UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

DISSERTAÇÃO DE MESTRADO

Processamento Confiável no Ambiente Operacional Seljuk-Amoeba

por

Érica de Lima Gallindo Ribeiro

Campina Grande, junho de 1998

UNIVERSIDADE FEDERAL DA PARAÍBA

CENTRO DE CIÊNCIAS E TECNOLOGIA

COORDENAÇÃO DE PÓS-GRADUAÇÃO EM INFORMÁTICA

Érica de Lima Gallindo Ribeiro

Processamento Confiável no Ambiente Operacional Seljuk-Amoeba

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

Área de Concentração: Redes de Computadores e Sistemas Distribuídos

Orientador: Francisco Vilar Brasileiro

Campina Grande, junho de 1998

R484p Ribeiro, Érica de Lima Gallindo.
Processamento confiável no ambiente operacional seljuk-amoeba / Érica de Lima Gallindo Ribeiro. - Campina Grande, 1998.
78 f.

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

1. Redes de Computadores. 2. Sistemas Distribuídos. 3. Processamento. 4. SELJUK-AMOEBA. 5. Dissertação - Informática. I. Brasileiro, Francisco Vilar. II. Universidade Federal da Paraíba - Campina Grande (PB). III. Título

CDU 004.7(043)

PROCESSAMENTO CONFIÁVEL NO AMBIENTE OPERACIONAL SELJUK-AMOEBA

ÉRICA DE LIMA GALLINDO RIBEIRO

DISSERTAÇÃO APROVADA EM 26.06.1998

PROF. FRANCISCO VILAR BRASILEIRO, Ph.D
Orientador

PROF. ANGELO PERKUSICH, D.Sc
Examinador

PROFa. INGRID ELEONORA SCHREIBER JANSCH PÔRTO, Dr.
Examinadora

CAMPINA GRANDE - PB

Aos meus filhos

Agradecimentos

A Deus.

A toda minha família, principalmente a meus pais.

A todos os amigos da pós-graduação.

Ao pessoal do LSD.

Ao meu orientador.

A banca examinadora.

Aos funcionários da COPIN, COPEX e DSC.

Ao pessoal da cantina do DSC.

Resumo

Construir aplicações distribuídas não é uma tarefa simples. Projetistas de tais sistemas têm seguido duas abordagens complementares para reduzir a complexidade de projeto, a saber: i) o uso de ferramentas de desenvolvimento apropriadas; e ii) a escolha da semântica de falha mais restritiva possível para os componentes que formam a camada de execução do sistema. O modelo Seljuk usa estas duas abordagens para especificar uma maneira estruturada de prover serviços para tolerância a faltas no contexto de ambientes operacionais distribuídos, facilitando assim a construção e execução de aplicações com requisitos de confiança no funcionamento.

Neste trabalho, seguindo o modelo Seljuk, e tomando o sistema operacional distribuído Amoeba como substrato, nós apresentamos o projeto e a implementação do serviço de processamento confiável do Seljuk-Amoeba. Nossa proposta se baseia em estender a funcionalidade do serviço de processamento do Amoeba, criando um serviço de processamento confiável. O nosso objetivo é atingido através da introdução de um serviço de execução e pelo desenvolvimento de protocolos de gerência de redundância no serviço de comunicação do Amoeba. O novo serviço de execução desempenha as funções tradicionais do serviço de execução do Amoeba e ainda provê mecanismos para a criação de unidades de processamento replicado.

Abstract

Building dependable distributed applications is not an easy task. Designers of such systems have followed two complementary approaches to reduce design complexity, namely: i) the use of appropriate developing tools; and ii) the choice of the most restrictive failure semantics possible for the components that form the system's underlying execution layer. The Seljuk model uses these two approaches to specify a structured way of providing fault tolerance services in the context of distributed operating environments, thus facilitating the construction and execution of dependable distributed applications.

In this work we have followed the Seljuk model to develop a reliable processing service to the Amoeba distributed operating system. This service offers a fail-controlled semantics for applications, even in the presence of failures in the componentes of the system. Our aim is achieved by the introduction of a new processing service and the development of redundancy management protocols in the Amoeba's communication service. In addition to perform the standard funcionalities of the Amoeba's processing service, we provide mechanisms to create replicated nodes.

Sumário

Lista de Figuras

Capítulo 1

Introdução

Sistemas computacionais distribuídos consistem de uma grande quantidade de componentes de hardware e software que podem vir a falhar. As falhas destes componentes podem levar a um comportamento não previsível do sistema, ou causar a indisponibilidade do mesmo. Para alguns tipos de aplicações a interrupção temporária dos serviços pode ser aceitável. Entretanto, existem aplicações - denominadas aplicações críticas - nas quais as conseqüências de um comportamento não previsível do sistema pode ser muito significativo. Controle de tráfego aéreo, monitorização de equipamentos hospitalares, controle de transações bancárias e controle de centrais telefônicas são exemplos desses tipos de aplicações. Para minimizar as perdas decorrentes de um comportamento não previsível, estas aplicações utilizam-se de mecanismos para tolerância a faltas.

Quando a construção dos mecanismos necessários à obtenção de tolerância a faltas fica a cargo do programador da aplicação, o tempo do desenvolvimento aumenta, uma vez que a aplicação deve agora introduzir características de tolerância a faltas. De fato, um dos principais problemas enfrentados pelos projetistas e programadores de aplicações distribuídas consiste nas dificuldades introduzidas pela necessidade de se tratar e tolerar faltas [Cristian 91].

Quanto mais restritiva for a semântica de falha dos componentes do sistema (a maneira como o componente pode falhar), mais simples será a implementação dos mecanismos para tolerância a faltas. Assim, se os componentes nunca falham (semântica de

falha mais restritiva possível), não é necessário adicionar à aplicação nenhum mecanismo para tolerância a faltas. Infelizmente, todo componente de hardware tem uma vida útil finita, e irá falhar, possivelmente de uma maneira imprevisível. Componentes de software, por sua vez, embora não sofram desgastes com o tempo de vida, são normalmente muito menos confiáveis que os componentes de hardware.

A dificuldade em se construir mecanismos para tolerância a faltas, depende diretamente do tipo de semântica de falha que se assume para os componentes do sistema. Podemos classificar a semântica de falha dos componentes de acordo com os pressupostos que são assumidos sobre o comportamento do componente quando ele falha [Cristian 91]. Semântica de falha por parada, semântica de falha por valor, semântica de falha por omissão, semântica de falha por temporização e semântica de falha bizantina, são cinco categorias de semânticas de falha que podem ser assumidas para os componentes dos sistemas. As quatro primeiras semânticas de falha dos componentes são consideradas semânticas de falha controlada, isto é, o comportamento, na presença de falhas, dos componentes que possuem essas semânticas é previamente conhecido. Por outro lado, a semântica de falha bizantina é considerada uma semântica de falha não-controlada uma vez que não se pode prever que comportamento o componente terá se este vier a falhar.

Numa tentativa de simplificar seus projetos, muitos sistemas tolerantes a faltas descritos na literatura foram construídos assumindo que eles executam em unidades de processamento que apresentam comportamento seguro diante da ocorrência de falhas, ou seja, o processador quando falha, pára. Outros sistemas assumem que os processadores nunca falham, desta forma eles devem executar em unidades de processamento que possam garantir este comportamento por todo o tempo de missão[1] da aplicação.

A semântica de falha de um processador depende, não somente das características do hardware propriamente, mas também, entre outros fatores, dos requisitos de confiabilidade exigidos pela aplicação e da duração da execução da aplicação. Embora tenha-se conhecimento que processadores convencionais possam falhar de uma maneira

[1] Período máximo que o sistema deve funcionar sem a necessidade de manutenção.

arbitrária [Harper et al. 88, Lala 86], a probabilidade desse tipo de falha ocorrer é tão pequena que é possível assumir uma semântica de falha bem definida e comportada (semântica de falha controlada) para esses processadores, e ainda assim, atender aos requisitos de confiabilidade de um grande número de aplicações.

Entretanto existe um número crescente de aplicações cujos requisitos de confiabilidade são tão altos, que, para essas aplicações, não é possível assumir que processadores convencionais apresentem semântica de falha controlada. Para estas aplicações faz-se necessária a construção de unidades de processamento, denominadas nodos, que garantam o comportamento assumido. Nodos com semântica de falha estável [Brasileiro 97], nodos com semântica de falha silenciosa [Brasileiro et al. 92] e nodos com semântica de falha mascarada são três classes de nodos com semântica de falha controlada [Brasileiro 95b].

Uma solução para que um sistema continue a funcionar mesmo se alguns de seus componentes de hardware apresentem falhas é obtida se o próprio sistema operacional - sobre o qual a aplicação executa – disponibilizar mecanismos para tolerância a faltas (p. ex. unidades de processamento replicado) que possam ser usados pelos projetistas a fim de que as aplicações possuam requisitos de confiança no funcionamento[2].

Um modelo que utiliza a solução mencionada acima é apresentado em [Brasileiro 97]. Este modelo, denominado Seljuk, define um ambiente operacional para o desenvolvimento e a execução de aplicações distribuídas tolerantes a faltas. A característica principal desse modelo é construir os serviços para tolerância a faltas embutidos no sistema operacional, permitindo assim que as aplicações definam a semântica de falha assumida para o hardware em tempo de execução; isto torna a construção de aplicações tolerantes a faltas mais simples e flexível.

---

[2] O termo confiança no funcionamento é utilizado em [Lemos-Veríssimo 91] para traduzir o termo em inglês *dependability* apresentado em [Laprie 85]; este é definido como "a confiança depositada no serviço a ser fornecido pelo sistema". Confiança no funcionamento é um conceito geral que enfatiza diferentes atributos dependendo dos requisitos de confiabilidade da aplicação. Confiabilidade e disponibilidade são os atributos mais significativos deste conceito. Confiabilidade diz respeito à probabilidade do sistema prover o serviço correto em um determinado momento. Disponibilidade, por outro lado, refere-se à probabilidade do sistema estar pronto para executar num dado instante.

O objetivo do Seljuk é alcançado através da disponibilização de duas classes de serviços. A primeira classe de serviços é a responsável pela implementação de diferentes semânticas de falhas para os nodos do sistema distribuído. Em tempo de execução, cada aplicação poderá escolher a semântica de falha a ser usada para os nodos onde ela será executada. O modelo provê, de forma transparente, os mecanismos necessários para garantir tal semântica, dentro das limitações do hardware disponível, e de acordo com a semântica de falha desse hardware. Obviamente, quando a semântica de falha dos processadores disponíveis é pelo menos tão restritiva quanto a semântica de falha requerida pela aplicação, não há necessidade de se usar os serviços fornecidos.

A segunda classe de serviços para tolerância a faltas, inclui aqueles serviços de mais alto nível, como por exemplo: serviços sofisticados de comunicação, diagnóstico e recuperação de nodos em falha, manutenção de grupos de processos, processamento replicado, etc [Vasconcelos 97]. Esses serviços poderão ser colocados à disposição das aplicações na forma de chamadas ao sistema, de processos servidores, ou de forma transparente - como os serviços discutidos na primeira classe. Todos os serviços desta classe também deverão levar em consideração a semântica de falha do hardware, definida pela aplicação em tempo de execução.

O Amoeba - um dos sistemas operacionais distribuídos de maior destaque no meio acadêmico e científico - foi escolhido como sistema hospedeiro para a primeira implementação do Seljuk devido à disponibilidade tanto de código fonte quanto de farta documentação. O sistema, denominado Seljuk-Amoeba [Brasileiro et al. 97], consiste na implementação dos serviços definidos no modelo Seljuk, utilizando o sistema operacional distribuído Amoeba [Mullender et al. 90] como substrato.

O Amoeba é um sistema operacional distribuído baseado na tecnologia de micronúcleos. O princípio básico desta tecnologia é minimizar a parte do sistema operacional que executa em modo supervisor, com o objetivo de aumentar a flexibilidade do sistema. Todo processador do Amoeba executa uma parte do sistema operacional denominada micronúcleo. Toda a funcionalidade do sistema operacional que não é fornecida pelo micronúcleo fica a cargo de processos servidores que executam em modo

usuário.

O serviço de processamento do Amoeba é composto de dois servidores: um pertencente ao micronúcleo (*Process Server*) - responsável pela gerência dos processos (criação, escalonamento a curto prazo, etc), e outro que executa em modo usuário (*Run Server)* - responsável pelo balanceamento da carga nos processadores disponíveis.

Neste trabalho nós apresentamos o projeto e a implementação do serviço de processamento confiável do Seljuk-Amoeba. Nossa proposta baseia-se em estender a funcionalidade do serviço de processamento do Amoeba, criando um serviço de processamento confiável. O nosso objetivo é atingido através da introdução de um novo *Run Server*, denominado Seljuk-Amoeba *Run Server*, ou *SA-Run Server*; e pelo desenvolvimento de protocolos de gerência de redundância no serviço de comunicação do Amoeba. O *SA-Run Server* desempenha o mesmo papel do *Run Server* e ainda provê mecanismos para a criação de unidades de processamento replicado.

O serviço de comunicação proposto fará uso dos protocolos para construção de nodos replicados descritos em [Brasileiro 95a]. As aplicações executadas utilizando o serviço de processamento confiável poderão, em tempo de execução, informar a semântica de falha assumida para os nodos onde irão executar, bem como a semântica de falha real dos processadores. O Seljuk-Amoeba deve então prover, de forma transparente, a semântica de falha requerida a partir da replicação do processamento em processadores disponíveis, caso os mesmos tenham uma semântica de falha menos restritiva.

O serviço de processamento confiável do ambiente operacional Seljuk-Amoeba é uma importante ferramenta para a construção de aplicações distribuídas robustas na medida em que este provê a redução da complexidade do desenvolvimento da aplicação, por permitir que os programadores assumam que a semântica de falha dos nodos sobre os quais as aplicações executam seja mais restritiva do que aquela fornecida pelos processadores do sistema. O Seljuk-Amoeba fica encarregado da gerência dos processadores redundantes necessários para garantir a semântica de falha assumida para o nodo.

Esse serviço também oferece flexibilidade para aplicações, por permitir que estas escolham suas semânticas de falha em tempo de ativação. Desta forma, se os requisitos de confiança no funcionamento da aplicação mudam, é possível fornecer o serviço requerido sem a necessidade de recompilar a aplicação, já que o serviço de processamento confiável é implementado no micronúcleo do Seljuk-Amoeba.

O restante deste trabalho está organizado como segue. O Capítulo 2 apresenta alguns conceitos básicos da área de Tolerância a Faltas e uma discussão sobre a implementação de nodos com semântica de falha controlada. Uma vez que o nosso trabalho se insere na área de tolerância a faltas em sistemas distribuídos, faz-se necessário introduzir neste capítulo a definição de sistema distribuído a ser usada, além dos conceitos básicos da área de tolerância a faltas. Além disso são discutidas algumas arquiteturas tolerantes a faltas reportadas na literatura.

Tendo em vista que o serviço de processamento confiável será implementado sobre o sistema distribuído Amoeba, o Capítulo 3 apresenta a estrutura e os principais conceitos deste sistema operacional, mostrando principalmente como se comporta o seu serviço de processamento.

O serviço de processamento confiável proposto baseia-se na construção de nodos que apresentam uma semântica de falha controlada. Uma discussão de como construir nodos é apresentada no Capítulo 4; neste capítulo também é apresentado como estas unidades podem ser inseridas no serviço de processamento do Amoeba para prover o serviço de processamento do Seljuk-Amoeba.

No Capítulo 5 encontram-se detalhes da implementação do serviço de processamento confiável proposto. Embora no projeto deste serviço tenhamos contemplado nodos com semântica de falha mascarada e nodos com semântica de falha silenciosa, para a implementação escolhemos apenas os nodos com semântica de falha silenciosa. Implementamos um nodo com semântica de falha silenciosa composto de dois processadores, utilizando o modelo líder/seguidor para fazer a ordenação das mensagens. Neste capítulo apresentamos também uma pequena análise de desempenho dos nodos replicados. Fazemos uma comparação de uma aplicação cliente/servidor convencional com

uma aplicação cliente/servidor na qual o servidor executa em um nodo replicado.

No Capítulo 6 apresentamos nossas conclusões sobre o trabalho realizado e as direções para futuras pesquisas.

Capítulo 2

Sistemas Distribuídos e Tolerância a Faltas

Em um sistema distribuído, a profusão de processadores possibilita a distribuição uniforme de tarefas entre os mesmos, com a finalidade de aumentar o desempenho global do sistema. Adicionalmente, torna-se possível recuperar as computações que estão executando em processadores que venham a falhar.

Podemos afirmar que um sistema distribuído consiste de vários nodos. Cada nodo consiste de um processador com memória privada, inacessível aos outros nodos, além de um relógio local que governa a execução de instruções neste processador. Cada nodo também possui uma interface de rede. Dois nodos em um sistema comunicam-se, um com o outro, pela troca de mensagens através do enlace de comunicação. Finalmente, existe um software que governa a execução das instruções neste processador. Assim, os principais componentes de um sistema distribuído são processadores, enlaces de comunicação, relógios e software. O esquema de tolerância a faltas visa tratar as falhas desses componentes, de uma maneira que o sistema distribuído, como um todo, também não venha a falhar.

2.1. MODELOS DE SISTEMA

Existem duas maneiras de visualizar um sistema distribuído: da maneira definida pelos componentes físicos do sistema (modelo físico), e da maneira definida do ponto de vista de processamento, ou computação (modelo lógico). O ponto de vista da computação

é importante, uma vez que é o que o usuário percebe e os serviços que estão definidos nesta perspectiva são aqueles para os quais a confiabilidade é desejada. O modelo físico também é importante, já que a computação é realizada sobre a estrutura física, e os componentes desta estrutura são aqueles que podem vir a falhar. A meta da tolerância a faltas em sistemas distribuídos é garantir que alguma propriedade, ou serviço, no modelo lógico seja preservado mesmo na presença de falha de alguns componentes do modelo físico.

2.1.1. MODELO FÍSICO

O modelo físico de um sistema distribuído consiste de várias máquinas que estão em localizações geográficas diferentes. Todos as máquinas são autônomas, e comunicam-se umas com as outras através de um enlace de comunicação. Uma característica básica dos sistemas distribuídos é a separação geográfica e a natureza autônoma das várias máquinas.

Outra característica importante dos sistemas distribuídos é a ausência de uma memória compartilhada entre as diferentes máquinas. Isto é, não existe uma memória no sistema que possa ser usada por mais de uma máquina. Uma área de memória pertence a uma máquina e somente aquela máquina pode acessá-la. Em contraste a isto, alguns sistemas paralelos têm memória compartilhada. Em tais sistemas, diferentes máquinas podem comunicar-se através do uso dessa memória.

2.1.2. MODELO LÓGICO

Em um nível lógico, considera-se que um sistema distribuído consiste de um conjunto finito de processos e canais de comunicação entre estes. Existe um canal entre dois processos se estes interagem através de troca de mensagens. Assim, um sistema distribuído pode ser representado por um grafo, no qual os vértices simbolizam processos no sistema e os arcos simbolizam o fato de que os vértices interligados interagem através desta troca de mensagens.

Constantemente fazem-se suposições a respeito do tempo necessário ao envio de mensagens de um vértice a outro. Um sistema é dito ser **síncrono** se, sempre que está funcionando corretamente, este sempre executa a função pretendida dentro de um tempo

limitado e bem conhecido; caso contrário, o sistema é dito ser **assíncrono**. Um canal de comunicação síncrono é aquele no qual o tempo máximo de transmissão de uma mensagem é limitado e conhecido, enquanto que um processador síncrono é aquele no qual o tempo para executar uma seqüência de instruções é finito e limitado.

A principal vantagem de um sistema síncrono é que a falha de um componente pode ser deduzida pela ausência de resposta dentro de algum tempo pré-definido. Assim, se o sistema distribuído é síncrono, um esquema baseado em temporizadores pode ser empregado para detectar falha dos nodos ou perda de mensagens. Por exemplo, uma falha de um nodo pode ser detectada enviando-se uma mensagem para ele. Se uma resposta não é recebida dentro da duração de um temporizador, então é possível assumir que o nodo receptor falhou. Tal esquema não pode ser utilizado se o sistema for assíncrono, já que não se pode estimar o atraso máximo para a transmissão de uma mensagem de um nodo para outro no sistema .

2.2. TOLERÂNCIA A FALTAS

A tarefa de projetar arquiteturas tolerantes a faltas é bastante difícil. Além de manter o controle das atividades regulares do sistema, o projetista de uma arquitetura tolerante a faltas tem que se preocupar também com todas as complexas situações que podem ocorrer se um componente vier a falhar. Esta dificuldade pode ser aumentada ainda mais pelo uso de uma terminologia confusa. Apesar da área de tolerância a faltas não ser uma área nova[3], até hoje é bastante comum ver grupos diferentes utilizando diferentes termos para um mesmo conceito, ou utilizando o mesmo termo para diferentes conceitos. Por exemplo, o que um grupo chama de falha, um outro chama de falta. No momento existem alguns acordos sobre as definições destes termos em várias línguas e quais conceitos eles representam. Neste trabalho usaremos a terminologia proposta por [Lemos-Veríssimo 91] para a língua portuguesa, e também as extensões a esta terminologia propostas em [Vasconcelos 97].

[3] A principal conferência na área de tolerância a faltas - IEEE's *International Conference on Fault Tolerant Computing Systems* (FTCS) - começou há quase três décadas.

Uma falha do sistema ocorre quando o comportamento do sistema desvia-se daquele definido pelas suas especificações. Isto é, uma falha do sistema ocorre quando ele não pode fornecer o serviço desejado. Um erro é aquela parte do estado do sistema que pode levar a uma falha do mesmo. Se existe um erro no estado do sistema, então existe uma seqüência de ações que pode ser executada pelo sistema e que levará a uma falha do mesmo, a menos que alguma medida de correção seja empregada. Os erros são causados por faltas. As faltas caracterizam uma condição física anormal, que geralmente resulta de erros de concepção e fatores externos difíceis de prever, o que torna a tarefa de tolerar faltas extremamente complicada. De uma maneira geral, uma falta é definida como o defeito que tem o poder de gerar erros.

Quanto maior o número de componentes, maior o número de itens que podem falhar. De maneira similar, quanto mais complexo um componente de hardware, maiores são as chances de este estar em um estado errôneo. Um sistema é dito ser tolerante a faltas se este comportar-se de acordo com o que foi especificado mesmo quando ocorrerem falhas em alguns de seus componentes.

Para se tolerar falhas que possam ocorrer nos componentes de um sistema, é necessária a introdução de redundância. Esta é definida como a parte não necessária ao correto funcionamento do sistema se nenhuma tolerância a faltas é suportada [Jalote 94]. Isto é, o sistema funciona corretamente sem redundância, se as falhas não vierem a acontecer. A redundância em um sistema pode ser de hardware, de software, ou redundância no tempo.

A redundância de hardware inclui os componentes de hardware que são acrescentados ao sistema para prover tolerância a faltas. A redundância de software, por sua vez, inclui todos os programas e instruções que são utilizadas para o suporte a tolerância a faltas. Uma técnica comum para tolerância a faltas é executar alguma instrução (ou seqüência de instruções) várias vezes. Esta técnica requer redundância no tempo, isto é, tempo extra para executar tarefas para tolerância a faltas.

Freqüentemente, todas as três formas de redundância são usadas em sistemas distribuídos. Redundância de hardware é empregada na forma de processadores, memórias

ou canais de comunicação extras. Redundância de software é utilizada para gerenciar estes componentes extras de hardware e utilizá-los corretamente para prover a continuidade do serviço, no caso de falha de algum dos componentes. Entretanto, ao se fazer inserção de redundância a fim de se alcançar um certo grau de tolerância a faltas, está se aumentando o custo do desenvolvimento e também a complexidade da aplicação.

2.3. FASES NA TOLERÂNCIA A FALTAS

Existem algumas fases gerais que a maioria dos sistemas que empregam técnicas de tolerância a faltas têm que percorrer. Estas fases são: detecção do erro, confinamento do dano, recuperação do erro, e tratamento da falta e continuação do serviço.

Na primeira fase, a presença de uma falta é determinada pela detecção de um erro no estado do sistema. Desde que a detecção do erro é o ponto de partida para o suporte a tolerância a faltas, uma estratégia de tolerância pode ser, no máximo, tão boa quanto o seu método de detecção do erro.

Uma vez que o erro tenha sido detectado, isto implica que ocorreu uma falta em algum dos componentes do sistema. Qualquer dano causado devido a esta falta tem que ser identificado e delimitado na fase de confinamento do dano. O objetivo desta fase é identificar os limites dentro dos quais a extensão do erro é confinada. Depois disto, o erro no estado do sistema tem que ser corrigido. Isto é feito na fase de recuperação do erro. Desde que existe um erro no estado do sistema, é necessário removê-lo para que este não se propague em ações futuras e para que se evite a ocorrência de uma falha. Com a recuperação do erro, o sistema alcançará um estado livre do erro.

Na fase final, tratamento da falta e continuação do serviço, a falta ou o componente danificado tem que ser identificado. Uma vez que o componente defeituoso tenha sido identificado, o sistema será "corrigido" através de sua reconfiguração, e pelo uso da redundância embutida de forma que, ou o componente defeituoso não é usado, ou é usado de uma maneira diferente da usual.

2.4. SEMÂNTICA DE FALHA DO HARDWARE

A semântica operacional de um componente corresponde às especificações que são padrão do serviço deste componente, enquanto que a semântica de falha descreve o comportamento do componente quando ocorre um número limitado de falhas de seus sub-componentes.

Como exemplos de possíveis semânticas de falha assumidas para o hardware, temos: semântica de falha por omissão, semântica de falha por parada, semântica de falha por temporização, semântica de falha por valor e semântica de falha bizantina.

Quando um componente pode omitir a resposta a uma dada entrada (e.g., um canal de comunicação que perde mensagens) diz-se que o mesmo apresenta uma semântica de falha por omissão. Se após omitir a produção da saída pela primeira vez, o componente omite a produção da saída para todas as entradas subsequentes, até que seja reiniciado, diz-se que ele apresenta uma semântica de falha por parada (e.g., queda de alimentação de um componente).

Outra semântica de falha possível é a semântica de falha por temporização que ocorre quando a resposta do componente é funcionalmente correta porém fora de tempo, isto é, a resposta aconteceu fora do intervalo de tempo especificado. As falhas de temporização podem ser tanto falhas adiantadas como atrasadas, acontecendo quando o componente responde antes ou depois do tempo especificado. Este último caso é também denominado falha de desempenho. Quando o componente pode responder incorretamente, ou seja, o valor de sua saída pode ser incorreto, diz-se que a sua semântica de falha é por valor. A semântica de falha arbitrária é aquela associada a um componente que pode se comportar de maneira totalmente arbitrária durante a falha. A especificação do funcionamento correto de um componente pode ser feita em um domínio de tempo ou em um domínio de valor. O domínio do tempo refere-se ao intervalo de tempo necessário para que o componente responda a um determinado estímulo enquanto que o domínio do valor refere-se ao conjunto de valores que devem ser retornados por este componente.

As semânticas de falhas supracitadas formam uma hierarquia, com a semântica de

falha por parada sendo a mais simples e mais restritiva, e a semântica de falha bizantina sendo a menos restritiva.

2.5. NODOS COM SEMÂNTICA DE FALHA CONTROLADA

Nodos replicados são unidades de processamento construídas através do agrupamento de processadores redundantes. Estes processadores são escolhidos de maneira que falhem de forma independente, isto é, a falha de um processador não tem nenhuma interferência na falha de outro processador. Como um exemplo bem simplificado de dependência na maneira de falhar, podemos supor que se tenha um nodo formado por três processadores com cada um estando localizado em uma máquina distinta, e sendo todas as máquinas alimentadas pela mesma fonte de energia. Se em decorrência de um problema qualquer, esta fonte de alimentação vier a falhar, todos os três processadores que estão executando a computação replicada certamente irão falhar. Estes tipos de problemas devem ser levados em consideração quando se for projetar um nodo replicado.

Uma vez tendo-se escolhido os processadores que formarão o nodo, a computação, a ser executada de forma confiável, é então replicada e executada simultaneamente em cada um deles. Os resultados das computações de cada um dos processadores são avaliados por um mecanismo apropriado (ex. comparação, votação majoritária, etc.) que garante uma semântica de falha controlada para o nodo, como mostra a Figura 2.1.

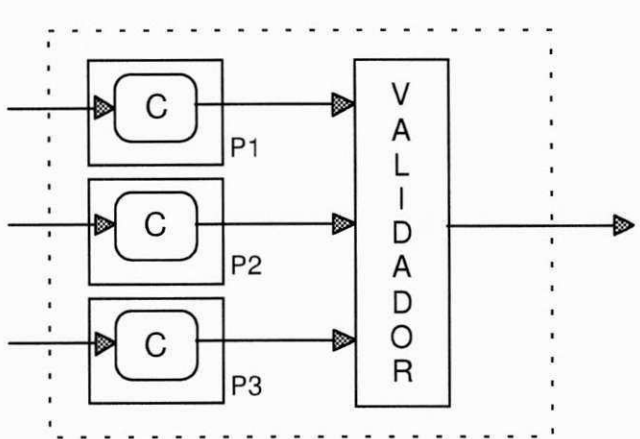


Figura 2.1: Um Nodo com Semântica de Falha Controlada

A maioria dos nodos com semântica de falha controlada apresentados na literatura são implementados em hardware. Os processadores formando o nodo são sincronizados a cada ciclo de relógio, e a cada ciclo os resultados das operações de cada um dos processadores são validados por um circuito que implementa a semântica de falha do nodo. A principal vantagem dessa abordagem é a reduzida perda de desempenho introduzida pelos circuitos especiais. Entretanto, existem alguns problemas associados com essa técnica: primeiro, os processadores precisam ser construídos de tal maneira que eles tenham um comportamento determinístico a cada ciclo de relógio; segundo, a introdução de circuitos especiais aumenta a complexidade do projeto, o que, em última instância, pode reduzir a confiabilidade do nodo como um todo; terceiro, é difícil incorporar avanços tecnológicos, sem que seja necessário um considerável esforço de (re)projeto do nodo; finalmente, o fato de se ter os mecanismos para tolerância a faltas fisicamente incorporados ao nodo, implica na forçosa distribuição do custo associado a esses mecanismos por todas as aplicações executando no nodo, inclusive por aquelas que não necessitam de serviços para tolerância a faltas.

Outra estratégia adotada é implementar em software tanto os mecanismos para sincronização dos processadores redundantes quanto a validação das saídas do nodo. Os projetistas do SIFT [Wensley et. al 78] foram os pioneiros a utilizar esta estratégia. Essa abordagem é muito mais flexível do que a abordagem baseada em hardware. Pode-se inclusive, utilizar diferentes tipos de processadores em um mesmo nodo, já que os mesmos não precisam estar sincronizados a cada ciclo de relógio. Esta diversidade de tipos de processadores possibilita o tratamento de faltas de concepção[4] na construção dos processadores. Além disso, a atualização tecnológica desses nodos é trivial, pois os protocolos implementados em software precisam apenas ser recompilados para a nova plataforma. Por outro lado, o principal problema da abordagem de software é a possibilidade de uma queda acentuada no desempenho do sistema. No SIFT, por exemplo, os recursos do processador utilizados para a gerência da redundância somam mais de 80%

[4] "Faltas de concepção" é o termo proposto em [Lemos-Veríssimo 91] para traduzir o termo em inglês *design faults* e é definido como as faltas resultantes de erros humanos introduzidos na fase de concepção ou de manutenção do sistema.

do total disponível. Devido a este problema, surge uma outra abordagem, denominada abordagem híbrida, que tenta fazer uso das vantagens presentes nas abordagens baseadas em software e hardware. O problema desta nova abordagem é que ao se tentar capturar, ao mesmo tempo, as vantagens das abordagens baseadas em software e baseadas em hardware, também se herda os problemas inerentes de cada uma delas. A abordagem híbrida pode ser usada em arquiteturas que pretendam atender classes de aplicações diferentes (aplicações críticas e não-críticas, por exemplo). Assim aplicações que tenham algum requisito de tempo podem ser executadas em unidades de processamento nas quais os mecanismos de gerência da redundância são implementados em hardware, evitando assim a perda do desempenho associado à abordagem baseada em software.

Existem várias arquiteturas tolerantes a faltas que utilizam o conceito de nodos com semântica de falha controlada para prover um processamento replicado confiável. Como a abordagem híbrida é pouco utilizada, veremos agora exemplos de arquiteturas que implementam os nodos replicados utilizando apenas as abordagens baseadas em software e em hardware.

2.5.1. NODOS IMPLEMENTADOS EM HARDWARE

- SEQUOIA

Sequoia [Bernstein 88] é um sistema comercial distribuído para executar o processamento de transações *on-line*. Este assume que os processadores possuem semântica de falha arbitrária e emprega replicação ativa para tolerar faltas. A arquitetura é caracterizada pelo uso extensivo de técnicas de detecção de falhas em hardware, possuindo a habilidade de rapidamente recuperar todos os processos depois de um falha, fazendo isto de forma transparente para o usuário.

Uma arquitetura simplificada do Sequoia é mostrada na Figura 2.2. Um sistema Sequoia consiste de um número de unidades de processamento (PEs) e unidades de memória (MEs), conectados via barramentos redundantes. Cada PE é construído utilizando dois processadores que são sincronizados a cada ciclo de relógio (operam em *lock-step*). Um comparador detecta qualquer diferença entre as saídas geradas pelos processadores de um PE, garantindo que estes tenham uma semântica de falha silenciosa.

Cada PE contém uma memória cache. Um processo P executando em uma PE utiliza esta memória cache para armazenar seus dados; todas as operações de entrada e saída são feitas na cache. Periodicamente, os conteúdos da cache são colocados em duas MEs distintas. Com isso então, estabelece-se um ponto de verificação replicado.

O código de um processo é armazenado em uma ME. Se um PE falha, cada processo que está executando neste PE é reiniciado em outro PE, começando a execução deste em seu último ponto de verificação.

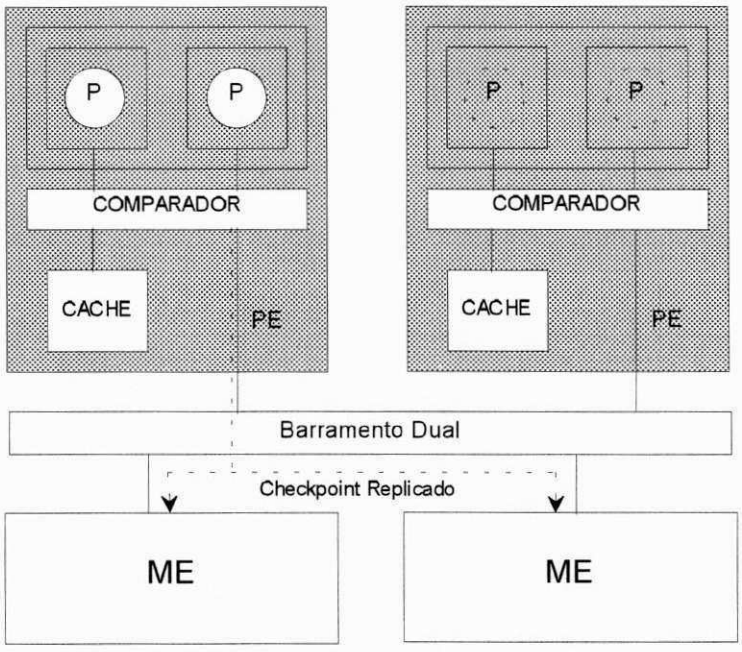


Figura 2.2: O Sistema Sequoia

Os dados são protegidos por código de detecção de erros enquanto armazenados em uma cache do PE, ME ou enquanto estiverem em trânsito no barramento. O hardware que implementa todo o armazenamento dos dados é particionado de uma maneira tal que qualquer dois bits do mesmo byte, incluindo o bit de paridade, não tenham qualquer componente comum. Assim sendo, a falha de um único componente pode produzir somente um único erro em qualquer byte, e tais erros são detectáveis. Se um erro é detectado, o byte pode ser descartado e o componente que o enviou então apresenta uma semântica de falha silenciosa.

Todos os componentes do Sequoia possuem semântica de falha silenciosa, ou

através de replicação ativa (PEs) ou através da detecção do erro (MEs ou barramentos). A replicação de todo código e dados, usando pontos de checagem e cópias em disco, garante que a execução de um processo que falhou possa ser retomada em qualquer ponto do seu processamento.

Na ocorrência de uma falha, a execução de um processo é reiniciada por outro PE. O novo processo herda a história (seu caminho de execução imaginário) do processo antigo através do último ponto de checagem. Assim o caminho da execução do processo não precisa ser determinístico, e isto não causa divergência no estado do sistema.

- STRATUS

Stratus [Webber-Beirne 91] é um outro sistema comercial distribuído para o processamento de transações on-line. Como no Sequoia, no Stratus obtém-se um nodo com semântica de falha silenciosa através do agrupamento de dois processadores convencionais, que operam em *lock-step*, e cujas saídas são comparadas por um circuito comparador confiável, antes de se tornarem disponíveis. Para manter os processos sincronizados, os processadores são controlados por um relógio confiável comum.

Um sistema Stratus consiste de um número de módulos de processamentos (PM) conectados por um conjunto de barramentos denominado StrataLINK, como mostrado na Figura 2.3.

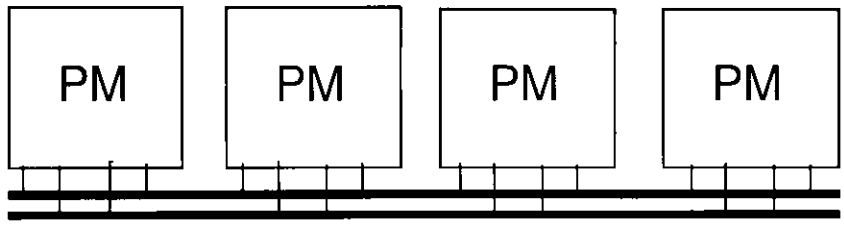


Figura 2.3: O Sistema Stratus

Cada módulo de processamento consiste de dois elementos de processamento idênticos (PEs), conectados por um barramento redundante denominado StratusBUS, como mostrado na Figura 2.4. Cada unidade de processamento é formada por um número de componentes (CPU, memória, etc) e constitui a menor unidade substituível do sistema.

Um componente CPU é construído a partir de dois processadores idênticos que operam em *lock-step* (como no Sequoia). As saídas dos dois processadores são comparadas por mecanismos de hardware. Enquanto estas estiverem combinando, serão colocadas no StratusBUS. Se uma discordância é detectada, as saídas não são propagadas, e o componente é isolado do StratusBUS, desta forma tem-se uma semântica de falha silenciosa.

Um componente de memória contém circuitos de detecção/correção de erros que previne a propagação de dados errôneos. Um componente de memória portanto, também possui uma semântica de falha silenciosa.

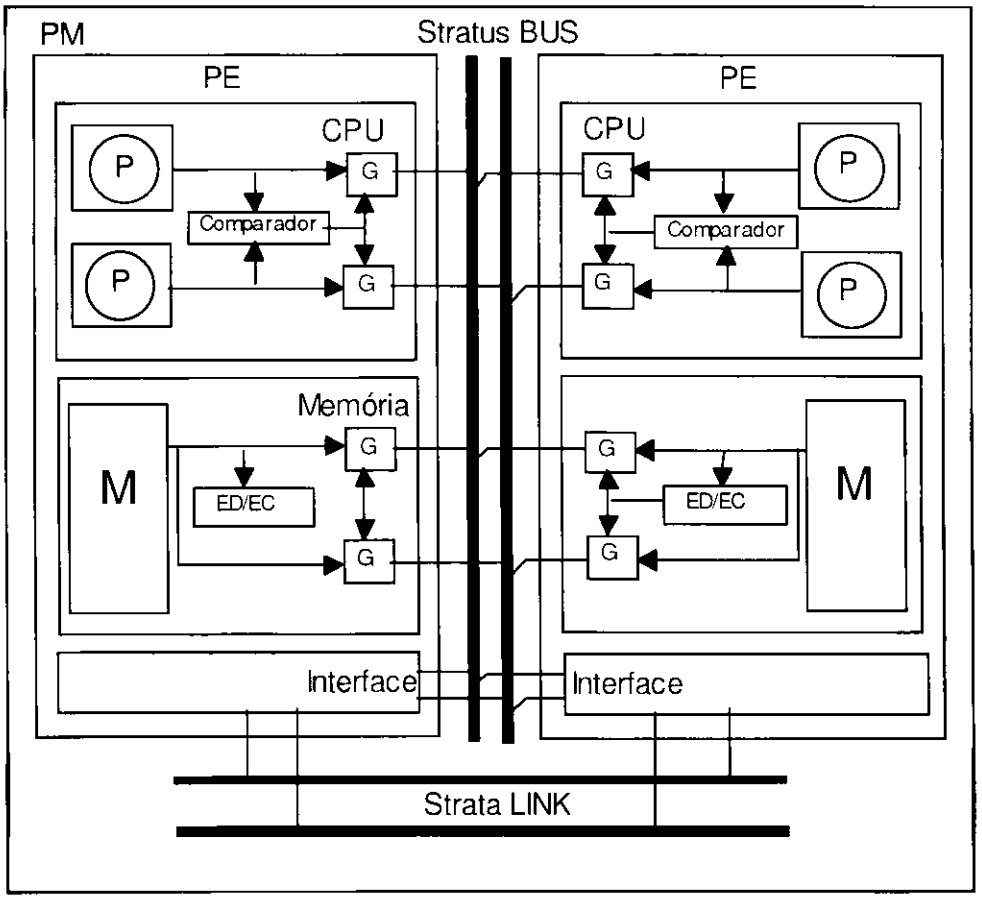


Figura 2.4: Um Módulo de Processamento do Stratus

Quando um componente em um PE emite/recebe uma mensagem para/do StratusBUS, o componente correspondente (no outro PE do mesmo PM) também emite/recebe a mesma mensagem (na ausência de falhas). Assim, quando uma única falha ocorre, existirá sempre, no mínimo, uma mensagem emitida/recebida e a falha é mascarada.

Os processadores que formam os dois componentes CPU em um módulo de processamento são alimentados exatamente com as mesmas entradas e na mesma ordem. Logo, todos os quatro processadores em um módulo de processamento operam em *lock-step* e desta forma os processos a serem executados não precisam ser determinísticas.

- FTMP

O FTMP [Hopkins et al. 78] é um computador multiprocessado que foi projetado como parte de um programa de controle de vôo de aeronave comercial. A estrutura do FTMP consiste de um número *i* de processadores, cada qual com sua própria memória M_i, que juntos formam um módulo de processamento. Os processadores são conectados aos barramentos múltiplos; a memória permanece privada para o seu processador. Existe um número de módulos de memória global, também conectados aos barramentos e acessíveis a todos os processadores. Qualquer módulo, seja de memória ou de processamento, pode receber dados em todos os barramentos porém pode enviar dados apenas em um (a escolha deste barramento é configurável).

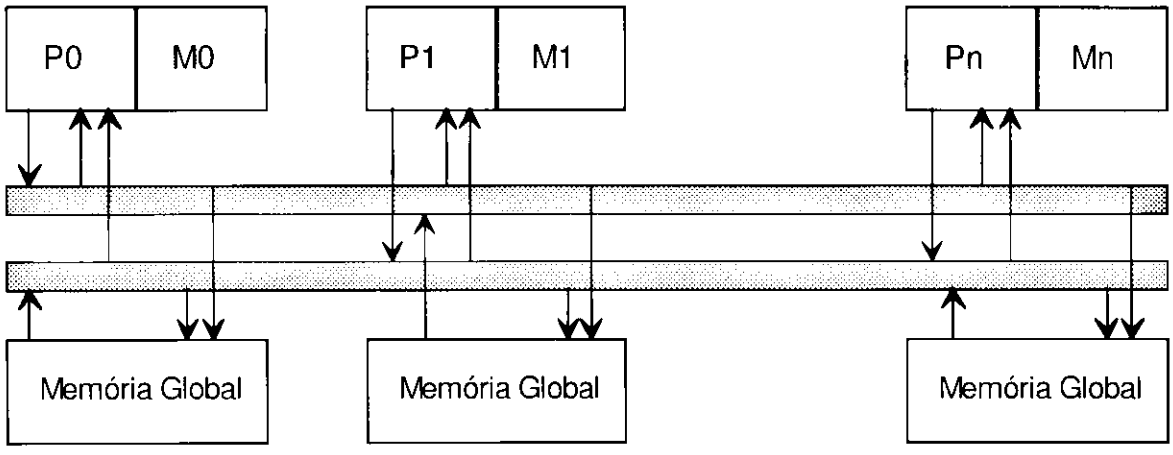


Figura 2.5: A Arquitetura do FTMP

Todo processador executa exatamente um processo (em uma dado instante). Todo processo é replicado (três vezes) e cada réplica executada em um membro diferente do módulo de processamento. Quando um módulo de processamento envia uma mensagem para outro, três mensagens são enviadas, uma para cada réplica. Estas três mensagens serão enviadas, necessariamente, em barramentos diferentes. Cada réplica no módulo destino recebe todas as três mensagens, uma de cada barramento, e as combina (em

hardware) para conseguir uma maioria. A falha de uma réplica ou a falha de um barramento serão mascaradas. Embora a maioria seja conseguida por mecanismos de hardware, cada membro do módulo de processamento tem seu próprio elemento de votação. Desta forma, a falha de um elemento de votação será tratada como se fosse a falha de todo o módulo do qual ele faz parte.

Todos os relógios dos módulos são fortemente sincronizados. Assim, os processos em um módulo de processamento permanecem operando em *lock-step* e processam os pedidos de entrada e saída de maneira síncrona. Embora os módulos de processamento estejam operando independentemente, o acesso aos barramentos é estritamente controlado de forma que os processadores ganhem acesso simultâneo e exclusivo.

2.5.2. NODOS IMPLEMENTADOS EM SOFTWARE

- SIFT

O SIFT [Wensley et al. 78] é um sistema computacional extremamente confiável projetado para executar aplicações críticas de controle de aeronaves. A tolerância a faltas no SIFT é alcançada através da replicação de tarefas entre unidades de processamento. As principais unidades de processamento são minicomputadores de prateleira, com microcomputadores convencionais servindo como a interface de entrada/saída do sistema. A meta do projeto SIFT é alcançar um probabilidade de falha menor que 10^{-10} por hora, por um período máximo de 10 horas de vôo.

Sempre que possível, os mecanismos para tolerância a faltas neste sistema são implementados em software ao invés de hardware. Isto inclui detecção e correção de erro, diagnósticos, reconfiguração e prevenção para que uma unidade defeituosa não venha a prejudicar o funcionamento do sistema como um todo.

A estrutura do hardware do SIFT é mostrada na Figura 2.6. A computação é realizada pelos processadores principais. Os resultados de cada um dos processadores são armazenados em memórias que são unicamente associadas a cada processador. Um processador e sua memória são conectados por um canal convencional com grande largura de banda. Os processadores de I/O e as memórias são estruturalmente similar aos

processadores principais e memórias, porém de menor capacidade computacional e de armazenagem. Estes estão conectados às unidades de entrada e saída do sistema que, para esta aplicação, são os sensores e atuadores da aeronave. Cada processador P_i e sua memória associada M_i, formam um módulo de processamento, e cada módulo é conectado a um barramento múltiplo do sistema.

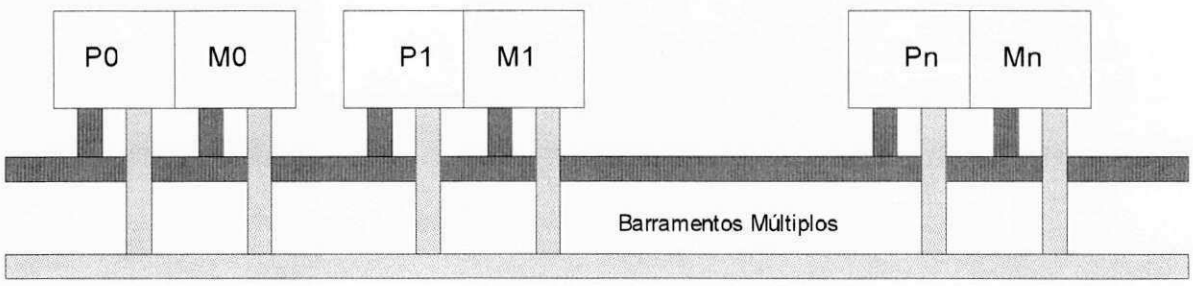


Figura 2.6: A Arquitetura do SIFT

O sistema executa um conjunto de tarefas. Cada tarefa consiste de uma seqüência de iterações. A entrada para cada iteração de uma tarefa é a saída produzida pela iteração anterior de algum conjunto de tarefas (que pode incluir a própria tarefa). As entradas e saídas, do sistema como um todo, são obtidas das tarefas executadas nos processadores de I/O. A confiabilidade é alcançada devido a execução de cada iteração de uma tarefa ser feita de forma independente por um número de módulos. Depois de executar a iteração, um processador coloca o resultado na sua memória. Um processador que utiliza esta saída da iteração determina seu valor examinando as saídas geradas por cada processador que executou a iteração. Tipicamente, o valor é escolhido por uma votação. Se todas as cópias das saídas não são idênticas, então um erro ocorreu. Tais erros são registrados na memória do processador, e estes registros são usados pelo sistema executivo para saber que módulo está defeituoso.

O mecanismo de comunicação entre processos e entre tarefas do SIFT permite um grau de assincronismo entre processadores e evita a forte sincronização necessária aos sistemas ultra confiáveis. Os relógios de todos processadores são sincronizados dentro de 50μs. Periodicamente, os relógios dos processadores precisam ser resincronizados para garantir que um relógio não esteja muito defasado em relação a nenhum outro.

O número de processadores que executam uma tarefa varia de acordo com a tarefa, e podem ser diferentes para a mesma tarefa em diferentes momentos - e.g., se uma tarefa não é crítica em algum instante, esta pode tornar-se crítica em outro momento. A alocação de tarefas para os módulos é, em geral, diferente para cada módulo. Isto é determinado dinamicamente por uma tarefa chamada de sistema executivo, que diagnostica erros a fim de determinar quais módulos e, ou, barramentos estão defeituosos. Quando o sistema executivo decide que um módulo tornou-se defeituoso, este reconfigura o sistema fazendo mudanças na alocação de tarefas aos módulos.

- **Voltan**

Outra arquitetura que assume uma abordagem puramente baseada em software para a gerência da redundância necessária à obtenção de tolerância a faltas é a família Voltan de nodos replicados [Shrivastava et al. 92]. Estes nodos são construídos através do agrupamento de processadores que falham de maneira independente. Dentre os nodos com semântica de falha controlada definidos pela família Voltan estão os nodos com semântica de falha segura e os nodos com semântica de falha mascarada.

Diferentemente do SIFT, os nodos Voltan caracterizam-se por poderem ser utilizados para prover tolerância a faltas para aplicações de propósitos gerais. O funcionamento dos nodos da família Voltan será detalhado na Seção 4.3, onde também mostraremos como iremos utilizá-los no projeto de um serviço de processamento confiável para o Seljuk-Amoeba.

Capítulo 3

O Sistema Operacional Distribuído Amoeba

O Amoeba é um sistema operacional distribuído de propósito geral que se baseia no modelo cliente/servidor. O princípio básico deste sistema é fazer com que um conjunto de máquinas cooperem a fim de realizar determinada tarefa, de forma que pareça um único sistema integrado. De uma maneira geral, os usuários não estão cientes do número e localização dos processadores que executam seus comandos, nem do número e localização dos servidores que armazenam seus arquivos. Para o usuário, o Amoeba assemelha-se a um sistema operacional centralizado de tempo compartilhado.

A primeira versão do Amoeba - Amoeba 1.0 - foi lançada em 1983, como fruto do trabalho de Andrew Tanenbaum e três de seus estudantes de Ph.D., Frans Kaashoek, Sape J. Mullender e Robert van Renesse, na *Vrije University* - Amsterdã. Atualmente, o Amoeba encontra-se disponível em sua versão 5.3. Embora ainda não tenha penetrado no mercado comercial, o Amoeba tem sido usado amplamente como base para testes de aplicações distribuídas e paralelas.

O Amoeba é um sistema operacional baseado na tecnologia de micronúcleos. Um micronúcleo é a parte central do sistema operacional que possui uma função limitada e bem conhecida. O princípio básico da tecnologia de micronúcleos é minimizar a parte do sistema operacional que executa em modo supervisor, com o objetivo de aumentar a flexibilidade do sistema. Toda a funcionalidade do sistema operacional que não é disponibilizada pelo micronúcleo fica a cargo de processos servidores que executam em

modo usuário. Desta forma, os serviços providos por esses servidores podem ser modificados/adaptados/configurados muito mais facilmente, atendendo às diferentes exigências das aplicações. Ao contrário de um sistema operacional convencional, que é planejado de forma a suportar aplicações diretamente, um micronúcleo é projetado de maneira a possibilitar que os programadores de sistemas o utilizem como base para construir vários outros sistemas operacionais sobre ele. Os serviços que os micronúcleos oferecem são aqueles necessários para projetistas de sistemas operacionais, não aqueles necessários a usuários habituais.

3.1. ASPECTOS DE PROJETO

Muitos projetos de pesquisa em sistemas operacionais distribuídos partem de um sistema existente (e.g., UNIX) e adicionam novas características a fim de torná-lo distribuído. No projeto do Amoeba tomou-se uma abordagem diferente. Os projetistas do sistema preferiram iniciar um novo projeto e experimentar novas idéias sem ter que se preocupar em manter compatibilidade com as funcionalidades de algum sistema já existente. Soluções para as inúmeras questões levantadas foram discutidas ao longo do projeto do Amoeba. A partir destas discussões, a equipe deste projeto enumerou as principais características que o Amoeba se proporia a ter:

Distribuição - O Amoeba é um sistema distribuído no qual múltiplas máquinas podem ser interconectadas. Estas máquinas não necessitam ser todas do mesmo tipo, podendo estar espalhadas em uma rede local, ou podendo múltiplas redes locais serem conectadas através de redes de longa distância, como a Internet.

Paralelismo - Um único programa no Amoeba pode utilizar múltiplos processadores a fim de obter maior velocidade de processamento.

Transparência - O usuário não necessita saber o número nem a localização de processadores que executam seus processos, nem o lugar onde os seus arquivos estão armazenados.

Desempenho - Nenhum dos objetivos acima pode ser considerado aceitável se o desempenho for fraco. Dessa forma o desempenho do sistemas foi uma das principais

preocupações de seus projetistas. Em particular, os mecanismos básicos de comunicação foram otimizados para permitir que o envio e o recebimento de mensagens fosse feito com o menor atraso possível.

3.2. ARQUITETURA DO SISTEMA

A arquitetura do Amoeba é composta de quatro componentes principais, cada qual executando o seu próprio micronúcleo. Estes micronúcleos provêem um número limitado de funções, tais como: gerência de processos, serviços de comunicação e entrada/saída. A figura abaixo mostra o modelo de um sistema Amoeba, e seus principais componentes.

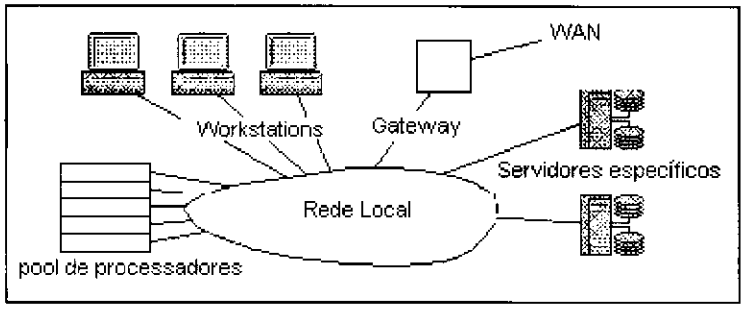


Figura 3.1: Arquitetura do Amoeba

- **Estações de trabalho** (*Workstations*) - Cada usuário possui uma estação de trabalho. Estas são usadas para realizar edições de textos e outras tarefas que requerem uma resposta interativa e rápida. As estações de trabalho são inteiramente dedicadas a executar a interface com o usuário, não fazendo nenhuma outra computação.

- **Infra-estrutura de processamento** (*Processor Pool*) - O poder de processamento do Amoeba está localizado em um ou mais *pools de processadores*. Isto significa que todos os processos do usuário no Amoeba, são executados nos processadores pertencentes a estes *pools*. Estes *pools* consistem de um número substancial de processadores, cada qual com sua própria memória local e uma conexão de rede. Memória compartilhada não é necessária, mas se existir pode ser usada para otimizar a troca de mensagens, fazendo cópia memória-a-memória ao invés de enviar mensagens através da rede de comunicação.

- **Servidores especializados** - No Amoeba existem servidores de arquivos, servidores de diretórios, servidores de inicialização do sistema e vários outros servidores com funções

específicas.

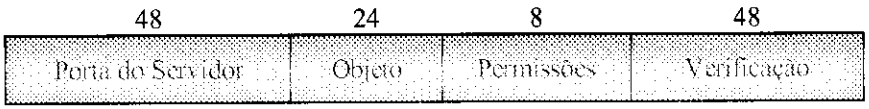
• ***Gateways*** - Estes são usados para unir diferentes redes Amoeba existentes em diferentes locais, de forma a torná-las um único sistema. Os *gateways* isolam o Amoeba das peculiaridades dos protocolos de comunicação que devem ser usados em redes de longa distância. Embora o Amoeba use um protocolo de rede proprietário, um servidor no nível do usuário (servidor TCP/IP) é fornecido para permitir comunicação TCP/IP com outras redes.

3.3. OBJETOS E CAPABILITIES

Todos os servidores do Amoeba funcionam com base no conceito de objeto – que é um tipo de dado sobre o qual são realizadas operações bem definidas. Para um objeto do tipo arquivo, por exemplo, como operações típicas podemos citar: a operação de leitura, a operação de gravação e a operação de remoção de um arquivo. As operações sobre os objetos são executadas de maneira síncrona, isto é, ao fazer uma chamada remota de procedimento[5] ao servidor que gerencia o objeto, o cliente fica bloqueado esperando por uma resposta.

Os objetos, no Amoeba, são gerenciados por servidores específicos. Para cada objeto do Amoeba, existe uma *capability* associada a ele. Uma *capability* é uma espécie de chave que permite ao seu detentor executar alguma (não necessariamente todas) operação sobre o objeto.

Quando um objeto é criado, o servidor que o gerencia devolve uma *capability* ao processo criador. Nas operações subsequentes, para utilizar o objeto, o cliente deverá apresentar essa *capability* ao servidor, a fim de identificar o objeto e comprovar a permissão de manipulá-lo. Uma *capability* consiste em uma estrutura de dados de 128 bits, como mostra a Figura 3.2.



[5] RPC - *Remote Procedure Call* [Birrell-Nelson 84]

Figura 3.2: Exemplo de uma *capability*

Na *capability*, o primeiro campo (Porta do Servidor) é um endereço lógico onde se pode encontrar o servidor que gerencia o objeto. O segundo campo (Objeto) é utilizado pelo servidor para identificar o objeto em questão. Por exemplo, um servidor de arquivos gerencia diversos arquivos; com esse campo, o servidor poderá saber que arquivo em especial ele está manipulando. Já o terceiro campo (Permissões) informa as operações que o detentor da *capability* pode realizar sobre o objeto. Por último, existe um quarto campo (Verificação) que serve para validar a *capability*. Como as *capabilities* são manipuladas diretamente por processos do usuário, se não existisse alguma forma de proteção estes poderiam falsificá-las.

3.4. GERÊNCIA DE PROCESSOS

No Amoeba, cada processo tem seu próprio espaço de endereçamento e uma ou mais linhas de execução (*threads*) que executam em paralelo. Cada *thread* tem seu próprio PC (*program counter*) e sua própria pilha, mas compartilha código e dados globais com as outras *threads* do mesmo processo. Do ponto de vista do programador, cada *thread* comporta-se como um processo seqüencial, exceto pelo fato de que as *threads* de um processo podem se comunicar usando memória compartilhada.

O objetivo de se ter várias *threads* dentro de um processo é obter o aumento do desempenho através do paralelismo. A possibilidade de existirem várias *threads* dentro de cada processo enquadra-se perfeitamente dentro das necessidades do modelo de computação distribuída e paralela. Por exemplo, um servidor de arquivos pode ter múltiplas *threads*. Inicialmente, cada *thread* espera pela chegada de uma requisição. Quando o pedido é recebido, ele é aceito por uma das *threads*, que então começa a processá-lo. Se em seguida a *thread* bloqueia esperando por alguma operação de entrada/saída, outras *threads* podem continuar executando. Todas as *threads*, embora tenham um controle de certa forma independente, podem acessar uma *cache* comum, e usar semáforos para prover a sincronização entre elas. Este projeto de se ter várias *threads* dentro de um mesmo processo torna mais fácil programar servidores e aplicações paralelas.

3.5. GERÊNCIA DE MEMÓRIA

O Amoeba tem um modelo de memória extremamente simples. Um processo pode ter o número de segmentos que sejam necessários, localizados no seu espaço de endereçamento virtual. Os segmentos não são migrados ou paginados, assim um processo deve estar inteiramente na memória para que possa ser executado. Além do mais, mesmo usando uma MMU - *Memory Management Unit*, cada segmento é armazenado na memória por contiguidade.

3.6. ESTRUTURA DE COMUNICAÇÃO

Sistemas operacionais distribuídos são estruturados, tipicamente, em torno do paradigma cliente/servidor. Neste modelo, um processo, denominado *cliente*, pede a outro processo, denominado *servidor*, que faça um trabalho para ele. O cliente então fica bloqueado até que o servidor lhe envie uma resposta. O mecanismo de comunicação normalmente utilizado para implementar o modelo cliente/servidor é o mecanismo de RPC.

Embora o RPC seja uma boa abstração para o tipo de comunicação pedido/resposta, existe um grande número de aplicações que requer que um grupo de processos interaja de maneira fechada. A comunicação em grupo permite que uma mensagem seja enviada confiavelmente de 1 remetente para n receptores. Esta estratégia pode ser útil, por exemplo, para aplicações que replicam dados a fim de obter tolerância a faltas. Tal aplicação necessitará de comunicação em grupo para manter consistentes os dados que estão replicados.

No Amoeba encontramos tanto RPC quanto comunicação em grupo. Para dar suporte a RPC e comunicação em grupo, o Amoeba dispõe de um protocolo de rede denominado FLIP - *Fast Local Internet Protocol* [Kaashoek et al. 93].

Na Figura 3.3 pode-se ver a organização dos protocolos de comunicação no micronúcleo do Amoeba. O micronúcleo contém duas camadas: uma camada superior que implementa RPC e comunicação em grupo, e uma camada inferior que implementa o FLIP. Uma interação entre a camada RPC e a camada FLIP, por exemplo, acontece quando um cliente faz uma chamada *trans* (primitiva RPC que envia uma mensagem de um cliente a

um servidor e espera por uma resposta). Neste caso, a camada RPC examina o cabeçalho e o *buffer* da mensagem enviada pelo *trans*, constrói uma outra mensagem a partir destes campos, e passa esta mensagem para a camada de baixo (camada FLIP), que se encarrega de transmiti-la para o destino.

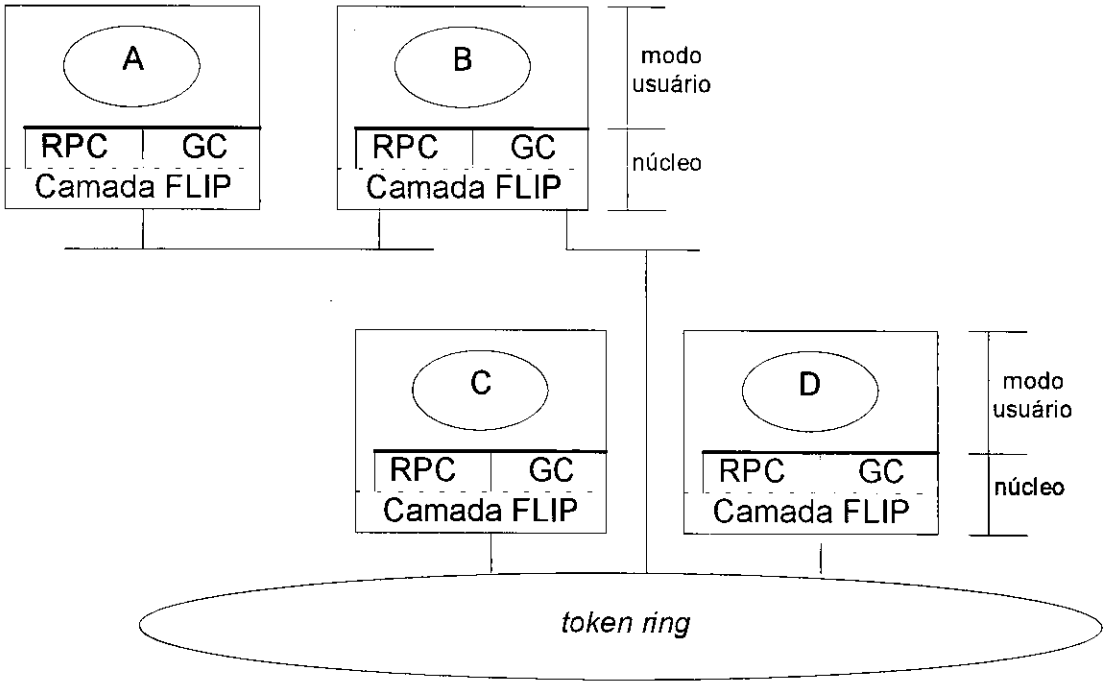


Figura 3.3: Estrutura de Comunicação no Amoeba

3.6.1. COMUNICAÇÃO EM GRUPO

Um grupo consiste de um ou mais processos, denominados membros, tipicamente executando em diferentes processadores e cooperando a fim de fornecer algum serviço. Processos podem ser membros de mais de um grupo. Grupos são fechados, o que significa que somente membros do grupo podem enviar mensagens para o grupo. Processos que não são membros e desejam se comunicar com o grupo podem unir-se ao grupo ou usarem RPC para se comunicar com um dos membros do grupo; este último, por sua vez, pode repassar a requisição para os outros membros do grupo.

Um grupo é identificado por uma porta. Todas as primitivas de comunicação em grupo devem apresentar esta porta. Um grupo é explicitamente criado por uma chamada a `grp_create`. O processo que chama a primitiva é o primeiro membro do grupo. Este processo se torna o sequenciador para o grupo, garantindo uma ordenação total das

mensagens e mantendo um histórico delas para uso no caso de falhas. Outros processos podem tornar-se membros do grupo fazendo uma chamada à primitiva `grp_join`, fornecendo como parâmetro a porta com a qual o grupo foi criado. Existem algumas maneiras através das quais o criador do grupo pode passar a porta para outros processos de modo que estes também possam vir a fazer parte do grupo. Uma delas é publicar a porta no servidor de diretório de modo que os processos possam achá-la. O conhecimento da porta é suficiente para se permitir a entrada de um processo no grupo.

Uma vez que um processo tenha se tornado membro de um grupo, ele pode fazer chamadas às primitivas `grp_send`, `grp_receive`, `grp_info`, `grp_forward` e `grp_leave`. Quando um membro do grupo deseja difundir uma mensagem, ele chama `grp_send`. Esta primitiva garante a entrega da mensagem para todos os membros do grupo (incluindo quem chamou a primitiva), mesmo na presença de canais de comunicação não-confiáveis. Para receber uma mensagem que foi difundida, um membro de um grupo deve chamar `grp_receive`. Se a mensagem difundida chega e nenhuma primitiva está pronta para recebê-la, a mensagem é armazenada em uma área temporária (*buffer*). Quando o membro finalmente fizer um `grp_receive`, este pegará a próxima mensagem na seqüência. `grp_info` permite que um membro obtenha informações a respeito do estado do grupo. As informações obtidas a partir da chamada a esta primitiva são: o número de membros do grupo, a identificação do membro que chamou a própria primitiva `grp_info` e o número seqüencial da próxima mensagem esperada. `grp_forward` integra RPC com comunicação em grupo. Esta primitiva permite que um membro de um grupo repasse um mensagem que foi recebida com um `getreq` para outro membro do grupo. Finalmente, `grp_leave` permite que um membro deixe o grupo; quando o último processo sai do grupo, o grupo deixa de existir.

Todas as primitivas, exceto `grp_info` são bloqueantes. Uma maneira usual de se programar utilizando primitivas de comunicação em grupo é dedicar uma *thread* para fazer as chamadas `grp_receive` e outra para fazer o trabalho propriamente dito.

3.6.2. RPC

O Amoeba oferece três primitivas RPC para a comunicação entre processos:

`getreq`, `putrep` e `trans`. Uma *thread* cliente solicita uma transação (um pedido seguido de uma resposta) chamando a primitiva `trans`. Uma *thread* servidora recebe o pedido chamando `getreq` e retorna a resposta chamando `putrep`. Todas estas primitivas são bloqueantes, isto é, uma chamada a `trans` suspende a *thread* até que o pedido tenha sido enviado, processado e a resposta tenha sido recebida; `getreq` suspende a *thread* até que um pedido tenha sido recebido e `putrep` suspende a *thread* até que a resposta tenha sido recebida pelo núcleo da máquina na qual o cliente executa.

Quando o cliente chama a primitiva `trans`, antes que a mensagem de pedido seja enviada, é necessário saber em qual endereço FLIP existe um servidor associado à porta RPC desejada. Assim, a primitiva `trans` difunde uma mensagem de LOCATE através da rede. Esta mensagem contém, entre outros parâmetros, a porta RPC com a qual o cliente deseja se comunicar. O cliente então bloqueia esperando pela resposta do servidor.

Depois de receber uma mensagem de LOCATE, todo micronúcleo da rede verifica se existe um servidor "escutando" na porta informada. Se existe tal servidor, o micronúcleo então envia uma mensagem HEREIS de volta para o cliente.

3.6.3. FLIP

O Amoeba usa um protocolo proprietário denominado FLIP - *Fast Local Internet Protocol*, para a troca de mensagens. Este protocolo suporta tanto RPC quanto comunicação em grupo, e está abaixo destes na hierarquia dos protocolos. Comparado com o modelo OSI, o FLIP funciona como um protocolo da camada de rede, enquanto que RPC e comunicação em grupo são equivalentes aos protocolos da camada de seção. No entanto, no Amoeba não encontramos o equivalente à camada de transporte; o trabalho de re-transmitir mensagens perdidas é feito no nível de RPC e comunicação em grupo. Conceitualmente, o FLIP pode ser substituído por outro protocolo de rede, como por exemplo o IP, embora com isto se perca um pouco da transparência do Amoeba. Embora o FLIP tenha sido projetado no contexto do Amoeba, ele também pode ser usado em outros sistemas operacionais.

Como um protocolo da camada de rede, o FLIP também fragmenta pacotes. Isto é feito quando o tamanho da mensagem a ser transmitida usando FLIP é maior que o

tamanho máximo definido para um pacote da rede em questão (por exemplo, o tamanho máximo de um pacote Ethernet é 1526 bytes [Soares et al. 95]). Fica a cargo dos programas de nível superior (por exemplo: RPC e comunicação em grupo) restaurar a mensagem original remontando os fragmentos.

Para se ter uma idéia de como o FLIP funciona, vamos considerar um exemplo usando a configuração da Figura 3.3. O processo A é um cliente e o processo B é um servidor. Quando B é criado, o núcleo solicita um novo endereço FLIP para B e o registra. Depois de sua inicialização, B faz uma chamada a `getreq` passando sua porta privada. A camada RPC então procura pela porta pública de B. Se a porta pública de B não estiver na *cache*, então a camada RPC computa esta porta pública através da primitiva `priv2pub` e coloca o resultado na *cache*. Quando encontrar a porta pública de B, a camada RPC coloca uma informação na tabela de portas, indicando que o processo que está aceitando requisições naquela porta, é o processo B. A primitiva `getreq` então fica bloqueada até que um pedido chegue.

Posteriormente, o processo A faz uma chamada a `trans` passando a porta pública de B. A camada RPC (da máquina onde B executa) procura em suas tabelas para ver se ela conhece o endereço FLIP do processo servidor que está aceitando requisições na porta fornecida. Se não encontrar, a camada RPC envia um pacote de *broadcast* para localizá-lo. Este pacote contém um *hop count* máximo para garantir que o *broadcast* seja confinado dentro da rede local (quando um *gateway* encontrar um pacote cujo *hop count* já é igual, ou é maior ao seu máximo *hop count*, o pacote é descartado ao invés de ser repassado). Se o *broadcast* falhar, o prazo de localização do endereço FLIP (da camada RPC que enviou o *broadcast*) expira e a camada envia um novo *broadcast*, desta vez com o *hop count* máximo incrementado de uma unidade; e isto é feito continuamente até que o servidor seja localizado.

Quando um pacote de *broadcast* chega na máquina onde B executa, a camada RPC daquela máquina envia uma resposta de volta anunciando seu endereço FLIP. Como acontece com todos os pacotes que chegam, a camada FLIP cria uma entrada para aquele endereço FLIP antes de repassar o pacote para a camada RPC. A camada RPC então cria

uma entrada em sua própria tabela mapeando a porta pública em endereço FLIP. Em seguida envia o pedido para o servidor. Uma vez que a camada FLIP agora tem uma entrada para o endereço FLIP do servidor, ela agora pode construir um pacote contendo o endereço de rede e enviá-lo sem nenhum trabalho adicional. Pedidos subsequentes para a porta pública do servidor utilizarão a *cache* da camada RPC para encontrar o endereço FLIP e utilizarão a tabela de roteamento da camada FLIP para encontrar o endereço de rede. Dessa forma, a difusão é usada somente a primeira vez que o servidor é contatado. Depois desta vez, as tabelas do núcleo provêem a informação necessária.

3.7. PRINCIPAIS SERVIDORES DO AMOEBA

Nos sistemas baseados na tecnologia de micronúcleo, toda a funcionalidade do sistema que não está embutida no micronúcleo fica a cargo de processos servidores que executam no modo usuário. A idéia desta forma de projeto é minimizar o tamanho do núcleo a fim de aumentar a flexibilidade do sistema. Devido ao sistema de arquivos e outros serviços padrões não estarem dentro do núcleo, eles podem ser facilmente mudados, e ainda múltiplas versões podem executar simultaneamente, cada qual atendendo aos diferentes requisitos dos usuários.

O Amoeba disponibiliza servidores para oferecer diversos serviços. Entre estes servidores encontramos: servidor de arquivos, servidor de diretório, servidor de balanceamento de carga, servidor de replicação, e outros. As próximas seções descrevem o funcionamento destes servidores.

3.7.1 SERVIDOR DE ARQUIVOS

O servidor de arquivos do Amoeba (*bullet server*) foi projetado para ser um servidor que apresenta um alto desempenho. Ao invés de armazenar arquivos como uma coleção de blocos de disco de tamanho fixo, o servidor de arquivos armazena os arquivos de forma contínua, tanto no disco quanto na memória principal do servidor (o Amoeba não possui o conceito de memória virtual).

Os arquivos são armazenados de forma imutável neste servidor. Este só pode ler ou criar, mas não modificar arquivos. Quando um processo emite um pedido de leitura de

um arquivo, o servidor deve transferir o arquivo completo para o cliente em uma única RPC, a menos que o arquivo seja muito grande onde múltiplas RPCs serão necessárias. O cliente faz as modificações localmente e quando termina salva as modificações no arquivo criando um novo arquivo. O servidor de arquivos não fornece nenhum serviço de nomes, assim, para ter acesso a um arquivo um processo deve fornecer a *capability* que é provida pelo servidor de diretórios.

3.7.2. SERVIDOR DE DIRETÓRIO

O servidor de arquivos, como visto na seção anterior, apenas trata da armazenagem de arquivos. O nome dos arquivos e de outros objetos é manipulado pelo servidor de diretório. Sua função primária é prover um mapeamento entre nomes representados em ASCII e *capabilities*.

Um entrada no diretório pode conter uma única *capability* ou um conjunto delas, a fim de permitir que um nome de arquivo seja mapeado num conjunto de arquivos replicados. Quando o usuário localiza um nome no diretório, o conjunto completo de *capabilities* é retornado, a fim de prover uma alta disponibilidade. Estas réplicas podem estar em diferentes servidores de arquivos, potencialmente distantes uns dos outros (o servidor de diretório (*directory server*) não sabe de que tipo de objeto são as *capabilities* que ele detém, ou onde elas estão localizadas).

3.7.3. SERVIDOR DE BALANCEAMENTO DE CARGA

O servidor de balanceamento de carga (*Run Server)* é o responsável pela escolha de um processador, que esteja com a menor carga de processamento no momento, para executar todo processo no Amoeba.

Para escolher a CPU, o *Run Server* verifica que arquiteturas dispõe para executar o programa. Isto é feito verificando o diretório onde localizam-se os vários *pools de processadores,* denominado **pooldir** (Figura 3.4).

Figura 3.4: Diretório de *pools de processadores*

A seleção de qual processador usar é feita da seguinte forma. Em primeiro lugar, faz-se a interseção dos descritores de processos enviados pelo *shell* e das arquiteturas disponíveis no *pooldir*. Por exemplo, se existem descritores de processos para as arquiteturas 386, SPARC e 68000, e no *pooldir* encontram-se *pools de processadores* de arquiteturas 386, SPARC e VAX, somente as arquiteturas 386 e SPARC serão consideradas.

Em seguida, o *Run Server* verifica quais das máquinas consideradas têm memória suficiente para executar o programa, eliminando aquelas que não satisfazem as exigências do processo. O *Run Server* mantém-se informado da memória e da CPU usada por cada um de seus processadores fazendo, regularmente, uma chamada *getload* a cada um, requisitando estes valores. Assim os números na tabela do *Run Server* estão continuamente atualizados.

Por último, para cada uma das máquinas restantes, faz-se uma estimativa do poder computacional que está disponível para executar o novo processo. Cada CPU faz sua própria estimativa. Para esta estimativa leva-se em consideração o poder computacional da CPU e o número de *threads* ativas rodando nela. Por exemplo, se uma máquina de 20 MIPS tem quatro *threads* ativas no momento, a soma de mais uma *thread* significará que cada uma, incluindo a nova, terá disponível 4 MIPS em média. Se outro processador tem 10 MIPS e uma *thread* ativa, nesta máquina o novo processo poderá ter até 5 MIPS disponíveis. O *Run Server* escolhe então o processador que pode liberar a maior quantidade de MIPS e retorna para o *shell* a *capability* para este processador.

3.7.4. SERVIDOR DE INICIALIZAÇÃO

O servidor de inicialização (*boot server*) é usado para verificar se todos os servidores configurados para executar, de fato, estão executando, tomando as devidas ações corretivas quando isto não for verdade. Um servidor que desejar fazer uso deste serviço deve ser incluído no arquivo de configuração do servidor de inicialização. Cada entrada informa de quanto em quanto tempo será feita a verificação. Se o servidor inspecionado não responder no tempo pré-estabelecido, o servidor de inicialização assume que ele falhou e toma as devidas providências a fim de reinicializá-lo. Desta maneira,

serviços críticos podem ser re-inicializados automaticamente se por ventura vierem a falhar. O próprio servidor de inicialização pode ser replicado, para se proteger de uma eventual falha dele mesmo.

3.7.5. SERVIDOR DE TCP/IP

Embora internamente o Amoeba use o protocolo FLIP a fim de atingir um alto desempenho, algumas vezes se faz necessário que este utiliza o protocolo TCP/IP, por exemplo, para se comunicar com terminais X, para enviar e receber *e-mails* de outras máquinas que não sejam máquinas Amoeba e interagir com outros sistemas Amoeba via uma internet. Para permitir que o Amoeba realize estas atividades, um servidor de TCP/IP (*tcp/ip server*) foi desenvolvido.

Para estabelecer uma conexão, um processo Amoeba faz uma RPC ao servidor de TCP/IP enviando a este um endereço TCP/IP. O cliente então permanece bloqueado até que a conexão tenha sido estabelecida ou recusada. RPCs subseqüentes podem enviar e receber pacotes da máquina remota sem que o processo Amoeba tenha que saber que o TCP/IP está sendo utilizado. Este mecanismo de comunicação é menos eficiente que o FLIP, assim ele só é usado quando não é possível se usar o FLIP.

3.7.6. OUTROS SERVIDORES

Outros serviços têm sido propostos utilizando o sistema operacional Amoeba como base. Verstoep e Sharp criaram um serviço de reserva de processadores do sistema, que permite a um usuário o uso exclusivo de um subconjunto dos processadores disponíveis. Este serviço é útil, por exemplo, quando o usuário deseja medir o desempenho de sua aplicação distribuída [Verstoep-Sharp 94].

Em [Vasconcelos 97], é proposto um serviço de replicação de componentes de software utilizando as técnicas de replicação ativa, semi-ativa e passiva. Nesse artigo, Vasconcelos sugere também a criação de um servidor de diagnóstico, que verifica se a execução do componente replicado falhou. Se o resultado da verificação for positivo, um servidor de reconfiguração do sistema é acionado.

Capítulo 4

Um Serviço de Processamento Confiável para o Seljuk-Amoeba

O principal objetivo do ambiente operacional Seljuk-Amoeba é facilitar o desenvolvimento e a execução de aplicações distribuídas com diferentes requisitos de confiança no funcionamento. A execução de uma aplicação distribuída robusta no ambiente operacional Seljuk-Amoeba utiliza basicamente três serviços distintos: i) o serviço de processamento, que é o responsável pela execução dos componentes da aplicação em nodos replicados; ii) o serviço de comunicação, que provê os mecanismos necessários para que dois ou mais processos possam trocar mensagens de forma confiável; e iii) o serviço de gerência da redundância, que implementa os protocolos que garantem os requisitos de confiança no funcionamento requeridos pela aplicação.

Uma das características mais importantes do Seljuk-Amoeba é a sua flexibilidade, que possibilita a coexistência de aplicações distribuídas com diferentes requisitos de confiança no funcionamento, num mesmo sistema distribuído. Além disso, o ambiente também permite que os requisitos de confiança no funcionamento das aplicações e a semântica de falha dos componentes que formam a infra-estrutura de execução, sejam definidos no momento da ativação das mesmas. Isto possibilita, entre outras coisas, o balanceamento entre as medidas de desempenho e de confiança no funcionamento que se deseja alcançar para as aplicações, em tempo de ativação, e sem a necessidade de qualquer mudança no código executável destas.

O objetivo deste capítulo é discutir questões relacionadas à construção de diversos tipos de nodos com semântica de falha controlada, mostrando como estes nodos podem ser utilizados no projeto de um serviço de processamento confiável para o ambiente operacional distribuído Seljuk-Amoeba.

4.1. TIPOS DE NODOS COM SEMÂNTICA DE FALHA CONTROLADA

Podemos destacar duas classes representativas de nodos que apresentam uma semântica de falha controlada: nodos com semântica de falha segura e nodos com semântica de falha mascarada, como mostra a Figura 4.1.

Os nodos da primeira classe são incluem aqueles cuja semântica é considerada segura, isto é, após a detecção de uma falha em qualquer componente do nodo, este não libera nenhuma saída, ele simplesmente pára; enquanto que a característica principal dos nodos da segunda classe é que estes ainda provêem o seu serviço padrão mesmo na ocorrência de um número limitado de falhas de seus componentes replicados, que são mascaradas.

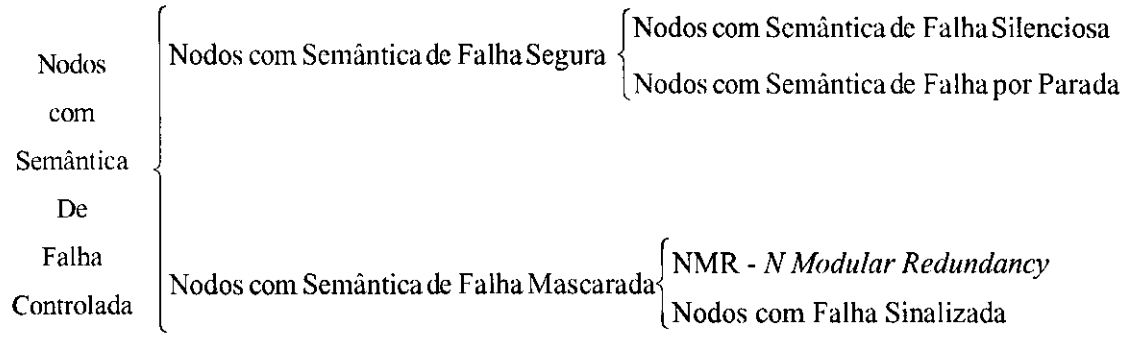


Figura 4.1: Tipos de Nodos com Semântica de Falha Controlada

Os nodos com semântica de falha silenciosa [Bernstein 88, Shrivastava et al. 92, Webber-Beirne 91, Brasileiro et al. 92] e os nodos com semântica de falha por parada [Schlichting-Schneider 84, Schneider 84] são os principais representantes da classe de nodos com semântica de falha segura. Nos nodos com semântica de falha silenciosa, uma vez que a falha tenha sido detectada, o nodo como um todo pára, em vez de executar qualquer operação errônea que pode ser visível fora do nodo. Já os nodos com semântica de falha por parada, além de pararem quando uma falha é detectada, possuem duas outras

propriedades que os diferenciam dos nodos com semântica de falha silenciosa, e os tornam mais difíceis de serem implementados. São elas: i) qualquer nodo no sistema deve ser capaz de detectar que um outro nodo com semântica de falha por parada parou de funcionar e; ii) cada nodo tem uma memória estável associada a ele cujo estado não é afetado pela falha do nodo, e continua acessível aos outros nodos do sistema, mesmo depois do nodo ter parado de funcionar.

Os nodos com semântica de falha mascarada são normalmente construídos através da replicação do processamento em um número de processadores que falham de forma independente, e da execução de uma função de validação sobre as saídas desses processadores, com o intuito de mascarar as falhas de parte dos processadores.

Um nodo NMR - *N Modular Redundancy* é formado por N processadores que falham de maneira distinta e que têm suas saídas validadas por componentes votadores. Tal nodo é capaz de mascarar π faltas de processadores do nodo, desde que $N \geq 2\pi+1$. Uma forma particular na qual a técnica NMR se apresenta, é o TMR - *Triple Modular Redundancy*, onde a unidade de processamento é triplicada. Esta é umas das mais bem conhecidas técnicas de tolerância a faltas, e tem sido utilizada em muitos sistemas.

Um nodo com semântica de falha sinalizada [Shrivastava et al. 91] é outro exemplo de nodo com semântica de falha mascarada. O funcionamento deste nodo na presença de falhas é similar ao NMR, exceto pelo fato de que assim que uma falha é detectada, este nodo é capaz de sinalizar a ocorrência dela. Esta característica é muito útil para computações críticas, pois com isto essas computações podem ser migradas para outros nodos que, no momento, possam ser considerados mais confiáveis do que este que acabou de sofrer uma falha em um componente.

4.2. FAMÍLIA VOLTAN DE NODOS REPLICADOS

A família Voltan de nodos replicados [Shrivastava et al. 92] segue a abordagem na qual o projeto SIFT [Wensley et. Al 78] foi pioneiro, ou seja, os mecanismos necessários para a gerência da redundância requerida para a obtenção de tolerância a faltas são implementados em software. Como discutido no Capítulo 2, com a implementação em

software tenta-se evitar a complexidade envolvida no projeto de circuitos especiais necessários para a abordagem baseada em hardware. Além disto, os nodos Voltan não são utilizados somente para prover tolerância a faltas para aplicações específicas. Uma vez que o objetivo do ambiente operacional Seljuk-Amoeba é prover tolerância a faltas para diferentes tipos de aplicações, através da replicação do processamento, utilizando a abordagem baseada em software, o modelo Voltan de nodos replicados foi o escolhido para o projeto do serviço de processamento do Seljuk-Amoeba.

Nos nodos Voltan, assume-se que as aplicações distribuídas não-replicadas são compostas de um número de processos que não compartilham memória; a interação é feita somente por troca de mensagens. Outra suposição feita é que os processos replicados comportam-se de maneira determinística [Brasileiro 97]. Uma computação é dita determinística se as saídas geradas por esta computação dependem unicamente das entradas e de seu estado inicial.

Assume-se também que as mensagens trocadas entre os processos são assinadas, de forma que nenhum processador incorreto possa falsificar mensagens e que o meio de comunicação, no qual as mensagens trafegam, é síncrono.

Além das considerações supracitadas, para implementar um nodo com semântica de falha controlada é necessário garantir que todas as réplicas recebam e processem a mesma sequência de pedidos. Para se alcançar isto é necessário que o nodo apresente as duas propriedades enumeradas abaixo:

- **Acordo**: Toda réplica executando em um processador correto recebe todos os pedidos.
- **Ordem**: Todas as réplicas executando em processadores corretos processam os pedidos na mesma ordem.

A computação é replicada em N processadores dos quais π podem falhar de forma arbitrária; onde $N=\pi+1$ para nodos com semântica de falha silenciosa e $N=2\pi+1$ para nodos com semântica de falha mascarada. A Figura 4.2 apresenta uma visão geral da arquitetura dos nodos Voltan. Além dos processos da aplicação (**Servidor**), cada

processador correto de um nodo executa cinco outros processos, denominados **Receptor, Transmissor, Ordenador, Validador** e **Remetente**.

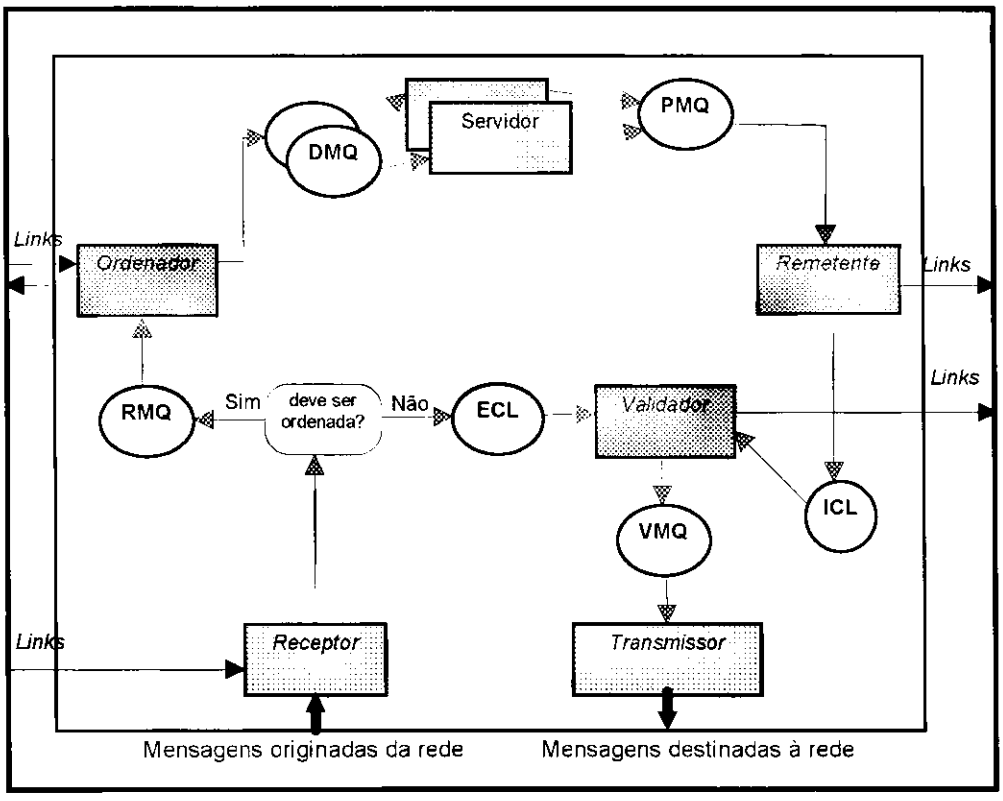


Figura 4.2: Visão de um Processador Correto Executando Processos em um Nodo Voltan

A função de cada processo é descrita a seguir.

Remetente: este processo recebe as mensagens produzidas pelo **servidor** daquele processador, assina-as e as envia para os outros processadores do nodo para que possam ser validadas.

Ordenador: executa um protocolo de ordenação em conjunto com os processos ordenadores dos outros processadores do nodo. Sua função é construir filas de mensagens para serem tratadas pelos servidores de todos os processadores corretos do nodo; essas filas devem conter as mesmas mensagens e na mesma ordem.

Validador: a função deste processo depende do tipo do nodo. No caso de nodos com semântica de falha mascarada, o processo de validação é feito através de votação. Faz-se uma comparação das mensagens assinadas e enviadas por outros processadores

com as que foram geradas localmente. Se a comparação falha, a mensagem é descartada; caso contrário, a mensagem é contra-assinada. Se na mensagem existirem π+1 assinaturas válidas, ela é considerada uma mensagem válida. Em seguida, esta é repassada ao processo Transmissor para entrega à aplicação. Se existirem menos que π+1 assinaturas na mensagem, ela é enviada para os outros processadores do nodo que ainda não a assinaram. Já no caso de nodos com semântica de falha segura, o processo Validador comporta-se como um comparador. O processo de comparação é semelhante à votação explicada anteriormente, só com a diferença de que ao se detectar uma falha, ao invés de simplesmente descartar a mensagem recebida, este Validador pára por completo, como também o processo Remetente. Isso garante que o nodo não mais irá produzir mensagens.

Transmissor: este é o processo responsável por enviar as mensagens com π+1 assinaturas para a aplicação destino.

Receptor: este processo autentica as mensagens recebidas de outros processadores do nodo ou de outros nodos, descartando qualquer mensagem cuja autenticação falhe, ou que seja duplicada. Mensagens autenticadas originárias da rede de comunicação são enviadas para o processo Ordenador local; mensagens autênticas vindas de outro processador do nodo que tenham menos que π+1 assinaturas, são enviadas para o processo Validador local.

A comunicação entre dois processos executando no mesmo processador é realizada através de listas e filas de mensagens. A diferença básica entre listas e filas é que na lista os processos podem retirar mensagens de qualquer posição, enquanto que na fila os processos só podem ter acesso às mensagens que estejam no topo da mesma. Na Figura 4.3 são mostradas as seguintes filas e listas:

RMQ - Fila de Mensagens Recebidas: contém mensagens válidas originadas da rede e autenticadas pelo processo Receptor, prontas para serem ordenadas.

DMQ - Fila de Mensagens Entregues: contém mensagens ordenadas prontas para serem processadas pelo processo Servidor.

PMQ - Fila de Mensagens Processadas: contém mensagens ainda não assinadas

produzidas pelo processo Servidor local. Estas têm que ser validadas pelo processo Validador, antes de sua transmissão ao seu destino.

ECL - Lista Externa de Mensagens Candidatas: contém mensagens assinadas e autenticadas que tenham sido recebidas de outros processadores para validação.

ICL - Lista Interna de Mensagens Candidatas: contém mensagens não assinadas, a espera de mensagens na lista ECL que coincidam com elas.

VMQ - Fila de Mensagens Válidas: contém mensagens com $\pi+1$ assinaturas (válidas) prontas para serem transmitidas através da rede.

4.3. CONSTRUINDO NODOS COM SEMÂNTICA DE FALHA CONTROLADA

Os nodos apresentados na seção anterior toleram falhas arbitrárias nos seus componentes. Se a semântica de falha do componente for mais restritiva, algumas simplificações são possíveis. Basicamente, a construção de um nodo replicado depende da semântica de falha dos processadores sobre os quais o nodo será construído; do comportamento que se espera que o nodo apresente na presença de falhas; do número de falhas em componentes que se espera que o nodo tolere (π); e da técnica de replicação a ser empregada. Dependendo destes fatores, o grau de replicação do nodo e a complexidade dos protocolos serão diferentes. O grau de replicação é definido como o número mínimo de processadores distintos no qual a computação deve ser replicada.

Nodos com semântica de falha silenciosa podem ser construídos com processadores que possuem as mais diversas semânticas de falha. Obviamente, se é possível assumir que os processadores falham por parada, ou simplesmente não falham, não há necessidade de replicar o processamento. No caso dos processadores falharem por valor, necessitamos de $\pi + 1$ processadores, uma vez que é necessário apenas que um dos processadores esteja correto para que falhas de até π processadores possam ser detectadas e o nodo não libere nenhuma saída para a aplicação.

Nodos com semântica de falha mascarada construídos com componentes que

falham por parada precisam ser compostos de $\pi + 1$ processadores. Com isto garantimos que mesmo que π processadores venham a falhar, pelo menos um continuará executando a computação. Além disso, como a semântica de falha dos processadores é de falha por parada, esta única saída gerada não precisa ser validada já que se o mesmo não parou é porque está correto.

4.4. PROJETO DE UM SERVIÇO DE PROCESSAMENTO CONFIÁVEL

Para se propor um serviço de processamento confiável para o Seljuk-Amoeba é necessário que se entenda como funciona o serviço de processamento do Amoeba. Nesta seção detalharemos como acontece a criação de um processo no Amoeba e em seguida apresentaremos uma proposta de como este serviço pode ser aprimorado para prover confiabilidade.

4.4.1. SERVIÇO DE PROCESSAMENTO NÃO-REPLICADO

O principal servidor existente no serviço de processamento do Amoeba é o servidor de balanceamento de carga (*Run Server*). O *Run Server* é o responsável pelo escalonamento de tarefas no Amoeba. Para fazer o balanceamento da carga dos processadores existentes, o *Run Server* leva em consideração fatores como: memória disponível, velocidade do processador, arquitetura do hardware, fragmentação de memória, entre outros.

O *Run Server* faz o balanceamento da carga utilizando os *pools de processadores* existentes. Um *pool de processadores* é formado por um conjunto de processadores interligados através de uma rede e constitui todo o poder computacional do Amoeba. Cada processador do *pool* tem sua própria memória local e executa um micronúcleo do Amoeba.

Antes de executar um processo, o *Run Server* tem que decidir em qual tipo de arquitetura o processo deve executar e qual processador deve ser o escolhido. No Amoeba, as tarefas são submetidas ao *Run Server* através de um interpretador de comandos (*shell*). Quando um comando é invocado a partir do *shell*, este extrai a primeira palavra da linha de comando, assume que é o nome de um programa executável, procura

por este programa no sistema de arquivos, e se o encontrar, faz com que este seja executado.

Antes de solicitar a execução do programa, o *shell* localiza as arquiteturas para as quais este programa está disponível. Para localizar as arquiteturas disponíveis, o *shell* procura o comando no diretório /bin (diretório onde encontram-se os executáveis no Amoeba). Se o programa estiver disponível para várias arquiteturas, este não será um arquivo, e sim um diretório contendo os executáveis para cada arquitetura disponível. Em seguida o *shell* faz uma RPC ao *Run Server,* enviando a este os descritores de processos para todas as arquiteturas disponíveis do programa em questão, e solicitando ao *Run Server* a escolha de uma arquitetura e uma CPU específica para executar o programa.

De posse da *capability* para o processador, o *shell* então faz uma RPC ao servidor de processos do processador escolhido, enviando a *capability* fornecida pelo *Run Server.* Junto com a *capability,* o *shell* envia, ao servidor de processos, o descritor de processo da arquitetura escolhida pelo *Run Server* para que o processo possa ser criado. A Figura 4.3 mostra todos os passos envolvidos na criação de um processo no Amoeba.

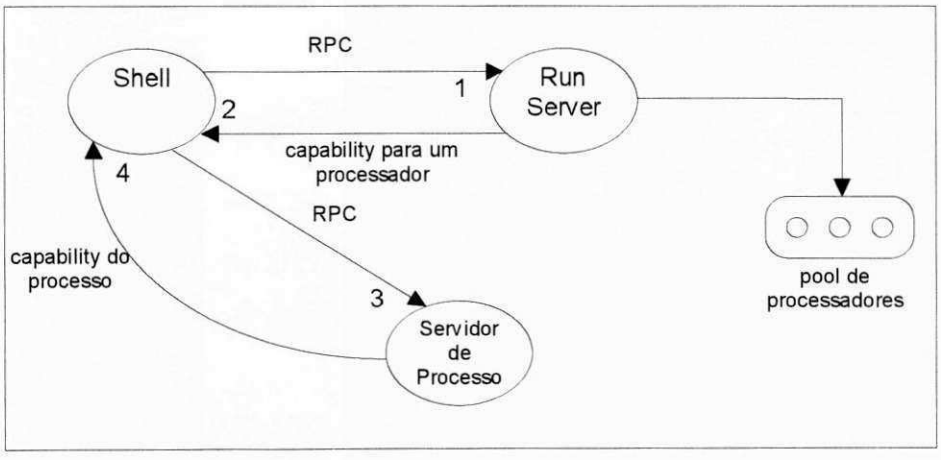


Figura 4.3: Fases da Criação de um Processo no Amoeba.

4.4.2. SERVIÇO DE PROCESSAMENTO CONFIÁVEL

No projeto do serviço de processamento confiável do Seljuk-Amoeba, verificamos a necessidade de aprimorar serviços existentes e de introduzir alguns serviços adicionais, como por exemplo, um serviço de gerência de nodos replicados. Os serviços existentes

foram modificados de forma que as mudanças não afetassem os clientes desses serviços. Nenhuma funcionalidade existente foi removida; apenas acrescentamos as novas funcionalidades que se fizeram necessárias.

A Figura 4.4 identifica as entidades presentes no serviço de processamento confiável proposto, mostrando como se apresenta o cenário da criação de um processo no Seljuk-Amoeba depois de se introduzirem mecanismos para tolerância a faltas.

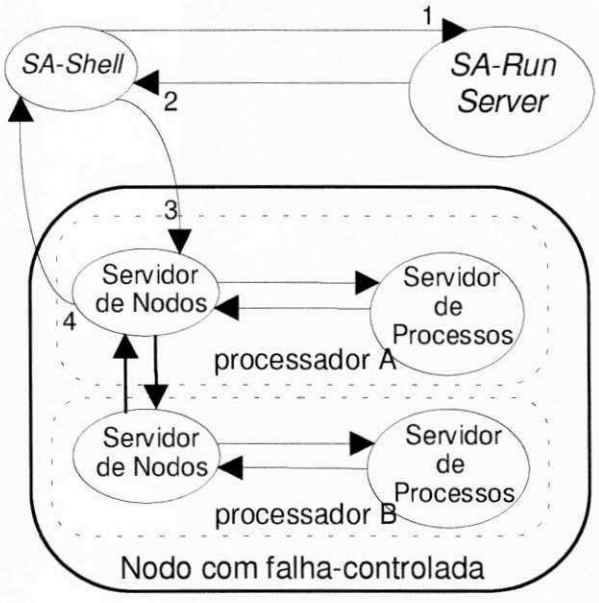


Figura 4.4: Criação de um Processo Replicado no Seljuk-Amoeba

As entidades presentes no serviço de processamento confiável do Seljuk-Amoeba são:

• *SA-Run Server* - executa as mesmas funções realizadas pelo *Run Server* e ainda aquelas responsáveis pela criação dos nodos replicados. Cabe ao *SA-Run Server* selecionar os processadores que formarão o nodo através da avaliação do grau de confiabilidade requerido para o nodo e da semântica de falha assumida para os processadores que irão compor este nodo

• *Servidor de Nodos* - responsável por gerenciar os nodos replicados formados pelo SA-Run Server. O *Servidor de Nodos* é projetado de tal forma que ele consiga prover as mais diversas semânticas de falha para os nodos replicados formados.

SA-Shell - é a interface que a aplicação utiliza para fazer uso do serviço de processamento replicado.

A seguir detalharemos cada uma das entidades supracitadas.

4.4.2.1. SA-SHELL

Para fazer uso do serviço de processamento confiável, aplicações executadas no Seljuk-Amoeba são invocadas a partir do *SA-Shell*. O *SA-Shell* faz uma RPC com o *SA-Run Server* pedindo a este que crie uma unidade de processamento confiável para executar a aplicação, e permanece bloqueado esperando por uma resposta. O *SA-Run Server*, então, retorna ao *SA-Shell* um descritor do nodo criado. Este descritor contém todas as informações necessárias à identificação do nodo no sistema. Algumas destas informações são: um identificador para o nodo, uma lista contendo todos os processadores que fazem parte do nodo, os descritores de processos correspondentes que foram escolhidos para cada processador, e ainda uma *capability* para o nodo. Com essas informações, o *SA-Shell* faz uma RPC com o servidor de processos do primeiro processador nodo, solicitando a criação das réplicas da aplicação a ser executada confiavelmente.

As aplicações executadas pelo *SA-Shell* podem escolher, em tempo de ativação, a semântica de falha do nodo no qual elas irão ser executadas, como também a semântica de falha efetiva dos processadores que formam o nodo. O *SA-Shell* deve informar ao *SA-Run Server* qual a semântica de falha que está assumindo para os processadores que formarão o nodo replicado, e qual será a forma como este nodo irá se comportar.

Quando o *SA-Shell* faz uma RPC com o *SA-Run Server* solicitando a criação de um nodo, ele envia uma série de parâmetros nessa RPC. Um desses parâmetros é uma lista contendo todos os descritores de processos disponíveis para aquela aplicação. Além dos descritores de processos, outros parâmetros extras devem ser enviados nessa RPC. Primeiro, é necessário que o *SA-Shell* informe ao *SA-Run Server* qual a semântica de falha dos processadores do sistema, sobre os quais os nodos com semântica de falha controlada serão construídos.

Depois de selecionar qual a semântica de falha do nodo sobre o qual a aplicação

executará, o *SA-Shell* deve informar qual o grau de resiliência que este nodo terá, isto é, quantos processadores, no máximo, poderão falhar. O *SA-Run Server* usa este parâmetro para decidir quantos processadores formarão o nodo. Esta decisão é tomada de acordo com a semântica de falha escolhida para o nodo e para os processadores que o compõe.

Outro parâmetro passado pelo *SA-Shell* para o *SA-Run Server*, é uma lista contendo a identificação dos processadores que a aplicação não deseja utilizar. Este parâmetro pode ser usado, por exemplo, se o projetista de uma aplicação acha que determinado processador não é confiável o bastante, e não quer submeter nenhuma tarefa para este. Este parâmetro pode ser usado também, para prevenir que um mesmo processador faça parte de mais de um nodo no qual um outro processo de uma mesma aplicação esteja executando.

4.4.2.2. SA-RUN SERVER

O *Run Server,* responsável pelo escalonamento de processos no Amoeba, é um processo que executa no modo usuário, podendo ser facilmente substituído por outro que desempenha uma função similar. Desta forma, é possível introduzir um novo servidor que desempenhe as mesmas funções antes desempenhadas pelo *Run Server*, e ainda aquelas necessárias à formação da estrutura de um nodo - a ser utilizado como base para o processamento replicado. Este servidor é o *SA-Run Server.*

Além de tolerar faltas físicas, os nodos formados pelo *SA-Run Server* estão também aptos a tolerar faltas de concepção. O Amoeba suporta processadores heterogêneos, isto é, permite-se a existência de processadores de arquiteturas diferentes em uma mesma instalação do sistema operacional. Assim sendo, torna-se possível alcançar tolerância a faltas de concepção do hardware, executando um processo em um nodo formado por processadores que sejam de arquiteturas distintas. Existe também, a possibilidade de se tolerar faltas de concepção do programa replicado propriamente dito. Isto é conseguido pela execução de diferentes versões do programa em cada um dos processadores do nodo, no caso de existirem diferentes versões de um mesmo programa disponíveis.

Ao se optar por uma replicação de componentes de software, é necessário levar alguns pontos em consideração, no momento de projetar as diferentes versões. Muitas

vezes existe a necessidade de se ter diferentes equipes de programação para o projeto das versões. Isto se dá devido ao fato que um programador tendo implementado uma versão, muito provavelmente usará algoritmos e estruturas de dados similares em uma outra versão de um mesmo componente de software. Além disto, ele está sujeito a cometer os mesmos erros, particularmente se os erros decorrem de uma incompreensão das especificações. Ter diferentes equipes de programadores, trabalhando totalmente independente uns dos outros, reduz a probabilidade de existir uma dependência entre as falhas que ocorrem em diferentes versões.

4.4.2.3. SERVIDOR DE NODOS

O *Servidor de Nodos* provê a semântica de falha requerida, de uma maneira transparente, fazendo a gerência do processamento replicado e garantindo que uma função de validação adequada seja aplicada às saídas geradas pelos processadores que formam um nodo. Como foi visto anteriormente, a escolha do número e do tipo de processadores a serem usados em um nodo depende dos requisitos de confiabilidade da aplicação, ou seja, do número de faltas que o nodo deve tolerar, e esta escolha é realizada pelo *SA-Run Server*.

O comportamento ideal do nodo do ponto de vista da aplicação é aquele em que o nodo não falha. No Seljuk-Amoeba é possível oferecer um serviço de processamento com essa semântica, mesmo que os processadores do sistema possam falhar. A falha desses componentes será mascarada pelo nodo. Uma outra possibilidade mais econômica em termos de desempenho e recursos consumidos, porém menos ideal, é o comportamento oferecido por um nodo com semântica de falha segura (ex: falha silenciosa). O Seljuk-Amoeba também implementa esse serviço, mesmo que a semântica de falha dos processadores seja menos restritiva que falha por parada.

Nodos mantidos pelo *Servidor de Nodos* utilizam os protocolos da família Voltan de nodos replicados para o controle das réplicas. A Figura 4.5, a seguir, mostra como os processos, filas e listas dos nodos Voltan, como também o fluxo de informação entre eles, podem ser logicamente inseridos dentro do micronúcleo do Seljuk-Amoeba, a fim de implementar o serviço de processamento confiável.

Quando uma mensagem chega em uma adaptador de rede de um processador, seu destino deve ser considerado, a fim de decidir se a mensagem precisa ser ordenada, ou não. Se a mensagem foi enviada para um processo replicado, esta deve ser ordenada, antes de ser repassada para a parte do núcleo que implementa o recebimento de mensagens nos protocolos de mais alto nível (representados na Figura 4.5 por $RPC_{in}/GC_{in}/FLIP_{in}$). Desta forma, todas as réplicas da aplicação irão receber as mesmas mensagens e na mesma ordem. Caso contrário a mensagem deve ser imediatamente repassada para RPC_{in}/GC_{in} $FLIP_{in}$.

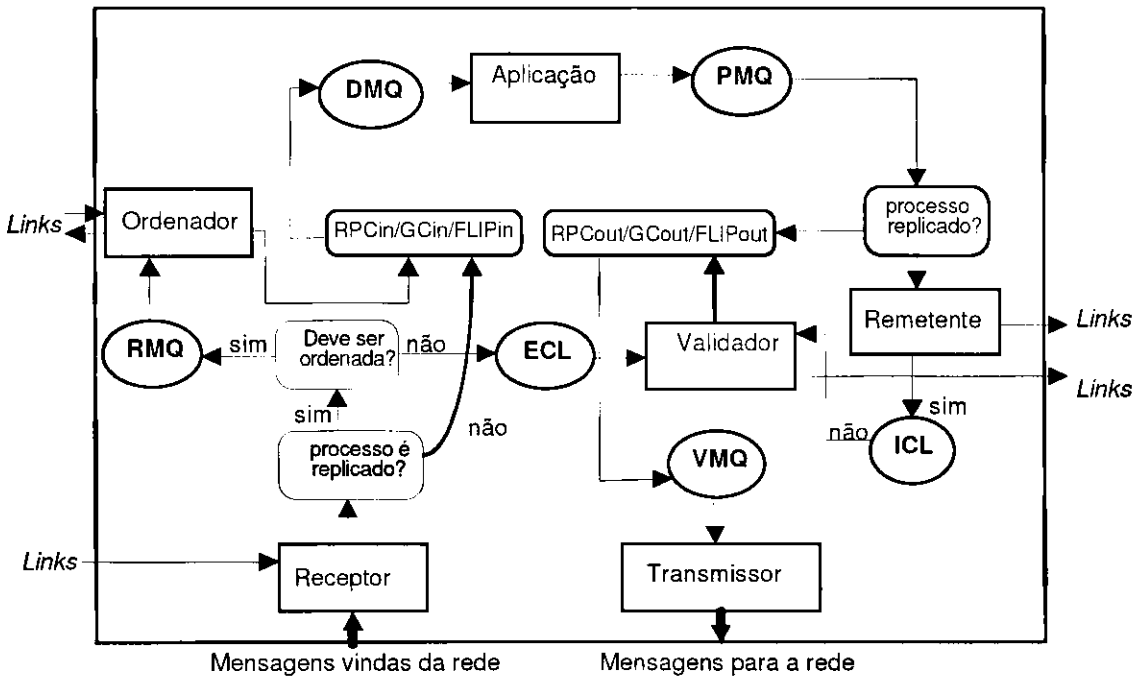


Figura 4.5: Visão de uma Réplica Implementando o Serviço de Processamento Confiável

Outros dois tipos de mensagens podem ser recebidos pelo processador: mensagens enviadas pelas outras *threads* Ordenador (dos outros processadores componentes do nodo); e mensagens enviadas pelas *threads* Validador (dos outros processadores do nodo replicado), que são entregues ao Validador local para validação. As mensagens produzidas pelas aplicações, são depositadas na PMQ. Depois disto, a mensagem é entregue à *thread* Remetente. Se a mensagem foi gerada por um processo replicado, esta é depositada na ICL e permanecerá lá esperando por uma mensagem na ECL que case com ela; além disto uma cópia da mensagem é assinada e enviada para os outros processadores do nodo. Caso contrário, a mensagem é imediatamente entregue através dos protocolos de comunicação

de mais alto nível (representados na Figura 4.5 por $RPC_{out}/GC_{out}/ FLIP_{out}$). Depois de ser processada por $RPC_{out}/GC_{out}/ FLIP_{out}$ a mensagem é enviada a *thread* Transmissor que se encarregará de enviá-la para o seu destino.

Capítulo 5

Aspectos de Implementação

5.1. AMBIENTE DE IMPLEMENTAÇÃO

O sistema operacional distribuído Amoeba pode executar em vários tipos diferentes de computadores. Recomenda-se que ele seja instalado em uma rede que possua, no mínimo, cinco computadores, mesmo que tecnicamente seja possível executá-lo utilizando apenas um computador. Todas as máquinas Amoeba devem estar conectadas por uma rede de comunicação, embora estas máquinas não necessitem estar na mesma rede, uma vez que as redes podem estar interconectadas por *gateways*.

Para obter melhores resultados do sistema, deve-se ter uma máquina separada executando o servidor de arquivos, uma *workstation* por usuário e um grupo de processadores para executar os processos dos usuários e os outros servidores do sistema.

No LSD/DSC/UFPB[6], o Amoeba foi instalado em um conjunto de máquinas compostas de processadores da arquitetura 80386, conectadas através de uma rede Ethernet. O sistema instalado conta com cinco computadores executando a versão 5.3 do Amoeba. Dentre esses computadores, um é dedicado ao servidor de arquivos; os computadores restantes fazem parte do *pool de processadores*. Todos os processadores possuem memória própria associada a eles; não existe memória compartilhada. O computador onde o servidor de arquivos executa possui 16Mbytes enquanto que os

[6] Laboratório de Sistemas Distribuídos do Departamento de Sistemas e Computação da Universidade Federal da Paraíba

computadores restantes possuem 8Mbytes de memória cada um.

Nós optamos por não instalar *workstations;* utilizamos uma máquina unix, rodando Solaris 2.4, como estação de trabalho através do uso do comando *amsh*. Este comando é usado no unix para inicializar um *shell* no Amoeba que usa um terminal unix para sua entrada e saída. O *amsh* utiliza o FLIP *driver*, que deve estar instalado na máquina que executa o *amsh,* para se comunica com o *pool de processadores* do Amoeba.

5.2. PROGRAMANDO APLICAÇÕES CLIENTE/SERVIDOR NO AMOEBA

O modelo de programação do Amoeba é baseado na existência de programas, possivelmente executando em diferentes processadores, cooperando para executar uma determinada tarefa. Isto é inerente à natureza de sistemas distribuídos. As aplicações cliente/servidor são implementadas utilizando o mecanismo de RPC. Outra forma de desenvolver aplicações distribuídas é utilizando comunicação em grupo. Tanto RPC como comunicação em grupo executam sobre o FLIP (como visto no Capítulo 3). O FLIP também pode ser utilizado diretamente pela aplicação. Entretanto, esta não é a maneira mais indicada, devido ao fato das camadas de RPC e comunicação em grupo já fornecerem muitas abstrações interessantes para o programador de aplicações distribuídas. Por exemplo, no nível do FLIP existe um tamanho máximo para as mensagens que trafegam, mensagens maiores devem ser quebradas antes de serem enviadas através do FLIP, e ainda precisam ser remontadas no lado destino; tarefa esta também deixada para a aplicação quando se usa diretamente o FLIP. Dessa forma, é interessante que a aplicação utilize os mecanismos de RPC e comunicação em grupo que escondem estes detalhes do programador.

O mecanismo de comunicação mais utilizado no Amoeba é o RPC. O mecanismo RPC do Amoeba é baseado em quatro primitivas: `getreq`, `putrep`, `trans` e `timeout`. Estas primitivas formam a base da comunicação entre processos. As primitivas `getreq` e `putrep` são usadas pelos servidores. Uma vez que o servidor tenha sido inicializado, ele faz uma chamada à primitiva `getreq` com a porta na qual ele estará aceitando pedidos. Quando um pedido chega, o servidor executa a operação requerida e envia a resposta de volta ao cliente utilizando `putrep`.

Uma *thread* cliente solicita uma transação (um pedido seguido de uma resposta) chamando a primitiva `trans`. Todas estas primitivas são bloqueantes, isto é, uma chamada a `trans` suspende a *thread* até que o pedido tenha sido enviado, processado e a resposta tenha sido recebida; `getreq` suspende a *thread* até que um pedido tenha sido recebido e `putrep` suspende a *thread* até que a resposta tenha sido recebida pelo núcleo da máquina na qual o cliente executa.

Ao ser inicializado, cada servidor escolhe randomicamente - através de uma chamada à primitiva `uniqport` - a porta na qual este estará recebendo pedidos. Uma vez de posse desta porta, o servidor criptografa a porta escolhida fazendo uma chamada à primitiva `priv2pub`, obtendo uma porta criptografada como resultado. Esta porta criptografada é publicada em um lugar previamente conhecido pelo lado cliente da aplicação, através de uma chamada à primitiva `name_append`. Esta primitiva recebe como parâmetro a porta a ser publicada e o local (nome do arquivo) no qual ela será publicada. Sempre que um cliente desejar enviar um pedido ao servidor, ele deverá utilizar esta porta publicada como identificador.

Depois de publicar a porta criptografada, o servidor faz uma chamada à primitiva `getreq`. A porta passada como parâmetro para esta primitiva é a porta real na qual o servidor está aceitando pedidos; esta porta é chamada de porta privada. Com este esquema de portas públicas e privadas, não é possível que um servidor possa receber pedidos em uma porta que outro servidor está utilizando, já que as portas privadas são de conhecimento exclusivo dos próprios servidores, e além disso a porta pública também não pode ser deduzida a partir da porta privada.

Quando um servidor faz uma chamada a `getreq`, a porta pública correspondente é computada pelo núcleo e armazenada numa tabela de portas em uso no momento. A primitiva `trans` utiliza portas públicas, assim quando um pacote chega em uma máquina, o núcleo compara a porta pública contida no cabeçalho do pacote com as portas públicas em sua tabela para ver se estas casam. Uma vez que as portas privadas nunca trafegam na rede, o esquema é seguro. Detalhes deste esquema podem ser vistos em [Tanenbaum et al. 86].

As Figuras 5.1 e 5.2 mostram, respectivamente, trechos de código do lado servidor e do lado cliente de uma aplicação Amoeba que utiliza o mecanismo de RPC. É necessário estar ciente que os trechos de códigos apresentados não representam a implementação de uma aplicação Amoeba real, já que apenas os detalhes referentes à comunicação cliente/servidor são mostrados. Em particular, o serviço específico que o servidor oferece não é relevante neste ponto.

```
#define SERVER_CAP   "/super/cap/default"
#define REQ_BUFSZ     8

main ()
{
    capability getport, putport;
    header hdr;
    char * buffer;
    int   err;

    uniqport (&getport);
    priv2pub (&getport, &putport);

    /* Publica a capability no servidor de diretorios */
    if ((err = name_append (SERVER_CAP, &putport)) != STD_OK ) {
          printf ("Nao pude acrescentar  a %s: %s\n", SERVER_CAP, err_why(err));
          exit (1);
    }

   for ( ; ; ) {
        hdr.h_port = getport;
        getreq(&hdr, buffer, REQ_BUFSZ);

        /* Processa a mensagem recebida */

        putrep(&hdr, buffer, REQ_BUFSZ);

   }
}
```

Figura 5.1: Exemplo de uma Aplicação Cliente/Servidor - Lado Servidor

Na Figura 5.1, o servidor solicita a geração de uma porta através de uma chamada à primitiva `uniq_port`. Esta porta será utilizada para este servidor receber pedidos dos clientes. Depois de escolhida a porta, o servidor converte a porta privada obtida em uma porta pública através de uma chamada a primitiva `priv2pub`. A porta resultante é então

publicada no servidor de diretórios, através da chamada à primitiva `name_append`.

Sempre que o cliente desejar se comunicar com o servidor, ele terá que obter a porta pública do servidor através de uma chamada à primitiva `name_lookup` (Figura 5.2). Esta primitiva recebe como parâmetro o nome de um arquivo e retorna à porta que foi publicada sob aquele nome. De posse desta porta, o cliente então pode fazer pedidos ao servidor através de chamadas à primitiva `trans` (ver Figura 5.2).

A primitiva `trans` possui seis parâmetros. Os três primeiros parâmetros do `trans` fornecem informações sobre o cabeçalho da mensagem, e sobre a mensagem a ser enviada; os três últimos parâmetros, por sua vez, fornecem as mesmas informações a respeito das respostas recebidas.

```
#define SERVER_CAP  "/super/cap/default"
#define REQ_BUFSZ    8
main (argc, argv)
int argc;
char * argv[];
{
    header hdr;
    char * buffer;
    int err;
    capability srv;

    /* Resgata a capability do servidor */
    if ( (err = name_lookup(SERVER_CAP, &svr)) != STD_OK ) {
        printf ("%s: lookup of %s failed: %s\n", argv[0], SERVER_CAP, err_why(err));
        exit (2);
    }
    /*Envia o comando para o servidor */
    hdr.h_port = svr.cap_port;
    hdr.h_priv = svr.cap_priv;
    trans (&hdr, buffer, REQ_BUFSZ, &hdr, buf, REQ_BUFSZ);

    .
    /* processa o resultado do trans */

    .
    exit (0);
}
```

Figura 5.2: Exemplo de uma Aplicação Cliente/Servidor- Lado Cliente

5.3. PROPRIEDADES DO SISTEMA DISPONÍVEL

No Capítulo 4 vimos que algumas suposições são feitas para o modelo Voltan de nodos replicados. Vimos que o modelo supunha: i) a existência de mecanismos de assinaturas digitais; ii) que o canal de comunicação no qual as mensagens trafegavam tinha que ser síncrono; e iii) que as aplicações a serem replicadas tinham que se comportar de maneira determinística. Esta seção define o sistema implementado face às limitações do ambiente e a necessidade de reduzir a complexidade da solução.

Nos nodos Voltan, a exigência de mecanismos de assinatura digital é feita para que o cliente possa detectar se uma mensagem é, ou não, válida. Se a mensagem tem $\pi+1$ assinaturas é sinal que pelo menos um processo correto (uma vez que π é o número máximo de processadores que podem falhar simultaneamente) validou aquela mensagem. Esta verificação das assinaturas se faz necessária quando a validação das mensagens é feita do lado servidor da aplicação. Na nossa implementação, optamos por colocar o mecanismos de validação do lado do cliente. Dessa forma, não se fez necessário a introdução de assinaturas nas mensagens trafegadas em nosso nodo replicado.

No modelo Voltan, as unidades de processamento formadas podem possuir semânticas de falha silenciosa ou semântica de falha mascarada. Em nosso trabalho nos preocupamos apenas com a implementação de nodos com semântica de falha silenciosa. Isto ocorreu porque o Amoeba não provê um serviço de comunicação síncrono, isto é, não existe nenhum mecanismo que garanta que uma mensagem enviada de um ponto a outro na rede é recebida dentro de um intervalo de tempo pré-determinado. Por esta razão, a gerência dos nodos com semântica de falha mascarada não é possível, já que os protocolos de ordenação destes nodos assumem a existência de um meio de comunicação síncrono [Brasileiro 95a]. Assim, em uma primeira versão, o serviço de processamento confiável do Seljuk-Amoeba foi construído em cima de processadores que falham por valor, construindo nodos que falham de forma silenciosa, ou seja, ou estes nodos funcionam corretamente, ou param de funcionar.

A última suposição feita pelo modelo Voltan é mantida no nosso sistema, ou seja, as aplicações que serão executadas no Seljuk-Amoeba têm que ser necessariamente

determinísticas.

5.4. DETALHES DE IMPLEMENTAÇÃO

No Capítulo 4, delineamos todas as características que achávamos necessárias ao serviço de processamento confiável do Seljuk-Amoeba. Entretanto, o serviço de processamento proposto difere do serviço implementado em vários aspectos. A primeira questão está relacionada com os nodos replicados formados. O projeto proposto trata de nodos com semântica de falha mascarada e de nodos com semântica de falha silenciosa. A nossa implementação foi realizada apenas para nodos com semântica de falha silenciosa, uma vez que para implementar nodos com semântica de falha mascarada necessitaríamos de um meio de comunicação síncrono; sincronismo este necessário para o funcionamento dos protocolos de ordenação desses nodos. Dessa forma optamos por implementar apenas a segunda classe de nodos com semântica de falha controlada. Além disso, dentre os nodos com semântica de falha silenciosa, optamos por aquele formado de apenas dois processadores por estes oferecerem uma solução prática e econômica para o problema da construção de nodos com semântica de falha controlada [Brasileiro et al. 96].

Uma outra questão relaciona-se com os mecanismos de comunicação utilizados pelo serviço de processamento confiável. Devido a maioria das aplicações distribuídas no Amoeba utilizarem RPC para a comunicação, optamos por implementar as *threads* necessárias à composição de um nodo, na camada RPC. Assim, aplicações que desejarem usar o serviço de processamento confiável terão que utilizar o mecanismo de comunicação de RPC.

Outra restrição na implementação diz respeito ao tipo de falha tolerado pelo nodo replicado formado. Os nodos formados estão aptos a tolerar apenas falhas no domínio do valor, falhas no domínio do tempo não são toleradas devido a ausência de um sincronismo no canal de comunicação. Uma vez enumeradas as suas características, vamos agora descrever como foi feita a implementação do serviço de processamento confiável para o ambiente operacional Seljuk-Amoeba.

Com o objetivo de inserir a redundância necessária à obtenção de confiabilidade no

serviço de processamento do Amoeba implementamos mudanças neste serviço em dois níveis. Inicialmente implementamos modificações no servidor de balanceamento de carga do Amoeba, como mencionado na Seção 4.4. Este servidor incorpora agora funcionalidades relacionadas com a criação das unidades de processamento replicado. Uma outra mudança implementada foi a introdução, no micronúcleo do Amoeba, de um serviço de gerência dos processos replicados; este serviço é o responsável pela gerência dos nodos replicados formados pelo *SA-Run Server*.

5.4.1. SA-SHELL

Como visto na seção 4.4.1, toda aplicação a ser executada deve ser invocada a partir do *SA-Shell*. O *SA-Shell* então solicita ao *SA-Run Server* dois processadores disponíveis para executar a aplicação de forma replicada. O *SA-Run Server* retorna um par de processadores para o *SA-Shell*. Em seguida, o *SA-Shell* conversa com o servidor de processos de cada processador solicitando a criação da réplica naquele processador (como pode ser visto na Figura 5.3).

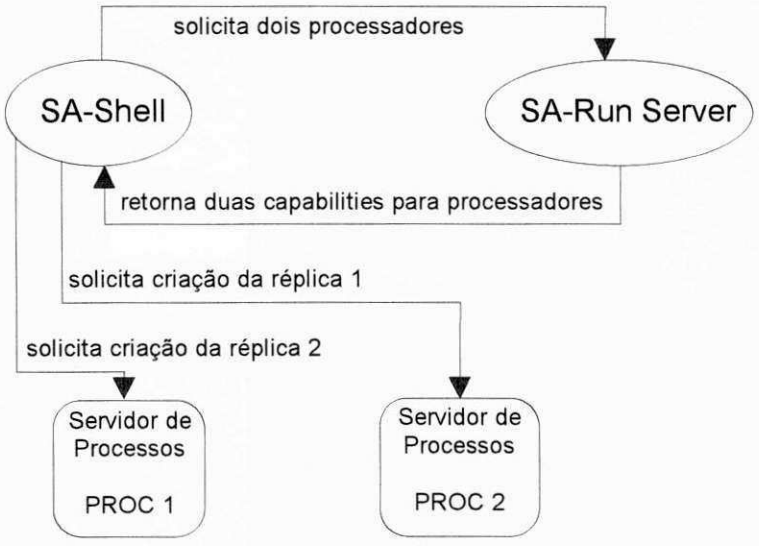


Figura 5.3: Processo de criação das réplicas no Seljuk-Amoeba

Ao criar cada uma das réplicas, o *SA-Shell* insere uma informação na estrutura de dados que identifica unicamente um processo no Amoeba; esta informação pode ser usada, em qualquer instante durante a execução da réplica, para identificar se a mesma é uma réplica de um processo replicado. Esta informação é utilizada pelas primitivas de

comunicação para que o serviço de gerência do nodo seja ativado. Dessa forma, nenhuma aplicação que não seja replicada perderá desempenho por conta da introdução dos protocolos de gerência.

5.4.2. SA-RUN SERVER

Conforme descrito no capítulo anterior, os nodos utilizados para o serviço de processamento do Seljuk-Amoeba são formados pelo *SA-Run Server*. Os nodos formados são capazes de tolerar faltas físicas e de concepção do hardware dos componentes do nodo. Para também tolerar faltas de concepção de software, a organização dos programas executáveis no Amoeba foi aprimorada. Em sua versão original, o Amoeba só trata de versões diferentes de programas para arquiteturas distintas, ou seja, só se pode ter uma única versão de um mesmo programa para cada arquitetura. Fizemos mudanças na organização do sistema de arquivos do Amoeba de forma que um programa em particular possa ter múltiplas versões disponíveis para cada arquitetura. Como discutido no capítulo anterior, é necessário que as diferentes versões sejam independentemente projetadas para satisfazer a especificação do programa.

A Figura 5.4 mostra a organização do diretório /bin (diretório do Amoeba no qual se encontram os programas executáveis), depois da introdução do suporte a múltiplas versões de um mesmo programa.

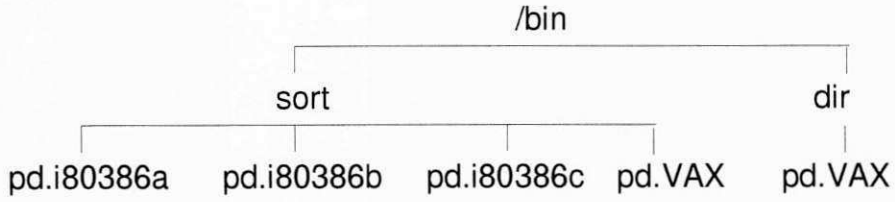


Figura 5.4: O Sistema de Arquivos do Seljuk-Amoeba

No exemplo da figura acima, o diretório /bin contém dois comandos: *dir* e *sort*. O comando *dir* está disponível apenas para a arquitetura VAX, enquanto que o comando *sort* possui três implementações disponíveis para a arquitetura 80386, além de uma implementação disponível para a arquitetura VAX. Este arranjo permite que projetistas de aplicações, solicitem que o *SA-Run Server* execute suas aplicações em nodos replicados, nos quais cada réplica executa uma implementação diferente do mesmo programa. Note

que com a diversidade das arquiteturas dos processadores e a disponibilidade de versões diferentes para cada réplica executando neles, o grau de tolerância a faltas obtido pode ser muito alto.

5.4.3. NODOS REPLICADOS FORMADOS NO SELJUK-AMOEBA

Como mencionado anteriormente, o modelo que utilizaremos para a construção dos nodos replicados é o modelo Voltan. Para se construir um nodo com semântica de falha silenciosa é necessário que os processos **Validador** e **Ordenador** estejam executando em cada processador componente do nodo. O processo **Validador** de um nodo com semântica de falha silenciosa é um processo **Comparador**. Sua função é comparar as saídas geradas pelo processo da aplicação que está executando no processador local com as saídas geradas pelas réplicas executando nos outros processadores do nodo.

A característica principal de um nodo com semântica de falha silenciosa é que este pára tão logo detecte um erro. Por esta razão o processo **Comparador** desempenha um papel muito importante no projeto do nodo. Qualquer falha que afete a execução correta dos processos da aplicação, ou dos processos do sistema[7], é detectada pelos processos **Comparador** dos processadores corretos[8]. Assim, todos os processos do sistema, exceto o **Comparador**, podem ser projetados assumindo que estes executam em um ambiente livre de falhas. Particularmente, esta propriedade pode ser usada para projetar um processo **Ordenador** mais simples e mais eficiente.

Para que se construa um nodo com semântica de falha controlada é necessário que as réplicas processem as mesmas entradas e na mesma ordem. Vários esquemas para se alcançar os requisitos de acordo e ordem foram pesquisados para a família Voltan [Brasileiro 95a]. No esquema mais prático e eficiente, um nodo possui uma semântica de falha silenciosa sendo formado por dois processadores [Brasileiro et al. 96]. As réplicas que executam no nodo desempenham papéis específicos e devem estar executando em

[7] Chamamos de processos do sistema aqueles processos necessários à formação de um nodo replicado (e.g. Ordenador, Comparador, etc.)

[8] Chamamos de processadores corretos todos aqueles que estão funcionando sem apresentar falhas.

processadores distintos; uma réplica é denominada líder (*leader)* e a outra é denominado seguidor (*follower*). O líder é responsável por definir a ordem na qual as mensagens serão processadas e é responsável também por indicar esta ordem ao seguidor.

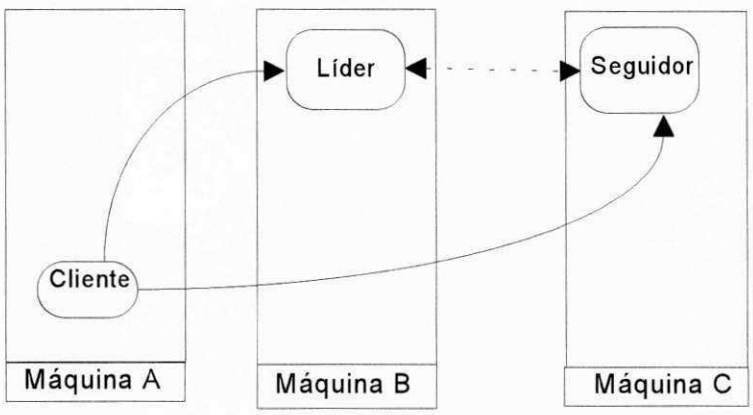


Figura 5.5: Um típico nodo Voltan

A Figura 5.5 mostra o comportamento de um típico nodo Voltan[9]. O cliente faz uma requisição ao servidor líder, e em seguida faz a mesma requisição ao servidor seguidor. Esses servidores, por sua vez, conversam entre si para entrarem em um acordo sobre a ordem na qual as mensagens foram recebidas do cliente. Depois de estabelecida uma ordem nas mensagens, estas são processadas ao mesmo tempo pelo servidor líder e pelo servidor seguidor. Após esses processamentos, se os resultados de ambos não divergirem, as respectivas mensagens de saída são enviadas para o cliente, que fica encarregado de descartar as mensagens duplicadas. Se os resultados dos processamentos do líder e do seguidor forem diferentes, nenhuma resposta é enviada de volta ao cliente, e o nodo pára, caracterizando a semântica de falha silenciosa do nodo.

Uma vez que o objetivo do ambiente Seljuk-Amoeba é, principalmente, oferecer um serviço de processamento replicado que seja transparente para as aplicações, o modelo mostrado na Figura 5.5 não é aceitável. Neste caso o projetista das aplicações **cliente** e **servidor** teria que incorporar os mecanismos para tolerância a faltas no próprio código das mesmas, além disso, deveriam existir duas versões da aplicação **servidor**, uma

[9] Nesta seção, quando estivermos utilizando o termo "nodos Voltan" estaremos nos referindo aos nodos Voltan com semântica de falha silenciosa compostos de dois processadores.

comportando-se como líder e outra como seguidor. Para se evitar isto, os mecanismos para tolerância a faltas estão inseridos no sistema operacional e são fornecidos pelo serviço de processamento do Seljuk-Amoeba.

No serviço proposto, todas as entidades (adicionais aos processos cliente e servidor) necessárias ao gerenciamento das réplicas são implementadas como *threads* do sistema operacional. Na Figura 5.6 podemos identificar as seguintes *threads*: O-líder (ordenador líder), O-seguidor (ordenador seguidor), comparador e duas outras *threads* que denominamos rt_1 (roteador 1) e rt_2 (roteador 2).

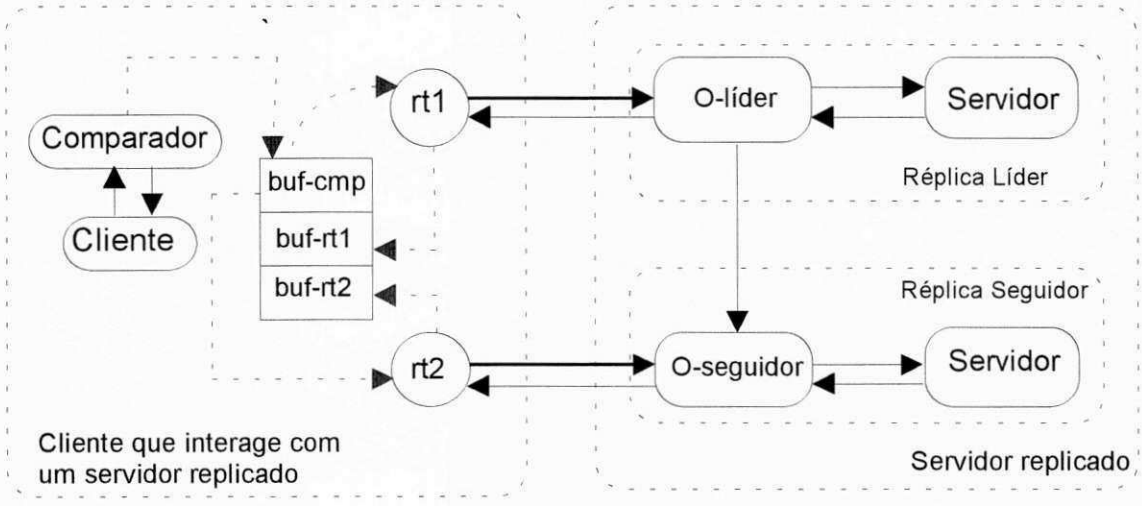


Figura 5.6: Um nodo com semântica de falha silenciosa composto de dois processadores

No estado inicial do sistema tem-se um cliente que deseja fazer uso de um serviço provido por um determinado servidor que se encontra replicado. A inicialização das réplicas do servidor é feita pelo *SA-Shell*, que conversa com os servidores de processos dos dois processadores do nodo, solicitando a criação das réplicas. Para poder conversar com o servidor, o cliente precisa da porta na qual o servidor recebe pedidos. Para obter esta porta, o cliente faz uma chamada à primitiva `name_lookup`, que por sua vez dispara as *threads* comparador, rt1 e rt2.

O servidor, por outro lado, deve fazer uma chamada à primitiva `name_append` para publicar a porta na qual ele estará aceitando pedidos. Quando `name_append` é chamada, ela dispara as *threads* O-líder e O-seguidor (detalhes sobre o funcionamento das primitivas `name_lookup` e `name_append` serão apresentados mais adiante).

Utilizando este estado inicial (Figura 5.6) como base, vamos agora entender como se dá o comportamento do nodo replicado implementado.

O cliente faz uma RPC ao comparador. O comparador, por sua vez, tem como função inicial repassar a mensagem enviada pelo cliente para os outros processos do nodo. O comparador desempenha este papel depositando a mensagem em uma área de armazenagem bem conhecida das *threads* rt_1 e rt_2. Quando o comparador deposita as mensagens nas referidas áreas de armazenagem, ele emite um sinal para as *threads* rt_1 e rt_2 indicando que estas já podem ler da área de armazenagem.

Ao receberem o sinal indicando que as áreas de armazenagem já foram preenchidas, as *threads* rt_1 e rt_2 lêem as respectivas mensagens. Em seguida, cada *thread* faz uma RPC com seu respectivo ordenador enviando como parâmetro a mensagem lida; a *thread* rt_1 faz uma RPC com o ordenador da réplica líder (O-líder), enquanto que a *thread* rt_2 faz uma RPC com o ordenador da réplica seguidora (O-seguidor).

O ordenador líder recebe as mensagens vindas do cliente através da *thread* rt1. O papel do ordenador líder é indicar (para o ordenador seguidor) a ordem na qual as mensagens devem ser processadas. O ordenador líder envia a mensagem recebida do cliente ao ordenador seguidor, além de repassá-la também ao servidor (réplica líder) que processará essa mensagem.

Por outro lado o ordenador seguidor recebe a mensagem enviada pelo cliente por dois lugares diferentes: recebe a mensagem enviada pelo ordenador líder e a mensagem enviada pela *thread* rt2. O ordenador seguidor repassa a primeira mensagem que ele recebe ao servidor (réplica seguidora), descartando a mensagem que chega depois (essa mensagem pode ser usada para tratar alguns tipos de falhas de comunicação, mas nessa implementação nós não fazemos qualquer tipo tratamento; para exemplos desse tipo de tratamento ver [Brasileiro 95]).

Uma vez que as duas réplicas tenham recebido as mensagens, estas processam-nas e enviam as respectivas respostas de volta para quem fez o pedido; no caso os ordenadores. Os ordenadores, por sua vez, enviam as respostas geradas pelas réplicas para

as *threads* rt_1 e rt_2. Essas *threads* depositam as mensagens recebida dos ordenadores na mesma área de armazenagem que o comparador utilizou para passar o pedido vindo do cliente. Uma vez que isso tenha sido feito, um sinal é gerado informando ao comparador que este já pode ler os resultados daquela área de armazenagem. De posse destas mensagens o comparador faz uma checagem para ver se os resultados são iguais. Se houver coincidência de resultados, uma das mensagens é enviada de volta ao cliente; caso contrário, uma mensagem de erro é retornada. Esta mensagem de erro fez-se necessária devido à natureza bloqueante das primitivas de RPC. Uma vez que o cliente fica bloqueado esperando uma resposta, não seria prudente o nodo omitir uma resposta para o cliente no caso de falha (o que garantiria a semântica de falha silenciosa do nodo). Entretanto, o cliente pode utilizar esta mensagem de erro retornada para identificar uma falha do nodo.

5.5. MODIFICAÇÃO NA INFRA-ESTRUTURA DE COMUNICAÇÃO

Como visto na seção 5.1, o Amoeba disponibiliza, além das primitivas usadas efetivamente para permitir a comunicação entre *threads* dos processos, algumas primitivas que fornecem serviços de apoio à comunicação, como os serviços de registro de nomes (primitiva `name_append`), busca de nomes (primitiva `name_lookup`) e de liberação de nomes (primitiva `name_delete`).

Efetuamos mudanças no comportamento destas primitivas para que pudéssemos criar os mecanismos necessários à ordenação das mensagens enviadas pelo cliente para os servidores replicados. A seguir detalharemos as mudanças efetuadas nessas primitivas.

- PRIMITIVA NAME_LOOKUP

A primitiva `name_lookup` recebe uma *capability* para um nome e retorna uma *capability* para a porta de comunicação que aquele nome representa. Com a modificação desta primitiva, as mensagens que antes eram enviadas direto para o servidor, são enviadas aos processos ordenadores Líder e Seguidor de forma que sejam ordenadas antes de serem processadas pelas réplicas. Do ponto de vista da implementação da aplicação cliente nada muda. O código fonte é o mesmo; existe apenas a necessidade de compilação da aplicação já que as primitivas `name_append` e `name_lookup` são disponibilizadas nas bibliotecas

de funções do Amoeba.

Um esqueleto da primitiva `name_lookup` é mostrado a seguir (Figura 5.7).

```
errstat name_lookup (name, object)
char *name;
capability *object;
{
/* Se o ordenador do líder e o ordenador do seguidor estão executando */
if ( (dir_lookup ((capability *)0, OrderLName, &pLeader) == STD_OK) &&
    (dir_lookup ((capability *)0, OrderFName, &pFollower) == STD_OK) ){

    /* Crio uma porta para o Comparador ficar "escutando" */
      uniqport (&aux.cap_port);
      priv2pub (&aux.cap_port, &router_pub.cap_port);
      router_pub.cap_priv = aux.cap_priv;

    /* Retorno a porta do comparador para o cliente*/
    object->cap_port = router_pub.cap_port;
    object->cap_priv = router_pub.cap_priv;

    /* Crio a thread do Comparador */
    if (!thread_newthread (Comparator, STACKSZ, (char *) 0, 0)){
        return (FT_ERROR1);
    }

/* Crio a thread do RT1 */
    if (!thread_newthread (RT1, STACKSZ, (char *) 0,  0) {
        return(FT_ERROR2);
    }

    /* Crio a thread do RT1 */
    if (!thread_newthread (RT2, STACKSZ, (char *) 0,  0) {
        return(FT_ERROR3);
    }
    /* Se não deu erro até agora, retorno ok */
    return (STD_OK);
} else
    /* Se os ordenadores não existem, sigo o caminho normal*/
    return (dir_lookup((capability *)0, name, object));
}
```

Figura 5.7: Esqueleto de name_lookup

Inicialmente verifica-se se as *threads* Ordenador Líder e Ordenador Seguidor já foram criadas; este é o indicador de que a aplicação que invocou a primitiva name_lookup é cliente de um servidor replicado. Em seguida criamos uma porta que

será usada pelo comparador para receber requisições. Esta porta é enviada de volta ao cliente, que a partir deste momento envia requisições para o comparador achando que está enviando diretamente para o servidor. O comparador por sua vez, fica num laço recebendo as requisições do cliente. O comparador desempenha o mesmo papel com todo pedido que chega do cliente: coloca a requisição num *buffer*, indica para as *threads* rt_1 e rt_2 que o *buffer* foi preenchido, e fica bloqueado esperando que estas *threads* respondam. As *threads* rt_1 e rt_2, por sua vez, lêem do *buffer* o pedido do cliente, repassam os pedidos para os respectivos ordenadores e ficam bloqueadas esperando respostas (ver Figura 5.7). Quando as respostas chegam, estas *threads* colocam estas mensagens em *buffers* conhecidos do comparador e enviam um sinal a ele. O comparador então é desbloqueado, lê as respostas contidas nos respectivos *buffers* e as compara. Se as respostas forem iguais, o comparador envia uma delas de volta ao cliente; caso contrário um erro é sinalizado.

- PRIMITIVA NAME_APPEND

Quando um processo servidor é inicializado, ele tem que escolher uma porta na qual ele estará recebendo pedidos, além de publicar esta porta em um previamente conhecido pelo cliente. Para efetuar esta publicação é que a primitiva name_append foi criada. Esta primitiva recebe como parâmetros um nome de um arquivo e uma porta. Com isto, cada vez que o cliente fizer uma chamada a name_lookup passando o mesmo nome de arquivo como parâmetro, ele obterá a porta na qual o servidor estará aceitando pedidos.

Quando um servidor faz uma chamada à primitiva name_append, esta tem que saber se o servidor é replicado (ver Figura 5.8). Se a resposta for negativa, a primitiva segue o seu curso normal; entretanto se o servidor for replicado, uma série de passos adicionais são necessários. Primeiramente, verifica-se se a porta do ordenador líder já foi publicada ; esta porta é usada pela thread rt_1 para repassar os pedidos que chegam do cliente para o ordenador líder. Se a porta ainda não tiver sido publicada, a publicação é feita e a thread ordenador líder é disparada.

Se a porta do ordenador líder já tiver sido publicada, publica-se a porta e dispara-se a thread do ordenador seguidor. Uma vez que as portas dos ordenadores estejam

publicadas, as threads rt_1 e rt_2 podem localizá-las para repassar os pedidos vindos do cliente para eles. Os ordenadores, por sua vez, enviam estes pedidos aos servidores replicados e esperam pelas respostas. De posse das respostas, os ordenadores as enviam de volta às *threads* rt1 e rt2.

```
errstat
name_append(name, object)
char *name;
capability *object;
{

/* Se esta rotina foi chamada por um processo replicado */
if (AmIReplicated()) {

/* Se a porta thread ordenador líder ainda não foi publicada */
    if ((dir_lookup ((capability *)0, LeaderP, &cap_aux)) != STD_OK) {

/* publico a porta do ordenador líder */
        if ((dir_append ((capability *)0, LeaderP, object)) == STD_OK)

/* disparo a thread ordenador do líder */
            if (!thread_newthread(OrderL, STACKSZ, LeaderP, sizeLeader)){
                return (FT_ERRO1);
            } else
                return (STD_OK);

/* Se a porta do ordenador líder já foi publicada */
   } else {

/*Se a thread ordenador do líder já existe, publico a porta do ordenador seguidor */
        if ((dir_append ((capability *)0, FollowerP, object)) == STD_OK)

/* Disparo a thread ordenador seguidor */
            if (!thread_newthread(OrderF, STACKSZ, FollowerP, sizeFollower))
                return (FT_ERRO2);
            else
                return (STD_OK);
    }

/* Se nao for replicado segue o curso normal */
} else
        return dir_append((capability *)0, name, object);
}
```

Figura 5.8: Esqueleto de name_append

- PRIMITIVA NAME_DELETE

Uma vez que a primitiva name_append publica a porta de um servidor no servidor de diretório, se faz necessário uma primitiva que remova esta porta depois do seu uso. A primitiva name_delete foi criada com este objetivo. Como fizemos várias mudanças na primitiva name_append, de forma que esta publique algumas portas adicionais à porta do servidor propriamente dito, foi necessário mudarmos também name_delete para que esta removesse todas as portas de servidores criadas em name_append.

5.6 CONCLUSÕES

O teste feito consiste em uma aplicação cliente, que executa fora de um nodo replicado, requisitando um serviço a um servidor que está executando em um nodo replicado. O cliente envia um pedido ao servidor e espera pela resposta. O servidor recebe o pedido do cliente, processa e envia a resposta de volta ao cliente. Depois de receber a resposta do servidor, o cliente emite um novo pedido. Nós medimos o intervalo de tempo entre o pedido do cliente e a resposta do servidor; fizemos isto para um servidor executando em um nodo replicado e um executando sem replicação.

Para um melhor resultado nos testes, a computação executada pelo servidor é mínima. Na realidade o servidor apenas recebe os pedidos e os envia de volta sem nenhum processamento. O objetivo deste processamento mínimo é o de tentar medir mais precisamente o tempo gasto pelos mecanismos de gerência dos nodos replicados. A idéia é que a computação não seja levada em consideração nos cálculos.

Nós fizemos uma análise superficial do desempenho do serviço de processamento replicado, comparando o desempenho de uma aplicação cliente/servidor convencional, com uma aplicação cliente/servidor na qual o servidor executa em um nodo replicado com semântica de falha silenciosa. Como já era esperado, houver uma considerável perda no desempenho do serviço. Vale salientar que os resultados obtidos não refletem a sobrecarga real imposta pelos protocolos de gerência das réplicas, uma vez que o código analisado ainda não foi otimizado por conta de restrições de tempo. Pretendemos, como trabalho futuro, perfilar a execução do nodo replicado para tentar identificar o gargalo do sistema e eliminá-lo se for possível.

Entretanto, existem aplicações nas quais o serviço de processamento confiável ainda é adequado, mesmo sem fazer uma otimização no código. Essas aplicações caracterizam-se por possuirem um alto tempo de processamento no servidor, ou possuirem uma baixa freqüência de interação cliente-servidor.

Capítulo 6

Conclusões

O serviço de processamento confiável do ambiente operacional Seljuk-Amoeba é uma importante ferramenta para a construção de aplicações distribuídas robustas. Este provê a redução da complexidade do desenvolvimento da aplicação, por permitir que os programadores assumam que a semântica de falha dos nodos sobre os quais as aplicações executam seja mais restritiva do que aquela fornecida pelos processadores que compõem o nodo. O Seljuk-Amoeba é encarregado da gerência dos processadores redundantes necessários para garantir a semântica de falha assumida para o nodo.

Este serviço também oferece flexibilidade para aplicações, por permitir que estas escolham suas semânticas de falha em tempo de ativação. Desta forma, se os requisitos de confiança no funcionamento da aplicação mudarem ao longo do tempo, será possível continuar a fornecer o serviço requerido sem a necessidade de adaptar a aplicação, já que o serviço de processamento confiável é implementado no micronúcleo do Seljuk-Amoeba.

Quando a semântica de falha dos processadores disponíveis for, no mínimo, tão restritiva quanto a semântica de falha requerida pela aplicação, o serviço de processamento do Seljuk-Amoeba irá se comportar exatamente como o serviço de processamento original do Amoeba, e não irá impor nenhuma redução no desempenho do sistema como um todo, uma vez que os protocolos de gerências das réplicas não estarão sendo utilizados..

Apesar da vantagens apresentadas, o Seljuk-Amoeba apresenta algumas restrições como, por exemplo, o fato do ambiente não suportar aplicações com requisitos temporais. Isto acontece devido a algumas limitações, a saber:

A comunicação no Seljuk-Amoeba é baseada em barramento, portanto não é possível tolerar falhas arbitrárias [Harper et al. 88] dos componentes de comunicação. Por exemplo, nada impede que um processador ou um controlador defeituoso inunde o barramento com pacotes, impossibilitando a comunicação entre os outros componentes corretos.

Um sistema de tempo real tem que executar um conjuntos de tarefas concorrentes de maneira que todas as tarefas com requisitos de restrição de tempo sejam executadas dentro de um tempo máximo conhecido. Para que isto ocorra, o sistema tem que dispor, entre outros requisitos, de uma rede de comunicação na qual se conhece o atraso máximo para a transmissão de mensagens, aliado a uma política de escalonamento que cumpra as exigências dos prazos de execução das tarefas. O Seljuk-Amoeba não dispõe de nenhum desses mecanismos.

Referências Bibliográficas

[Bernstein 88] P.A. Bernstein, "Sequoia: A Fault-Tolerant Tightly Coupled Multiprocessor for Transaction Processing," IEEE Computer, Vol. 21, No. 2, pp. 37-45, February 1988.

[Birrell-Nelson 84] A.D. Birrell e B.J. Nelson, "Implementing Remote Procedure Calls", ACM Transactions on Computer Systems, Vol. 2, No. 1, pp. 39-59, fevereiro de 1984.

[Brasileiro 95] F.V. Brasileiro, "Constructing Fail-Controlled Nodes for Distributed Systems," Ph.D. Thesis, University of Newcastle upon Tyne, May 1995.

[Brasileiro 97] 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, pp. 45-59, Julho de 1997.

[Brasileiro et al. 92] F.V. Brasileiro, P.D. Ezhilchelvan, S.K. Shrivastava, N.A. Speirs, e S. Tao, "Efficient Protocols for Fail-Silent Nodes in Distributed Systems," Technical Report No. 413, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, December 1992.

[Brasileiro et al. 96] F.V. Brasileiro, P.D. Ezhichelvan, S.K. Shrivastava, N.A.

Speirs e S. Tao, "Implementing Fail-Silent Nodes for Distributed Systems", IEEE *Transactions on Computers*, Vol. 45, No. 11, pp. 1226-1237, Novembro de 1996.

[Brasileiro et al. 97] F.V. Brasileiro, E.L.Gallindo, S.R. Vasconcelos e V.S. Catão, "On the Design of the Seljuk-Amoeba Operating Environment", *Journal of the Brazilian Computer Society*, dezembro de 1997.

[Budhiraja et al. 93] N. Budhiraja, K. Marzullo, F.B. Schneider e S. Toueg, "The Primary-Backup Approach", In Sape Mullender, editor, *Distributed Systems*, pp. 199-216. ACM Press, 1993.

[Catão-Brasileiro 97] V.S. Catão e F.V. Brasileiro, "Serviço de Comunicação Síncrona para Nodos Replicados", Anais do VII Simpósio de Computadores Tolerantes a Falhas, Julho de 1997.

[Cristian 91] F. Cristian, "Understanding Fault-Tolerant Distributed System", Communication of the ACM, Vol. 34, No. 2, Fevereiro de 1991, pp. 57-78.

[Czeck et al. 85] E. W. Czeck, D. P. Siewiorek e Z. Segall, "Fault Free Performance Validation of a Fault-Tolerant Multiprocessor: Baseline and Synthetic Workload Measurement", Carnegie Mellon University, Departament Computer Science, CMU-CS-85-117.

[Harper et al. 88] R.E. Harper, J. H. Lala e J. J. Deyst, "Fault Tolerant Processor Architecture Overview", Digest of Papers, FTCS-18, Tokyo, Japan, pp. 252-257, Junho de 1988.

[Hopkins et al. 78] A.L. Hopkins, T.B. Smith, and J.H. Lala, "FTMP - A Highly Reliable Fault-Tolerant Multiprocessor for Aircraft," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1221-1239,

October 1978.

[Jalote 94] Pankaj Jalote, Fault Tolerance in Distributed Systems, Prentice Hall, New Jersey, 1994, ISBN 0-13-301367-7.

[KRST93] M.F. Kaashoek, R. Renesse, H. van Satveren,e A.S. Tanenbaum, "FLIP: An Internetwork protocol for Supporting Distributed Systems", ACM Transactions on Computer Systems, Vol. 11, No. 2, pp. 73-106, Fevereiro de 1993.

[Lala 86] J.H. Lala, "A Byzantine Resiliente Fault Tolerant Computer for Nuclear Power Plant Applications", Digest of Papers, FTCS-16, Vienna, Austria, pp. 338-343, Julho de 1986.

[Lemos-Veríssimo 91] R. de Lemos e P. Veríssimo, "Confiança no funcionamento - Proposta para uma terminologia em Português", comunicação pessoal, dezembro de 1991.

[Mullender et al. 90] 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.

[Palumbo-Butler 85] D. L. Palumbo e R. W. Butler, "Measurement of SIFT operating system overhead", NASA Tech. Memo. 86322, 1985.

[Powell 92] D. Powell (Ed.), Delta-4 – A Generic Architecture for Dependable Distributed Computing, Spring-Verlag, 1992, ISBN 3-540-54985-4.

[Schlichting- Schneider 84] R.D. Schlichting, and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing

Systems," ACM Transactions on Computer Systems, Vol. 1, No. 3, pp. 222-238, August 1984.

[Schneider 84] F.B. Schneider, "Byzantine Generals in Action: Implementing Fail-Stop Processors," ACM Transactions on Computer Systems, Vol. 2, No. 2, pp. 145-154, May 1984.

[Schneider 90] F.B. Schneider, "Implementing Fault Tolerant Services Using the State Machine Approach: a Tutorial", ACM Computing Surveys, Vol. 22, No. 4, pp. 299-319, dezembro de 1990.

[Shrivastava et al. 91] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, and D.T. Seaton, "Fail-Controlled Computer Architectures for Distributed Systems," Technical Report No. 333, Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, UK, July 1991.

[Shrivastava et al. 92] S.K. Shrivastava, P.D. Ezhilchelvan, N.A. Speirs, S. Tao, and A. Tully, "Principal Features of the Voltan Family of Reliable Node Architectures for Distributed Systems," IEEE Transactions on Computers, Vol. 41, No. 5, pp. 452-549, May 1992.

[Smith 84] T.B. Smith, "Fault Tolerant Processor Concepts and Operation," Digest of Papers, FTCS-14, Kissimmee, USA, pp. 158-163, June 1984.

[Soares et al. 95] L.F.G Soares, G. Lemos e S. Colchet, Das LANs, MANs e WANs às Redes ATM, 2a. edição, Ed. Campus, 1995, ISBN 85-7001-954-8.

[Tanenbaum et al. 86] A.S. Tanenbaum, S.J. Mullender e R. Van Renesse, "Using Sparse Capabilities in a Distributed Operating System",

Proc. Sixth Int'l Conf. On Distributed Computing Systems, IEEE, pp. 558-563, 1986.

[Theuretzbacher 86] N. Theuretzbacher, "'VOTRICS': Voting Triple Modular Computing System," Digest of Papers, FTCS-16, Vienna, Austria, pp. 144-150, July 1986.

[Vasconcelos 97] S.R.A. Vasconcelos, "Provendo Serviços para Tolerância a Faltas em Sistemas Distribuídos", Dissertação de Mestrado, Coordenação de Pós-graduação em Informática, Universidade Federal da Paraíba, Campina Grande, Brasil, julho de 1997.

[Verstoep-Sharp 94] Kees Verstoep e Gregory J. Sharp, "The Amoeba Processor Reservation System", Faculteit Wiskunde en Informatica, Vrije Universiteit, Amsterdam.

[Webber-Beirne 91] S. Webber, and J. Beirne, "The Stratus Architecture," Digest of Papers, FTCS-21, Montréal, Canada, pp. 79-85, June 1991.

[Wensley et al. 78] J.H. Wensley, J.H. Wensky, L. Lamport, J. Goldberg, M.W. Green, K.N. Levitt, P.M. Melliar-Smith, R.E. Shostak, e C.B. Weinstock, "SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control," Proceedings of the IEEE, Vol. 66, No. 10, pp. 1240-1255, October 1978