de P_i, MC_i, só pode ter sido enviada antes de P_j ter enviado m. Assim, para uma mensagem m originada de P_j contendo apenas uma assinatura (ou seja, $|m.S|$ igual a 1), P_i deve esperar no máximo um intervalo de tempo d (ao invés de $2d$) após a recepção de m, antes de invalidar qualquer outra mensagem m', originada por P_j com número de seqüência menor ou igual a MC_i. Generalizando para mensagens contendo um número qualquer de assinaturas, temos que P_i deve esperar no máximo o intervalo de tempo $(d+2(|m.S|-1)d)$ antes de invalidar uma mensagem m' originada por P_j, com $m'.TS$ menor ou igual a MC_i. Essa condição pode ser formalizada assim:

Condição 2: uma mensagem m, recebida por um processador P_i, cujo número de seqüência $m.TS$ é menor ou igual ao valor do relógio lógico local MC_i, e cujo processador de origem $m.O$ é j, é considerada válida apenas se foi recebida antes do instante $t+d+2(|m.S|-1)d$, em que t é menor instante no qual uma outra mensagem m', $m'.O = j$, foi recebida e tornou MC_i maior que $m.TS$. Caso contrário, P_j é garantidamente incorreto e m deve ser descartada.

Após efetuar a verificação de validade das mensagens é possível determinar quais mensagens são estáveis, isto é, quais mensagens podem ser entregues ao processo destino. Assim, supondo que o valor de *d* seja convenientemente escolhido, podemos garantir que se sabe quando uma mensagem é estável. Uma vez que teremos $\pi+1$ rodadas de troca de mensagens, e que em cada rodada se espera no máximo um intervalo de tempo *2d* para se efetuar as verificações de validade de mensagens (*timeliness checks*), temos que o intervalo máximo de estabilização de mensagens é equivalente a $2d(\pi+1)$.

No entanto, o problema para se implementar os protocolos de ordenação de mensagens de um nodo replicado num ambiente de rede assíncrono é exatamente garantir o valor de *d*. Num ambiente de rede assíncrono não se pode garantir um valor máximo para o tempo efetivo de transmissão de mensagens (T_c como foi visto no início do capítulo); conseqüentemente, não se pode garantir um valor máximo para *d*. Isso acontece porque um atraso excessivamente elevado para a recepção de uma mensagem, pode ter sido causado tanto por sobrecarga no sistema, como também devido a uma falha [Crist95]. Por outro lado, o próprio tempo de processamento T_o e T_d dos processos que implementam o protocolo de ordenação (que também está incluído em *d*) não é um fator determinístico, dificultando ainda mais a garantia de *d*.

Dessa forma, temos a motivação para o restante do trabalho: como possibilitar o cálculo do valor de *d*, considerando-se o uso de redes assíncronas, e levando-se em conta que os processadores no nodo podem ser submetidos a uma carga qualquer de processamento.

2.4. CONCLUSÃO

Neste capítulo estudamos o modelo de nodos replicados com semântica de falha controlada do ambiente **Seljuk-Amoeba**. Vimos também porque o modelo necessita que se tenha um meio de comunicação síncrono para transmissão de mensagens.

O fato de o modelo de nodos descrito acima supor um suporte de comunicação síncrono, afeta diretamente a implementação que se pretende fazer no ambiente **Seljuk-Amoeba**. Uma vez que o ambiente usa LANs (*Local Area Networks*) comuns para a comunicação entre os processadores dos nodos replicados, composta por COTS que são inerentemente assíncronos, essa restrição não pode ser satisfeita. Assim, não se pode garantir o valor de *d*, a menos que mecanismos especiais sejam adicionados ao sistema.

No próximo capítulo, veremos por que o tipo de LAN utilizada no **Seljuk-Amoeba** não possui as características necessárias à comunicação síncrona.

De qualquer forma, é importante identificar precisamente em que pontos da arquitetura de nodos replicados a restrição de comunicação síncrona se faz necessária. Pelo que vimos, quando não se faz necessário tolerar faltas no domínio do tempo, apenas os processos *Difusor* e *Broadcast*, encarregados de transmitir as mensagens para ordenação, é que precisam ter garantido o tempo máximo para a transmissão de mensagens.

Um outro fator que também deve ser levado em consideração é o tempo de processamento dos processos *Difusor* e *Broadcast*, bem como o tempo de execução do próprio sistema operacional. Enquanto se faz o envio de uma mensagem, ou mesmo enquanto se espera que alguma mensagem chegue, os processos que executam o protocolo de ordenação podem ser submetidos ao escalonamento de tarefas do próprio sistema operacional. Assim, o tempo levado até uma mensagem ser efetivamente transmitida, ou o tempo transcorrido desde o momento em que uma mensagem chega no processador destino até ela ser realmente entregue ao processo destinatário pode variar bastante, sendo difícil de ser medido.

Assim, no valor de *d*, além do tempo necessário à transmissão de mensagens (no nosso caso, T_e), deve ser considerado também o maior intervalo necessário ao processamento de uma mensagem qualquer, seja no seu recebimento, seja no seu envio. Em implementações anteriores do modelo de nodos Voltan [Brasi95], esse tempo é estimado, supondo uma carga máxima submetida aos processadores; entretanto, num ambiente aberto, como é o caso do **Seljuk-Amoeba**, esse intervalo não é determinístico. No Capítulo 4, abordaremos como contornar as restrições mencionadas acima, possibilitando o cálculo correto do valor de *d*.

3.

Considerações sobre Comunicação Síncrona

3.1. INTRODUÇÃO

Num ambiente assíncrono utilizando redes locais comuns não há garantias de tempo para a transmissão de mensagens (numa rede a longa distância existem menos garantias ainda). Os fatores que determinam isso se relacionam com o próprio tipo de rede utilizada (*Ethernet*, *Token-Ring*, FDDI) e com o fato de geralmente se utilizar um sistema operacional de propósito geral. Estes fatores contribuem para que o atraso na transmissão de mensagens (T_t, como vimos, formado por $T_o + T_e + T_d$) seja aleatório. Neste capítulo, vamos abordar esses fatores com relação ao ambiente **Seljuk-Amoeba**, além de mostrar algumas alternativas disponíveis na literatura na área de redes em tempo real que se preocupam em prover comunicação síncrona.

O capítulo está dividido como se segue. Na seção 3.2 discutimos o método de acesso CSMA/CD que serve de base para o tipo de rede local que pretendemos usar - *Ethernet*; assim, trataremos das dificuldades relacionadas com o uso de uma rede *Ethernet* ao se tentar prover comunicação síncrona. Já na seção 3.3, abordamos as dificuldades determinadas pelo uso de um sistema operacional de propósito geral - o Amoeba. A seção 3.4, por sua vez, trata de algumas considerações necessárias à construção de redes de tempo real, e aborda duas soluções em particular encontradas na literatura. As tecnologias abordadas neste capítulo servirão de referencial teórico para a proposta que apresentamos no Capítulo 4. O capítulo termina com algumas considerações finais presentes na seção 3.5.

3.2. O MÉTODO DE ACESSO CSMA/CD

A principal característica que distingue um tipo de rede local de outra é o método de acesso ao meio de comunicação. É esse método que determina qual processador tem a vez de transmitir mensagens a cada instante. Uma vez que o ambiente **Seljuk-Amoeba** se

utilizará de uma rede tipo *Ethernet*, é importante analisarmos que dificuldades devem ser tratadas ao se tentar disponibilizar comunicação síncrona para o ambiente. No nosso caso, os problemas se encontram no método de acesso CSMA/CD, principal característica de uma rede local *Ethernet*. Nesta seção vamos abordar o método de acesso CSMA/CD e discutir por que é um fator de complexidade para nosso objetivo: permitir comunicação síncrona sobre um ambiente assíncrono. As discussões levantadas aqui podem ser encontradas com maior detalhe em [SLC95] (pp. 175-178 e pp. 188-195) e [Tanen89] (pp. 141-164).

Uma rede tipo *Ethernet* é baseada no padrão IEEE 802.3, o qual faz parte do padrão internacional IEEE[1] 802. Este padrão especifica os níveis de enlace de dados e físico do modelo RM-OSI[2], para diversos tipos de redes locais (*Token Ring*, *Token Bus*, FDDI, etc.). A Figura 3.1 (retirada de [SLC95], p.141) mostra as relações entre o padrão proposto pelo IEEE (apresentando alguns padrões de redes locais) e o modelo RM-OSI .

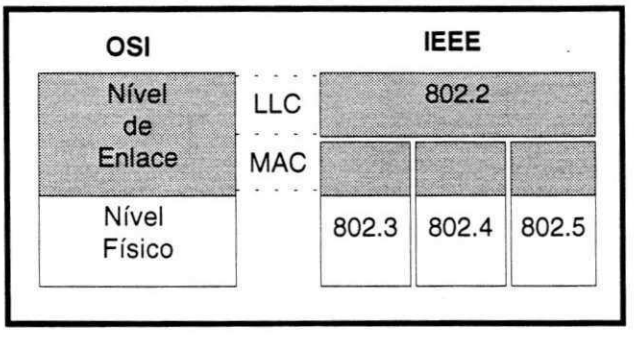

Figura 3.1 - Relação entre o modelo de referência OSI e o padrão IEEE 802

Basicamente, o que foi feito foi dividir a camada de enlace de dados em duas subcamadas. A primeira subcamada, a camada LLC (*Logical Link Control*), oferece a interface para o nível superior (nível de rede do modelo OSI); é responsável pela realização das funções de controle de erro e de fluxo no enlace, além da definição de diferentes classes de serviço (serviço com/sem conexão, com/sem reconhecimento). A outra subcamada, chamada de MAC (*Medium Access Control*), desempenha o controle de acesso à rede. Em suma, todo o tratamento comum aos diferentes tipos de redes locais situa-se na camada LLC; as funções específicas de cada tecnologia permanecem na camada MAC.

[1] IEEE: acrônimo para **I**nstitute of **E**lectrical and **E**lectronics **E**ngineers, órgão internacional de padronização.

[2] RM-OSI: **R**eference **M**odel - **O**pen **S**ystems **I**nterconnection; modelo para o desenvolvimento de padrões para a comunicação de sistemas abertos [DZ83], proposto pela ISO (**I**nternational **S**tandards **O**rganization), órgão internacional de padronização.

O padrão IEEE 802.3 utiliza o método de acesso conhecido como **CSMA/CD** (***Carrier Sense Multiple Access with Collision Detection***). Neste método, sempre que um processador deseja transmitir dados, ele verifica antes se o meio de transmissão está livre para transmissão, ou seja, verifica se existe alguma transmissão em curso. Se o meio é detectado livre, a transmissão acontece imediatamente; se o meio está ocupado, o processador espera até que fique livre e possa iniciar a transmissão.

É possível que dois ou mais processadores detectem que o meio está livre simultaneamente, ou que, devido ao tempo de propagação na rede, uma transmissão não seja detectada a tempo por algum outro processador que deseja transmitir, acontecendo assim uma colisão.

Dessa forma, sempre que um processador inicia uma transmissão, ele monitora o meio de forma a detectar se houve alguma colisão. Os processadores que tiveram sua transmissão corrompida por alguma colisão aguardam um intervalo aleatório, após o qual executam o mesmo procedimento novamente: verificação se o meio está livre; transmissão; e verificação de colisão. Esse processo é repetido até que a transmissão seja efetuada com sucesso.

A possibilidade de colisões determina que o método de acesso CSMA/CD, utilizado em redes *Ethernet*, é não-determinísitico. Na média, a eficiência (isto é, o percentual de mensagens entregues com sucesso) pode chegar a 95%. Com certeza é um percentual elevado, mas pode ser facilmente degradado com o aumento do tráfego na rede. Assim, no pior caso, a eficiência cai bastante.

De qualquer maneira, mesmo na média, não há garantias. Dessa forma, uma mensagem pode ter que ser retransmitida várias vezes, aumentando o tempo de sua transmissão. Fica bastante claro, portanto, que este tipo de rede não pode ser utilizado diretamente num ambiente que necessita de um suporte de comunicação síncrono. Uma vez que o acesso ao meio de comunicação é inerentemente não-determinístico, não há como garantir a componente T_e (ver Capítulo 2) relativa ao tempo de transmissão de mensagens.

3.3. SISTEMAS OPERACIONAIS DE PROPÓSITO GERAL

Assim como o uso de uma rede local tipo *Ethernet* traz um fator de não-determinismo para a transmissão de mensagens, o uso de um sistema operacional de

propósito geral (no nosso caso, o Amoeba) também contribui para que os atrasos na transmissão de mensagens sejam não-determinísticos. Esses fatores, responsáveis pelas componentes T_o e T_d relativas ao tempo de transmissão de mensagens (ver Capítulo 2), são os seguintes:

1• ao se fazer uma solicitação de envio de mensagens, não há garantia de que este pedido será atendido imediatamente. Por ser uma operação de entrada e saída, o processo que a executou pode entrar em modo bloqueado no sistema operacional, o qual só executará a tarefa quando o processo estiver pronto para executar. Como o escalonamento do processo não é determinístico, pode haver um retardo adicional até que se consiga realmente começar a transmitir a mensagem;

2• a camada de enlace de dados no nível LLC pode retardar o envio de quadros (e portanto das mensagens dos processos) devido a um eventual controle de fluxo e à possibilidade de tratamento de erros. Essas tarefas geralmente necessitam de quadros adicionais para reconhecimento e outros ainda para um eventual reenvio;

3• durante o próprio envio das mensagens, eventos como disparo de temporizadores ou mesmo interrupções de dispositivos podem ocorrer, e necessitam ser tratados. Uma vez que estamos lidando com um sistema operacional de propósito geral, no qual a carga de processamento não é limitada, as rotinas para tratamento dessas interrupções podem ser chamadas quantas vezes seja necessário. Como não se pode prever quantas vezes elas serão chamadas, não podemos precisar quanto atraso será inserido na transmissão da mensagem;

4• por causa de diferenças de processamento entre a CPU e a placa de rede, ou mesmo devido a um congestionamento na rede, a taxa com que um processo solicita envio de mensagens pode ser maior que a taxa efetiva a que o hardware ou mesmo os *drivers* do sistema operacional podem transmitir. Dessa forma, pode ser necessário colocar as mensagens sendo transmitidas em *buffers* do sistema operacional até que a transmissão possa ser efetivada, o que também determina um atraso adicional no tempo de transmissão de uma mensagem.

Os fatores acima, mais os fatores mencionados na seção anterior sobre os problemas relacionados com o uso de uma rede tipo *Ethernet*, são os maiores desafios que devem ser tratados numa solução para o nosso problema: prover comunicação síncrona no sistema operacional Amoeba, utilizando uma rede *Ethernet*. Essa solução é detalhada no Capítulo 4 a seguir. Na próxima seção abordaremos algumas alternativas nas quais nos baseamos para a solução apresentada mais adiante.

3.4. REDES DE TEMPO REAL

Uma área que pode nos ser de muito interesse para o nosso problema é a área de redes de tempo real. Aqui procura-se assegurar a comunicação em tempo real, para dar suporte às aplicações distribuídas de tempo real. Neste tipo de aplicação, existem prazos (*deadlines*) a serem cumpridos, e para isso a comunicação entre os processadores também precisa ser em tempo real.

Em [Veris93], definem-se os atributos que uma rede de tempo real deve possuir. São eles:

- atraso para transmissão de mensagens conhecido e limitado;
- comportamento determinístico na presença de fatores de distúrbio (por exemplo, sobrecarga, falhas);
- reconhecimento de urgência, ou seja, classes especiais de tráfego que ofereçam baixa latência em detrimento do tráfego comum;
- garantia de conectividade, ou seja, capacidade de tratar eventuais partições na rede.

Quando se fala em redes de tempo real, existem duas áreas distintas de aplicações: a das redes locais (LANs), e a das redes a longa distância e redes metropolitanas (as WANs, *Wide Area Networks*, e MANs, *Metropolitan Area Networks*). Vamos nos concentrar apenas nas redes locais por serem mais fáceis de serem estudadas, uma vez que o comportamento de redes de maior distância não é bem controlado do ponto de vista do retardo na transmissão e de sua confiabilidade.

Mesmo para as redes locais, existem duas abordagens que tentam assegurar o comportamento tempo real em uma rede: desenvolvimento de soluções proprietárias, com replicação de componentes de *hardware*, ou mesmo com uso de componentes

especialmente criados para lidar com as possíveis situações de falha na rede; e uma outra abordagem, de menor custo e complexidade, a qual prefere o uso de redes comuns baseadas em COTS, e sem uso de replicação de componentes. A primeira abordagem é obrigatória para sistemas críticos (i.e., controle de tráfego aéreo, controle de processos em usinas nucleares, aplicações militares, etc.).

Para aplicações menos "exigentes", em que não há risco de perda de vida humana e para as quais é possível assumir uma semântica de falha mais restritiva do que a semântica de falha arbitrária[1] para os componentes do meio de comunicação, a segunda alternativa é mais indicada.

No entanto, o uso de uma LAN comum ainda não assegura o comportamento tempo real de que se necessita. Mesmo com retardos finitos em situações normais (ou em algumas situações de falha, como a perda do *token* em redes tipo *token ring* ou *token bus*), ainda é possível a perda de mensagens da aplicação. Com esse tipo de componente, o comportamento tempo real é obtido através de uma abordagem sistêmica, definindo um modelo para a semântica de falha que se espera, e fornecendo protocolos que garantam o comportamento esperado, desde que os pressupostos do modelo se verifiquem. Como o ambiente **Seljuk-Amoeba** usa COTS, a abordagem que visa oferecer um comportamento tempo real a uma LAN é a mais indicada para o problema que temos a resolver.

A seguir veremos duas tecnologias para comunicação entre aplicações de tempo real. A primeira, destina-se a aplicações periódicas, com requisitos rígidos de Qualidade de Serviço (aplicações *hard real time*). Aplicações periódicas são aquelas em que o tempo é o fator de disparo das tarefas; são também chamadas aplicações *time triggered*. Já a segunda tecnologia que discutiremos, não supõe a existência de nenhuma regra para o disparo de tarefas; são assim destinadas a aplicações *event triggered*. Nossa proposta de um serviço de comunicação síncrono foi baseada na combinação das duas tecnologias a seguir.

É importante notar que nem todos os itens que determinam uma rede de tempo real (segundo a classificação apresentada em [Veris93]) são contemplados nas soluções a seguir. No entanto, satisfazem às expectativas das aplicações a que se destinam.

[1] Um componente com semântica de falha arbitrária pode falhar de qualquer forma. Esse tipo de semântica é menos restritiva que a semântica de falha silenciosa, por exemplo, na qual um componente ao falhar simplesmente pára de funcionar. Em [Jalot94] encontramos uma classificação para os tipos de falha de um componente.

3.4.1. Arquiteturas Time Triggered

Numa arquitetura *time-triggered*, todas as tarefas (inclusive envio de mensagens) são periódicas. Para Kopetz e Grünsteidl, apenas arquiteturas *time-triggered* podem garantir tempo de resposta finito para as aplicações [KG92]. Apesar de menos flexível, esse tipo de arquitetura facilita a análise e testes do sistema como um todo (o que é particularmente difícil em sistemas de tempo real).

O **TTP** (*Time Triggered Protocol*) [KG92] foi projetado seguindo essa idéia, sendo utilizado no projeto MARS [KM85]. Objetiva aplicações em que a latência máxima entre um estímulo e sua resposta pelo ambiente deve ser conhecida e pequena; seu foco principal é, portanto, aplicações com requisitos rigorosos de tempo de execução. Daí a necessidade de um serviço de comunicação que garanta a transmissão de mensagens em tempo finito e que esse tempo seja o menor possível.

No TTP garante-se tempo finito em transmissões de mensagens através de um método TDMA de acesso ao meio físico. Num método de acesso TDMA, cada processador na rede possui um intervalo próprio - *slot* - para transmissão de mensagens, o qual se repete seguidamente como num ciclo; a transmissão de mensagens só pode ocorrer durante o *slot* de cada processador. Com este método, sempre é possível saber em que momento um processador poderá transmitir, facilitando assim a construção e testes de sistemas de tempo real. No entanto, uma desvantagem deste método é a baixa taxa de utilização da rede em situações de tráfego pouco intenso. Um processador que necessite fazer uma transmissão no momento em que não se encontra no seu *slot*, terá que esperar os *slots* dos outros processadores (que eventualmente podem não ter o que transmitir) até poder transmitir. Vemos então que a taxa de utilização da rede, em situações de baixo tráfego é inversamente proporcional ao número de processadores envolvidos no TDMA.

No caso do TTP, cada nodo (um nodo na terminologia adotada pelo TTP é equivalente a um processador em nosso ambiente de nodos replicados) recebe uma fatia de tempo para enviar mensagens e o faz em intervalos conhecidos. No entanto, a fatia de tempo para o envio de mensagens não é necessariamente a mesma para todos os nodos; aqueles que geram mais tráfego podem ser privilegiados. Para se garantir que haja sincronização entre todos os nodos de forma que estes só acessem a rede nos seus intervalos específicos, os relógios dos nodos são sincronizados através de *hardware* específico, com a implementação de circuitos especiais nas próprias placas de rede.

A topologia do meio físico de comunicação é baseada num barramento. Para tolerar faltas do próprio meio de comunicação, esse barramento é duplicado, determinando que cada nodo tenha duas placas de comunicação que se conectam com cada um dos barramentos.

A principal vantagem do TTP é o tempo de resposta garantido, mesmo sob condições de pico para o tráfego na rede. Esta é uma grande vantagem para sistemas de tempo real e se constitui num bloco básico para a construção de arquiteturas tolerantes a faltas, ou nodos com semântica de falha controlada. O problema é que para se garantir que o tempo de transmissão de mensagens seja mínimo, o TTP deve ser implementado no próprio *hardware* das placas controladoras de rede (na forma de circuitos VLSI), o que reduz a flexibilidade de adotá-lo em ambientes de rede quaisquer.

3.4.2. Arquiteturas Quase-Síncronas

Essa solução (apresentada em [AV96a, AV96b e AV95]) é menos rigorosa quanto a garantir atraso finito em transmissões na rede. Destina-se a aplicações cujos requisitos de confiabilidade são também menos rigorosos. Para entendermos melhor para que tipo de aplicações é indicada, é necessário distinguir as aplicações distribuídas em tempo real em duas classes diferentes.

A primeira classe se refere às aplicações críticas, que podem apresentar risco à vida humana ou causar graves prejuízos; neste caso, deve-se saber com antecedência todas as situações às quais o ambiente estará sujeito, e os requisitos de tempo devem ser assegurados impreterivelmente, mesmo em situações de sobrecarga do sistema. Podemos citar como exemplo de aplicações dessa classe aquelas que lidam com equipamentos nucleares ou controle de vôo de aviões.

A outra classe de aplicações em tempo real abrange uma variedade muito maior. Para essas aplicações, os requisitos de tempo são importantes, mas eventualmente pode-se aceitar uma diminuição gradual na Qualidade de Serviço (QoS) da rede; tais aplicações normalmente executam em ambientes em que é difícil estimar um cenário de pior caso; se fosse possível, bastaria supor um valor máximo para a latência na rede, o qual nunca seria ultrapassado. Nestes ambientes, o pior caso é tão discrepante com relação à situação normal, que não vale a pena ser considerado. É para essas aplicações que uma arquitetura quase-síncrona se destina. Aplicações multimídia, que precisam gerar tráfego em tempo real são exemplos de aplicações que podem utilizar esta alternativa.

Assim, opta-se por fornecer um serviço de comunicação não totalmente síncrono (quase-síncrono), que garante entrega de mensagens em tempo finito apenas para aquelas de maior prioridade (mensagens de controle), pertencentes a um serviço de detecção de falhas de temporização. É através desse serviço que se consegue detectar a ocorrência de uma falha de comunicação e reportá-la para a aplicação.

O sistema funciona como se possuísse dois barramentos: um de controle, síncrono, i.e., com entrega garantida de mensagens em tempo finito, e outro, assíncrono, para tráfego de mensagens quaisquer. Essa abstração pode ser conseguida através de um único barramento com o uso de uma rede *token bus* (padrão IEEE 802.4), na qual as mensagens de controle e apenas elas circulam com a maior prioridade possível para a rede. De forma análoga, também se pode conseguir essa abstração com o mesmo mecanismo de prioridades numa rede *token ring* (padrão IEEE 802.5). O tráfego normal deve seguir com menor prioridade.

Para o tráfego que passa pelo canal assíncrono (de mais baixa prioridade, portanto) estima-se um valor para o atraso na rede, o qual é bem inferior ao que corresponderia ao pior caso, mas mais próximo da situação de carga normal. O serviço de detecção de falhas de temporização, funcionando no canal síncrono, monitora o canal assíncrono de forma que quando o tempo estimado não for obtido (por perda da mensagem ou chegada com atraso), a aplicação tem como ser notificada e pode tratar essa falha da forma mais conveniente.

Apesar de não ter a característica de uma rede de tempo real propriamente dita, para a qual todas as mensagens têm entrega garantida num tempo máximo, essa propriedade é conseguida para o tráfego de controle, pertencente ao serviço de detecção de falhas.

Essa solução não prevê nenhum *hardware* específico que a suporte, sendo por isso apta a ser utilizada em qualquer ambiente de rede que permita a abstração de um canal síncrono. No entanto, o fato de se permitir que as falhas ocorram (perda de mensagens, por exemplo) entra em desacordo com o que pretendemos. Os protocolos do nodo Voltan necessitam que a entrega de mensagens seja feita dentro de um intervalo de tempo conhecido. Apenas detectar que houve uma falha não é suficiente ainda para nossos propósitos.

3.5. CONCLUSÃO

Neste capítulo vimos os problemas que teremos que enfrentar para inserir comunicação síncrona num ambiente assíncrono: tratamento de não-determinismo causado pelo uso de redes tipo *Ethernet* e sistemas operacionais de propósito geral. Vimos ainda duas alternativas para redes de tempo real: abordamos o protocolo TTP e arquiteturas Quase-Síncronas.

Com relação às duas tecnologias de redes de tempo real abordadas podemos identificar duas características principais que consideramos importantes: o método de acesso TDMA utilizado pelo TTP, que oferece acesso ordenado (sem colisão) a um meio de comunicação compartilhado, como um barramento; e a separação de duas classes distintas de tráfego na rede (tráfego síncrono e assíncrono) presente em arquiteturas Quasi-Síncronas, que permite separar mensagens de controle de mensagens relativas ao tráfego comum das aplicações.

No próximo capítulo detalhamos como nossa proposta se utilizou das idéias presentes nestas duas tecnologias de redes de tempo real para fornecer um serviço de comunicação síncrono utilizando uma rede local comum, tipo *Ethernet*.

4.

Projeto de um Serviço de Comunicação Síncrono sobre uma Rede Assíncrona

4.1. INTRODUÇÃO

Inicialmente, vamos recordar o problema que temos que resolver. Precisamos que cada processo *Ordenador* possua um tempo máximo para transmissão de mensagens, de maneira que consiga determinar com precisão a ordem em que as mensagens que chegam devem ser processadas. No Capítulo 2 definimos esse tempo máximo como sendo d, e mostramos que d é composto de três outros intervalos de tempo: o tempo de

processamento na origem, antes de a mensagem chegar ao dispositivo (placa) de rede para ser efetivamente transmitida (T_o); o próprio tempo de envio da mensagem (T_e); e o tempo de processamento no destino, até a mensagem ser efetivamente entregue ao processo destinatário (T_d). Assim, nosso objetivo neste capítulo é propor uma solução que permita calcular o valor de *d* para o ambiente **Seljuk-Amoeba**. Para tanto, é necessário estabelecer valores máximos para T_o, T_e e T_d.

Uma vez que a implementação do **Seljuk-Amoeba** irá se utilizar de uma rede tipo *Ethernet*, utilizando o método de acesso CSMA/CD, não é possível estabelecer um valor máximo para T_e, a menos que mecanismos especiais sejam adicionados ao sistema. Nossa intenção é de utilizar as idéias apresentadas no capítulo anterior, presentes no TTP e em arquiteturas Quase-Síncronas, mas aplicadas sobre uma rede tipo *Ethernet*. Assim, para se evitar as colisões, simula-se um método TDMA em *Ethernet*, identificando dois tipos de tráfegos, síncrono e assíncrono. Apenas para o tráfego síncrono é possível determinar o valor de T_e. No entanto, é preciso adicionar mecanismos de controle de fluxo para se evitar pedidos de transmissão de mensagens além da capacidade permitida para o tráfego síncrono.

Como discutido anteriormente, apenas estabelecer um valor máximo para T_e não é suficiente para se garantir comunicação síncrona entre dois processos corretos. É preciso ainda estabelecer valores máximos para T_o e T_d. Estes, por sua vez, são influenciados por diversos fatores como escalonamento, tempo de troca de contexto no sistema operacional, ou mesmo tratamento de interrupções.

O capítulo está estruturado da seguinte forma: na seção 4.2 encontra-se uma discussão sobre os mecanismos nos quais baseamos nossa proposta. A seção 4.3, por sua vez, detalha nossa proposta para se garantir *d*, ou seja, o valor máximo de $T_o + T_e + T_d$. Já na seção 4.4, é construída uma arquitetura que identifica os módulos componentes do serviço de comunicação proposto, com o objetivo de facilitar sua posterior implementação; esses módulos levam em conta as considerações apresentadas na seção 4.3. Na seção 4.5 explica-se a necessidade de se inserir mecanismos de controle de fluxo para os processos que compõem o protocolo de ordenação; também se aponta onde esses mecanismos devem ser inseridos. Por fim, na seção 4.6, é discutida a solução apresentada, mostrando seus principais pontos positivos e negativos.

4.2. MOTIVAÇÃO PARA A SOLUÇÃO APRESENTADA

Uma das premissas do ambiente **Seljuk-Amoeba** é de não utilizar componentes proprietários na sua implementação; esses componentes devem poder ser disponíveis livremente no mercado. Seguindo essa orientação, o meio de comunicação utilizado no ambiente é uma rede local baseada em *Ethernet*.

Devido à possibilidade de colisões, comum no método de acesso CSMA/CD utilizado nesse tipo de rede, o atraso na transmissão é não determinístico; assim, precisamos de algum mecanismo que evite a ocorrência de colisão de mensagens. Se for possível simular um método de acesso TDMA (de forma análoga ao utilizado no protocolo TTP abordado no capítulo anterior) sobre uma rede tipo *Ethernet*, eliminamos a possibilidade de colisões, facilitando o cálculo do atraso máximo na transmissão. Assim, garante-se acesso sem colisões ao meio de comunicação, o que elimina a competição por recursos da rede entre os processadores; além disso, acrescenta-se um determinismo no método de acesso à rede, pois sempre é possível saber quando um processador poderá transmitir.

No entanto, apesar de o TDMA possibilitar a eliminação da competição por recursos entre os processadores da rede, este não elimina a competição por recursos entre os processos executando num mesmo processador. Assim, é possível que um pedido de transmissão de mensagens seja atrasado de forma não previsível, dependendo das requisições de outros processos. Faz-se necessário então, distinguir dois tipos de tráfego que podem ser gerados pelos processos: o tráfego síncrono, e o tráfego assíncrono. Essa idéia, utilizada em arquiteturas Quase-Síncronas, permite estabelecer prioridades entre as mensagens originadas pelos processos em execução.

A idéia da separação do tipo de tráfego gerado pelos processos aplica-se diretamente ao modelo de nodos com o qual estamos lidando. Teremos o tráfego gerado pelos processos *Ordenadores* do nodo (que deve ser síncrono, devido à ordenação de mensagens) e o tráfego gerado por outros processos do nodo (ou outros processos quaisquer do sistema), que não precisa necessariamente ser síncrono. Assim, podemos reservar parte dos recursos disponíveis na rede (largura de banda), priorizando o envio de mensagens geradas pelos processos *Ordenadores*. A reserva pode ser feita sem problema no caso dos processos *Ordenadores* porque a informação de ordenação que necessita circular no tráfego síncrono é apenas a parte de controle relativa à tupla <TS, μ, S, O>

definida no Capítulo 2, isto é, sem a componente μ; portanto, o tamanho desse cabeçalho é fixo e conhecido.

Para que esse mecanismo funcione corretamente, é preciso ainda limitar a carga a ser submetida ao tráfego síncrono, sendo para isso necessário executar algum tipo de controle de fluxo. Assim, ao se reservar recursos para o tráfego relativo à ordenação, deve-se estabelecer quantas mensagens serão transmitidas, efetuando a reserva considerando o tamanho do cabeçalho relativo a cada mensagem. A cada rodada de ordenação, o controle de fluxo deve evitar que se transmita mais que o número máximo de mensagens previsto na reserva.

4.3. UM SERVIÇO DE COMUNICAÇÃO SÍNCRONO PARA O SELJUK-AMOEBA - ESPECIFICAÇÃO

Nossa proposta é simular um método de acesso ao meio de comunicação baseado em TDMA, mas construído sobre uma rede tipo *Ethernet*. Além disso, vamos fornecer uma API de comunicação que permita aos processos *Ordenadores* dos nodos replicados reservar para si parte do *slot* TDMA do processador em que estão executando. Maiores detalhes sobre a API podem ser encontrados no capítulo seguinte, que trata de aspectos de implementação. Neste capítulo, e particularmente nesta seção, abordaremos apenas as características que essa API deve ter.

Cada nova reserva vai ser mapeada pelo sistema operacional para uma porção do *slot* do processador no qual o processo está executando; assim, é como se cada processo que fizesse uma reserva tivesse garantido para si um intervalo de tempo dentro do *slot* TDMA do processador no qual executa. Mensagens de mais baixa prioridade, ou seja, o tráfego assíncrono, serão enviadas na porção restante do *slot* TDMA. Dessa forma, esse tipo de mensagem só poderá ser transmitido depois de se ter enviado as mensagens para as quais foi feita alguma reserva no *slot*. Na Figura 4.1 a seguir vemos uma situação em que alguns processadores (os processadores 2 e 3 na figura) executam processos que solicitaram reservas para tráfego síncrono, enquanto que nos outros processadores da rede há apenas o tráfego assíncrono.

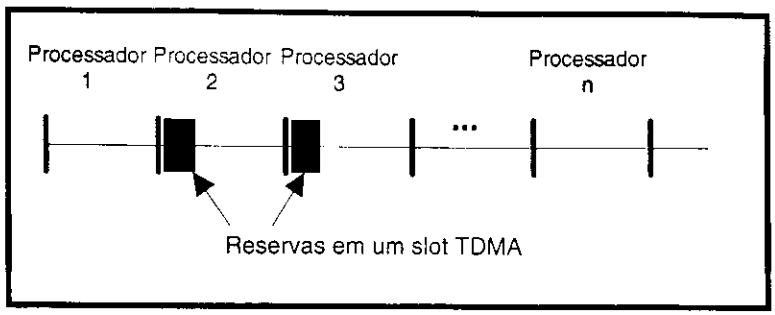

Figura 4.1 - Acesso baseado em TDMA + reserva de QoS

De maneira a permitir a transparência dessa proposta, todo o mecanismo de acesso TDMA deve ser implementado no núcleo do próprio sistema operacional. Assim, nenhuma aplicação (quer seja replicada ou não) precisará se preocupar com qual tipo de acesso ao meio está sendo feito. Isso garante que as aplicações já existentes no sistema e que não possuem requisitos para transmissão síncrona de mensagens (ou seja, não executam de forma replicada), não terão que ser modificadas para se adaptar a esse novo esquema. Dessa forma, não serão necessárias alterações de código fonte e nem mesmo será preciso recompilar tais aplicações.

Do ponto de vista da implementação, o mecanismo TDMA será executado em cada processador na rede da seguinte forma: para cada solicitação de envio de mensagens de um processo qualquer, coloca-se essa mensagem numa fila mantida dentro do núcleo do sistema operacional. Quando o processador que deseja enviar mensagens tem a vez de fazê-lo (i.e., seu *slot* TDMA chegou), é priorizado o envio de mensagens geradas por processos que fizeram alguma reserva, e só depois são enviadas as mensagens dos outros processos.

É necessário ainda considerar que uma mensagem contém também informações de controle relativas ao próprio protocolo de comunicação; esse fator deve ser calculado para que se aloque espaço suficiente no *slot* TDMA, comportando toda informação a ser efetivamente transmitida. Assim, ao enfileirarmos mensagens, na verdade estaremos enfileirando os pacotes que a compõem. Conseqüentemente, durante um *slot* TDMA a unidade mínima para transmissão é formada pelos pacotes que compõem uma mensagem, e não a mensagem propriamente dita. O tamanho máximo desse pacote é dependente do método de acesso ao meio em uso (numa rede *Ethernet*, por exemplo, esse tamanho corresponde a 1572 *bytes* [SLC95], p. 210).

Apesar de a implementação parecer ser relativamente simples, existem ainda alguns outros pontos que precisam ser levados em consideração de forma que essa proposta fique completa. São eles: necessidade de sincronização de relógios; previsão do tráfego a ser gerado pelos processos; cálculo do tamanho livre num *slot*; fatores de incerteza como tratamento de interrupções e tempo de troca de contexto entre tarefas em execução no sistema operacional; e o escalonamento dos processos *Broadcast* e *Ordenador* que compõem o protocolo de ordenação de mensagens do **Seljuk-Amoeba** e para os quais se faz necessária a comunicação síncrona de mensagens. Esses tópicos serão abordados com maior detalhe nas subseções seguintes.

4.3.1. Definições

Para o método TDMA proposto, definimos que o período de um ciclo TDMA[1] corresponde a T unidades de tempo (abreviadamente, **ut**) e existem n processadores na rede participando do TDMA. Assim, temos que num ciclo do TDMA cada processador possui um *slot* para transmitir, de tamanho T/n **ut**.

Chamaremos de B a taxa de transmissão da rede; B é então definido em termos do número de bits por **ut** que se pode transmitir. Supomos também que cada mensagem enviada por um processo é composta por pacotes de tamanho máximo p bits; desses p bits, consideramos que e bits são informações de controle para o protocolo de comunicação. Essas definições serão importantes para as próximas seções.

4.3.2. Semântica de uma reserva

A reserva dentro de um *slot* TDMA será feita em termos da Qualidade de Serviço (*Quality of Service* - QoS) de que um processo precisa, expressa na forma $<y,t>$. Isso significa que cada pedido de envio de mensagens do processo cujo tamanho não exceda y bits de dados será enviado em até t **ut**. Além disso, dentro de um intervalo de tempo t, medido a partir da primeira solicitação de envio de mensagens, o processo não solicitará o envio de mais que y bits. Se isso acontecer, considera-se que o pedido violou a reserva solicitada; este pedido deve ser então colocado numa fila de espera, sendo transmitido apenas quando se assegure que não mais violará a reserva feita.

[1] O período de um ciclo TDMA é o intervalo de tempo compreendido entre dois *slot* consecutivos de um mesmo processador.

O nosso principal objetivo é fazer com que os processos *Broadcast* e *Difusor* do nodo replicado, que necessitam da restrição de transmissão síncrona de mensagens, utilizem-se da reserva como especificada acima antes de começar a transmitir mensagens. Assim, o valor de y corresponderia ao tamanho da informação de ordenação utilizada pelo protocolo, enquanto que t seria equivalente ao tempo máximo para a transmissão da mensagem.

Com o objetivo de facilitar a verificação de aceitação ou não de uma reserva (que trataremos mais adiante), criamos também a notação $<x>$ para identificar uma reserva. Essa notação será utilizada pelo núcleo do sistema operacional para "normalizar" a reserva a ser feita de forma a se adequar a um único *slot*. Isso é necessário porque pedidos de transmissões de mensagens feitos a partir de uma reserva $<y,t>$ podem requerer mais de um *slot* para serem efetuados, já que t pode ser maior que o período T de um ciclo TDMA. Por exemplo, se t é maior ou igual que nT, e menor que (n+1)T, onde n é um número natural maior que zero, então um pedido de transmissão de até y bits pode ser efetuado em até n *slots*, cada *slot* transmitindo até $x=\lceil y/n \rceil$ bits.

Assim, na notação $<x>$ determina-se que se deseja reservar exatamente x bits em cada *slot*. Dessa forma, dada uma reserva no formato $<y,t>$, podemos calcular seu correspondente segundo a notação $<x>$ da seguinte forma: caso $t \geq T$, então $x=\lceil y / (\lfloor t / T \rfloor) \rceil$, caso contrário, i.e. $t < T$, então $x=y$, onde t e T são diferentes de 0. Em última instância, é o valor de x que será reservado em cada *slot* TDMA.

É possível que um pedido de uma reserva não seja aceito; isso dependerá das reservas que já foram feitas até o momento, e dos valores dos parâmetros y e t da reserva $<y, t>$. Dessa forma, para se aceitar uma nova reserva $<y,t>$ é preciso que o valor $<x>$ relativo à normalização dessa reserva possa ser comportado no *slot* TDMA, ou seja, o espaço livre disponível no *slot* deve comportar $<x>$. O cálculo do espaço livre num *slot* será tratado na seção 4.3.5. Com relação ao parâmetro t, este deve ser sempre maior que o atraso máximo estabelecido pelo serviço de comunicação síncrono, i.e. $t \geq d$. O valor de d será discutido na seção 4.3.7 que trata da política de escalonamento dos processos *Broadcast* e *Difusor*. Assim, para se aceitar uma reserva $<y,t>$ qualquer, é preciso que o *slot* TDMA comporte seu valor normalizado $<x>$ e que a componente t seja maior ou igual a d.

4.3.3. Taxa de envio de mensagens

O esquema de reservas deve se prevenir quanto à possibilidade de um processo violar a reserva feita, tentando enviar mais bits do que solicitado. Faz-se necessária, portanto, a existência de um mecanismo de policiamento que verifique cada novo pedido de envio de mensagens de um processo e determine se ele está ou não dentro dos padrões estabelecidos na sua respectiva reserva. Neste caso, só serão enviadas as mensagens de acordo com a Qualidade de Serviço reservada; se o processo enviar mais do que o que havia solicitado como reserva, o atraso da transmissão será maior do que o esperado.

Essa restrição determina que antes de se fazer uma reserva, o processo conheça qual o tamanho máximo das mensagens que irá gerar e o número de transmissões que poderá fazer ao longo de um intervalo de tempo *t*; é claro que nem sempre se conhece essa informação. Assim, é preciso que o processo se previna também quanto a isso, controlando o fluxo com que as mensagens são enviadas.

No Capítulo 2, mostramos que a comunicação síncrona só é necessária no momento em que uma mensagem tem que ser ordenada, ou seja, no momento em que ela sai do processo *Broadcast* para ser difundida entre os outros processadores do nodo (ver Capítulo 2). Assim, no processo *Broadcast*, antes de enviá-la para a ordenação, podemos deixá-la em fila de espera até que o fluxo de mensagens se normalize. O fluxo estará normalizado quando se garantir que o envio de uma nova mensagem não violará a reserva solicitada. Mais adiante neste capítulo, sugerimos um mecanismo para controle de fluxo a ser usado antes de uma mensagem ser difundida para ordenação pelos processadores dos nodos replicados.

4.3.4. Sincronização de relógios

A primeira implicação da utilização de um método de acesso baseado em TDMA, é que os relógios dos processadores na rede precisam estar sincronizados uns com os outros. Se houver perda de sincronismo, é possível que um processador envie dados no *slot* de outro processador, podendo ocasionar perda de dados. Para resolver esse problema, sem ter que usar *hardware* específico para isso (como é feito no TTP), teremos que fazer a sincronização de relógios em *software*. Ou seja, para poder implementar o esquema TDMA, é necessário que o núcleo do sistema operacional também execute algum protocolo de sincronização de relógios em conjunto com os outros processadores na rede.

Um fator que é inerente à sincronização de relógios em *software*, é que os relógios, mesmo sincronizados, ainda podem diferir uns dos outros; entretanto, essa diferença é limitada por uma constante, digamos ε **ut**. Portanto, apesar de os relógios dos processadores estarem sincronizados entre si, pode haver um erro entre as leituras de quaisquer dois relógios corretos; esse erro é equivalente a no máximo ε **ut**, o que determina que ainda é possível que os *slots* de dois processadores quaisquer se sobreponham. Dessa forma, é preciso que cada processador leve essa diferença em consideração, esperando um intervalo de tempo adicional de ε **ut** além do tempo necessário para seu *slot* "chegar", antes de realmente começar a transmitir suas mensagens. A Figura 4.2 a seguir ilustra uma situação que pode ocorrer se o valor de ε não for levado em conta.

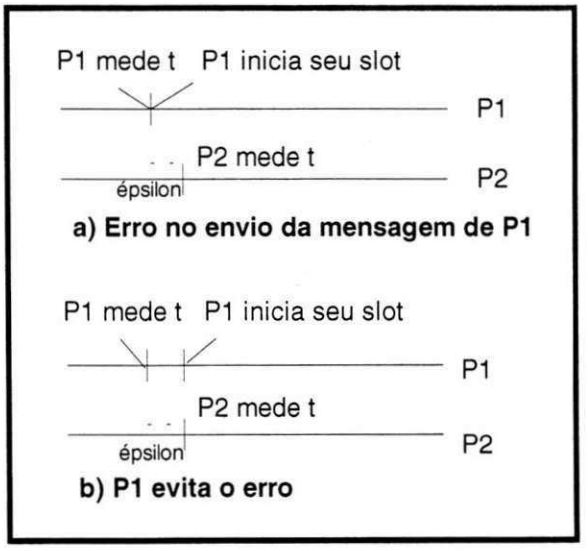

Figura 4.2 - Tratando o intervalo ε para sincronização de relógios

Digamos que o processador P1 esteja medindo o tempo corretamente e esteja esperando a chegada do seu *slot* que deve vir no instante *t*. O problema é que o processador P2 pode ainda estar transmitindo mensagens; no pior caso, este processador só medirá o instante *t* (que indica o fim do seu *slot*) dentro de ε **ut** (situação 4.2 a)). Dessa forma, P1 deve atrasar o início de seu *slot* em ε **ut**, para garantir que não haverá problemas na transmissão de suas mensagens (situação 4.2 b)).

O problema de sincronizar relógios em sistemas distribuídos é largamente estudado na literatura e existem vários protocolos para isso disponíveis. Uma descrição sucinta desse problema pode ser encontrada em [Jalot94], pp 89-99. Na nossa implementação usaremos o protocolo descrito em [CM96], devido ao fato de poder ser implementado

utilizando o tráfego assíncrono e por permitir um valor pequeno para ε (da ordem de 150 milisegundos). Mais adiante neste capítulo, na discussão relativa à arquitetura proposta, detalharemos o funcionamento deste protocolo e onde este deverá ser inserido.

4.3.5. Cálculo da porção utilizável de um slot

Como vimos anteriormente, para se aceitar uma nova reserva $<y,t>$, temos que verificar se o *slot* TDMA pode comportar seu valor normalizado $<x>$. Para isso devemos determinar o espaço livre disponível para reservas num *slot* TDMA.

A maneira mais imediata de se fazer isso é simplesmente diminuir cada reserva feita do total de recursos disponíveis (espaço total no *slot*), e se o resultado dessa diminuição comportar a nova reserva, ela pode ser aceita. No entanto, veremos mais adiante que alguns outros itens devem ser levados em conta no momento de se calcular o espaço livre no *slot*.

Vamos seguir inicialmente o raciocínio mais simples. O espaço total do *slot* (ou o tamanho do *slot*) pode ser expresso em termos do total de bits que podem ser enviados naquele *slot*. Então, supondo que uma rede consiga transmitir a uma taxa de B bits por **ut**, e que o período de um ciclo TDMA equivale a T **ut**, deduz-se que ao final de um período TDMA teremos BT bits para serem compartilhados entre todos os processadores na rede. Mais ainda, uma vez que existem n processadores, podemos ter n *slots* TDMA de tamanho $\lfloor (B\mathrm{T})/\mathrm{n} \rfloor$, ou seja, poderemos enviar $\lfloor (B\mathrm{T})/\mathrm{n} \rfloor$ bits em cada *slot* ao final de um período do ciclo. Suponhamos agora que i processos fizeram suas reservas num processador qualquer, solicitando respectivamente as reservas (normalizadas) $<x_1>$, $<x_2>$, ..., $<x_i>$. Dessa forma, a porção livre do *slot* TDMA desse processador, corresponde a: $\lfloor \mathbf{(BT)/n} \rfloor - (\mathbf{x_1 + x_2 + ... + x_i})$.

Nas próximas seções, veremos alguns outros fatores que devem ser considerados no cálculo do tamanho livre de um *slot*. Sempre que for proposta uma correção neste cálculo, esta correção virá sublinhada.

4.3.5.1. Considerando uma alocação mínima para o tráfego assíncrono

É importante levar em conta a necessidade de se garantir justiça para o envio de mensagens, sejam elas originadas de uma reserva ou não. Da forma como foi detalhado até aqui, é possível que quase todo o *slot* TDMA de um processador seja reservado

unicamente para tráfego síncrono. Apesar de terem menor prioridade, as mensagens originadas do tráfego assíncrono também têm que ser enviadas em algum momento. Assim, sugerimos que parte do *slot* seja reservado exclusivamente para mensagens pertencentes ao tráfego assíncrono, podendo ser configurada como um parâmetro passado em tempo de *boot* do sistema. Dessa maneira, supondo que *A* bits devam ser reservados para transmissão de mensagens assíncronas, analogamente à seção anterior, o espaço livre dentro de um *slot* TDMA utilizável em outras reservas passa a ser: $\lfloor (BT)/n \rfloor - A - (x_1 + x_2 + ... + x_i)$.

4.3.5.2. Considerando o valor de ε

Precisamos também levar em conta um item que foi levantado na seção: a máxima diferença entre os relógios de quaisquer dois processadores corretos na rede. Se cada processador precisa esperar um tempo adicional ε **ut** como foi visto anteriormente, antes de começar a transmitir, isso implica que o tamanho livre do seu *slot* TDMA também será reduzido na mesma proporção; essa redução será equivalente a Bε. Portanto, a porção livre do *slot* (ou seja, o número de bits que podem ser usados numa nova reserva) passa agora a ser igual a $\lfloor (BT)/n \rfloor - A - B\varepsilon - (x_1 + x_2 + ... + x_i)$.

4.3.5.3. Considerando cabeçalhos do protocolo de comunicação

Da forma que estamos propondo, as unidades de transmissão num *slot* TDMA são os pacotes que compõem a mensagem; esses pacotes possuem, portanto, informações de controle além da mensagem propriamente dita. O problema é que quando uma reserva é solicitada, a aplicação só tem como dimensionar o número de bits gerado por ela própria, ou seja, a aplicação desconhece que para cada mensagem enviada, bits extras serão adicionados à mensagem em si; é claro que a aplicação não deve ter que se preocupar com isso. Assim, no momento da reserva, esses bits extras que são adicionados à mensagem devem ser levados em consideração pelo núcleo do sistema operacional, que em última instância é o responsável pela alocação do *slot*, e conhece quanta informação extra deve ser adicionada.

De acordo com a discussão inicial desta seção, vamos considerar que um processador qualquer possui num dado momento *i* processos em execução, os quais solicitaram as reservas $<x_1>$, $<x_2>$, ..., $<x_i>$. Devemos então relacionar a reserva solicitada pelo processo com o número máximo de pacotes que ela pode gerar. Considerando que um pacote tem no máximo *p* bits numa rede qualquer, e que desses *p* bits, *e* bits

correspondem a informações de controle, uma reserva de x_i bits determina que o processo poderá gerar até $\lceil x_i / (p-e) \rceil$ pacotes. O espaço máximo que esses pacotes irão ocupar no *slot* é então equivalente a $\lceil x_i / (p-e) \rceil p$. Dessa forma, podemos refazer o cálculo do espaço livre disponível num *slot* TDMA como sendo:

$$\lfloor (\mathbf{BT})/\mathbf{n} \rfloor - A - \mathbf{B}\varepsilon - p * \sum_{j=1}^{i} \lceil x_j / (p-e) \rceil .$$

4.3.5.4. Tratamento de faltas

O modelo de faltas que será tratado aqui diz respeito apenas à possibilidade de perda de mensagens pelo meio de comunicação ou de a mensagem chegar corrompida ao destino. Dessa forma, para o modelo de faltas do nodo de forma geral, não estaremos prevendo faltas arbitrárias do meio de comunicação, por exemplo, provocadas por um processador que "inunda" a rede com mensagens de forma descontrolada ou mesmo processadores cujos relógios podem atrasar ou adiantar aleatoriamente. Apesar de esse modelo ser mais restritivo, ainda está de acordo com o tipo de semântica de falha que se definiu no Capítulo 2 para a implementação dos nodos replicados no ambiente **Seljuk-Amoeba**, isto é, semântica de falha por valor.

Nossa sugestão é modificar um pouco os pressupostos do protocolo de ordenação. Precisamos considerar que, dentro de um intervalo de tempo de duração T' **ut**, só poderá haver no máximo k falhas de transmissão de mensagens como detalhado acima. Essa consideração é semelhante ao tratamento de faltas abordado em [Veris93].

De acordo com a nossa proposta, para cada mensagem transmitida, enfileiramos os pacotes que a compõem. Assim, para assegurar que um pacote pertencente a uma mensagem gerada por um processador correto efetivamente chegue a seu destino, devemos enviá-lo k+1 vezes, considerando o modelo de faltas acima. Esse tratamento também é sugerido em [Veris93].

Como estamos supondo que o tamanho de um pacote da rede é de no máximo p bits, chegamos à conclusão de que para cada pacote que se deseja enviar, devemos ter na verdade k+1 transmissões de pacotes de tamanho máximo p bits, ou seja, é como se o processo desperdiçasse kp bits em cada *slot* para cada pacote enviado. Esses bits adicionais têm que fazer parte da reserva feita pelo processo, da mesma forma que se considerou os bits extras correspondentes a informações de controle do protocolo de

comunicação. Portanto, se considerarmos que o intervalo T' corresponde exatamente ao período T de um ciclo TDMA, essa sobrecarga adicional também pode ser levada em conta no momento de uma reserva.

De acordo com as discussões anteriores, podemos supor que qualquer processo *i* enviará no máximo $\lceil x_i / (p\text{-}e) \rceil\ p$ bits em cada *slot* (notar que esse valor já inclui a informação adicional relativa ao protocolo de comunicação); agora, esses dados serão enviados *k*+1 vezes. Assim, de forma análoga às seções anteriores, o espaço livre dentro de um *slot* TDMA apto a ser usado em outras reservas corresponde a: $\lfloor (BT)/n \rfloor - A - B\varepsilon - \underline{(k+1)}\ p * \sum_{j=1}^{i} \lceil x_j / (p-e) \rceil$.

4.3.6. Outros fatores de incerteza

Por simplificação, a especificação acima desconsiderou os fatores tratados na seção 3.3 do capítulo anterior. Esses fatores são especialmente difíceis de se medir num sistema operacional de propósito geral além de, como vimos, serem potenciais fontes de atraso para a transmissão de mensagens. Dessa forma, no Capítulo 3 vimos a necessidade de tratar os seguintes itens:

1. escalonamento não-determinístico da rotina encarregada de fazer a transmissão de mensagens, que no nosso caso será ativada durante o *slot* TDMA;
2. controle de fluxo e tratamento de erros na camada de enlace de dados (nível LLC);
3. disparo de temporizadores e interrupções de dispositivos durante a transmissão de mensagens, isto é, durante um *slot* TDMA;
4. necessidade de uso de *buffers* do sistema operacional, devido à diferença entre a taxa com que as mensagens são enviadas durante o *slot* TDMA e a taxa efetiva a que podem ser transmitidas.

Para tratarmos a situação 1) acima, sugerimos que a rotina responsável pela transmissão das mensagens seja escalonada com a maior prioridade possível, de forma que o tempo necessário à troca de contexto relativa à sua ativação (devido à chegada do *slot* TDMA) possa ser mensurado. Assim, é possível medir o atraso para a ativação da rotina. Mais adiante, propomos também mecanismos especiais de escalonamento que devem ser introduzidos no **Seljuk-Amoeba** para atender não só aos requisitos da rotina de

transmissão de mensagens durante o slot TDMA, mas também aos processos *Broadcast* e *Difusor*.

Para se tratar um eventual retardo devido ao controle de fluxo e ao tratamento de erros do protocolo LLC, como detalhado na situação 2), propomos que se permita apenas o serviço sem conexão e sem reconhecimento para o protocolo LLC. Esse serviço elimina o controle oferecido pelo protocolo LLC, eliminando também o retardo associado. Isso pode ser feito já que o esquema TDMA permite que apenas um processador faça uso do meio a cada instante, evitando assim colisões.

Com relação à situação 3), para se tratar eventos durante a transmissão de mensagens, podemos separar os eventos em dois grupos: aqueles que devem ser tratados impreterivelmente, mesmo durante o *slot* TDMA, e os que podem ser tratados após o *slot* corrente, ao término da tarefa de transmissão de mensagens. Eventos como as interrupções de dispositivos podem ser desabilitados logo na entrada do *slot* TDMA, e reabilitados no final do *slot*; a idéia é retardar o acionamento da rotina que irá tratar aquela interrupção. No entanto, essa idéia não pode ser usada para se tratar a interrupção do relógio, pois as interrupções certamente ocorrerão várias vezes durante o mesmo *slot*, e não é possível "enfileirar" interrupções pendentes. Apesar disso, uma vez que a taxa e a duração com que as interrupções de relógio ocorrem são conhecidas, é possível calcular o atraso provocado por essas interrupções. Já eventos como disparo de temporizadores só serão executados ao final do *slot* TDMA, uma vez que a tarefa de transmissão detém a maior prioridade possível para execução. Vemos então que dos eventos considerados acima, apenas o processamento das interrupções de relógio traria algum retardo para as transmissões durante o *slot*.

Para lidar com a situação 4) acima, que trata da possibilidade de ser necessário colocar mensagens em *buffers* do sistema é preciso conhecer a taxa real com que mensagens podem ser transmitidas. Essa taxa deve refletir a taxa com que a tarefa de transmissão de mensagens dentro do *slot* pode conseguir enviar dados. Assim, podemos acomodar essa taxa no valor de B.

Os atrasos decorridos dos fatores 1) e 3) podem ser contabilizados em cada *slot* TDMA, da mesma forma que contabilizamos outros atrasos mencionados na seção 4.3.6, como o valor de ε, o tratamento de faltas, etc. Para contabilizar o atraso, devemos

considerar o número de bits úteis de um *slot* que são perdidos para cada atraso exposto acima e subtrair do espaço livre no *slot* destinado a reservas.

O grande problema, no entanto, é que esses valores (não só os atrasos presentes nas situações 1 e 3, mas também a medição da taxa real de transmissão de mensagens) são difíceis de serem medidos e só uma implementação com testes que levem em consideração situações de sobrecarga no sistema pode nos dar uma precisão correta da ordem de grandeza dos valores desses atrasos.

4.3.7. Escalonamento dos processos Broadcast e Difusor

Uma das suposições do protocolo de ordenação dos nodos replicados proposto em [Brasi95] é de que o tempo de processamento das instruções do protocolo de ordenação poderia ser calculado e inserido no valor de *d*; para validação do protocolo utilizou-se um ambiente de testes mais "bem comportado", no qual se conhecia a carga máxima a ser submetida em cada processador do sistema. Nosso problema é que em situações normais de uso, não há como prever corretamente a carga a que serão submetidos os processadores na rede.

Portanto, ao se transmitirem as mensagens de ordenação nos processos *Broadcast* e *Difusor*, não temos como garantir o intervalo de tempo para a recepção dessas mensagens pelo processo *Difusor* de cada processador destino. No Capítulo 2 vimos que $d=max(T_o+T_e+T_d)$; com o esquema de reservas discutido nas seções anteriores é possível conhecer o valor de T_e, mas não o de T_o e T_d, os quais são dependentes do escalonamento e do tempo de execução dos processos *Broadcast* e *Difusor*, envolvidos na ordenação de mensagens.

Assim, para possibilitarmos o cálculo de *d*, nossa proposta é fazer com que os processos *Difusor* e *Broadcast* executem em modo não-preemptivo e com a maior prioridade possível em cada processador componente de um nodo. A execução em prioridade mais alta assegura que assim que o processo puder sair do modo bloqueado, ele será ativado o mais rápido possível; por outro lado, uma vez que seja ativado, é preciso que o seu processamento não seja interrompido, daí a necessidade da execução não-preemptiva. Além disso, é preciso que esses processos sejam alocados durante um intervalo de tempo fixo em cada processador, de forma que se possa prever os instantes

em que as mensagens de ordenação serão geradas (por um processo *Broadcast* ou *Difusor*) e os instantes em que serão recebidas por cada processo *Difusor* do nodo.

No entanto, existe uma outra entidade que teria que ter as mesmas características de escalonamento dos processos acima: a rotina de transmissão de mensagens durante o *slot* TDMA que tratamos anteriormente. Como essa rotina trata da transmissão das mensagens a cada *slot*, ela não pode ser interrompida durante seu processamento e sua ativação tem que ocorrer dentro de um intervalo máximo de tempo, sob pena de não se conseguir garantir as reservas solicitadas. Dessa forma, temos três entidades com mesma prioridade competindo pelo processador.

Nossa proposta define um escalonamento estático das três tarefas acima de forma que não possam competir entre si pelo processador, mas que mantenham as características de máxima prioridade e execução não-preemptiva de que necessitam.

A rotina de tratamento do *slot* deve ser escalonada independentemente dos outros dois processos do protocolo de ordenação; além disso, sua duração é dependente apenas do período T do TDMA e do número n de processadores envolvidos no TDMA. Assim, sua duração corresponde a T/n.

Os processos *Broadcast* e *Difusor* por sua vez, devem ser escalonados pelo menos uma vez entre dois *slots* consecutivos de um mesmo processador. Idealmente, esses processos devem ser escalonados imediatamente antes de um *slot*; isso permite que se possa enfileirar um maior número de mensagens a serem tratadas por cada um deles.

A duração da execução do processo *Broadcast* vai depender da reserva <*y,t*> feita; dessa forma, esse processo deve executar tempo suficiente para processar a quantidade máxima de mensagens determinada pela reserva.

Com relação ao processo *Difusor*, seu tempo de execução depende do tráfego gerado pelo processo *Broadcast*. Cada mensagem difundida pelo *Broadcast* deve ser difundida pelo processo *Difusor* de todos os outros processadores do nodo. Para um nodo com semântica de falha mascarada, o número de mensagens geradas a partir de cada mensagem difundida pelo *Broadcast* é dado pela equação:

$$(N-1)\sum_{j=1}^{\pi} A_{(j-1)}^{(N-2)}(N-j-1).$$

Esta equação é derivada da seguinte forma: cada processo *Difusor* pode receber mensagens cuja difusão foi iniciada por até N-1 processos *Broadcast* de outros processadores do nodo (primeiro termo da equação, equivalente a N-1); o segundo termo é relativo ao somatório de todos os possíveis arranjos de mensagens contendo de 1 até π assinaturas; cada mensagem do arranjo, por sua vez, deve ser difundida para todos os outros processadores que ainda não a assinaram. Dado que j processos assinaram a mensagem e que a mensagem difundida conterá mais uma assinatura, o número de processos que não assinaram a mensagem é dado por N-j-1.

Assim, o tempo que o *Difusor* irá executar deve ser suficiente para tratar esse número de mensagens (tratar uma mensagem significa recebê-la e difundi-la para os outros processadores que ainda não a assinaram). Portanto a reserva que deve ser feita para o *Difusor*, tem que levar em conta o tráfego que pode ser gerado pelo processo *Broadcast*; para cada mensagem enviada por um processo *Broadcast*, serão necessárias tantas difusões (realizadas por processos *Difusor*) quantas determinadas na equação acima.

Entretanto, o esquema acima ainda possui um problema: durante a execução do *Difusor* ou do *Broadcast*, é possível que outro processador do nodo esteja transmitindo e, portanto, podem chegar mensagens para serem tratadas pelo processador no qual esses processos executam. Isso determina que durante sua execução, esses processos podem ser interrompidos para que seja possível tratar mensagens provenientes de outros processadores do nodo, introduzindo um fator de incerteza no cálculo dos seus tempos de execução. Para evitar isso, podemos inserir no TDMA processadores que nunca transmitem, fazendo com que esses processos executem exatamente durante o *slot* desses processadores. Na verdade, não é necessário que eles existam realmente; basta determinar que o número de processadores participando do TDMA seja maior do que os que efetivamente irão transmitir, e associar a cada processador do nodo posições não consecutivas com relação a seus *slots*. Para ilustrar um exemplo de como esse esquema funciona, na Figura 4.3 mostra-se o escalonamento descrito acima para o caso de um nodo de falha mascarada com três processadores (também conhecido como TMR – *Triple Modular Redundancy*). Concentramos nossa discussão em nodos com esse número de processadores devido ao fato de simplificar seu estudo e por ser essa a abordagem mais comum para replicação de processos em nodos com semântica de falha mascarada. Entretanto, o tratamento pode ser estendido para outros tipos de nodos.

Na Figura 4.3 identificamos D, B e S como sendo os intervalos de tempo, respectivamente, para a execução dos processos *Difusor*, *Broadcast* e do *slot* TDMA em um processador qualquer do nodo. Note que na figura, cada processo *Broadcast* e *Difusor* de um processador é executado uma única vez a cada período do TDMA.

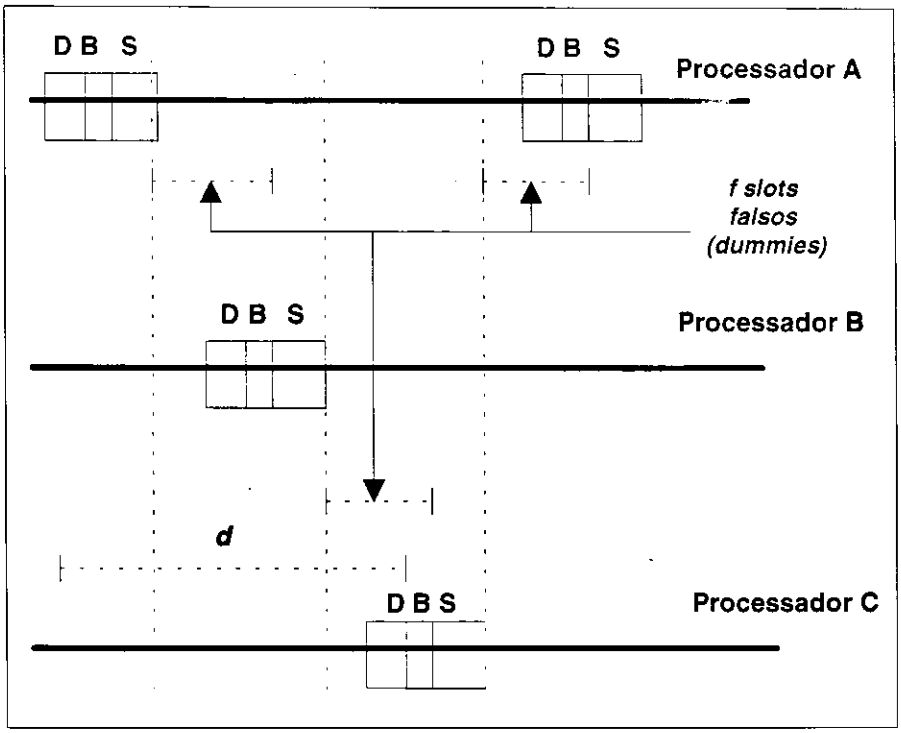

Figura 4.3 - Escalonamento para os processos *Broadcast* e *Difusor*

Na Figura 4.3 podemos identificar também qual o valor de *d* que se deve utilizar para cada processo *Broadcast* e *Difusor*. O valor de *d* deve compreender o maior intervalo possível compreendido entre o início do escalonamento de um processo *Broadcast* ou *Difusor* até a recepção por todos os outros *Difusores* do nodo. De acordo com o escalonamento acima proposto para um nodo TMR, e supondo que foi necessária a inserção de *f slots* falsos, podemos ver que $d = (D+2S+2fS)$. O número *f* de *slots* falsos a ser inseridos vai depender dos tempos de execução de D e B.

Determinar D e B significa determinar por quanto tempo cada processo *Broadcast* e *Difusor* deve ser executado após suas respectivas ativações. Esse tempo dependerá da reserva que se pretende solicitar para cada um. No caso apresentado na Figura 4.3, vamos supor que a cada execução do *Broadcast* seja gerada apenas 1 mensagem. Essa mensagem será então recebida pelos processos *Difusores* executando nos outros processadores do nodo, os quais retransmitem essa mensagem em seguida, um para outro. No nosso caso, temos que N=3 e π=1; pela equação acima, deduzimos que cada *Difusor* poderá transmitir

até 2 mensagens. Assim, o tempo de execução de cada *Broadcast* deve ser suficiente para processar uma mensagem, enquanto que cada *Difusor* deve conseguir processar até duas mensagens. Esse tratamento é análogo para qualquer número de mensagens que deveriam ser transmitidas por cada processo *Broadcast.*

Podemos ainda eliminar o inconveniente de se usar *slots* falsos. Isso pode ser possível, desde que seja garantido que no momento em que se ativa cada processo *Broadcast* ou *Difusor*, não haja tráfego destinado ao processador em que esses processos executam. Se forçarmos que os processadores "falsos" façam parte de outro nodo, é garantido que durante a transmissão de mensagens relativas ao tráfego síncrono desses processadores, nenhum processador de outro nodo receberá estas mensagens. Dessa forma, cada processo *Broadcast* e *Difusor* executaria na parcela relativa ao tráfego síncrono desses processadores pertencentes a outro nodo.

Entretanto, de acordo com nossa suposição de que os processadores possuem semântica de falha por valor, é possível que devido a uma falha, uma mensagem que deveria conter o endereço físico de um processador pertencente a um nodo, possua o endereço de um outro processador de um outro nodo; é possível então que essa mensagem acabe chegando durante o processamento dos processos *Broadcast* e *Difusor* desse processador. No entanto, se a probabilidade com que uma situação desse tipo ocorra for pequena o suficiente, essa alternativa pode ser seguida.

4.4. UM SERVIÇO DE COMUNICAÇÃO SÍNCRONO PARA O SELJUK-AMOEBA - ARQUITETURA

De acordo com as discussões apresentadas na seção anterior, podemos identificar os seguintes módulos componentes da nossa proposta de um serviço de comunicação síncrono: módulo de Reserva, módulo de Policiamento, módulo de Sincronização de Relógios, e módulo de Envio de mensagens. Esses módulos devem ser executados no núcleo do sistema operacional.

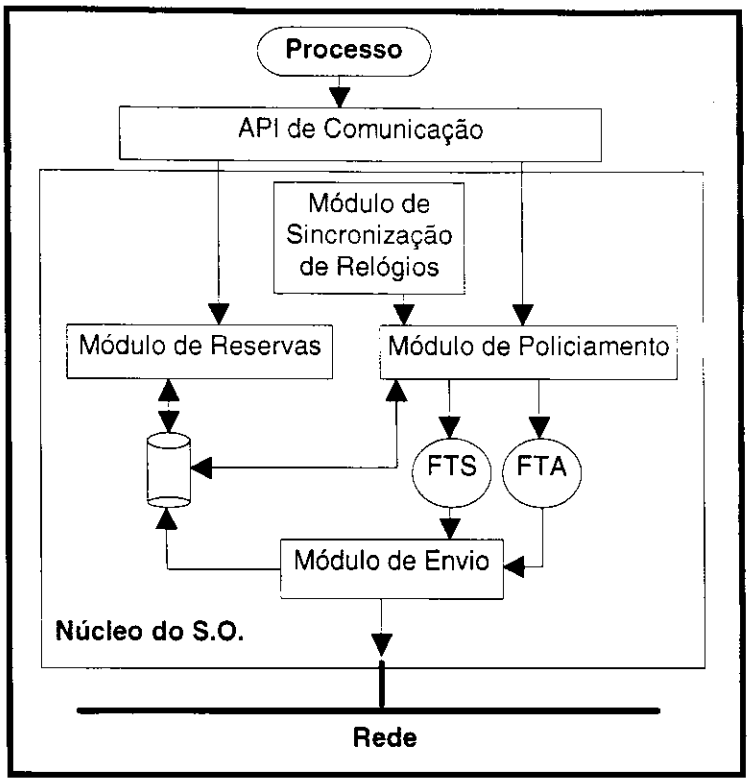

Figura 4.4 - Arquitetura proposta

A arquitetura encontra-se esboçada na Figura 4.4 acima, sendo mostrada apenas a visão de um único processador. Na figura pode-se ver que um processo pode tanto efetuar reservas como enviar dados à rede, através de uma API de comunicação; essa API será detalhada no próximo capítulo, o qual contém os aspectos de implementação da proposta aqui apresentada. A seguir vamos esboçar as funções de cada módulo e os relacionamentos que os módulos guardam entre si.

4.4.1. Módulo de Reservas

É neste módulo que se faz as verificações de aceitação ou não de uma nova reserva. Assim, devemos levar em conta as discussões para determinar o espaço livre dentro de um *slot* levantadas na seção 4.3. Para tanto, deve-se utilizar a notação <*x*> a partir da reserva <*y,t*> que é especificada pelo processo.

Este módulo deve manter as reservas já efetuadas numa estrutura de recuperação rápida (numa tabela por exemplo); o módulo também é responsável pela posterior retirada das reservas presentes na tabela. As informações na tabela de reservas irão servir tanto para o módulo de Envio de mensagens, quanto para o módulo de Policiamento. Nas próximas seções detalha-se algumas informações que devem estar presentes na tabela de reservas e que serão necessárias para cada um desses módulos.

4.4.2. Módulo de Policiamento

Este módulo intercepta as requisições de envio de mensagens para verificar se estão de acordo com uma reserva eventualmente feita. Caso não haja nenhuma reserva para o processo, a mensagem (na verdade, os pacotes físicos que a compõem) deve seguir para a fila de tráfego assíncrono (a fila FTA da Figura 4.4).

Caso tenha sido feita alguma reserva para o processo, deve-se recuperar a informação dessa reserva e verificar se o novo pedido está de acordo com a reserva efetuada. Se estiver, a mensagem, ou melhor, os pacotes da mensagem são armazenados na fila para o tráfego síncrono (a fila FTS da Figura 4.4). Caso a solicitação de envio esteja excedendo a reserva feita inicialmente, os pacotes da mensagem devem ir para uma fila de espera relativa àquele processo (mantida junto com a informação da sua reserva), até que possam ser corretamente inseridos na fila FTS.

Para se determinar se uma solicitação de envio está ou não excedendo uma reserva feita, basta que se disponha do número de bits já enfileirados em FTS para aquele processo. Essa informação é incrementada pelo próprio módulo de Policiamento a cada inserção na fila FTS, e posteriormente zerada pelo módulo de Envio.

É no módulo de Policiamento também que se escalona a retirada dos pacotes enfileirados na fila de espera do processo. Maiores detalhes de como isso é feito podem ser encontrados no próximo capítulo, que trata de detalhes de implementação.

De maneira a simplificar o tratamento do módulo de Envio, é no módulo de Policiamento também que são inseridas as k+1 cópias dos pacotes das mensagens, necessárias para contornar possíveis faltas durante a transmissão de mensagens.

4.4.3. Módulo de Envio

Este módulo é o responsável pelo envio dos pacotes relativos às mensagens dos processos. Ele tem sua execução determinada pelo período T do método TDMA e pelo tempo de espera adicional relativo a ε. Assim, ao final de cada *slot*, este módulo tem sua execução escalonada para se iniciar após T+ε **ut**.

Quando este módulo é ativado (isto é, quando chega o *slot* do processador no qual está executando), deve-se percorrer a fila FTS enviando os pacotes relativos ao tráfego síncrono; quando se exaurir a fila FTS, percorre-se a fila FTA enviando agora os pacotes

dos processos que geraram tráfego assíncrono. Os pacotes do tráfego assíncrono são enviados até acabar a fila FTA, ou se chegar até o fim do *slot*.

Neste módulo é feita também a atualização do número de bits enfileirados em FTS por cada processo. Assim, ao final do *slot*, essa informação é zerada na tabela de reservas, permitindo ao módulo de Policiamento verificar corretamente se um pedido de transmissão de mensagens não viola uma reserva previamente feita.

4.4.4. Módulo de Sincronização de Relógios

Este módulo executa um protocolo de sincronização de relógios com todos os outros processadores da rede. O protocolo que pretendemos seguir é descrito com maior detalhe em [CM96]. Ele se utiliza das idéias de outros três protocolos de sincronização de relógios encontrados na literatura.

Uma grande vantagem deste método é que ele não exige um limite máximo para o atraso na transmissão de mensagens, restrição que se faz presente numa grande parte de outras alternativas discutidas. Na verdade, existe uma restrição para o atraso, mas o valor utilizado como base em [CM96] é tão alto (da ordem de 30 segundos) que consideramos que esta não seria uma restrição de tempo importante. Além disso, este protocolo não determina uma sobrecarga excessiva com relação ao número de mensagens enviadas, evitando o desperdício de recursos da rede.

O método desenvolvido por Keith Marzullo e Matthew Clegg [CM96] é baseado nos seguintes protocolos: o protocolo de sincronização de relógios de baixo custo (*low cost clock synchronization*) desenvolvido por Drummond e Babaoglu [DB93]; o protocolo probabilístico (*probabilistic clock synchronization*) de Flaviu Cristian [Crist89]; e o protocolo de sincronização a posteriori (*a posteriori clock synchronization*) de Veríssimo e Rodrigues [VR92].

O protocolo de baixo custo de Drummond e Babaoglu possui esse nome devido ao fato de que nenhuma informação de sincronização de relógios é trocada entre os processadores na rede. Ao invés disso, os autores fazem uma série de suposições que permitem derivar as leituras de relógios remotos implicitamente a partir dos *broadcasts* gerados por outras entidades no sistema. Entretanto, supõe-se que os exatos instantes em que os *broadcasts* vão ocorrer sejam conhecidos a priori.

O protocolo de Cristian, por sua vez, se baseia no fato de que os processadores na rede são logicamente conectados ponto-a-ponto uns aos outros. Um processador P_i que deseje saber o valor do relógio remoto de outro processador P_j, envia uma mensagem solicitando que P_j retorne o valor de seu relógio local. Quando a resposta chega até P_i, é possível calcular o erro da leitura através do tempo decorrido desde o pedido de leitura do relógio por P_i até a chegada da resposta e fazer uma estimativa correta do valor atual do relógio de P_j. No entanto, uma desvantagem deste protocolo é a necessidade de se trocar duas mensagens a cada leitura de um relógio remoto; para uma rede com n processadores, isso implica $2n(n-1)$ trocas de mensagens.

Por fim, o protocolo de Veríssimo e Rodrigues se vale do fato de que um *broadcast* numa rede local baseada em barramento é recebido quase que instantaneamente em todos os processadores da rede. Assim, a máxima diferença entre o tempo transcorrido desde o recebimento de um *broadcast* pelo primeiro processador até o momento em que o último processador o recebe, é muito menor que o tempo de transmissão propriamente dito; essa diferença entre os instantes de recebimento do *broadcast* é chamada de *tightness*. Assim, quando um processador faz o *broadcast* do valor corrente do seu relógio, sabe-se que os outros processadores receberão este valor aproximadamente no mesmo instante. Uma desvantagem desse protocolo é que a efetiva entrega de uma mensagem aos processadores na rede pode sofrer atrasos decorrentes do tratamento de interrupções e escalonamento de tarefas no destino. Assim, a suposição da propriedade *tightness* pode ser eventualmente violada, invalidando as leituras dos relógios. Um outro problema, é que o *tightness* não se verifica para o processador que está fazendo o *broadcast*; a mensagem chega no processador de origem bem antes da sua chegada nos outros processadores, devido a otimizações no próprio *driver* de rede.

4.4.4.1. Descrição do protocolo

Para superar as desvantagens de cada método isoladamente, utiliza-se o esquema a seguir. A cada mensagem m disseminada via *broadcast* por um processador P_i qualquer, são concatenadas as seguintes informações de controle: o valor corrente do relógio local de P_i (que vai no campo $m.T$), a identificação do último processador P_j do qual P_i recebeu mensagens ($m.prev\_sender$), o valor do relógio de P_j ao enviar a última mensagem que P_i recebeu ($m.prev\_T$), e o valor do relógio de P_i ao receber essa mensagem ($m.prev\_R$).

Um processador P_i que acabou de receber uma mensagem m de outro processador P_j qualquer, pode deduzir o valor atual do relógio de P_j a partir das informações acima. Para tanto, deve escolher o método apropriado dependendo de qual foi o último processador que enviou mensagens a P_j. Se o último processador do qual P_j recebeu mensagens é o próprio P_i, estima-se o relógio de P_j através do método probabilístico de Cristian [Crist89]; caso contrário, utiliza-se o método de Rodrigues e Veríssimo [RV92].

Para o cálculo através do método proposto em [Crist89], precisamos, além das informações já presentes em m, do valor do relógio local de P_i ao receber m (R), o valor mínimo para o atraso da transmissão na rede (Δ_{min}), e o valor máximo da taxa com que um relógio correto se afasta do tempo real (ρ). Assim, temos que o relógio de P_j pode ter um valor mínimo igual a $[m.T+\Delta_{min}(1-\rho)]$, e valor máximo igual a $[m.prev\_R + (R - m.prev\_T)(1 + 2\rho) - \Delta_{min}(1+\rho)]$. Para maiores detalhes sobre como derivar esses valores, consultar [Crist89].

Já com relação ao cálculo com o método de [RV92], precisamos conhecer, além das informações já presentes em m, o valor do *tightness* para a rede (δ_i). Assim, o valor mínimo do relógio de P_j é igual a $[m.prev\_R - \delta_i(1+\rho)]$, e seu valor máximo é $[m.prev\_R + \delta_i(1+\rho)]$.

De posse desses valores cada processador pode ressincronizar seu relógio local, escalonando a atualização apropriadamente. Em [CM96] mostra-se como escalonar cada ressincronização. O intervalo de ressincronização sugerido é da ordem de 45 segundos.

Fica claro que o protocolo necessita que um número mínimo de *broadcasts* seja feito por cada processador na rede, caso contrário, não se consegue obter as informações necessárias a cada ressincronização. Assim, se a taxa com que os *broadcasts* são realizados pelos processo for menor que o necessário para o protocolo, faz-se necessária a transmissão de *broadcasts* adicionais pelo próprio protocolo de sincronização de relógios. Para o protocolo descrito aqui, é necessário que a freqüência para os *broadcasts* seja de um por segundo.

4.5. SUGESTÃO DE UM MECANISMO DE CONTROLE DE FLUXO

A discussão que levantaremos aqui é direcionada às restrições do protocolo de ordenação de mensagens do ambiente **Seljuk-Amoeba**.

Na seção 4.3, determinamos que o processo *Broadcast* é encarregado de controlar o fluxo com que as mensagens são entregues para ordenação, de maneira a se evitar a violação da reserva feita para o tráfego síncrono. No entanto, o controle de fluxo necessário é bastante simples, devido ao escalonamento fixo dos processos *Broadcast* no sistema.

Uma vez que se sabe o número de mensagens que se considerou no momento da reserva, sempre que se escalonar o processo *Broadcast*, ele deve retirar da fila FMR (alimentada pelo processo *Receptor* local) no máximo o número de mensagens previsto na reserva. Dessa forma, evita-se extrapolar as reservas feitas pelos processos *Difusores* em execução nos outros processadores do nodo e para os quais o processo *Broadcast* deve difundir mensagens.

4.6. CONCLUSÃO

Neste capítulo vimos a especificação das características de um serviço de comunicação síncrono para uso no ambiente **Seljuk-Amoeba**. Apesar de a discussão ter sido voltada para as restrições do protocolo de ordenação dos nodos replicados, qualquer processo que possua características semelhantes também poderia fazer uso do serviço especificado aqui. No entanto, a garantia de tempo finito fim-a-fim não poderia ser conseguida, uma vez que só estamos propondo um escalonamento especial para os processos que executam o protocolo de ordenação num nodo replicado. Essa característica poderia ser conseguida se implementássemos políticas de escalonamento dentro do próprio Amoeba, podendo ser utilizadas por quaisquer processos. Essa idéia faz parte dos melhoramentos que prevemos para esta proposta (ver seção sobre trabalhos futuros no Capítulo 6).

Especificou-se um serviço baseado num método de acesso TDMA, no qual se permite aos processos reservar partes de um *slot* TDMA para seu uso exclusivo. Discutimos também os fatores que devem ser considerados para que esse esquema funcione corretamente: necessidade de sincronizar os relógios dos processadores que executam o TDMA, conhecimento a priori da Qualidade de Serviço que deve ser solicitada, e que situações devem ser consideradas ao se aceitar ou não um novo pedido de reserva, baseado nas reservas já feitas. Para assegurar que um processo não violará uma reserva efetuada, sugerimos também um algoritmo para controle de fluxo das mensagens que deve ser executado antes de elas serem transmitidas. Com o objetivo

também de garantir a comunicação síncrona, propomos uma forma de escalonar os processos *Difusor* e *Broadcast*, de maneira a considerar o processamento correspondente ao envio/recepção de mensagens. Apenas esses processos foram considerados por ser aí que reside a necessidade de se limitar o tempo de transmissão de mensagens no ambiente **Seljuk-Amoeba**.

Neste capítulo discutimos apenas as características e as restrições que devem ser consideradas na implementação de um serviço de comunicação síncrono. Este serviço deve ainda ser disponibilizado aos processos sob a forma de alguma API. No próximo capítulo, vamos sugerir essa API, e detalhar como as características aqui discutidas podem ser inseridas no sistema operacional Amoeba.

5.

Aspectos de Implementação

5.1. INTRODUÇÃO

Neste capítulo encontram-se os detalhes de implementação da proposta apresentada no Capítulo 4; todas as considerações levantadas aqui são direcionadas para o sistema operacional Amoeba. Como vimos no capítulo anterior existem, uma série de fatores de potencial atraso que são difíceis de se mensurar e só uma implementação pode nos dar uma idéia melhor dos valores envolvidos. Infelizmente, não houve tempo para se iniciar a implementação da especificação apresentada no Capítulo anterior. No entanto, é possível dar uma idéia dos algoritmos a serem desenvolvidos para implementá-la e identificar em que pontos no Amoeba eles devem ser inseridos. Esse é o objetivo deste capítulo.

Antes de se tratar como deve ser a implementação dos módulos especificados no Capítulo 4, é necessário um estudo sucinto do suporte de comunicação oferecido pelo sistema operacional Amoeba. Assim, esse estudo será orientado às necessidades de um programador de aplicações, enfatizando o suporte de comunicação do ponto de vista da API (*Application Programer Interface*) disponível, e comentando a forma de usá-la.

Após esse breve estudo sobre a programação de aplicações distribuídas no Amoeba, vamos sugerir uma extensão à API de comunicação oferecida, no sentido de permitir aos processos *Broadcast* e *Difusor* pertencentes ao protocolo de ordenação de mensagens dos nodos replicados, reservar a Qualidade de Serviço necessária, de acordo com as diretrizes desenvolvidas no capítulo anterior. A seguir, detalharemos a proposta apresentada no Capítulo 4, enfocando as principais estruturas de dados necessárias, bem como um esboço dos algoritmos envolvidos de forma a implementar o esquema TDMA no núcleo do Amoeba. A extensão proposta é transparente às aplicações que utilizam a API de comunicação da forma convencional: não será necessário modificar código, nem mesmo recompilar tais aplicações para se adaptarem à extensão proposta para a API.

O capítulo encontra-se estruturado da seguinte maneira: a seção 5.2 aborda a comunicação no Amoeba, dando ênfase em como usar a API disponível; nesta seção

mostramos também como estender a API para permitirmos a especificação da Qualidade de Serviço desejada. Já na seção 5.3, detalhamos cada módulo presente na arquitetura proposta no capítulo anterior, enfatizando como as considerações levantadas no Capítulo 4 podem ser inseridas no Amoeba. Por fim, a seção 5.4 faz uma avaliação da solução como um todo.

5.2. COMUNICAÇÃO NO AMOEBA

A comunicação no Amoeba é dividida em duas camadas lógicas: uma camada de mais baixo nível que se encarrega da comunicação propriamente dita, implementada pelo protocolo FLIP (***Fast Local Internet Protocol***), e uma segunda camada que se utiliza das primitivas definidas pelo FLIP para fornecer ao projetista das aplicações a abstração de RPC (*Remote Procedure Call*) [BN84] e de Comunicação em Grupo (*Group Communication*) [BJ87]. Eventualmente, uma aplicação pode fazer uso direto do FLIP sem necessariamente passar pelo nível de RPC ou de Comunicação em Grupo.

O FLIP responde pelo nível de rede (camada 3 do modelo de referência OSI [DZ83]). A camada superior de RPC e Comunicação em Grupo, por sua vez, funciona a nível de sessão (camada 5). Dessa forma, não existe um nível de transporte, correspondente à camada 4 do modelo OSI; todo o controle de perda de mensagens com seu respectivo reenvio é feito pelas camadas de RPC e Comunicação em Grupo.

As seções a seguir descrevem o funcionamento do FLIP e como propomos alterá-lo de forma a permitir a uma aplicação fazer a reserva da Qualidade de Serviço desejada.

5.2.1. Funcionamento do Protocolo FLIP

O FLIP é um protocolo não confiável baseado em datagramas, ou seja, não é orientado a conexão (*connectionless protocol*). Além do mais, pode haver perda de pacotes, os pacotes podem ser corrompidos, ou ainda, podem chegar numa ordem diferente da ordem de envio. É possível também ocorrer fragmentação de pacotes; isso é necessário quando o tamanho de um pacote FLIP a ser enviado é maior que o tamanho máximo de um quadro definido para o nível de enlace de dados. É importante notar que o reagrupamento dos fragmentos deve ser feito num nível mais alto; não cabe ao FLIP esta tarefa. Assim, a camada de RPC e Comunicação em Grupo é que se responsabiliza por reagrupar os fragmentos de um pacote FLIP no destino.

Cada máquina na rede é conectada por uma *FLIP Box* [KRST93]. Uma *FLIP Box* consiste de três módulos: *Host Interface, Packet Switch* e *Network Interface*. Na Figura 5.1 vemos as relações entre cada módulo de uma *FLIP Box* e as relações desses módulos com outras camadas.

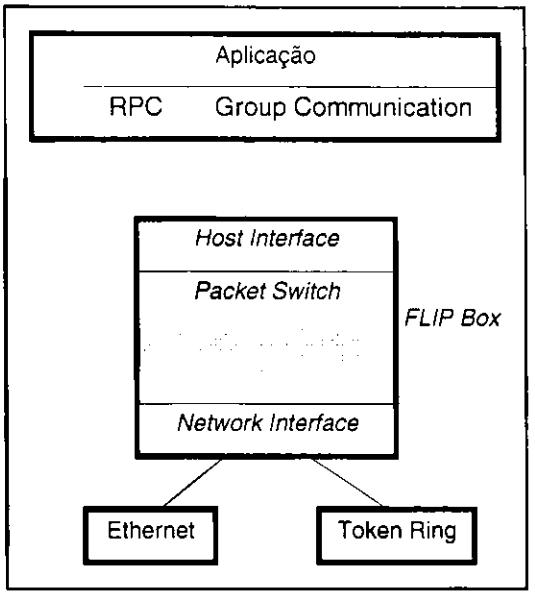

Figura 5.1 - Módulos de uma *FLIP Box*

O módulo *Host Interface* define a interface, ou seja, a API através da qual as camadas superiores fazem uso dos serviços oferecidos pelo FLIP. A API é oferecida ao programador na forma de uma biblioteca a qual é estaticamente ligada ao código da aplicação.

Já o módulo de *Packet Switch* é o coração do FLIP. É o módulo *Packet Switch* que implementa os serviços definidos no módulo *Host Interface*. Neste módulo ocorre a transmissão de fragmentos FLIP entre as máquinas na rede, e se mantém o controle de roteamento de fragmentos. Este módulo é implementado no núcleo do Amoeba.

Por fim, o módulo *Network Interface* representa os *drivers* para comunicação com as interfaces de rede específicas. Na verdade, este módulo é a interface entre o nível de rede e o nível de enlace de dados.

5.2.2. Primitivas FLIP

A API FLIP disponibiliza sete primitivas: *flip_init, flip_end, flip_register, flip_unregister, flip_unicast, flip_multicast,* e *flip_ broadcast.*

A primitiva *flip_init* serve para passar ao FLIP duas funções (*callbacks*) que tratarão os eventos de: chegada de mensagens para o processo; e ocorrência de erro durante a transmissão de mensagens. Notar que uma aplicação que usa o FLIP não fica bloqueada à espera de mensagens; quando as mensagens chegam, a função *callback* especificada em *flip_init* é chamada, e só então se faz o tratamento necessário para a chegada de mensagens. Um mecanismo análogo é feito quando da ocorrência de erros na transmissão da mensagem.

A ativação da primitiva *flip_init* resulta na criação de uma entrada numa tabela do FLIP, referenciando as duas funções *callback* passadas como parâmetros. A primitiva *flip_init* retorna ainda o valor da posição na tabela dessa entrada recém criada; esse retorno será usado em chamadas subseqüentes às outras primitivas da API, servindo como uma espécie de *handle*, ou seja, um código de controle que permitirá o uso da API. Para retirar essa entrada da tabela do FLIP, usa-se a primitiva *flip_end*, passando como parâmetro o *handle* retornado pela primitiva *flip_init*.

A transmissão de mensagens é feita via NSAPs (*Network Service Access Points*). Um NSAP é um número de 64 bits que identifica os processos na rede. Um NSAP é equivalente ao conceito de porta + endereço IP em ambientes TCP/IP. Cada entidade (processo) que deseja receber mensagens via FLIP, necessita registrar um NSAP junto ao protocolo; isso é feito com a primitiva *flip_register*. Por sua vez, a primitiva *flip_unregister* retira um registro de um NSAP feito previamente.

O envio de mensagens entre NSAPs (em última instância, entre processos) é feito através de três primitivas: *flip_unicast*, que envia uma mensagem ponto a ponto para um NSAP na rede; *flip_multicast*, que envia uma mensagem para um número mínimo de processadores que estejam esperando mensagens no mesmo NSAP (i.e., fizeram *flip_register* para o mesmo NSAP); e *flip_broadcast* que envia uma mensagem para todos os processadores na rede, o qual será recebido por aqueles processadores que estejam esperando mensagens no NSAP 0.

Notar mais uma vez que cada chamada a uma função da API com exceção de *flip_init* deve ter sempre como um dos parâmetros o valor retornado por uma chamada anterior a *flip_init*, ou seja, o *handle* retornado por *flip_init*. Esse fator é importante para a alteração que pretendemos fazer ao FLIP, a qual detalhamos mais adiante.

As primitivas para *unicast*, *multicast* e *broadcast* mencionadas acima, recebem também um parâmetro adicional (*hop count*) que indica o número máximo de processadores pelos quais a mensagem pode passar antes de chegar até o destino. Uma vez que o FLIP pode ser usado para interligar redes distintas, formando uma *internet*, esse controle é importante para o roteamento dos fragmentos FLIP.

Outro ponto que é interessante observar é a existência de um protocolo de resolução de endereços, o qual mapeia NSAPs para endereços físicos na rede. Esse protocolo é executado para as primitivas *flip_unicast* e *flip_multicast*, mas não para *flip_broadcast*; neste caso é usado o próprio recurso de transmissão para todos os processadores da rede, disponível pelo nível de enlace de dados da rede particular. O protocolo funciona da seguinte maneira: antes de se enviar a mensagem da aplicação propriamente dita, é enviado um pacote de controle perguntando a todos os processadores na rede quais deles esperam pacotes em um determinado NSAP. Para o caso de um *unicast*, a primeira resposta é usada, descartando-se as outras; no caso de um *multicast*, se tiver havido pelo menos n respostas (onde n é especificado na chamada a *flip_multicast*), todos os processadores que responderam receberão a mensagem. Uma vez que se resolve o endereço de um NSAP, este endereço é mantido localmente para futuras solicitações de envio de mensagens. A espera por respostas se dá dentro de um intervalo máximo (*timeout*), após o qual futuras respostas são descartadas.

5.2.3. Alterando o FLIP para permitir a reserva da Qualidade de Serviço desejada

Segundo a proposta apresentada no Capítulo 4, para uma aplicação poder fazer uso de um serviço de comunicação síncrono, é preciso permitir que se especifique qual a Qualidade de Serviço (QoS) necessária. Essa Qualidade de Serviço determina duas coisas: a previsão do tráfego gerado pela aplicação, onde a reserva é especificada através do par $<y,t>$, em que y representa a maior quantidade de bits que a aplicação poderá gerar dentro de um intervalo de tempo t; e a garantia de entrega dentro do intervalo t.

Para permitir uma reserva desse tipo, propomos adicionar uma nova primitiva à API do FLIP, na qual se indica qual a reserva pretendida, ou seja, qual o valor do par $<y,t>$. Queremos que essa nova primitiva se assemelhe à primitiva *flip_init*, de forma que uma vez chamada, a nova primitiva retorne um identificador (*handle*) que deverá ser usado em qualquer outra chamada à API. Isso determina que a Qualidade de Serviço especificada

estará relacionada com o *handle* retornado. Dessa forma, sempre se pode saber qual o valor da reserva, já que o próprio *handle* tem que ser passado como parâmetro em toda chamada a alguma outra primitiva da API.

Portanto, criaremos uma nova primitiva que tenha a mesma funcionalidade que a primitiva *flip_init* comentada anteriormente, mas que além disso, permita solicitar a reserva da Qualidade de Serviço desejada. Teremos assim, a primitiva *flip_init_qos*, que recebe os mesmos parâmetros que a original *flip_init*, e mais um parâmetro que especifica a Qualidade de Serviço desejada, o par <*y*,*t*>. O parâmetro relativo ao par <*y*,*t*> será mantido como informação associada ao *handle* retornado pela primitiva *flip_init_qos*. Sempre que fizermos alguma chamada às outras primitivas da API usando esse *handle*, estaremos utilizando a Qualidade de Serviço especificada.

Precisamos ainda tratar um outro fator, necessário para a reserva da Qualidade de Serviço, mas que não foi previsto no Capítulo 4. No capítulo anterior, foi especificado como fazer a reserva em um *slot* TDMA a partir da previsão do tráfego máximo que poderia ser gerado por cada aplicação; nesta previsão se incluía fatores como o tamanho máximo dos pacotes gerados, tratamento de faltas na transmissão, entre outros. No entanto, vimos no início da seção 5.2 que existe um fator extra que está relacionado com a necessidade de se executar um protocolo de resolução de endereços lógicos para endereços físicos, determinado pelo FLIP. Isso implica que será gerado mais tráfego do que os cálculos inicialmente previstos para a reserva do *slot* TDMA. No entanto, esse tráfego adicional só é necessário até a resolução de endereços terminar, posto que ao se resolver um NSAP, seu endereço físico correspondente é mantido localmente para uso futuro. Assim, seria um desperdício desnecessário incluir esse tráfego extra no cálculo do espaço livre num *slot*.

Propomos então que a aplicação, antes de começar a transmitir as mensagens de que necessita para um determinado NSAP, verifique se o endereço físico relativo a este NSAP já está na cache local. Para tanto, propomos a inclusão de mais uma primitiva ao FLIP a primitiva *flip_resolve_NSAP* que será responsável por resolver o endereço físico relativo a um NSAP. Essa primitiva recebe como parâmetro o NSAP a ser resolvido e o número mínimo de processadores que devem responder ao protocolo de resolução de endereços (necessário para futuras chamadas a *flip_multicast*). O retorno de *flip_resolve_NSAP* deve indicar três situações possíveis: 1) o endereço foi resolvido corretamente; 2) não existe o número requisitado de processos esperando mensagens naquele NSAP; ou 3) o protocolo

de resolução ainda não terminou, ou seja, ainda não se passou o intervalo máximo para espera das respostas. Uma vez que o NSAP já está presente na cache local, qualquer futura chamada a *flip_unicast*, *flip_multicast* ou *flip_broadcast* irá gerar no máximo apenas o tráfego previsto no capítulo anterior. Por outro lado, se a aplicação utilizar-se dessas primitivas sem a garantia de que o NSAP já foi resolvido, corre-se o risco de a aplicação violar a reserva efetuada, apesar de estar gerando o tráfego de acordo com a reserva.

No caso específico do modelo de nodos replicados do Seljuk-Amoeba, deve-se chamar a primitiva *flip_resolve_NSAP* após a criação do nodo (para maiores detalhes sobre como um nodo é criado, consultar [GBC97]). Assim, resolve-se os NSAPs relativos aos processos *Difusor* e *Broadcast*, que fazem parte do protocolo de ordenação de mensagens, e para os quais é necessária a transmissão síncrona de mensagens.

Como a API do FLIP foi apenas estendida para comportar as novas primitivas *flip_init_qos* e *flip_resolve_NSAP*, e uma vez que os mecanismos que implementam a nova funcionalidade executam no núcleo do sistema operacional, outros módulos (RPC, Comunicação em Grupo, ou mesmo as próprias aplicações) que faziam uso do FLIP da forma usual, não são afetados por essa extensão à API. Portanto, a alteração fica transparente para as aplicações que fazem uso do FLIP da forma convencional.

A diferença entre as aplicações que usam a nova primitiva *flip_init_qos* e as que continuam usando a primitiva *flip_init* é que o tráfego gerado pelas últimas será tratado pelo módulo de Policiamento (o qual veremos na próxima seção) como tráfego assíncrono; isto é, fragmentos relativos ao tráfego assíncrono terão menor prioridade que aqueles gerados por aplicações que usam *flip_init_qos*.

5.3. IMPLEMENTAÇÃO DO SERVIÇO DE COMUNICAÇÃO SÍNCRONO NO AMOEBA

Na Figura 4.4 apresentada no Capítulo 4, vimos a arquitetura para implementar nossa proposta de um serviço de comunicação síncrono. Nossa intenção agora é detalhar cada módulo apresentado no capítulo anterior, mas agora sob o aspecto da implementação no Amoeba. Assim, a Figura 5.2 mostra uma nova arquitetura, em que se faz o relacionamento com a API FLIP.

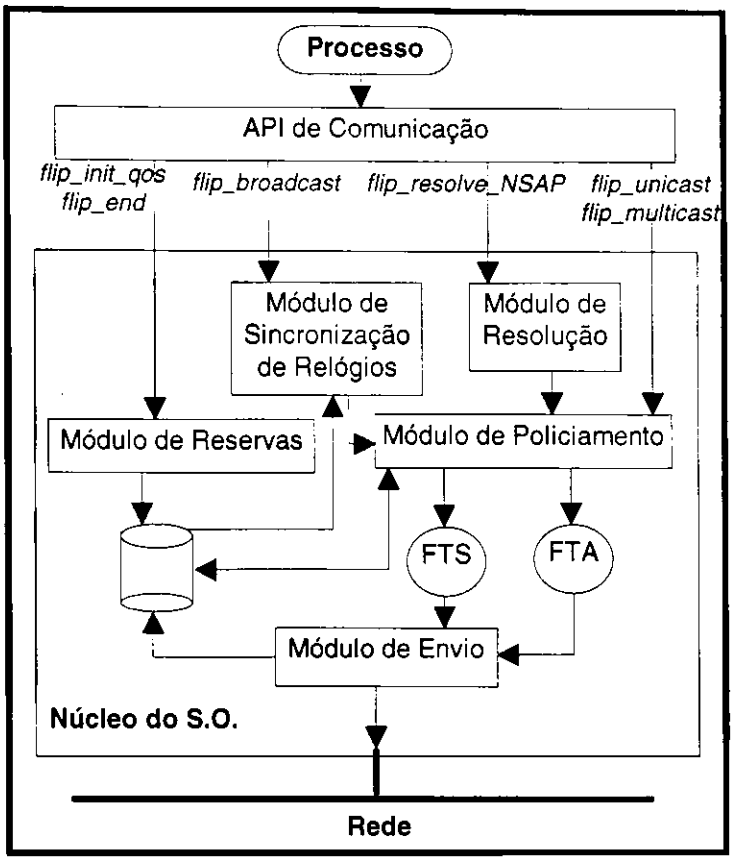

Figura 5.2 - Relacionando a API FLIP com a arquitetura proposta

É importante notar que as primitivas *flip_init*, *flip_register* e *flip_unregister* não necessitam ser modificadas para dar o suporte à comunicação síncrona da forma que pretendemos. O que precisa ser tratado é a reserva da Qualidade de Serviço (através de *flip_init_qos*), a retirada de uma reserva (com *flip_end*), a solicitação para a resolução de NSAPs (com a primitiva *flip_resolve_NSAP*) e a transmissão das mensagens propriamente ditas (efetuada através das primitivas *flip_unicast*, *flip_multicast*, e *flip_broadcast*).

Com relação à Figura 4.4, houve as seguintes mudanças: o módulo de Sincronização de Relógios intercepta mensagens provenientes de *broadcasts* e acessa as informações sobre as reservas já feitas, antes de repassar a mensagem ao módulo de Policiamento (ver os detalhes a respeito do módulo de Sincronização de Relógios mais adiante na seção 5.3.5). Além disso, inserimos também o módulo de Resolução, devido à necessidade de mapear NSAPs para endereços físicos na rede; este módulo é tratado na seção 5.3.2.

Para os algoritmos aqui descritos, supõe-se que os valores das constantes B, T, ***A***, n, ε, *d*, ***e*** e ***p*** que foram definidas no Capítulo 4 são inicialmente conhecidos; eles podem ser passados como argumentos ao núcleo em tempo de *boot*. Recordando o significado dessas constantes, temos: **B** = taxa de transferência na rede; **T** = período do TDMA; ***A*** = espaço mínimo no *slot* para tráfego assíncrono; **n** = número de processadores envolvidos no TDMA; **ε** = máxima diferença entre as leituras de quaisquer dois relógios de processadores corretos; *d* = tempo máximo de transmissão de mensagens fim-a-fim; ***e*** = número de bits extras adicionados a uma mensagem relativos ao protocolo de comunicação; e por fim, ***p*** = tamanho máximo do pacote para a rede local específica.

5.3.1. O módulo de Reservas

O módulo de reservas recebe o par <*y*,*t*> de uma chamada a *flip_init_qos* e trata de armazená-lo numa tabela interna; essa tabela pode ser indexada pelo próprio *handle* retornado por *flip_init_qos*, de forma análoga à indexação usada na tabela mantida pela primitiva *flip_init*.

A estrutura de uma linha da tabela de reservas deve conter a tupla **<*y*, *t*, NumBitsEnfileirados, FilaEspera>**. Os componentes dessa tupla correspondem, respectivamente a: o número de bits para a reserva, o intervalo máximo de tempo em que esses bits devem ser enviados, o número de bits enfileirados até o momento na fila FTS, e uma fila que deve conter os fragmentos que não puderam ser enviados por terem violado a reserva feita. O campo **NumBitsEnfileirados** é incrementado pelo módulo de Policiamento, e zerado a cada *slot* pelo módulo de Envio; deve ser inicializado com zero. Já o campo **FilaEspera** deve ser inicializado com uma fila vazia; esta fila será atualizada (com retirada/inserção de elementos) através do módulo de Policiamento. Um esboço dos algoritmos que implementam o módulo de Reservas é mostrado na Figura 5.3 e na Figura 5.4.

```
int _TamSlot = (B*T)/n - B*ε - A;        /* inicialização do tamanho do slot; */
Tupla _TabReservas[MAXHANDLES] = { VAZIO, VAZIO,...VAZIO } /* inicialização
                                                    da tabela de reservas */
Boolean IncluiReserva( int y, int t, int Handle )
INÏCIO
1  int x; /* para normalizar uma reserva */
2  int MaxBits; /* número máximo de bits que podem ser gerados */
3  Se t < d
4      Retorne FALSO;
5  x = ⌈y / ⌊(t/T)⌋⌉;
6  MaxBits = (k+1)*⌈x / (p+e)⌉*p;
7  Se MaxBits > _TamSlot
8          Retorne FALSO;
9  Se _TabReservas[Handle] != VAZIO
10     _TabReservas[Handle] = <y, t, 0, VAZIO>
11 Senão
12     Retorne FALSO;
13 _TamSlot -= MaxBits;
14 Retorne VERDADEIRO;
FIM
```

Figura 5.3 - Módulo de Reserva (incluindo uma reserva)

Na Figura 5.3, encontramos o algoritmo para a função **IncluiReserva**; nesta figura, podemos identificar alguns pontos importantes que foram levantados no capítulo anterior. No passo 3, por exemplo, é feito o teste se o tempo *t* que a aplicação solicita pode ser assegurado, considerando-se o atraso máximo *d* para transmissão de mensagens. Já no passo 7, verifica-se se a reserva pode ser comportada no *slot* TDMA, considerando a notação normalizada e o número máximo de bits que serão gerados, prevendo as situações de faltas e a inclusão de informação de controle relativa ao protocolo de comunicação.

Cada reserva deve ser feita pela primitiva *flip_init_qos*, chamando a função **IncluiReserva**; se esta função retornar erro, ou seja, não há espaço disponível no *slot* para se efetuar a reserva, *flip_init_qos* também deve retornar erro, de forma semelhante a uma situação de erro ocorrida após uma chamada a *flip_init*. Para se aceitar uma nova reserva, são considerados os cálculos apresentados na seção 4.3.3. Por sua vez, uma reserva é excluída através da função **RetiraReserva**. Essa função se encarrega de incrementar o

espaço livre no *slot* (ver Figura 5.4). Ela deve ser chamada quando a aplicação terminar o uso do *handle* FLIP, ou seja, na chamada a *flip_end*.

```
void RetiraReserva( int Handle )
INÍCIO
1   int x;   /* para normalizar a reserva */
2   int MaxBits;   /* número máximo de bits que pode ser gerado */
3   Se _TabReservas[Handle] != VAZIO
4       x = ⌈_TabReservas[Handle]. y / ⌊(_TabReservas[Handle].t/T)⌋ ⌉;
5       MaxBits = (k+1)*⌈ x / (p+e)⌉ *p;
6       _TamSlot += MaxBits;
7   _TabReservas[Handle] = VAZIO;
FIM
```

Figura 5.4 - Módulo de Reserva (excluindo uma reserva)

A função **RetiraReserva** libera o espaço ocupado pela reserva, considerando o mesmo número de bits que foi incluído pela função **IncluiReserva**. Isso é feito no passo 6 do algoritmo descrito na Figura 5.4.

Notar que todas as variáveis cujos nomes começam por "_" são consideradas globais aos módulos (por exemplo, a tabela de reservas, o tamanho do *slot*). Assim, podem ser atualizadas concorrentemente por qualquer módulo. Portanto, é importante o uso de mecanismos de controle de acesso simultâneo - *race conditions* - para evitar o uso de valores inconsistentes. No algoritmo acima não utilizamos nenhum mecanismo deste tipo (regiões críticas, estruturas de exclusão mútua - *mutexes*, etc.) para efeito de simplificação, mas esse fator deve ser levado em conta.

5.3.2. O módulo de Policiamento

Este módulo enfileira os fragmentos originados pelos processos para serem transmitidos pelo módulo de Envio, e determina se as requisições de envio de mensagens estão de acordo com uma reserva eventualmente feita.

Este módulo deve ser implementado no ponto do núcleo para o qual convergem os fragmentos das mensagens (os quadros), já com informações de controle do FLIP. Assim, a unidade de manipulação deste módulo é um fragmento FLIP. Uma vez que qualquer fragmento (seja originado de uma primitiva *flip_unicast*, *flip_multicast* ou *flip_broadcast*) é direcionado para esse mesmo ponto, permite-se que se consiga interceptar qualquer quadro relativo a uma mensagem gerada. No nosso caso, essa funcionalidade é

implementada no núcleo do Amoeba através da função **ethsend**. Assim, todos os fragmentos originados das primitivas acima, convergem para a função **ethsend**.

Nossa idéia é fazer com que no ponto em que ocorre o envio de um fragmento (no código da função **ethsend**, por exemplo), ao invés de se enviar efetivamente o fragmento, este deve ser direcionado para o módulo de Policiamento. No momento apropriado, o módulo de Envio fará com que esse fragmento seja transmitido. Os algoritmos que implementam este módulo são descritos na Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags** e na Figura 5.6.

Assim, ao invés de se enviar o fragmento propriamente dito, deve-se passá-lo para a função **ArmazenaFrags**, mostrada na Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags**. A função primeiro verifica se o fragmento pertence ao tráfego com reservas (passo 5 no algoritmo da Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags**); se não pertence, esse fragmento deve ser inserido na fila do tráfego assíncrono, FTA (passo 6). Se é um fragmento de uma mensagem síncrona, deve-se verificar se está de acordo com a reserva feita (passo 10 no algoritmo da Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags**). Caso o fragmento esteja violando a reserva, ele deve ser inserido na fila de espera relativa ao *Handle* atual (passo 13). Se não existem elementos nesta fila, deve-se ligar um temporizador que irá executar a função **TrataTimer** após o tempo máximo de um *slot*, T/n (passo 12). Após esse intervalo, o fragmento com certeza pode ser inserido na fila FTS sem a possibilidade de se violar a reserva. Se o pedido não viola a reserva feita, deve-se incrementar o número de bits enfileirados pela aplicação na fila FTS até o momento (passo 15). O algoritmo finaliza com a inserção das k+1 cópias do fragmento na fila FTS (passo 16).

```
Fila _FTS;          /* fila do tráfego síncrono */
Fila _FTA;          /* fila do tráfego assíncrono */
void ArmazenaFrags( int Handle, char *Frag, int TamFrag )
INÍCIO
1  ItemFila  IFila;      /* elemento a ser incluído em FTS, FTA ou mesmo em fila de espera */
2  int Sobra;            /* total de bits que ainda podem ser enviados sem violar a reserva */
3  int x;  /* para normalizar a reserva */
4  IFila.Info = <Frag, TamFrag, Handle>;
5  Se _TabReservas[Handle] = VAZIO /* não foi feita reserva para esse handle */
6      Insere(IFila, _FTA);
7      Retorne;
8  x = ⌈_TabReservas[Handle]. y / ⌊( _TabReservas[Handle].t/T)⌋ ⌉;
9  Sobra = x - _TabReserva[Handle].NumBitsEnfileirados;
10 Se (TamFrag  - e) > Sobra /* temos que guardar este fragmento */
11     Se !ExisteElemento (_TabReserva[Handle].FilaEspera)
12         Escalone( TrataTimer, Handle, T/n);
13     Insere(IFila, _TabReserva[Handle].FilaEspera);
14     Retorne;
15 _TabReserva[Handle].NumBitsEnfileirados += x;
16 Para i=1 até k+1 faça
17     Insere( IFila, _FTS);
FIM
```

Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags**

Notar que é usada a notação <x> para se testar se o pedido de transmissão não viola a reserva. Sempre que formos utilizar o tamanho do fragmento a ser enviado (passos 9 e 16 do algoritmo acima) devemos desprezar os *e* bits relativos à informação de controle do protocolo de comunicação, já que na notação <x> isso não é incluído.

Na Figura 5.6 mostramos o algoritmo para a função que será chamada no disparo do temporizador, a função **TrataTimer**. Essa função deve obter o primeiro elemento da fila de espera da aplicação (passo 4 no algoritmo), e verificar se o fragmento FLIP em espera não viola a reserva feita (passo 8); essa verificação é análoga àquela executada na função **ArmazenaFrags** que aparece na Figura 5.5 - Módulo de Policiamento: função **ArmazenaFrags**. Se não é violada a reserva, o fragmento é retirado da fila de espera e armazenado na fila FTS (passos 9, 10, 11, 12), de forma análoga ao que é feito na função **ArmazenaFrags**. O ciclo continua até se exaurir a fila de espera ou até que se encontre um fragmento que esteja violando a reserva; neste caso, a função é novamente escalonada para executar após o intervalo de duração de um *slot* (passo 14), quando com certeza se pode garantir que o fragmento não violará a reserva feita.

```
void TrataTimer( int Handle )
INÍCIO
1   ItemFila   IFila;
2   int Sobra;
3   int x; /* para normalização */
4   Enquanto ExisteElemento(_TabReservas[Handle].FilaEspera)
5       IFila  = ObtemElemento ( _TabReservas[Handle].FilaEspera ); /* não retira da fila ainda */
6       x = ⌈ _TabReservas[Handle]. y / ⌊( _TabReservas[Handle].t/T)⌋ ⌉;
7       Sobra = x - _TabReservas[Handle].NumBitsEnfileirados;
8       Se (IFila.Info.TamFrag  - e) <= Sobra /* podemos enfileirar este fragmento em FTS */
9           RetiraElemento(_TabReservas[Handle].FilaEspera );      /* agora podemos retirar da fila */
10          _TabReservas[Handle].NumBitsEnfileirados += x;
11          Para i=1 até k+1 faça
12              Insere( IFila, _FTS);
13      Senão
14          Escalone( TrataTimer, Handle, T/n);
15          Retorne;
FIM
```

Figura 5.6 - Módulo de Policiamento: função de tratamento de temporizadores

É importante observar mais uma vez o uso de variáveis globais (_FTS, _FTA, _LE); essas variáveis devem ter seu uso protegido através de algum mecanismo de controle de acesso simultâneo.

5.3.3. O módulo de Envio

É neste módulo que a transmissão de mensagens (ou melhor, de fragmentos) efetivamente ocorre. Para efeito de simplificação, considera-se que os processadores na rede já estão sincronizados (maiores detalhes sobre a sincronização dos relógios dos processadores da rede são discutidos na seção 5.3.4 a seguir).

Este módulo será implementado como uma linha de execução - *thread* - independente, cuja ativação é iniciada de tempos em tempos, de acordo com o período dos *slots* TDMA. A *thread* executa num ciclo ininterrupto, retirando os fragmentos da fila FTS e da fila FTA, enviando-os efetivamente. Ao término do *slot*, a *thread* fica bloqueada durante o tempo necessário até o próximo *slot*. Isso deve ocorrer dentro de $(T + \varepsilon)$ unidades de tempo, já que devemos contabilizar o valor de ε a cada período T do ciclo TDMA.

É importante que esta linha de execução (*thread*) execute em modo não-preemptivo, de maneira que não possa ser interrompida durante seu processamento, e que execute com a maior prioridade possível, para garantir que se consiga medir o tempo necessário para sair do modo bloqueado. Para conseguirmos isso, teremos que modificar um pouco o escalonamento de processos dentro do Amoeba. Apesar de parecer uma tarefa difícil, a modificação é relativamente simples.

Dentro dos arquivos que implementam a funcionalidade do núcleo do Amoeba, podemos encontrar a função **scheduler**. Esta função é encarregada de escalonar processos a serem executados, habilitando a execução de *threads* destes processos; esta função é chamada sempre que o escalonador de processos se faz necessário (devido ao término da fatia de tempo de algum processo, por exemplo).

É possível dividir o código da função **scheduler** em duas partes: a primeira é responsável por determinar qual a próxima *thread* a ser habilitada, seja do processo corrente, seja de um outro processo qualquer apto a ser posto em execução; a segunda parte da função é responsável por efetivamente executar a *thread* escolhida.

A modificação a ser feita é forçar com que sempre seja habilitada para execução a *thread* responsável pela transmissão dos fragmentos do módulo de Envio (a **FunçãoEnvio**, como veremos adiante). Assim, desde que esta *thread* esteja pronta para executar, ao invés de se tentar determinar qual a próxima *thread* a ser habilitada para

execução, basta habilitar a *thread* responsável pelo módulo de Envio. Uma vez que *threads* são representadas internamente através de estruturas de dados, ao criar a *thread* do módulo de Envio, basta salvar esta estrutura para ser referenciada posteriormente.

Um esboço da função que implementa a *thread* acima encontra-se na Figura 5.7. Inicialmente, percorre-se a fila FTS transmitindo cada fragmento encontrado; a cada fragmento transmitido, incrementa-se o tamanho corrente do *slot* (em número de bits) e é zerado o componente NumBitsEnfileirados para o *handle* relativo ao fragmento enviado. Essas ações são executadas ao longo dos passos 6 a 10 do algoritmo da Figura 5.7. Após a transmissão dos fragmentos correspondentes ao tráfego síncrono, passamos à transmissão do tráfego assíncrono (passos 11 a 19 no algoritmo). Assim, percorre-se a fila FTA tentando enviar os fragmentos encontrados. O ciclo é executado até se exaurir a fila FTA, ou até que algum fragmento não seja comportado no *slot* (passo 19).

Após enviar todos os fragmentos presentes até o momento nas filas FTS e FTA, a *thread* bloqueia durante um intervalo de tempo necessário até o próximo *slot*. Esse intervalo depende não só do valor de $T+\varepsilon$ como também do tempo restante no *slot* corrente. Uma vez que o *slot* pode não ser totalmente utilizado para a transmissão de fragmentos, é necessário levar esse fator em consideração para se determinar quanto tempo a *thread* deve ficar bloqueada até o próximo *slot* (passo 20 do algoritmo).

Ao se bloquear a *thread* do módulo de Envio, permite-se que outras *threads*, do núcleo ou de qualquer outro processo à espera de ser executado, possam ser habilitadas para execução.

```
void FunçãoEnvio()
INÍCIO
1  Boolean   FimSlot = FALSO;   /* determina se é possível enviar fragmentos no slot corrente */
2  int       TamMaxSlot = B*T/n - B*ε;    /* quantos bits se podem enviar num slot */
3  int       TamAtualSlot=0; /* quantos bits já forma enviados até o momento */
4  ItemFila  IFila;       /* elemento de uma fila */
5  Loop para sempre
6      Enquanto ExisteElemento(_FTS)
7              IFila = RetiraElemento(_FTS);
8              Transmite(IFila.Info);
9              TamSlotAtual += IFila.Info.TamFrag;
10             _TabReservas[IFIla.Info.Handle].NumBitsEnfileirados = 0;
11     FimSlot = FALSO;
12     Enquanto ExisteElemento(_FTA) E !FimSlot
13             IFila = ObtémElemento (_FTA);    /* não retira o elemento da fila ainda */
14             Se IFIla.Info.TamFrag <= (TamMaxSlot  - TamAtualSlot)
15                 RetiraElemento( _FTA );    /* agora o elemento é retirado */
16                 Transmite(IFila.Info);
17                 TamSlotAtual += IFila.Info.TamFrag;
18             Senão
19                 FimSlot = VERDADEIRO;
20     Bloquear( T + ε - (TamMaxSlot - TamSlotAtual)/B);
FIM
```

Figura 5.7 -Módulo de Envio: função de envio de fragmentos do núcleo

5.3.4. Execução dos processos Broadcast e Difusor

No Capítulo 4, propusemos um escalonamento especial para os processos *Broadcast* e *Difusor* do ambiente **Seljuk-Amoeba**. Na proposta apresentada, estes processos deveriam ser executados logo antes da rotina de tratamento do *slot* TDMA (ver Figura 4.3).

Devido a esta característica, podemos inserir a execução desses processos dentro da função que trata o envio de fragmentos em um *slot* (**FunçãoEnvio**, mencionada na seção anterior). Assim, no laço principal de execução da função, podemos tratar não só o envio de fragmentos no *slot*, como também podemos executar os processos *Broadcast* e *Difusor*.

```
void FunçãoEnvio()
INÍCIO
1  Boolean   FimSlot = FALSO;   /* determina se é possível enviar fragmentos no slot corrente */
2  int       TamMaxSlot = B*T/n - B*ε;     /* quantos bits se podem enviar num slot */
3  int       TamSlotAtual=0; /* quantos bits já foram enviados até o momento */
4  ItemFila  IFila;        /* elemento de uma fila */
5  int iMaxMsgBroadcast = 1; /* numero máximo de mensagens que o Broadcast pode tratar; depende
                                da reserva que se deseja fazer) */
6  int iTempoMsgBroadcast  = 10; /* tempo para o Broadcast processar uma mensagem; este valor está
                                em unidades de tempo (ut) ; */
7  int iMaxMsgDifusor = [ (n-1) Σ_{j=1}^{π} A_(j-1)^(n-2) (n-j-1) ] * iMaxMsgBroadcast; /* ver capítulo 4*/
8  int iTempoMsgDifusor = 10; /* tempo para o Difusor processar uma mensagem; este valor está
                                em unidades de tempo (ut); */
9  Loop para sempre
10     int iNumMsgTratadas = Broadcast();
11     Bloquear( (iMaxMsgBroadcast  - iNumMsgTratadas) * iTempoMsgBroadcast );
12     iNumMsgTratadas = Difusor();
13     Bloquear( (iMaxMsgDifusor  - iNumMsgTratadas) * iTempoMsgDifusor );
14     Enquanto ExisteElemento(_FTS)
15             IFila = RetiraElemento(_FTS);
16             Transmite(IFila.Info);
17             TamSlotAtual += IFila.Info.TamFrag;
18             _TabReservas[IFIla.Info.Handle].NumBitsEnfileirados = 0;
19     FimSlot = FALSO;
20     Enquanto ExisteElemento(_FTA) E !FimSlot
21             IFila = ObtémElemento (_FTA);     /* não retira o elemento da fila ainda */
22             Se IFIla.Info.TamFrag <= (TamMaxSlot  - TamAtualSlot)
23                 RetiraElemento( _FTA );     /* agora o elemento é retirado */
24                 Transmite(IFila.Info);
25                 TamSlotAtual += IFila.Info.TamFrag;
26             Senão
27                 FimSlot = VERDADEIRO;
28     Bloquear( T + ε - (TamMaxSlot - TamSlotAtual)/B – (iMaxMsgBroadcast *
                 iTempoMsgBroadcast) – (iMaxMsgDifusor * iTempoMsgDifusor) );
FIM
```

Figura 5.8 - Função de envio de fragmentos, modificada para tratar os processos *Broadcast* e *Difusor*

O código final da função apresentada na seção anterior ficaria da maneira descrita na Figura 5.8.

As diferenças do algoritmo da Figura 5.7 com relação ao da Figura 5.8 encontram-se nas linhas 5 a 8 e 10 a 13 (que foram inseridas), e na linha 28 (que foi alterada). Nas linhas 5 a 8 é feita a inicialização das variáveis relativas ao número máximo de mensagens que os processos *Broadcast* e *Difusor* podem tratar, e inicializa-se também o tempo que cada processo necessita para tratar uma mensagem. No caso do processo *Broadcast*, a variável iMaxMsgBroadcast deve ser inicalizada com um valor dependendo da reserva feita; já para o processo *Difusor*, a variável iMaxMsgDifusor é inicializada com o número total de mensagens que devem ser difundidas para cada mensagem gerada por um processo *Broadcast* (maiores detalhes ver Capítulo 4).

Nas linhas 10 a 13, os processos *Broadcast* e *Difusor* são efetivamente executados; supomos que eles retornam o número de mensagens que conseguiram processar, para podermos bloquear a execução da **FunçãoEnvio** no caso de não haver o número esperado de mensagens a serem tratadas.

Por fim, na linha 28, foi feita uma alteração para que a **FunçãoEnvio** seja bloqueada durante o intervalo correto até sua próxima ativação.

5.3.5. O módulo de Sincronização de Relógios

Como detalhamos no capítulo anterior, o protocolo que pretendemos seguir é descrito com maior detalhe em [CM96]. Nesta seção vamos mostrar que modificações devem ser feitas no Amoeba para dar suporte ao protocolo.

5.3.5.1. Inserindo o protocolo de sincronização de relógios no Amoeba

Para que as medições possam ser feitas corretamente, precisamos interceptar os pedidos de *broadcasts* vindos das aplicações, e acrescentar às mensagens as informações necessárias ao protocolo. Como estamos supondo que o protocolo não necessita de comunicação síncrona para funcionar corretamente, determinamos que acrescentaremos informações de controle apenas às mensagens pertencentes ao tráfego assíncrono. Daí a necessidade de se acessar a informação na tabela de reservas relativa ao *handle* passado em *flip_broadcast*, para se saber se foi feita alguma reserva para esse *handle* (ver Figura 5.2).

Assim, para cada chamada a *flip_broadcast* que chega ao núcleo, originada do tráfego assíncrono, são concatenadas à mensagem as informações de controle do protocolo descrito na seção 4.4.4; notar que a informação é concatenada à mensagem como um todo, não a um fragmento. Quando do recebimento de um *broadcast* (o qual é sempre endereçado ao NSAP 0), essas informações são extraídas, antes de se entregar a mensagem à aplicação destino. Uma vez que pode haver fragmentação da mensagem FLIP, a informação de controle irá constar no último fragmento da mensagem.

5.3.5.2. Inicialização

Para garantir que todos os processadores na rede estejam corretamente sincronizados antes de se começar o ciclo TDMA, sugerimos que o protocolo de sincronização seja inicialmente executado de forma independente do método TDMA. Assim, o esquema TDMA só deve começar no instante t' no qual se tenha certeza de que os relógios estejam corretamente sincronizados. Assim, t' pode ser grande o suficiente para que isso seja verdadeiro; o valor de t' deve ser o mesmo para todos os processadores, podendo ser passado como parâmetro no *boot* de cada um.

Dessa forma, a ativação da *thread* que desempenha as funções do módulo de Envio deve ser ajustada para começar de acordo com o valor de t' e da ordem do processador dentro do ciclo TDMA. Assim, cada processador P_i escalona a execução da *thread* de envio de mensagens no instante $[t' + \varepsilon + i(T/n)]$, onde T/n é o tamanho de um *slot* TDMA, em unidades de tempo - **ut**; ε, por sua vez, é o desvio máximo entre as leituras de quaisquer dois relógios corretos na rede.

Enquanto a *thread* do módulo de envio não é iniciada, solicitações de reservas através de *flip_init_qos* devem ser recusadas e todas as mensagens que chegarem ao módulo de Policiamento devem ser diretamente transmitidas, sem a necessidade de serem incluídas nas filas FTS e FTA presentes na Figura 5.2.

5.4. CONCLUSÃO

Neste capítulo apresentamos os detalhes de implementação envolvidos em cada módulo componente do serviço de comunicação síncrono especificado no Capítulo 4. Com relação à arquitetura inicialmente proposta no Capítulo 4, foi necessária a inclusão de mais um módulo, o módulo de Resolução, devido à necessidade de resolução de endereços lógicos para endereços físicos do protocolo FLIP.

O serviço de comunicação proposto é transparente para as aplicações que não necessitam que a transmissão de mensagens ocorra dentro de um intervalo de tempo máximo conhecido. Além disso, excetuando-se a necessidade do uso das primitivas *flip_init_qos* e *flip_resolve_NSAP*, a forma de se transmitir mensagens com garantias de atraso máximo segue as mesmas regras definidas na especificação original da API FLIP.

6.

Conclusões

6.1. CONSIDERAÇÕES FINAIS

Neste trabalho estudamos as dificuldades de se disponibilizar um serviço de comunicação síncrono para uso no sistema operacional Amoeba, a ser utilizado pelos protocolos de ordenação do ambiente **Seljuk-Amoeba**. Apresentamos ainda uma alternativa para superar as dificuldades especificadas direcionando-a para uma futura implementação no sistema operacional Amoeba. Com o objetivo de facilitar o trabalho numa possível implementação da proposta apresentada, mostramos também quais os passos necessários para que esta solução venha a ser incorporada ao núcleo do Amoeba.

A principal contribuição deste trabalho foi o detalhamento das dificuldades que deveriam ser tratadas ao se tentar fornecer um suporte síncrono sobre um ambiente assíncrono, utilizando redes *Ethernet*. Vimos que este é um problema de difícil solução e que envolveria, numa implementação efetiva da proposta aqui apresentada, a necessidade de se determinar os valores de diversas variáveis próprias do sistema operacional, como por exemplo, intervalos de tempo relativos ao tratamento de interrupções. Além disso, as alterações no núcleo do Amoeba não são simples de se fazer, devido ao fato de ser muito dependente de sincronização entre os núcleos das máquinas envolvidas na rede. Qualquer fator de perda de sincronização aumenta a complexidade da implementação.

Essa complexidade reflete as restrições que impusemos para a solução, isto é, uso de um sistema operacional de propósito geral (o Amoeba) junto com uma rede tipo *Ethernet*. Uma vez que nossa intenção foi de possibilitar que uma diversidade maior de aplicações pudessem utilizar-se do serviço de processamento confiável do ambiente **Seljuk-Amoeba**, sem usar componentes proprietários, o trabalho continua válido.

Outros sistemas também permitem a inserção de mecanismos para tolerância a faltas para as aplicações (como é o caso do projeto MARS [KM85], por exemplo). No entanto, são destinados a um número mais restrito de aplicações; nesses sistemas, o uso de

componentes de *hardware* especiais é um fator de limitação para sua utilização por aplicações de propósito geral.

6.2. TRABALHOS FUTUROS

De início, as discussões apresentadas neste trabalho precisam de uma implementação que as valide; em particular, para que se possa medir os valores máximos dos atrasos provocados por fatores de incerteza detalhados no Capítulo 4, e que são vitais para que o mecanismo proposto possa ser utilizado. Assim, esta seria uma das primeiras tarefas a se desenvolver num outro trabalho.

Nossa sugestão de escalonamento especial se aplica apenas aos processos *Difusor* e *Broadcast* dos protocolos de ordenação de mensagens do **Seljuk-Amoeba**; no entanto, é nosso interesse disponibilizar mecanismos de escalonamento de processos mais genéricos, de forma que outros processos possam utilizar o esquema de reservas aqui proposto. Assim, o próximo passo a ser seguido é prover facilidades para escalonamento de tarefas dentro do núcleo do Amoeba. Uma proposta para dissertação de doutorado que contempla a construção de um núcleo de tempo real no Amoeba [Galli97], disponibilizando escalonamentos mais sofisticados que o proposto aqui, já foi inclusive aprovada junto à Coordenação de Pós-Graduação em Engenharia Elétrica da UFPb, Campus II.

Vimos que o fato de termos nos baseado numa rede *Ethernet* foi um fator de limitação para a solução do problema proposto. Acreditamos que um ambiente que fizesse uso de uma rede ATM, a qual já permite especificar classes de tráfego para as aplicações (entre elas o tráfego síncrono), facilitaria bastante o desenvolvimento de um serviço de comunicação síncrono para os protocolos do **Seljuk-Amoeba**. Assim, um estudo sobre como deveria se comportar um serviço síncrono sobre uma rede ATM seria um importante trabalho a ser desenvolvido a seguir.

Referências

[AV96a] C. Almeida e P. Veríssimo, "Timing Failure Detection and Real-Time Group Communication in Quasi-Synchronous Systems," em Proceedings of the 8th Euromicro Workshop on Real-Time Systems, Áquila, Itália, junho de 1996.

[AV96b] C. Almeida e P. Veríssimo, "Real-Time Communication in Quasi-Synchronous Sytems. Providing Support for Dynamic Real-Time Applications," disponível em http://pandora.inesc.pt/docs/abstracts/rtcdynapp_abs.html, maio de 1996.

[AV95] C. Almeida e P. Veríssimo, "An Adaptive Real-Time Group Communication Protocol," em Proceedings of the First IEEE Workshop on Factory Communication Systems, Lausanne, Suíça, outubro de 1995.

[Bartl81] J.F. Bartlett, "A NonStop Kernel", ACM 8th Symposium on Operating Systems Principles, Pacific Grove, EUA, Vol. 15, No. 5, pp. 22-29, dezembro de 1981.

[BE95] F.V. Brasileiro e P.D. Ezhilchelvan, "Atomic Broadcast Using Time-outs instead of Syncronised Time," Anais do VI Simpósio de Computadores Tolerantes a Falhas, Canela, Brasil, pp.223-238, agosto de 1995.

[BESST96] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs e S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems," IEEE Transactions on Computer, Vol. 45, N. 11, pp. 1226-1238, novembro de 1996.

[BJ87] K.P. Birman e T.A. Joseph, "Reliable Communication in the Presence of Failures", ACM Transactions on Computer Systems, Vol. 5, No. 1, pp. 47-76, fevereiro de 1987.

[BN84] A.D. Birrell e B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, N. 1, pp. 39-59, fevereiro de 1984.

[Brasi95] F.V. Brasileiro, "Constructing Fail-Controlled Nodes for Distributed Systems: a Software Approach," Tese de PhD, University of Newcastle upon Tyne, Newcastle, Inglaterra, maio de 1995.

[Brasi97] F.V. Brasileiro, "Seljuk - Um Ambiente para Suporte ao Desenvolvimento e à Execução de Aplicações Distribuídas Robustas", Anais do VII Simpósio de Computadores Tolerantes a Falhas, Campina Grande, Brasil, pp. 45-74, julho de 1997.

[CASD85] F. Cristian, H. Aghili, R. Strong, e D. Dolev, "Atomic Broadcast: from Simple Message Difusion to Byzantine Agreement", Digest of Papers, FTCS-15, Ann Arbor, EUA, pp. 200-206, junho de 1985.

[CM96] M. Clegg e K. Marzullo, "Clock Synchronization in Hard Real-Time Distributed Systems", relatório técnico CS96-478, University of California, San Diego, Department of Computer Science and Engineering, fevereiro de 1996.

[Crist89] F. Cristian, "Probabilistic Clock Synchronization", Distributed Computing, Vol. 3, pp. 146-158, 1989.

[Crist91] F. Cristian, "Understanding Fault-Tolerant Distributed Systems", Communications of the ACM," Vol 34, No. 2, pp. 56-77, fevereiro de 1991.

[Crist95] F. Cristian, "Synchronous and Asynchronous Group Communication", Communications of the ACM, de 1995.

[DB93] R. Drummond e Ö. Babaoglu, "Low-cost Clock Synchronization", Distributed Computing, Vol.6, No. 4, pp. 193-203, julho de 1993.

[DZ83] J.D. Day e H. Zimmerman, "The OSI Reference Model," Proceedings of the IEEE, Vol. 71, N. 12, pp. 1334-1340, dezembro 1983.

[Galli97] E. L. Gallindo, "Especificação de uma Plataforma de Desenvolvimento de Aplicações Distribuídas de Tempo Real", plano de trabalho para doutorado junto à Coordenação de Pós-Graduação em Engenharia Elétrica da Universidade Federal da Paraíba, aceito em dezembro de 1997.

[GBC97] E.L. Gallindo, F.V. Brasileiro e V.S. Catão, "Reliable Processing on the Seljuk-Amoeba Operating Environment", em Proceedings of the XVII International Conference of the Chilean Computer Science Society, pp. 105-114, Valparaíso, Chile, novembro de 1997.

[GSTC90] A. Gopal, R. Strong, S. Toueg, e F. Cristian, "Early-Delivery Atomic Broadcast", Proceedings of the 9th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing, Cidade de Québec, Canadá, pp. 297-309, agosto de 1990.

[Jalot94] P. Jalote, Fault Tolerance in Distributed Systems, Prentice-Hall, 1994, ISBN 0-13-301367-7.

[KG92] H. Kopetz e G. Grünsteidl, "TTP - A Time-Triggered Protocol for Fault-Tolerant Real-Time Systems," Digest of Papers, The 23th International Symposium on Fault-Tolerant Computing, pp 524-533, Toulouse, França, junho de 1993.

[KM85] H. Kopetz, and W. Merker., "The Architecture of MARS", Digest of Papers, FTCS-15, Ann Arbor, EUA, pp. 274-279, junho de 1985.

[KRST93] M.F. Kaashoek, R. van Renesse, H. van Staveren e A.S. Tanenbaum, "FLIP: an Internetwork Protocol for Supporting Distributed Systems," ACM Transactions on Computer Systems, pp. 73-106, fevereiro de 1993.

[Lampo78] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System", Communications of the ACM, Vol. 21, No. 7, pp 558-565, Julho de 1978.

[Lapri89] J. C. Laprie, "Dependability: a Unifying Concept for Reliable Computing and Fault Tolerance", em Dependability of Resilient Computers, Anderson, T. (Ed.), BSP Professional Books, 1989, ISBN 0-632-02054-7.

[LV91] R. de Lemos e P. Veríssimo, "Confiança no Funcionamento - Proposta para uma Terminologia em Português," comunicação pessoal, dezembro de 1991.

[MRTRS90] S. J. Mullender, G. van Rossum, A. S. Tanenbaum, R. van Renesse e H. van Staveren, "Amoeba: A Distributed Operating System for the 1990's," IEEE Computer, Vol. 23, No. 5, pp.44-53, maio de 1990.

[Veris93] P. Veríssimo, "Real-Time Communication", em Distributed Systems, S. Mullender, (Ed.), ACM Press, 1993, ISBN 0-201-62427-3.

[VR92] P. Veríssimo e L. Rodrigues, "A *posteriori* Agreement for Fault-tolerant Clock Synchronization on Broadcast Networks", 22nd International Symposium on Faul-Tolerant Computing, julho de 1992.

[VRR91] P. Veríssimo, J. Rufino e L. Rodrigues, "Enforcing Real-Time Behaviour on LAN-based Protocols", Proceedings of the 10th IFAC Workshop on Distributed Control Systems, Semmering, Austria, setembro de 1991.