DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Relatório Técnico

RT-MAC-9607

Sistemas Operacionais Distribuídos: Conceitos, Exemplos e Tendências

Markus Endler

Agosto 96

Sistemas Operacionais Distribuídos: Conceitos, Exemplos e Tendências

Markus Endler
Departamento de Ciência da Computação
Instituto de Matemática e Estatística
Universidade de São Paulo

Prefácio

Esta monografia consiste de um resumo das notas da aula e de trabalhos apresentados por alunos na disciplina de pós-graduação *MAC-755 Sistemas Operacionais Distribuídos* ministrada no IME/USP no segundo semestre de 1994. A principal motivação foi a de elaborar um texto didático em português que apresentasse os principais conceitos e mecanismos usados em tais sistemas, e que ao mesmo tempo mostrasse também exemplos concretos de sistemas operacionais e plataformas distribuídas atualmente existentes.

Devido à impossibilidade de apresentar todos os sistemas, tive que escolher um subconjunto pequeno de sistemas que ao meu ver tiveram (e ainda têm) um papel significativo no desenvolvimento desta área. São eles o *Mach, V System, Chorus, Sprite, Isis, Horus, Synthetix* e o *Distributed Computing Environment* da OSF. Ainda assim, faltam artigos sobre sistemas *canônicos* como por exemplo o Amoeba, o X-Kernel e outros que não foram completados devido à desistência de alguns alunos.

O curso foi em grande parte baseado no excelente livro *Modern Operating Systems* de Andrew S. Tanenbaum[22], e a primeira parte desta monografia resume e elabora sobre os principais conceitos abordados nos capítulos finais este livro. À exceção do artigo introdutório sobre o sistema Mach, baseado também em um capítulo de *Modern Operating Systems*, todos os demais artigos são resumos de artigos de revistas e relatórios técnicos obtidos pela rede.

Gostaria de agradecer aos alunos Sergio Borger, Rodrigo Braz, Celia S. Fukusawa, Alfredo Goldman, Fabio Kon, Jorge Nakahara Jr. , Fabio Nakeno, Eugenio Nasser, e Vanderlei S. Porcel pela pesquisa e o trabalho que investiram na elaboração dos excelentes resumos que compõem a segunda parte desta monografia.

1 Introdução

Sistemas operacionais distribuídos têm recebido crescente interesse devido à tendência ao uso de redes de computadores e de sistemas multiprocessadores. Há várias vantagens no uso de sistemas distribuídos em comparação com os uniprocessadores, sejam eles computadores de grande porte (mainframes) ou os computadores pessoais.

A primeira é a possibilidade de aumentar a capacidade de processamento a um custo relativamente baixo, isto é, com uma boa relação custo/benefício, visto que é bem mais barato utilizar vários computadores ou processadores de média performance do que adquirir um computador paralelo ou vetorial de alto desempenho.

A segunda vantagem vem da adequação natural de tais sistemas a muitas aplicações intrinsecamente distribuídas, como é o caso, por exemplo, da automação bancária e da automação industrial, entre outras. Tais aplicações requerem o compartilhamento de recursos e de informações por parte de seus usuários.

A terceira é a possibilidade de se obter uma maior confiabilidade e disponibilidade do sistema, visto que teoricamente é menos provável que todas as máquinas de um sistema distribuído falhem, do que quando se tem apenas um uniprocessador. Esta vantagem, no entanto, depende fortemente da organização do software e da eficácia dos mecanismos de tolerância a falhas.

Uma outra vantagem está na possibilidade de uma expansão gradual do sistema à medida que crescem as necessidades de capacidade de processamento e/ou de interconexão do sistema. Tal necessidade de fato ocorre com muitos grupos de usuários e só pode ser satisfeita até certo ponto quando todo o processamento está centralizado em um mainframe. Tendo-se vários computadores pessoais, pode-se facilmente interconectá-los numa rede, mas o grau de integração do sistema dependerá do tipo de sistema operacional usado.

Uma quinta vantagem está na flexibilidade de configuração do sistema, isto é, na escolha de que função cada máquina vai exercer e se ela será dedicada a um único usuário, ou se ela atuará como gerenciadora de recursos compartilhados (servidora de disco ou de terminais).

Apesar de tantas vantagens, sistemas distribuídos também apresentam alguns problemas:

O principal é a dificuldade de se desenvolver o software para tais sistemas, que em geral é bem mais complexo do que o software para uniprocessadores. Em parte, esta dificuldade se deve à falta de conceitos e estruturas abstratas que facilitem a tarefa de programação, tal como o a programação estruturada e o conceito de procedimento que auxiliam a programação seqüencial. Pesquisadores atuando nas áreas de sistemas operacionais e linguagens de programação têm se empenhado em melhorar esta situação, e têm-se observado certo avanço, principalmente com a invenção do conceito de *chamada remota de procedimentos*, (Remote Procedure Call)[6] atualmente presente em muitas das linguagens e sistemas operacionais.

A outra razão é que programação paralela e distribuira é intrinsecamente mais complexa do que a seqüencial devido à impossibilidade de se captar o estado global do sistema. Isto vem da inexistência de um sincronismo na execução das máquinas do sistema, que impede um congelamento da execução, tal como ele é feito em depuradores para programas seqüenciais. Além disto, o projeto de algorítmos distribuídos requer o teste de ausência de bloqueios (*deadlocks*) e da satisfação de propriedades de vivacidade (*liveness*), isto é, que certas operações garantidamente são executadas periodicamente. Tais problemas estimularam muitas pesquisas práticas e teóricas nas áreas de especificação e verificação de protocolos distribuídos, bem como de padrões de interação de processos distribuídos.

Um outro problema do uso de sistemas distribuídos está ligado a não-confiabilidade do meio de interconexão. Em qualquer rede, mensagens podem se perder total ou parcialmente, e seu conteúdo pode ter erros de transmissão. Isto estimulou o desenvolvimento de protocolos de comunicação que garantam uma transmissão correta sobre o meio de comunicação. O problema é que o quanto mais elaborado

for o protocolo, pior será a performance do sistema. Como em redes locais (LANs) a probabilidade de ocorrência de erros é bem menor do que em redes metropolitanas (Metropolitan Area Networks - MANs) ou de redes de longo alcance (Wide Area Networks - WANs), têm-se adotado protocolos de comunicação mais simples - e conseqüentemente mais eficientes - em sistemas operacionais distribuídos.

Outro fator que pode influenciar dramaticamente o desempenho de um sistema distribuído é a carga sobre o meio de comunicação. Se por exemplo, o meio é de difusão, como um barramento tipo *Ethernet*, há um limite no número de máquinas que podem ser conectadas a este meio. Para este mesmo meio, há também a técnica de segmentação de redes locais através de pontes (bridges), que permitem aliviar a carga total sobre uma rede.

O último problema no uso de sistemas distribuídos está relacionado à segurança no acesso aos recursos compartilhados. Devido à interconexão em larga escala de redes locais, precisa-se de mecanismos eficientes de autenticação (descobrir a autenticidade do usuário), de controle de acesso aos recursos, e de criptografia (a codificação e decodificação de mensagens), para evitar que usuários não credenciados acessem recursos ou informações indevidamente.

1.1 Tipos de sistemas operacionais

Essencialmente, distingue-se dois tipos de sistemas operacionais para sistemas distribuídos:

Sistemas operacionais de rede são na verdade serviços distribuídos (sistema de arquivos, serviço de informação) operando sobre um conjunto heterogêneo de sistemas operacionais locais. A principal característica destes sistemas é que eles não implementam integralmente o acesso transparente aos recursos distribuídos na rede, eventualmente obrigando o usuário a conhecer o nome das máquinas e a localização exata dos recursos. O *Network File System* (NFS),talvez o mais conhecido dos serviços distribuídos, no entanto, representa um certo avanço com relação a este problema no sentido de que implementa um acesso transparente (de localização) a servidores de arquivos remotos.

Sistemas operacionais distribuídos por outro lado, implementam o acesso remoto a recursos de forma totalmente transparente, criando não só a nível de comandos do usuário, mas também a nível de processos da aplicação, a ilusão de um sistema uniprocessador. Estes sistemas tipicamente têm um mesmo kernel executando em todas as máquinas que cuida do gerenciamento de recursos locais e da comunicação transparente entre processos em diferentes máquinas.

A seguir, veremos a estrutura e as principais características deste segundo grupo de sistemas operacionais e depois analisaremos os seus representantes mais conhecidos.

1.2 A arquitetura Microkernel

Sistemas operacionais distribuídos (SODs) são compostos de um núcleo (kernel), que roda em todas as máquinas do sistema, e um conjunto de serviços realizados por servidores distribuídos sobre a rede, que executam em modo usuário. Na maioria dos SODs, optou-se pela implementação de um kernel contendo apenas o conjunto mínimo de funções necessárias à execução e a interação dos processos do sistema (os servidores) e dos processos de usuários (os clientes). Por causa desta característica, os mesmos são chamados de microkernels. A implementação de SODs segundo esta arquitetura oferece as seguintes vantagens:

Modularidade É possível desenvolver os serviços distribuídos de forma modular, testando-os individualmente e identificando precisamente as interdependências entre os serviços do sistema operacional.

Flexibilidade É possível configurar cada serviço da forma mais adequada às necessidades dos usuários e de suas aplicações. Em particular, pode-se escolher o grau de distribuição e de replicação dos servidores de um serviço a fim de influenciar a performance e a disponibilidade do serviço.

Heterogeniedade É possível ter serviços que emulem diversos sistemas operacionais tradicionais simultaneamente numa mesma rede. Tais serviços são chamados de personalidades (*personalities*) de um SOD e se basciam nas primitivas do microkernel.

Simplicidade Devido ao reduzido número de funções, o microkernel tende a ser menor e muito mais simples do que o kernel de sistemas operacionais monolíticos, o que tem efeitos diretos sobre a capacidade de adaptar o SOD para novas arquiteturas.

A única desvantagem da arquitetura microkernel (que no entanto é fortemente contestada pelos seus defensores) é a sua performance, que sem dúvida é pior do que a de sistemas com kernel grande. Isto se deve ao fato de que em SODs com microkernel, o uso de um certo serviço, na melhor das hipóteses envolve 4 trocas de modo (kernel para usuário, e vice-versa) e duas mensagens, (figura 1), ao passo que em sistemas monolíticos geralmente só ocorrem 2 trocas de modo.

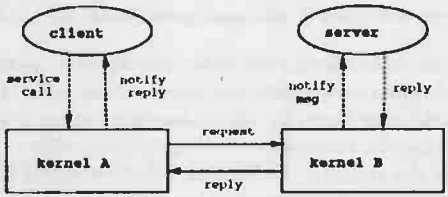

Figura 1: Acesso a um serviço remoto

Desta maneira, a performance de um SOD vai depender fortemente da rapidez na troca de modos (e contexto) e do protocolo usado para a comunicação entre o cliente e o servidor.

Tipicamente, as principais tarefas de um microkernel são:

- gerenciamento de interrupções, troca de contexto e escalonamento
- gerenciamento de memória
- gerenciamento de processos e/ou threads
- comunicação entre processos (local e remota)
- controle de dispositivos (drivers)

No entanto, não são todos os SODs que realizam todas estas funções no microkernel. Por exemplo, no sistema Amoeba, optou-se por implementar a maior parte do software de drivers na forma de processos executando em modo usuário. Em Plan9, por sua vez, optou-se por implementar boa parte do gerenciamento de processos na forma de um servidor em modo usuário.

Por outro lado, na maioria dos SODs, funções como sistema de arquivos, sistema de janelas, alocação de processadores e serviço de nomes são realizados através de servidores em modo usuário.

Algumas das funções do microkernel e de servidores serão abordadas com maior detalhe mais adiante, e depois veremos como os vários SODs estruturam estas funções.

2 O Microkernel

Apesar de haver algumas diferenças entre os microkernels dos diferentes SODs, principalmente no que diz respeito às funções por ele realizadas, pode-se identificar que algumas delas estão presentes em todos os microkernels. Estas são: o tratamento de interrupções, o escalonamento, o gerenciamento de memória, a comunicação entre processos e certo tipo de gerenciamento de processos/threads. A Figura 2 mostra um esquema simplificado do microkernel, onde as funções mais básicas aparecem na base.

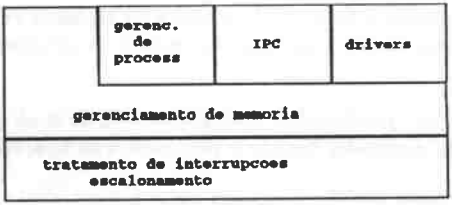

Figura 2: O Microkernel

Todas as funções do microkernel, exceto a comunicação entre processos (ou Inter Process Communication - IPC) e o gerenciamento de processos são implementadas usando essencialmente as mecanismos e políticas conhecidas de sistemas operacionais tradicionais. O IPC essencialmente consiste de uma componente que realiza a comunicação entre processos na mesma máquina (através da memória) e de uma componente que realiza a comunicação remota, mas sem que os processos do modo usuário percebam a diferença entre os dois tipos de comunicação. Na verdade, o IPC remoto pode ser visto como uma forma de E/S, no qual a máquina remetente está escrevendo dados em um dispositivo e a máquina receptora está lendo de um dispositivo externo. Por isto, a parte mais básica do IPC remoto nada mais é do que um driver de um dispositivo de E/S: a interface de rede.

Existe uma atual tendência de implementar microkernels configuráveis, isto é, cuja funcionalidade pode ser adicionada de forma modular. Com isto, pode-se adaptar melhor o SOD às características do hardware e das aplicações, eventualmente aumentando consideravelmente a performance do sistema. Por exemplo, para sistemas multi-computadores dedicados exclusivamente à aplicações paralelas de alto desempenho, pode-se optar por políticas bem simples (e eficientes) de gerenciamento de processos e de memória, pois há apenas um usuário e conseqüentemente não há a preocupação com o aspecto de segurança. De forma análoga, pode-se instalar em cada máquina apenas os drivers para os dispositivos efetivamente conectados à máquina. Exemplos de SOD com microkernels configuráveis são Chorus[16] e PEACE[18].

3 O gerenciamento de processos

Comparado com o que ocorre em sistemas operacionais para uniprocessadores, o gerenciamento de processos em SODs é bem mais complexo e custoso em termos de tempo de execução. A razão para isto está no fato deste gerenciamento envolver uma cooperação entre os microkernels nas diversas máquinas. Tal cooperação, que é realizada através do envio de mensagens, é necessária para implementar tanto os mecanismos básicos de criação remota de processos, como funções mais complexas, como por exemplo a escolha/verificação da máquina onde um novo processo será executado. Dentre as possíveis funções de gerenciamento, sem dúvida, a migração dinâmica de processos de uma máquina para a outra é a mais complexa e cara em termos de tempo de execução. Devido a este fato, não compensa utilizá-la para o balanceamento de carga no sistema, e por isto são poucos os SODs que a implementam.

Para possibilitar o gerenciamento de processos em SODs utiliza-se uma estrutura de dados descrevendo todo o estado de execução de um processo. Esta estrutura é o *descritor de processos (DP)*, que tipicamente contém o seguinte conjunto de informações acerca de um processo:

descritor da máquina contém os requisitos básicos do processo, e tipicamente contém o tipo da CPU (conjunto de instruções), tamanho mínimo de memória, dispositivos necessários, necessidade de co-processador

descritor de memória especifica o layout dos segmentos na memória virtual, incluindo, para cada segmento: o seu endereço inicial, o seu tamanho, sentido de crescimento, direitos de acesso e a localização em disco

descritor de threads contém, por thread, o conteúdo do PC, do stack pointer, dos registros e do estado dos system calls bloqueados (ponteiros para buffers de E/S, etc.)

descritor de arquivos abertos contém, para cada arquivo, o seu fileId e a posição corrente

Além destes itens, certos sistemas (ex. Amoeba) também incorporam as *capabilities* associadas ao processo dentro do DP. Capabilities são senhas criptografadas que determinam os direitos de acesso e manipulação de quaisquer objetos (arquivos, processos) por outros processos.

Além de um DP com formato igual para todo o sistema, o gerenciamento de processos e a migração requerem um compartilhamento global e transparente do sistema de arquivos em toda a rede. Isto porque o conteúdo dos segmentos usados por um processo em grande parte estará armazenado em arquivos e precisa estar acessível de qualquer máquina. Assim, dada a informação acerca do layout e da localização (p.ex. no disco) dos segmentos no descritor de memória do DP estes podem ser acessados de forma idêntica por todos os microkernels.

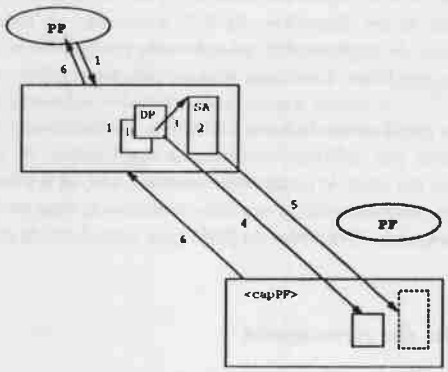

Figura 3: Criação remota de processos

Durante a criação remota de um processo (PF) por um processo pai (PP) são as seguintes as principais operações efetuadas no microkernel do processo pai (KP) e no microkernel do processo filho (KF).

1. PP faz um trap ao KP e fornece os argumentos de criação; KP faz uma cópia do DP do PP para o PF

2. KP cria um segmento especial (de ambiente), onde coloca os argumentos de criação e o valor das variáveis de ambiente (ex. BIN, USER, HOME, etc.)

3. KP altera o DP do PF[1], inserindo no descritor de memória o endereço do segmento de ambiente (este pode ser um arquivo ou a máquina do PP)

4. KP envia o DP para o KF, que aloca memória para o segmento de ambiente

5. KF copia o segmento de ambiente (da máquina pai ou de um arquivo) e inicia a execução de um dos threads do processo (carregando os registros, o PC e o SP correspondentes)

6. KF cria uma capability para o PF (<capPF>) e a envia para KP, que a repassa para o PP.

A seqüência acima descreve a criação *direta* de processos remotos, na qual a máquina destino é determinada pelo PP. Em muitos sistemas, têm-se, no entanto, uma criação *indireta* de processos, na qual há a intermediação por parte de um sistema de gerência de processos (Process Management System: PMS). O PMS é um serviço distribuído e pode tanto estar incorporado no microkernel, como pode estar implementado na forma de servidores em espaço usuário. As funções de um PMS diferem muito de sistema para sistema, podendo variar da simples verificação da disponibilidade da máquina destino escolhida pelo PP até a própria escolha desta, a partir de dados sobre a carga atual no sistema. Na criação indireta, portanto, de certa forma o PP delega a criação do PF ao PMS, que a efetua através de seu agente (servidor) na máquina destino. O quanto maior for o poder de decisão do PMS, mais transparente (de localização) será a criação remota de processos para o programa de aplicação.

3.1 Migração

Comparado com o mecanismo de criação remota de processos, a migração só difere nos seguintes aspectos:

- O pedido de migração de um processo P tipicamente não vem do seu processo pai e sim do PMS, que tem a informação sobre a carga das máquinas na rede.

- O PMS faz um system call `STUN`, que faz com que o kernel origem (KO) interrompa a execução de P e crie um novo DP para P, contendo o estado da CPU no momento da interrupção bem como os ponteiros para todos os segmentos de memória do processo P.

- Como no caso da criação remota, este DP é copiado para o kernel destino (KD)

- Ao contrário da simples criação, no caso da migração, no entanto, não apenas o segmento de ambiente, mas todos os segmentos de dado, de pilha, de heap, etc. têm que ser copiados para a máquina destino

- Os segmentos geralmente são copiados da máquina origem, mas em certos sistemas, como o Sprite[14], os segmentos de um processo a ser migrado são armazenados temporariamente em disco, o que simplifica a migração, mas aumenta o seu custo.

- Enquanto a execução de P não for reiniciada na máquina destino, o KO têm que interceptar todas as mensagens para P e pedir aos remetentes que enviem novamente as mensagens para o novo endereço.

[1] Se a criação de processos for do tipo *fork* (UNIX) o PF 'herda' os segmentos do PP, senão apenas o endereço do segmento de texto é alterado

Dado o custo relativamente alto da migração (devido a transferência de um número grande de segmentos), diz-se que ela só vale a pena quando o tempo restante de execução do processo na máquina origem for pelo menos 10 vezes maior do que o tempo gasto com a migração.

Há várias propostas para a otimização da migração, dentre as quais destacamos as usadas em MACH[23] e no sistema V[8]. Na primeira, propõem-se que inicialmente apenas o estado da CPU e as páginas do *working set* sejam copiadas para a máquina destino, fazendo-se a cópia das demais páginas quando estas forem requeridas (*copy on demand*). A segunda proposta é a de copiar de antemão todos os segmentos para a máquina destino, ainda durante a execução de P na máquina origem. Assim, no momento da migração propriamente dita, só se precisaria copiar os segmentos modificados desde o momento da cópia antecipada. Enquanto a primeira solução tem a desvantagem de eventualmente deixar o espaço de endereçamento de um processo espalhado pela rede durante um período longo de tempo, a segunda solução tem a desvantagem de que alguns segmentos certamente serão transferidos mais de uma vez.

Em resumo, migração invariavelmente é uma função custosa e só deveria ser usada em caso de necessidade, como por exemplo no caso de uma máquina executando processos vitais para o sistema ter que ser desligada.

Outras referências

Informações adicionais sobre o microkernel, o gerenciamento de processos e em especial, sobre a migração podem ser encontradas no capítulo 15 de [11].

4 Comunicação entre processos

A comunicação de processos é o aspecto que mais distingue sistemas operacionais tradicionais (para uni- e multi-processadores) dos sistemas operacionais distribuídos. A principal diferença está no fato de que nos primeiros a comunicação é implementada através do compartilhamento de memória e no segundo caso é através do uso de um meio de comunicação (ex: barramento, fibra ótica, etc.). Assim, a comunicação em sistemas distribuídos apresenta duas características particulares: ela gasta certo tempo e pode falhar. Infelizmente, programadores de sistemas distribuídos têm que conviver com estas características, mas é o papel do sistema operacional de tentar minimizar ao máximo as conseqüências destas características para o programador.

Nesta seção discutiremos três paradigmas de comunicação entre processos comumente encontrados nos SODs, a saber, o envio de mensagem, a chamada remota de procedimento e a comunicação de grupo.

4.1 Envio de mensagens

O envio de mensagem é a forma mais básica de comunicação entre processos de uma aplicação distribuída. Toda forma de comunicação é regida por um protocolo, ao qual as partes envolvidas (os kernels e os processos) têm que aderir. A principal diferença entre os protocolos de comunicação para WANs e LANs é que os primeiros são compostos de um grande número de camadas de serviço, onde cada camada é responsável pela garantia da comunicação com relação a certo problema (ex. correção de erros, controle de fluxo, roteamento, etc.), enquanto que os segundos tipicamente envolvem protocolos mais simples e menos tolerantes a falhas de comunicação, porém mais eficientes. A razão para isto é que os meios de comunicação em LANs geralmente são mais homogêneos (com relação às taxas de transmissão, ao tamanho dos pacotes, etc.), não requerem roteamento e são menos sucetíveis à perda de pacotes ou mensagens.

O protocolo básico para a comunicação em LANs é comumente chamado de *protocolo request-reply* e comparado com o modelo ISO/OSI[21], consiste em apenas dos níveis físicos, de enlace de dados e de sessão. Neste protocolo, um processo *cliente* envia um pedido (*request*) para outro processo *servidor*, que efetua a operação desejada e retorna um resultado ou confirmação (*reply*), conforme ilustrado na figura 1. Apesar da versão básica deste protocolo ser bastante simples, também por envolver apenas dois system calls, a saber *send(destino, &buffer)* e *receive(endereço, &buffer)*, existem as mais diversas variantes implementadas nos diversos sistemas.

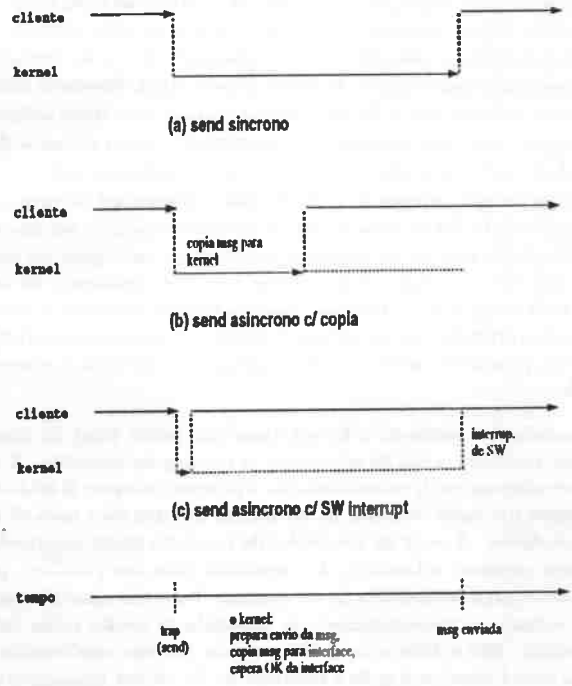

Figura 4: Primitiva send síncrona e asíncrona

A seguir, veremos as diferentes possibilidades com relação a formas de endereçamento, sincronismo, bufferização e confiabilidade.

Endereçamento A forma mais primitiva é incorporar no endereço *destino* o nome ou o número da máquina executando o servidor. Isto quebra a transparência de localização da comunicação, que traz consigo uma série de conseqüências negativas. A outra possibilidade é atribuir identificadores globais a processos ou caixas de correio (mailbox) e fazer com que o cliente envie uma mensagem de difusão (broadcast) questionando o endereço de rede do servidor antes do primeiro acesso ao mesmo. Uma vez obtido, este endereço será usado em todos os acessos subseqüentes. A terceira possibilidade é a implementação de um servidor de nomes, que mantém a informação da localização de cada servidor e que é consultado pelos clientes que desejem usar um determinado serviço. Apesar de se tratar de uma solução centralizada, ela é freqüentemente usada devido a

sua flexibilidade, pois possibilita incorporar certo poder de escolha no servidor de nomes com relação ao servidor mais adequado a cada cliente.

Sincronismo As primitivas de comunicação podem ser síncronas ou asíncronas. No caso do *send* síncrono, o cliente é bloqueado até que a interface de rede mande uma interrupção indicando que a mensagem foi enviada (figura 4(a)). Para aplicações que requerem alto grau de paralelismo, o *send* é implementado de forma assíncrona, isto é, o system call *send* retorna imediatamente e o processo pode prosseguir em paralelo (ou pseudo-paralelismo) com as atividades do kernel (montagem da mensagem, cópia para e controle da interface de rede, etc.). O principal problema do caso assíncrono é que enquanto a mensagem não tiver sido enviada, o cliente não deverá alterar o *buffer*, pois senão o conteúdo da mensagem poderá vir a ser corrompido. A solução mais freqüente é copiar *buffer* para dentro do kernel (figura 4(b)), liberando assim o acesso do cliente ao *buffer*, o que no entanto tem a desvantagem de causar uma cópia adicional e redundante dos dados da mensagem, visto que estes também são copiados para a memória da interface de rede. A outra possibilidade, que só é usada em sistemas especializados em aplicações de alto desempenho que requerem uma latência mínima de comunicação, é programar clientes sensíveis a interrupções de software (figura 4(c)). Neste caso, o cliente poderia prosseguir em seu processamento e seria avisado pelo kernel (através de tal interrupção) de que a mensagem foi enviada e que portanto o seu buffer pode ser re-escrito. Esta solução tem a desvantagem de tornar a programação dos processos bem complicada. Também existem versões síncronas e assíncronas da primitiva *receive*, sendo que a primeira é mais comum. Quando a primitiva é assíncrona, o SOD geralmente também tem uma primitiva adicional *wait*, usada para sincronizar o processo com a chegada de uma mensagem.

Buferização Na maioria dos sistemas, o kernel aloca um buffer (fila) de tamanho fixo para cada objeto (processo, mailbox), a fim de armazenar as mensagens recebidas. A ausência de buffers só é possível em sistemas nos quais garantidamente o processo receptor já está ativo antes de qualquer envio de mensagem e é capaz de tratar as mensagens em uma taxa mais alta do que a maior taxa de envio de mensagens. Apesar de um buffer de tamanho finito não resolver completamente o problema de uma eventual sobrecarga de mensagens para um processo, pelo menos é possível absorver uma sobrecarga temporária de mensagens. Para um controle mais efetivo de fluxo de mensagens, no entanto, o remetente deve ser impedido de enviar novas mensagens assim que o buffer estiver cheio. Isto é feito através da ausência de uma confirmação por parte do kernel receptor para o kernel remetente após o recebimento da última mensagem que encheu o buffer.

Confiabilidade A forma de envio de mensagem mais confiável é aquela no qual o recebimento tanto do request como do reply são confirmados pelo kernel receptor através de uma mensagem *acknowledgement (ACK)*. O problema é que neste caso, cada acesso ao servidor envolverá 4 mensagens. Outra opção é usar o próprio reply como confirmação do recebimento do request. O problema aqui é que o o kernel do servidor não terá como saber se o reply foi recebido pelo kernel do cliente. Além disto, será difícil definir qual deverá ser o tempo de espera (no kernel cliente) pelo reply, visto que este dependerá do serviço sendo executado no servidor. A solução mais freqüente para estes problemas é usar o reply como confirmação do request e fazer com que somente o recebimento do reply seja confirmado pelo kernel cliente. Neste caso, dependendo do serviço sendo executado, eventualmente o servidor não poderá atender a outros pedidos enquanto não receber a confirmação do kernel cliente. Para resolver o problema do timeout no kernel cliente, o servidor deve também ativar um timer ao receber o request e deve enviar uma confirmação do request caso não seja possível enviar o reply a tempo.

Em muitos sistemas existem, além do request e do reply, também outras mensagens que permitem um controle adicional sobre o estado dos servidores e do envio de mensagens, dentre as quais destacamos as seguintes:

Are You Alive (AYA) é usada pelo cliente para saber se o servidor continua ativo

I Am Alive (IAA) é a resposta do servidor à mensagem AYA

Try Again (TA) é usada pelo kernel servidor para indicar ao cliente que (a) o servidor está sendo incapaz de tratar de todos os pedidos ou (b) houve um problema temporário (p.ex. migração do servidor) no qual o servidor não pode tratar os pedidos

Address Unknown (AU) é usada pelo kernel do servidor para informar que o endereço destino indicado na mensagem não existe naquela máquina.

Outras referências

Mais informações sobre o envio de mensagens em geral podem ser obtidas em [11](cap. 9) e [19](cap. 4) e, para o caso de Unix, em [20].

4.2 Chamada remota de procedimento

A chamada remota de procedimentos (Remote Procedure Call ou RPC), inicialmente proposta em [6] é um mecanismo que visa adaptar o paradigma da programação estruturada (baseado na chamada de procedimentos) ao contexto de sistemas distribuídos. Seu principal objetivo é o de ocultar os detalhes de empacotamento, envio e recebimento de mensagens do programador, livrando-o assim do paradigma de entrada saída, que é pertinente ao envio de mensagens. Alguns autores chegam até a comparar o envio de mensagens simples ao estilo de programação seqüencial usando goto's. Neste sentido, o RPC representaria a solução para a programação estruturada de aplicações distribuídas.

A idéia central é obter exatamente a semântica da chamada de procedimento local para o acesso remoto a um serviço, fazendo com que o cliente não consiga distinguir se a chamada a uma função é local ou remota (*transparência de localização*). Todos os detalhes relativos à procura de um servidor, ao empacotamento e desempacotamento, ao envio de mensagem e à conversão da representação dos dados para a comunicação entre máquinas distintas, são tratados em *stubs* gerados automaticamente e fornecidos aos processos cliente e servidor sob forma de bibliotecas de execução. Portanto, para cada serviço no sistema, existe um stub no lado cliente (*client stub*) e um stub no lado do servidor (*server stub*).

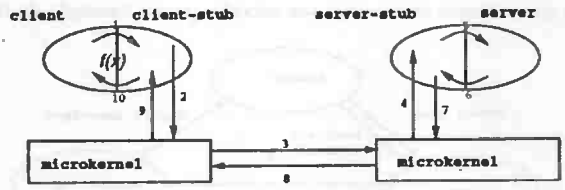

Figura 5: Caminho crítico do RPC

A Figura 5 mostra um modelo simplificado do funcionamento do RPC para uma função *f(x)* a ser executada por um servidor remoto. O client stub tem como tarefa empacotar e eventualmente

converter os argumentos da chamada da função *f*, para depois, usando um system call de envio de mensagem, enviar a mensagem contendo o endereço do servidor, o código da função e os seus argumentos para a máquina do servidor. Ao chegar ao kernel destino, este verifica a existência do server stub desejado (podem haver vários stubs em uma máquina), interrompe o processo correspondente e passa a mensagem para o server stub. Este desempacota (e converte) os argumentos de *f* e chama esta função localmente. Quando a função local retorna, o server stub empacota (e converte) o resultado de *f* em uma mensagem e envia esta ao client stub, através dos kernels. No lado cliente, então, o resultado de *f* é desempacotado da mensagem (e eventualmente convertido) e repassado ao programa que chamou o client stub. Todo este processamento e envio de mensagens envolvido em uma chamada RPC é chamada de caminho crítico (*critical path*). Os client e server stubs são gerados por um *stub generator* a partir de uma descrição da interface de um serviço (em termos das funções remotas que este serviço oferece).

4.2.1 Marshalling

A conversão dos dados (i.e. argumentos resultado da função), também chamada de *marshalling*, se torna necessária quando o RPC é usado em um sistema heterogêneo, seja a nível de hardware ou de linguagem de programação. No primeiro caso, diferentes máquinas podem apresentar uma diferença na numeração dos bytes de uma palavra (*big & little endian*), na codificação de caracteres (ASCII e EBCDIC), ou na representação de inteiros negativos ou números de ponto flutuante. No segundo caso, diferentes linguagens de programação podem apresentar diferenças na ordem dos argumentos na pilha para a chamada de uma função.

Para que seja possível fazer o marshalling, é necessário se conhecer o tipo e o sentido (de entrada, de saída, ambos) dos parâmetros da função, bem como as máquinas e as linguagens usadas em cada um dos lados. Basicamente, existem duas possibilidades para o marshalling. Uma é definir uma representação canônica de dados (sendo que a mais conhecida é a *ONC External Data Representation (XDR* da Sun) e fazer com que os argumentos (e o resultado) sejam sempre codificados antes de qualquer envio de mensagem e decodificados a cada recebimento de mensagens. A desvantagem óbvia desta solução é que tal conversão se torna supérfula quando a comunicação é entre máquinas linguagens iguais. A outra solução é converter os dados (geralmente no server stub) somente quando necessário, ou seja, quando a máquina linguagem do cliente difere da do servidor. Apesar de ser mais eficiente, esta solução tende a tornar os server stubs muito complexos, pois estes terão que conter procedimentos de conversão para todos os possíveis tipos de máquinas cliente.

4.2.2 Binding

Outro problema da implementação do RPC diz respeito à localização dos servidores, isto é, como o client stub sabe em que máquina se encontra um servidor para a execução da função remota?

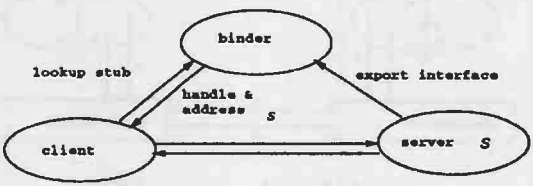

Figura 6: Endereçamento através do binder

Uma solução é ter o endereço do servidor na forma de uma constante dentro do client stub. Esta solução tem a desvantagem óbvia de requerer a alteração e recompilação de todos os client subs para um serviço sempre que o servidor correspondente for executado em uma máquina diferente. Uma solução mais flexível é implementar um servidor de nomes para o RPC, também chamado de *RPC server* ou *binder*(Figura 6). Neste caso, todo novo servidor inicialmente comunica (*exporta*) a sua interface e seu endereço ao binder. Então, à primeira vez que um cliente chama uma função remota, o client stub envia uma mensagem especial *lookup* para o binder indicando o stub que deseja acessar remotamente. O binder verifica a existência de um servidor para o serviço desejado e retorna um *handle* e o endereço de rede do servidor. O handle recebido pelo client stub serve para identificar unicamente o server stub em uma máquina. De posse destas informações, o client stub pode mandar as mensagens diretamente ao server stub e não precisa mais acessar o binder para as chamadas subseqüentes às funções do serviço. Outra vantagem do binder é que este pode incorporar as funções de um negociador (*trader*), que fornece o endereço do servidor mais adequado para cada cliente, como por exemplo, o mais próximo ou menos carregado.

4.2.3 Vantagens e Desvantagens

Portanto, além de prover um grau de abstração maior para a comunicação entre processos, o RPC ainda oferece as seguintes vantagens:

- Os stubs não precisam ser programados, e sim são gerados a partir de uma especificação de interface de alto nível
- Pode-se escolher qual é protocolo de comunicação a ser usado pelos stubs. Tal escolha geralmente depende das características da rede e da aplicação. Em redes locais o mais comum é usar o UDP/IP.
- A conversão da representação da dados em sistemas heterogêneos é feita de forma transparente para o usuário.
- Fica também transparente para o usuário qual é o servidor que estará executando a função. Assim, um serviço pode ser realizado por um grupo de servidores sem que o cliente tenha que tomar conhecimento deste fato.

No entanto, o RPC não consegue ter exatamente a semântica da chamada local de funções, pelos seguintes motivos:

- Uma função remota não consegue ter acesso a variáveis globais do programa ou da função que a chama.
- Não é possível definir funções que tenham como parâmetro vetores de tamanho indefinido. Para o RPC, é necessário indicar o tamanho máximo de todos os vetores que sejam argumentos da função.
- O custo operacional de passar argumentos a uma função remota que sejam estruturas de dados complexas envolvendo ponteiros é altíssimo. Devido a este fato, são pouquíssimos os sistemas que implementam esta funcionalidade.
- O RPC não pode ser usado para funções com um número variável de argumentos, como é, por exemplo, o caso do *printf* de C.

- Em sistemas em que há a possibilidade de ocorrência de falhas (de comunicação ou quebra de nós da rede) o RPC poderá ter ou a semântica "*pelo menos uma execução*" ou a semântica "*no máximo uma execução*", mas nunca conseguirá apresentar a semântica "*exatamente uma execução*" da chamada de funções locais. Enquanto no primeiro caso o stub cliente envia repetidamente a mensagem request até receber o reply do stub servidor, no segundo caso, apenas uma tentativa é feita para garantir que o servidor não tenha executado a função mais de uma vez. Tal garantia é imprescindível para certas aplicações, como por exemplo em transações bancárias.

Outro problema é que o RPC é intrinsecamente síncrono (o cliente é bloqueado até que a função seja executada) e portanto não serve para modos de interação asíncronos. Para suprir esta deficiência, existem trabalhos propondo formas de RPC asíncrono[2].

Além disto, há o problema de que o custo operacional do RPC tende a ser maior do que a do simples envio de mensagens, principalmente devido às múltiplas cópias e conversões dos argumentos e do resultado feitas pelos stubs.

Outras referências

O trabalho original propondo o RPC está em [12], e foi resumido em [6]. Como o RPC já foi implementado em muitos sistemas (de formas ligeiramente diferentes), são muitas as referências existentes sobre o assunto. Para uma apresentação geral, recomendamos, além do próprio *Modern Operating System*, o capítulo 9 de [11], e para um estudo detalhado sobre o RPC da NCS (Network Computing Architecture), da Sun e do OSF/DCE recomendamos [7].

4.3 Comunicação de grupo

A comunicação de grupo (CG) é outro mecanismo de interação entre processos que fornece ao usuário um grau de abstração maior do que o simples envio de mensagens. Ao contrário do RPC, no entanto, a CG tem como objetivo tornar a *replicação* transparente ao programador. Usando a CG, um processo cliente interage simultaneamente com todos os membros de um grupo de processos, como se eles fossem apenas um processo. Muitas aplicações e algoritmos distribuídos se baseiam em CG, como por exemplo serviços tolerantes a falhas implementados através de servidores replicados, sincronização de relógios, eleição de um coordenador, etc.

A facilidade de se implementar a CG depende fortemente do suporte à difusão de mensagens dado pelo meio de transmissão. Quando disponível, a CG pode utilizar diretamente a capacidade de difusão selecionada de mensagens (*multicast*), que por exemplo existe em redes Ethernet. Em outras redes com topologia de barramento que apenas têm a capacidade de difusão global para todas as máquinas (*broadcast*), o envio selecionado tem que ser implementado a nível de software, através do tratamento de interrupções geradas pela interface de rede. Ou seja, neste caso todas as máquinas sofrem uma interrupção de sua interface de rede, mas somente nas máquinas selecionadas, o kernel repassa a mensagem para o processo de aplicação. Por fim, em redes que não possuem nenhum suporte à difusão de mensagens, a CG tem que ser realizada através de uma série de envios ponto-a-ponto.

Além de primitivas do tipo *send_grp* e *receive_grp*, a CG requer primitivas adicionais para o gerenciamento de grupos, dentre as quais destacamos as mais comuns.

Primitiva	Descrição
create_grp()	criação de um grupo (fornece um handle *gh*)
destroy_grp(gh)	deleção do grupo
join(gh, pid)	entrada do processo *pid* no grupo *gh*
leave(gh, pid)	saida do processo *pid* do grupo *gh*

Além de poderem usar diferentes formas de envio de mensagem (vide seção 4.1), grupos podem diferir com relação ao direito de uso da CG, à organização e ao gerenciamento de um grupo.

Com relação ao direito de acesso à comunicação de grupo, podemos distinguir entre grupos *abertos* e grupos *fechados*. Enquanto em grupos abertos é permitido também a processos externos enviar uma mensagem para todos os membros de um grupo, em grupos fechados, somente os processos membros de um grupo podem efetuar tal operação. Note-se que isto não impede que possam haver interações ponto-a-ponto (envio de mensagem, RPC) entre membros e não-membros de um grupo.

Além disto, grupos podem ter uma organização plana ou hierárquica. No primeiro caso, todos os membros do grupos exercem exatamente as mesmas funções com relação ao funcionamento da comunicação de grupo, isto é, todos os membros são encarregados de dissipar mensagens aos demais membros (e a tratar de pedidos de inclusão de novos processos). Na organização hierárquica, existe um processo com a função de *coordenador* do grupo, que é encarregado de repassar todas as mensagens provenientes de outros processos aos demais membros do grupo, incluindo as mensagens de pedido de inclusão ou exclusão. A principal vantagem de grupos planos é o seu alto grau de tolerância à falha de membros do grupo, visto que todos exercem exatamente a mesma função. Em grupos hierárquicos, por sua vez, o coordenador é um elemento central vital ao funcionamento do grupo, que se tornar um gargalo na comunicação com o grupo. Além disto, a falha do coordenador desencadeia um processo de escolha de um novo coordenador e até que este esteja estabelecido, o grupo fica inacessível. Em compensação, grupos hierárquicos tipicamente requerem um menor número de interações entre os seus membros para fins de gerenciamento do grupo. Em Amoeba, optou-se por uma CG com grupos fechados e hierárquicos.

Também com relação a pedidos de inclusão e exclusão de elementos de grupo, tem-se a opção entre uma solução centralizada e fechada, ou uma solução descentralizada e aberta. No primeiro caso, usa-se um servidor de grupo (*group server*), que recebe todos os pedidos de inclusão e exclusão de membros a um grupo e que contém todas a informação com respeito aos atuais membros de um grupo. Em muitos casos, este servidor é o próprio coordenador do grupo. Alternativamente, pode-se implementar a adesão ou saída de um processo de um grupo através da difusão desta informação a todos os membros do grupo (*anúncio público*).

4.3.1 Características desejáveis

Para que a CG realmente realize a abstração a qual se propôs, é necessário que o mecanismo que a implemente tenha as seguintes características:

Confiabilidade É desejável que todas as mensagem de grupo realmente alcancem todos os membros do grupo. Em especial, é importante garantir que se um dado membro do grupo não recebeu uma mensagem, então todos os demais membros descartem esta mensagem. Esta característica do *tudo-ou-nada* é fundamental para muitos algorítmos distribuídos.

Ordenação consistente A outra característica desejável é que todos os membros de um grupo recebam as mensagens numa mesma ordem. A ausência desta característica pode causar problemas quando é necessário que todos os processos de um grupo mantenham os seus estados de processamento iguais e sincronizados, como por exemplo em serviços tolerantes a falha implementados através de *réplicas ativas*. A ordenação consistente de mensagens pode pode ser requerida em diversos graus de exigência:

- O mais forte é a *ordenação baseada em tempo global*, na qual se exige que as mensagens sejam recebidas exatamente na ordem absoluta de tempo que os respectivos *send_grp()* ocorreram.

- A exigência um pouco mais fraca é que o recebimento seja em ordem consistente com o tempo. Neste caso, exige-se apenas que as mensagens sejam recebidas por todos os membros numa dada ordem estabelecida por um elemento sequencializador, que pode ser tanto um processo, como o meio de transmissão.
- A exigência mais fraca é a *ordenação consistente com a causalidade*, que apenas impõe o seguinte: Se o envio da mensagem m_2 é direta- ou indiretamente dependente do envio da mensagem m_1, então todos os membros do grupo necessariamente têm que receber m_1 antes de m_2.

Sistemas como o Isis e Horus permitem escolher entre os diversos graus de ordenação.

Inclusão e exclusão síncrona Uma terceira característica importante é a de que a inclusão e a exclusão de um membro do grupo sempre aconteça de forma sincronizada com o recebimento de mensagens, evitando assim a incerteza quanto ao recebimento por um membro que esteja se juntando ao grupo em um momento de de difusão de mensagem.

Quando uma CG tem as primeiras duas características, diz-se que ela é uma *comunicação de grupo atômica.*

Dependendo dos requerimentos de tolerância à falha, consistência de répicas e de performance, diferentes áreas de aplicação podem eventualmente requerer apenas uma ou outra das características acima (ou versões mais fracas). Por exemplo, em Amoeba pode-se definir um *grau de flexibilidade* para cada grupo no sistema, e que e determina o número máximo de membros do grupo que podem deixar de receber uma mensagem de grupo.

Além disto, existem implementações de CG com outras características como por exemplo *limite de tempo máximo* para a difusão, que satisfazem as necessidades de aplicações de tempo-real ou de alta-performance nas quais não há necessidade de se garantir a atomicidade.

Como a realização das características *confiabilidade* e *ordenação consistente* envolve um custo adicional de processamento não desprezível, e como nem todas as aplicações necessitam tais características, está se permitindo cada vez mais a configuração da CG às necessidades particulares da aplicação. O principal exemplo disto é o sistema Horus, que implementa uma CG bem mais flexível do que o seu sistema antecessor, Isis.

Uma outra preocupação é com a escalabilidade da CG, uma vez que aplicações distribuídas estão usando sistemas com um número cada vez maior de componentes e estes estão distribuídos em uma área geográfica cada vez maior. Neste contexto, é provável que mais e mais sistemas passem a adotar uma organização hierárquica de vários níveis.

Outras referências

Como referência sobre os diferentes tipos de CG, recomendamos a leitura de [11](cap. 5), e para a implemantações particulares em Amoeba e em Isis, recomendamos [10] e [5], respectivamente.

5 Conclusão

A pesquisa em sistemas distribuídos, e em particular em sistemas operacionais distribuídos tem mostrado uma evolução muito acelerada nos últimos tempos, a cada ano abordando novos problemas e apresentando soluções inovadoras com relação a estruturação de sistemas, serviços, protocolos, paradigmas de interação e ambientes de programação.

Desta forma, qualquer texto sobre este tópico invariavelmente em pouco tempo torna-se ultrapassado se o mesmo só abordar os conceitos e os sistemas atuais. Por isto, é fundamental que se identifiquem as tendências mais significativas nesta área.

O aproveitamento do conhecimento adquirido na pesquisa em produtos comerciais notoriamente está sempre bastante defasado no tempo. No caso particular de sistemas operacionais, por exemplo, somente nos últimos anos firmas como Microsoft, Novell, Apple e IBM e outras estão mudando os seus sistemas operacionais para uma arquitetura microkernel, apesar de suas vantagens já estarem comprovadas há quase uma década. Assim, as muitas das idéias e tendências observadas hoje em dia na pesquisa influirão no projeto dos futuros sistemas.

De um modo geral, pode-se perceber as seguintes tendências na área de suporte ao processamento distribuído:

Configurabilidade A capacidade de um sistema poder ser configurado às características e necessidades particulares de usuários com problemas em diversas áreas de aplicação está se tornando um fator cada vez mais relevante para o sucesso de um sistema. Como se viu, muitos sistemas recentes já permitem uma configuração do próprio microkernel (ex. Peace), e dos mecanismos de interação, como o RPC (ex. Amoeba) e da CG (ex. Horus). Em diversos sistemas, o alto grau de configurabilidade é realizado através de uma implementação orientada a objetos.

Crescente grau de abstração Percebeu-se que abstrações como o RPC, CG e transações atômicas (não discutidas neste texto) podem facilitar muito a tarefa de programação de sistemas distribuídos. Por isto, continua-se pesquisando por outros paradigmas de programação abstratos que facilitem ainda mais o desenvolvimento de sistemas distribuídos. Um exemplo é a programação baseada em notificações[3]. Além disto, há uma crescente preocupação em aliviar o programador da implementação de serviços básicos, como por exemplo o serviço de nomes, o serviço de sincronização de relógios, serviço de arquivos, e outros etc., que também tornam a distribuição e a replicação do programa transparente para o programador. Tais serviço básicos já estão incorporadoe em plataformas como o OSF/DCE[15], ANSAware[1], CORBA[13], desenvolvidas por diferentes consórcios.

Heterogeneidade Devido à crescente necessidade de interação entre diferentes sistemas (hardware e sistemas operacionais) vem-se investindo muito trabalho no desenvolvimento das chamadas *middleware*, que implementam de forma padronizada um conjunto de serviços básicos, como RPC, threads, sincronização, serviço de nomes, etc. em cima de muitos sistemas operacionais. Assim, ambientes como o OSF/DCE, ODP, ANSAware já provêm uma API (interface para programas de aplicação) padrão, só que não a nível de microkernel, mas a nível de bibliotecas e servidores rodando em diferentes sistemas operacionais. Com o objetivo de esconder as diferenças entre estas instâncias de middleware acima mencionadas e de fornecer um modelo de programação unificado e mais abstrato estão se desenvolvendo também *middlewares* de serviços mais elevados, como por exemplo o CORBA (Common Object Request Broker Architecture)[13], que provê um modelo orientado a objetos, e o ambiente em desenvolvimento no projeto CORDS[4].

Sistemas Dinâmicos Com o surgimento dos labtops e palmtops, abriu-se todo uma nova área de pesquisa voltada para a computação desconectada (*mobile/wireless computing*), que visa dar suporte ao uso destes computadores em uma rede. Como estas máquinas não estão fixas em um dado ponto da rede, eles não permitem um endereçamento no esquema da atual internet, no qual o endereço IP de uma máquina está atrelado ao endereço da subrede em que se encontra. Além da elaboração de novos protocolos de encaminhamento e localização, pesquisa em sistemas operacionais para tais sistemas também tem que levar em conta aspectos como quantidade limitada

de energia (da pilha) e de banda de comunicação, e reduzido espaço em disco para o cacheamento de arquivos acessados através da rede. Para uma introdução na área, recomenda-se a leitura de [9] e [17].

Roteiro

Os artigos a seguir apresentam um resumo de diversos sistemas operacionais distribuídos. O primeiro, de Celia S. Fukusawa, trata das principais características do **V System**, um dos primeiros sistemas a implementar um RPC e comunicação de grupo.

Depois vêm três artigos sobre o **Mach**, sendo que no primeiro Sergio Borger apresenta um resumo das principais características e da arquitetura geral do sistema, e nos dois artigos seguintes Fabio Kon aborda serviços de gerenciamento de memória especiais implementados em cima do microkernel Mach.

No artigo seguinte Fabio Nakano dá detalhes sobre o suporte para a migração de processos no sistema operacional **Sprite**. A seguir, Vanderlei S. Porcel apresenta o sistema comercial **Chorus**, que ao lado de Mach é um dos mais conhecidos.

Em seguida, Jorge Nakahara Jr. apresenta o sistema **Isis** e Eugenio Nasser o sistema sucessor **Horus**. Ambos têm como ponto forte o suporte à diversas formas de comunicação de grupo. Rodrigo Braz então, apresenta o sistema **Synthetix**, que o possibilita a síntese e a configuração do kernel com o objetivo de otimizar as operações mais freqüentes.

Por fim, Alfredo Goldman apresenta o **OSF/DCE**, que apesar de não ser um sistema operacional propriamente dito, provê os mesmos serviços de um SOD, só que no topo de diversos sistemas operacionais.

Referências

[1] Advanced Networked Systems Architecture, Cambridge, UK. *ANSA Reference Model*, release 1.00 edition, March 1989.

[2] A.L Anada and B.H. Tay. A Survey of Asynchronopus Remote Procedure Calls. *ACM Operating Systems Review*, 26(2):92–105, April 1992.

[3] J. Bacon, J. Bates, R. Hayton, and K. Moody. Using Events to Build Distributed Applications. In *Proc. 2nd International Workshop on Services in Distributed and Networked Environments, Whistler, CA*, pages 148–155. IEEE, June 1995.

[4] M.A. Bauer, N. Coburn, D.L. Erickson, P.F. Finnigan, J.W. Hong, P.-A. Larson, J. Pachl, J. Slomin, D.J. Taylor, and T.J. Teorey. A Distributed System Architecture for a Distributed Application Environment. *IBM Systems Journal*, 33(3):399–425, 1994.

[5] K.P. Birman. The Process Group Approach to Reliable Distributed Computing. Tech. Report 1216, Dept. of Computer Science, Cornell University, Ithaca, USA, 1993. ftp.cs.cornell.edu:/pub/isis.

[6] A.D Birrel and B.J. Nelson. Implementing Remote Procedure Calls. *ACM Trans. on Computer Systems*, 2:29–59, February 1984.

[7] J. Bloomer. *Power Programming with RPC*. O'Reilly & Associates Inc., 1992.

[8] David R. Cheriton. The V distributed system. *Communications of the ACM*, 31(3):314–333, March 1988.

[9] T. Imielinski and B. R. Badrinath. Wireless Mobile Computing: Challenges in Data Management. *Communications of the ACM*, 37(10):19–28, October 94.

[10] M.F. Kaashoek, A.S. Tanenbaum, and K. Verstoep. Group Communication in Amoeba and its Application. *Distributed Systems Engineering Journal*, 1(1):48–58, July 1993.

[11] S. Mullender. *Distributed Systems.* Addison Wesley, 1993.

[12] B.J. Nelson. *Remote Procedure Call.* PhD thesis, Dept. of Computer Science, Carnegie-Mellon University, 1981.

[13] Object Management Group. *The Common Object Request Broker Architecture and Specification*, omg document 91.12.1 edition, December 1991.

[14] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. *IEEE Computer*, 21(2):23–36, February 1988.

[15] W. Rosenberry, D. Kenney, and G. Fisher. *Understanding DCE.* O'Reilly & Associates Inc., 1992.

[16] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemont, F. Herrmann, C. Kaiser, S. Langlois, P. Leonard, and W. Neuhauser. Overview of the CHORUS Distributed Operating Systems. Technical Report 25, Chorus Syste'mes, 1990.

[17] M. Satyanarayanan. Mobile computing. *IEEE Computer*, 26:81–82, 93.

[18] W. Schroeder-Preikschat. PEACE - A Distributed Operating System for High-Performance Multicomputer Systems. In *Proc. European Workshop in Progress in Distributed Operating Systems and Dsitributed Systems Management, LNCS 433*, pages 22–43, 1992.

[19] M. Sloman and J. Kramer. *Distributed Systems and Computer Networks.* Prentice-Hall International, 1987.

[20] W.R. Stevens. *UNIX Network Programming.* Prentice Hall, 1990.

[21] A.S. Tanenbaum. *Computer Networks.* Prentice Hall, 1988.

[22] A.S. Tanenbaum. *Modern Operating Systems.* Prentice Hall, 1992.

[23] E.R. Zayas. Attacking the Process Migration Bottleneck. In *Proc. 11th Symposium on Operating System Principles*, pages 13–24. ACM, 1987.

V Distributed System

Célia Satie Fukusawa

1 Introdução

O V System foi desenvolvido pelo grupo de sistemas distribuídos da Universidade de Stanford, California, Estados Unidos, como parte de um projeto de pesquisa que visava explorar as características dos sistemas distribuídos. Dentre estas características podemos citar a crescente demanda de aplicações inerentemente distribuídas, como os utilizados em sistemas bancários; a possibilidade de crescimento incremental; a maior confiabilidade; o compartilhamento de dados e recursos; entre outros.
Para implementação destes sistemas distribuídos foram criados novos conceitos e mecanismos, como por exemplo a execução de multiprocessos que executam em um ou mais máquinas e que cooperam mutualmente através de troca de mensagens via uma rede local; o RPC ("Remote Procedure Call") que foi criado como um novo conceito que possibilitava a utilização da comunicação entre processos em diferentes máquina de modo transparente à aplicação e a migração de processos que permite a melhor distribuição de carga entre as máquinas da rede.
Teve como principal responsável o Dr. David R. Cheriton do Departamento de Computação da Universidade de Stanford.
O V System é um dos precursores de sistemas operacionais distribuídos atuais tendo sido apresentado pela primeira vez em 1983. O principal objetivo era como em todos os sistemas distribuídos criar um sistema que possibilitasse um melhor aproveitamento dos recursos disponíveis ("hardware", "software" e dados), do que em sistemas centralizados ("mainframes") e computadores pessoais, onde cada máquina é alocada a um indivíduo e somente este tem acesso a seus recursos. Isso foi possível devido à uma crescente disponibilidade de equipamentos e tecnologias de alta performace e baixo custo, como estações de trabalho ("workstations") e redes locais, cada vez mais rápidas e confiáveis [2].
Em termos históricos o V System descende do Thoth que é um sistema operacional portável de tempo real. O Thoth [4] foi desenvolvido na Universidade de Waterloo, também pelo Dr. Cheriton. Sua primeira versão apareceu em 1976, rodando em dois minicomputadores (Texas Instruments 990 e Data General Nova). O suporte a multiprocessos concorrentes que se comunicavam via IPC ("Interprocess Communication") serviu de base para o desenvolvimento do V System.

2 Arquitetura Geral do Sistema

O V System é um sistema operacional projetado para trabalhar em uma rede de "workstations" conectados por uma rede local de alta performance. O sistema é composto de um microkernel, módulo de serviços, biblioteca de execução e um interpretador de comandos. Destaca-se por ser um dos precursores a se utilizar da arquitetura microkernel, suas vantagens e desvantagens serão discutidas na próxima seção. Em cada nó da rede executa uma cópia do microkernel tornando a existência de multiplas máquinas e da rede de conexão transparente para os processos de aplicação.

3 Microkernel

O V System foi criado baseado no conceito de um microkernel. A idéia principal era criar um suporte com funções básicas capazes de atender a serviços compexos que fossem implementados fora do kernel, em analogia a uma placa de hardware, onde a placa mãe fornece tensão, dados, endereços para a conexão de placas específicas como por exemplo de teclado, disco e vídeo. A função do microkernel é de permitir uma maior flexibilidade, confiança, configuração e fácil manutenção devido a sua simplicidade. Esta arquitetura porém apresenta um tempo de resposta maior a comparada a sistemas onde o kernel executa funções mais completas e complexas.
O kernel provê apenas funções de gerenciamento de tempo, de processos, de memória, a comunicação entre kernels e o controle de dispositivos.
O V kernel mantém o tempo corrente, e as funções de get/set time e delay são disponíveis para os processos locais em cada nó. A sincronização de máquinas é implementada por um processo externo ao kernel.
O gerenciamento de processos provê as funções: *Create*, *Destroy*, *Query*, *Modify* e *Migrate process*. Estas funções serão descritas com mais detalhes na próxima seção.
O gerenciamento de memória no kernel tem a função de garantir a a integridade da memória alocada a um processo evitando acessos maliciosos ou acidentais por parte de outros processos. Cabe ao gerenciador de memória a alocação de espaço de memória para os programas quando em sua execução. Porém a cópia do código dos programas e seu mapeamento em memória é realizada somente no momento da execução conforme são referenciadas.
Os dispositivos (disco, interface de rede, mouse, teclado, fita magnética , linha serial, etc) possuem servidores específicos, mas o kernel implementa a parte mais básica do controle a fim de gerenciar as interrupções, executar operações com privilégio e manter a integridade.

4 Gerenciamento de Processos

Como dito anteriormente o gerenciamento de processos implementa as funções de criação, destruição, modificação, migração e obtenção do estado de processos.
A inicialização de processos é simples bastando ao kernel alocar e inicializar um novo descritor de processo, sendo que a criação e a inicialização do espaço de endereçamento é feita pelo gerenciador de memória como visto no item 3. A terminação de processos caracteriza-se por sua simplicidade pois a maior parte dos recursos são gerenciados a nível de processos, isto é de tempos em tempos cada servidor se encarrega de verificar se o recurso está alocado a um processo morto. Como exemplo temos o servidor de arquivos que possui um coletor de lixo que a cada intervalo de tempo fecha arquivos alocados a processos mortos.
O kernel provê apenas prioridade simples na alocação de processos. Outros modos de alocação (por exemplo "time sharing") são implementados a nível de processo.
O tratamento de exceções é feito externamente ao kernel. No caso de erro o kernel detecta e envia uma mensagem para o processo responsável pelo tratamento. E finalmente, o suporte para a migração de processos, onde o estado de um processo a nível de kernel origem são obtidas e copiadas para o kernel destino. Funções de freeze e unfreeze foram introduzidas para suportar a migração.

5 Comunicação

O V System permite os seguintes modos de comunicação: envio de mensagens síncrono, RPC ("Remote Procedure Call") e comunicação em grupo.
O envio de mensagens segue o protocolo VMTP ("Versatile Message Transaction Protocol") que basicamente é o protocolo "request-reply".
Neste protocolo o processo cliente executa um comando *Send* que bloqueia o processo até o recebimento de uma resposta ("acknolegment" do servidor). Por outro lado o processo servidor executa um comando *Receive* e fica bloqueado até receber uma mesagem requisitando o seu serviço (mensagem gerada pelo *Send* to cliente), ao receber a mensagem o servidor executa o serviço e gera um *Reply/Return* que é a mensagem de resposta ao processo cliente (figura 7).

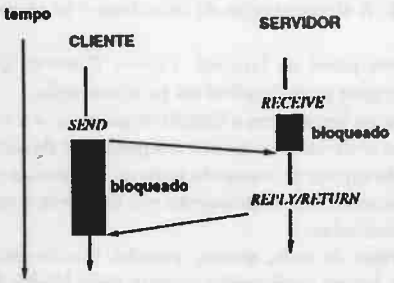

Figura 7: Comunicação Básica Usando V IPC

A comunicação entre processos pode também ser realizada através de RPC ("Remote Procedure Call") onde o local de execução (se remoto ou local), é transparente para o usuário. Quando um serviço é chamado por RPC o kernel verifica se o serviço pode ser executado localmente, caso não o kernel gera a interface correspondente para execução remota (stub cliente), localiza um servidor remoto e envia a mensagem até o servidor via rede, como ilustrado na figura 8.

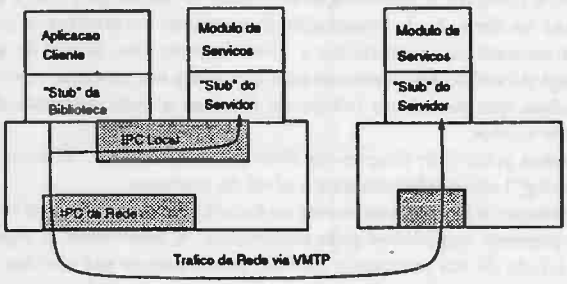

Figura 8: Comunicação Remota e Local

O protocolo VMTP possui algumas características interessantes como a noção de "Conversation" que implica na ação a nível de aplicação para a comunicação o que contrasta com a implementação

convencional onde as conecções para a comunicação é feita a nível de transporte. Esta política visa possibilitar o mínimo de redundância e uma comunicação flexível e de alto nível de forma a facilitar a inclusão de operações como a comunicação de grupo [3].

5.1 Comunicação de Grupo

O V System permite a comunicação em grupo ("multicast") onde é permitida a transmissão de um pacote, uma mensagem ou um RPC para um grupo de processos. Muitas são as aplicações para a comunicação em grupo, como por exemplo a procura de um nome, a sincronização de relógios, programas paralelos, gerenciamento de transações atômicas, replicação e outros.
O V System foi um dos primeiros sistemas a definir o conceito de grupo de processos. Cada grupo é descrito por um identificador de grupo que, como o identificador de processos possui 32 bits. O bit 17 indica se o identificador é um grupo (valor 0) ou de processo (valor 1). Na primeira versão do V System os bits mais significativos do identificador indicavam se o grupo era local ou não (figura 9) [1]. O grupo local foi definido como um grupo em que todos os processos do grupo pertencem ao um mesmo "host". Um grupo pode ser definido como restrito, no qual somente membros do grupo podem comunicar entre si, ou irrestrito, quando é desejável que processos externos ao grupo possam receber e/ou mandar mensagens para o grupo. Além disto, um grupo pode ser local ou global. Entende-se como grupo global aquele que possui membros distribuídos em vários pontos da rede.

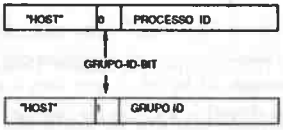

Figura 9: Identificador de grupos e processos

Cabe ao kernel administrar a localização de todos os processos de um grupo. São operações de grupo [1]:

Comando	Descrição
CreateGroup	criação de um novo grupo
JoinGroup	inclusão de um processo a um grupo já existente ou agrupamento de dois grupos
LeaveGroup	remoção de um processo do grupo
QueryGroup	obteção de informações do grupo
Send	envio de uma mensagem para todos os processos de um grupo
GetReply	obténção de resposta de um grupo. Pode-se definir o número de respostas desejadas, isto é, quantos membros do grupo devem responder para que o processo cliente seja desbloquado
Reply	envio de resposta
DestroyProcess	destruição de um grupo
ForceException	execução de um procedimento de excessão por todos os elementos do grupo

Para a implementação da comunicação em grupo definiu-se o conceito de grupo de "hosts", que é o grupo de todos os "hosts" que possuem pelo menos um processo do grupo. Este grupo de "hosts" deve satisfazer os seguintes requisitos:

- um simples pacote de endereço é necessário para endereçar todos os "hosts" pertencentes ao grupo, de forma análoga à comunicação ponto a ponto;
- a transmissão de uma mesagem para um "host" não necessita ser totalmente confiável, porém é necessário garantir que através de algumas poucas retrasmissões todos os "hosts" recebam pelo menos uma cópia da mensagem;
- seja possível a comunicação entre todos os membros do grupo de "host".

A comunicação de grupo no V System se utiliza do protocolo de rede apenas os níveis "network" e "datalink", provendo desta forma uma simples implementação e uma alta performance. Para a comunicação entre multiplas redes, há três opções para a conecção: Pontes ("Bridges"), "Internetwork Gateways" e "Transport-level Gateways" [3].
Utilizam-se Pontes quando as multiplas redes possuem o mesmo espaço de endereçamento e o mesmo formato de pacotes. É necessário também que a interconexão entre as redes sejam aciclicas, visto que cada Ponte envia todos os pacotes de "multicast" que recebe, assim no caso de uma rede ciclíca uma ponte receberia o mesmo "multicast" que enviou e esta mandaria novamente para a rede origem, criando um efeito de vai e volta levando a sobrecarga da rede. Nesta implementação as multíplas redes são vistas como uma rede única, sendo desta forma construídas apenas por limitações físicas, como por exemplo comprimento da rede.
A comunicação através de "Internetwork Gateways" traz uma série de problemas. Para transmitir os pacotes de comunicação do VMTP, estes tem que ser inseridos em pacotes para o envio através da rede. Isto requer um acréscimo de pelo menos 20 bytes, o que leva a um "overhead" de 25% e a perda de tempo na criação do cabeçalho e na demultiplexação do pacote. Outro importante fator é a falta de suporte de "multicast".
"Transport-level Gateways" baseiam-se em "gateways" inteligentes, nas quais o transporte é feito através de duas "gateways" ao invés de uma "gateway" e os "end-points", chamamos de "end-points" todos os objetos aos quais podem receber uma mensagem: modulos, "ports" ou "hosts". Para que um processo B possa comunicar com outro externo a sua rede local, processo A, deve ser criado um processo "alias" A' para o processo destino no "gateway" local. Como mostra a figura 10.

Os "gateways" locais e remotos são responsáveis pela tradução dos nomes e o transporte de dados entre as redes. O uso dessa arquitetura de "gateways" é transparente a nível de processos. Muitas são as vantagens desta implementação. Os "gateways" podem controlar todo o fluxo entre redes através dos "alias" o que permite implementar um sistema de segurança com autorização de acessos e prevenir "multicasts" não autorizados. Outro fator é a ausência de custos na rede local, pois todos os pacotes são manuseados pelos respectivos "alias" e transmitidos como em um processo local, não existindo a necessidade de um cabeçalho ou código de manipulação no pacote. Outra característica que torna esta implementação interessante é a possibilidade de incluir uma série de otimizações na comunicação entre redes, como por exemplo na mudança da taxa de transmissão ("retransmission damping"), agrupamento de pacotes e compreensão de dados.

6 Serviços Distribuídos

O V System suporta uma série de serviços que são implementados em espaço usuário. Estes serviços utilizam-se das funções do kernel e dos protocolos de entrada e saída (UIO), descrito a seção seguinte,

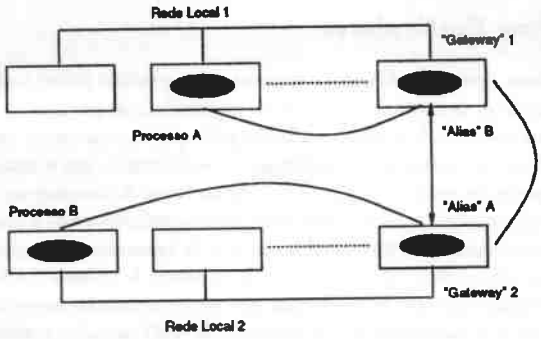

Figura 10: Comunicação entre redes usando "Alias"

e de nomes. Abaixo descrevemos alguns destes serviços.

- Servidor "pipe" ("Pipe server") suporta a buferização, mutiplas leituras e escritas.
- Servidor de rede ("Internet server") implementa o protocolo TCP/IP.
- Servidor de arquivo ("File server") deriva do sistema de arquivos do Thoth. Sua característica principal é a utilização de largos blocos de dados a fim de diminuir o "overhead" na transferência de grande quantidade de dados pela rede ou para disco.
- Servidor de impressora ("Printer server") suporta o "spooling" de "jobs" de impressão e multiplos protocolos clientes, permitindo que arquivos sejam submetidos usando-se V IPC e UIO ou usando conecção TCP.
- Servidor de grupo[2] ("Team server") ou gerenciador de programas controla a execução local de programas. São funções do servidor de grupo:
  - Prover a interface entre o programa cliente e o kernel no momento da execução do programa;
  - Implementar a divisão do tempo de processamento entre os programas através de classes de prioridades ("foreground", "background" e "guest");
  - Manipular a ocorrência de exceções em programas com erros na execução;
  - Manter informações de tempo de execução e tempo de uso de recursos dos processos;
  - Participar na distribuição de carga (programas) entre as máquinas do sistema e
  - Executar a terminação e a migração de processos.
- Servidor de vídeo ("Display server") implementa um sistema de multi-janelas.

Originalmente estava planejado outros servidores que deveriam ser incluidos em futuras versões do V System, como por exemplo, um servidor para disco ótico, um servidor para transações atômicas e um servidor para sincronização de tempo, mas como o V System foi abandonado, não é provável qu eles tenham sido implementados.

[2] O servidor de grupo é usado para referenciar um grupo de processos que dividem o mesmo espaço de endereçamento.

7 Características Particulares

A forma como um sistema operacional executa suas entradas e saídas define a sua comunicação com o mundo exterior. Tornando-se uma das características primordiais no projeto de sistemas operacionais. No V System o protocolo de entrada e saída é definida pelo que chamamos de interface UIO, que é um protocolo de alto-nível que se utiliza das facilidades de comunicação que o sistema fornece. A comunicação entre os programas ou módulo é feita pelo IPC ao invés de mensagens, numa implementação a nível de aplicação o que contrasta com a metodologia convencional onde a entrada e saída é feita a nível de "kernel" o que provoca um substancial acrescimo de tamanho e complexidade.
Uma das características principais do UIO é a sua uniformidade de acessos a diferentes dispositívos de saída. Como uma aplicação deve operar como uma grande variedade de serviços de entrada e saída e não somente com um serviço particular a uniformidade do UIO permite a utilização de uma mesma aplicação em redes heterogêneas onde a existe de uma grande variedade de serviços de I/O.
A entrada e saída é implementada através da criação de um objeto UIO que corresponde a um arquivo onde é possível ler, escrever, questionar e modificar. O objeto UIO é implementado como um bloco de dados o que permite um eficiente transferência e armazenamento de dados.

8 Modificações do Kernel

Uma das funções do V System era de prover uma plataforma de testes onde novas idéias fossem testadas e experimentadas. Neste intuito, deste a primeira versão alguns erros foram verificados e soluções foram propostas e implementadas.
Na primeira versão o identificador de grupo apresentava um campo discriminando se o grupo era local ou global, como o descrito no item 5.1. Porém com a introdução da migração de processos, não havia mais garantia de que um grupo local permaneceria local. Um dos processos a ele relacionado poderia migrar para outro "host", deixando o grupo de ser local. A solução proposta e posteriormente implementada era de eliminar a noção de grupo local ou global, fazendo com que todos grupos passassem a ser considerados como globais.
O identificador de processos apresentava também o mesmo problema. O campo "host" lógico ajudava na alocação dos identificadores de processos e no seu mapeamento, porém incluia numa série de restrições para a migração de processos. A solução, como foi feito para o identificador de grupo foi a eliminação deste campo. Outro problema encontrado era que inicialmente o gerenciamento de memória e de processos era feito internamente ao kernel como um servidor único, comunicando-se através de envio de mensagens, devido a complexidade deste servidor único sua implementação apresentava erros e uma baixa performance. A solução adotada foi utilizar a comunicação via procedural (RPC) e de dividir em vários servidores (memória, processo, dispositivo e comunicação) aumentando a modularidade do kernel e simplificando sua implementação.

9 Conclusão

O V System foi implementado como um sistema acadêmico e teve como objetivos testar e experimentar idéias novas no suporte a execução de programas distribuídos.
O conceito de microkernel aparece para prover esta flexibilidade. Um suporte básico é fornecido e a inclusão ou exclusão de novas implementações não só torna-se possível como também relativamente simples.
O V System foi um dos primeiros sistemas operacionais distribuídos sendo ele o percursor de conceitos

como já foi comentado. São exemplos destes conceitos a extrutura de microkernel, a comunicação de grupo e a migração. Não foi discutido os detalhes da migração neste trabalho para referência veja [5]. Outra preocupação do V System foi a de definir protocolos e interfaces, uma vez que é através deles que o sistema é definido. Assim boa parte do trabalho foi implementar novos protocolos para o transporte de dados, I/O, transações atomicas, execução remota, migração, sincronização de tempo, nomes, etc. Mesmo que os protocolos do V System não tenha sido usados em sistemas posteriores, a implementação do V System foi vital para o desenvolvimento posterior nesta área.

Referências

[1] Cheriton, D.R. and Zwaenepoel, W. *Distributed Process Groups in the V Kernel.* ACM Transactions on Computer Systems 3, 2 (Maio 1985), 77-107.

[2] Cheriton, D.R. *The V Distributed System.* Communications of the ACM 31, 3 (Março 1988), 314-333.

[3] Cheriton, D.R. *Request-Response and Multicast Interprocess Communication in the V Kernel.* Networking in Open Systems, Springer, LNCS 248 (Agosto 1989), 297-312.

[4] Cheriton, D.R., Malcom, M.A., Melen, L.S. and Sager, G.R. *Thoth, a Portable Real-Time Operating System.* CACM 22, 2 (Fevereiro 1979), 105-115.

[5] Theimer, M.M., Lantz, K.A. and Cheriton, D.R. *Preemptable Remote Execution Facilities in the V System.* In Proc. of the 10th Symposium on Operating System Principles, ACM SIGOPS, 1985.

Introdução ao Mach

Sergio Borger

1 Introdução

1.1 Histórico

Em 1975 o professor Richard Rachid da universidade de Rochester iniciou um projeto chamado RIG (Rochester Inteligent Gateway). Este projeto visava implementar um sistema operacional baseado numa coleção de processos que se comunicavam através de troca de mensagens. Esta implementação foi feita em um processador Data General e chamou-se *Eclipse*.

Em 1979 o professor Rachid tranferiu-se para a Universidade de Carnegie Mellon onde iniciou um projeto para melhorar o sistema operacional Eclipse. Desta vez ele utilizou uma estação de trabalho científica PERQ que possuia um monitor bitmapped, adaptadores de comunicação de rede, mouse, e que permitia microprogramação. Como resultado deste trabalho, em 1981 estava pronto o sistema operacional *Accent*, que possuia como principais características a operação em rede, um sistema de proteção, memória virtual de 32 bits. Em 1983, ja existiam aproximadamente 150 PERQ utilizando Accent, porém este sistema ainda perdia em popularidade para o BSD UNIX, pois o UNIX possuia mais aplicativos.

Nesta época, o Departamento de Defesa Americano, através do DARPA, estava em busca de um sistema operacional que pudesse suprir suas necessidades e fosse compatível com o UNIX.

Com o objetivo de aprimorar o sistema Accent, e contando com o suporte financeiro do DARPA, nasceu em 1984 o projeto Mach. Os principais objetivos do projeto Mach eram de implementar um sistema operacional compatível com o UNIX, que utilizasse o conceito de threads, que possuisse um bom mecanismo de comunicação entre processos e um bom sistema de gerência de memória e que pudesse ser executado em arquiteturas multiprocessadoras.

Para isto, foi compilado um Kernel que possuia todas as funções do UNIX e todas as funções de comunicação entre processos e gerência de memória. O resultado foi um kernel grande em termos de tamanho, porém nem sempre compatível com o UNIX 4.2BSD.

Em 1986 o Mach 2.5 foi portado para várias plataformas, entre elas o VAX, IBM RT e a Sun 3. Nesta versão a concepção do Mach era baseada em multiprocessamento em uma única estação e no conceito de um sistema operacional que distribuisse seus recursos de forma transparente.

Em 1988 a Open Software Foundation (OSF) escolheu o Mach 2.5 como base para o seu sistema operacional OSF/1. Esta foi uma tentativa do grupo de empresas que compõe a OSF em competir com a AT&T e demarcar o caminho que os sistemas Unix-like vinham trilhando.

Em 1989 a equipe do Mach retirou o a maior parte do código específico do UNIX do kernel e o implementou em espaço usuário, criando assim o Mach 3.0 microkernel. Esta é a versão atual, do qual trata o presente trabalho.

2 Arquitetura microkernel

A arquitetura microkernel implementada no Mach 3.0 tem como objetivos:

. Ser uma base para desenvolvimento de sistemas operacionais.

. Prover acesso de forma transparente a recursos existentes na rede.

. Suporte ao endereçamento de memória. Especialmente quando são necessários endereços muito grandes. O Mach possui um sistema muito elaborado para o endereçamento de memória virtual.

. Utilizar o máximo de paralelismo durante a execução, tanto no espaço kernel quanto no espaço usuário.

. Ser portável para o maior número de plataformas possível.

A arquitetura do Mach pode ser separada em duas partes distintas: aquala no espaço usuário e aquela no espaço kernel.

No espaço kernel é executado o microkernel do Mach 3.0. Ele implementa os mecanismos básicos de operação do sistema operacional, tais como: gerência de processos e threads, gerência de memória, controle básico de E/S e comunicação entre threads.

Em espaço usuário estão servidores que implementam as políticas de alocação de recursos, o sistema de arquivos e outras funções de sistemas operacionais tradicionais, tais como sistema de proteção e autenticação.

Além disto são implementadas em espaço usuário bibliotecas de emulaçao para diversos sistemas operacionais, tais como: 4.3BSD, UNIX System V, DOS e outros. Através destas bibliotecas de emulação, pode-se ter a execução simultânea de diversos sistemas operacionais sobre o microkernel do Mach (figura 11).

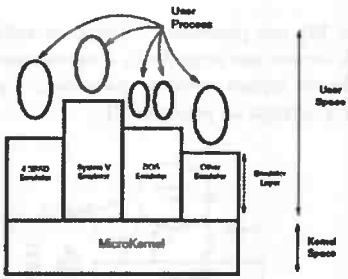

Figura 11: Arquitetura Mach

O microkernel Mach está baseado nas seguintes abstrações: processos (*tasks*), threads, objetos de memória, portas (*port*) e mensagens.

Para o Mach, o conceito de processo (task) caracteriza apenas um espaço de endereçamento, no qual um ou mais fluxos de execução (threads) podem executar. O Mach é um sistema baseado em mensagens, no sentido de que toda interação entre processos, e destes com o microkernel, é feito através do envio de mensagens para portas.

Portas são objetos de comunicação gerenciados pelo microkernel, que podem armazenar uma sequência de mensagens. Para cada *port*, apenas um processo (ou um thread) detêm o direito de

recebimento de mensagens daquela porta (*read capability*). No entanto, podem existir vários processos (ou threads) com o direito de envio para um port (*write capability*). Ambos os tipos de capability podem ser passados de um processo para outro através do envio envio das mesmas em mensagens.

Um objeto de memória é um conjunto de uma ou mais páginas do sistema de memória virtual do Mach que pode ser mapeada para o espaço de endereçamento de um processo.

A seguir iremos descrever duas formas de interação entre processos em espaço usuário e o microkernel, com o intuito de mostrar algumas características do sistema operacional.

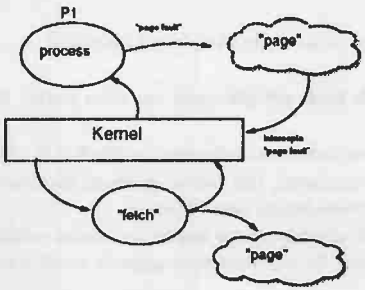

Figura 12: Exemplo de interação ("page fault")

Vamos considerar (figura 12) um processo P1 sendo executado (em espaço usuário). Em dado momento da execução de P1, ocorre um page fault, o microkernel intercepta este page fault e chama um processo *fetch*, executado em espaço usuário, que obtém a página necessária e a devolve para o microkernel, que por sua vez a entrega ao processo P1.

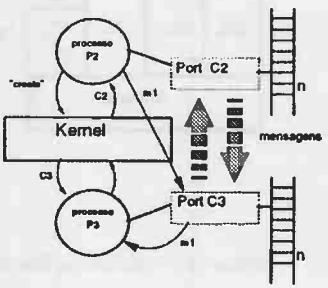

Figura 13: Exemplo de interação (envio de mensagem)

Vamos agora considerar (figura 13) os processos P2 e P3, onde P2 deseja enviar uma mensagem para P3. Inicialmente o processo P2 faz uma chamada ao microkernel para que este crie um port para

envio da mensagem, e recebe como resposta a indicação do port c2 criado. O processo P3 executa o mesmo procedimento, obtendo o port c3. Quando o microkernel cria um port, ele cria também uma estrutura de fila que mantém as mensagens que são envidas ao port, de forma a garantir a entrega de mensagens. Quando P2 esta pronto a enviar a mensagem m1 ao processo P3, ele escreve a mensagem m1 no port c3. Da mesma forma, quando o processo P3 estiver pronto a receber a mensagem, ele lê a mensagem m1 do port c3.

Note que P2 e P3 podem ser fluxos (threads) em dois processos distintos ou no mesmo processo. A comunicação atraves ports difere um pouco nas duas implementações, sendo a que descrevemos mais proxima da implementação através de Threads.

3 Processos

Em MACH um processo é um espaço de endereçamento compartilhado por um conjunto de Threads ou fluxos de processamento (figura 14). Dentro de um processo, os Threads se comunicam entre si e com o microkernel do Mach através de ports. A comunicação com o microkernel através de ports visa reduzir a um mínimo o número de chamadas ao sistema operacional através de system calls.

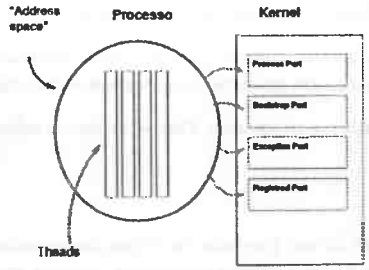

Figura 14: Threads e "Special Ports"

Um processo pode estar em um estado *pronto*, quando esta sendo executado ou pronto para execução, ou no estado *bloqueado*, quando o processo é interrompido por algum motivo (i.e. algum thread do processo fez uma chamada ao sistema operacional e está agurdando o término do I/O).

O escalonamento dos threads e dos processos possui parametros que podem determinar até em que processador um determinado thread, de um processo, será executado.

Para que as políticas de gerênciamento de recursos de um sistema operacional emulado sejam executadas e utilizadas corretamente (visto que este é executado em espaço usuário) é necessário que o microkernel possua o endereço de emulação (*emulation address*). Este endereço indica para o kernel a localização das entradas necessárias na biblioteca de emulação para que este saiba como manipular as políticas implementadas pelo processo de emulação do S.O. e como tratar as chamadas ao sistema.

Para comunicação entre os processos e o microkernel, existem ports específicos que são compartilhados pelos threads do processo (figura 14).

Estes ports específicos são:

Process port serve para que cada processo requisite dos sistema operacional os serviços que necessita.

Bootstarp port serve para inicialização do processo. É a partir dele que os processos obtém os outros ports do kernel; (i.e. o primeiro processo obtém todos os outros ports a partir deste port).

Exception port é utilizado para a indicação de erros (i.e. divisão por zero), é especialmente interessante para a depuração.

Registred port é utilizado para a localização de serviços básicos (i.e. serviço de nomes).

Além disto os processos possuem estatísticas sobre a utilização dos recursos destinados a eles pelo microkernel.

As primitivas para o controle de processos, fornecidas pelo Mach são:

Create cria um novo processo.

Terminate termina um processo.

Suspend para a execução de um processo.

Resume re-inicia a execuçao de um processo suspenso.

Priority designa a prioridade com a qual um processo será executado.

Assing designa o processador no qual um processo será executado, isto é especialmente interessante no caso de processadores que possuem características especiais.

Info fornece informações sobre o processo e Threads retorna informaçes sobre os threads do processo.

4 Threads

Threads são entidades ativas de um processo, ou fluxos independentes de processamento dentro de um processo. Elas podem estar em um estado bloqueado, prontas para execução, ou em execução. Cada thread pertence exatamente a um único processo e em cada processo existe pelo menos uma thread.

Todos os threads de um processo compartilham o mesmo espaço de endereçamento ("address space"), portanto todas as variáveis (e características) de cada fluxo de processamento estão disponíveis para utilização e compartilhamento por todos os threads do processo. Apesar desta possibilidade, cada thread possui recursos próprios. Por exemplo o *thread port*, apesar de acessivel por todos os threads, é um recurso específico de um único thread do processo (figura 15).

Dentro de um processo todos os threads são escalonados através de uma política de cotas de tempo (timesharing). O Mach também possuí mecanismos de escalonamento, sincronização e exclusão mútua necessários à execução de threads em arquiteturas multiprocessadoras.

O pacote de threads fornecido pelo Mach é chamado *C Threads*. É um pacote simples se comparado com outros, porém possui as primitivas básicas necessárias para a manipulação de threads. Além disto, é possivel realizar a sua extensão devido ao encapsulamento em uma biblioteca baseada em linguagem orientada a objetos. A utilização das funções de threads é feita através das primitivas:

Fork para a criação de um novo thread no processo.

Exit para terminar um thread.

Join para que um processo aguarde o término da execuçao de um thread para proseguir seu processamento.

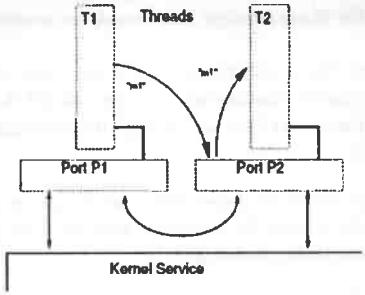

Figura 15: Comunicação entre threads

Detach para que um thread indique ser impossível que outro thread aguarde seu término através do join.

Yield quando um thread não tem nenhum processamento a ser executado naquele momento.

Self para a obtenção de informações sobre o próprio thread.

A sincronização entre threads em um processo pode ser feita através de mutexes, com primitivas básicas o **lock**, o **trylock** e o **unlock**. Alternativamente, a sincronização entre threads também pode ser feita através do uso de ports. Neste caso, a sincronização ocorre quando o thread receptor fica bloqueado em um port, esperando o outro thread enviar a mensagem.

A comunicação entre dois threads, T1 e T2, de um processo P é feita através de ports p1 e p2 respectivamente (figura 15). Após a criação dos ports, feita a pedido de cada um dos threads ao microkernel, em dado momento, o thread T1 escreve a mensagem m1 no port p2 do thread T2. Em um momento subseqüente, o thread T2 lê a mensagem m1 do port p2. Existem outras possibilidades para a comunicação entre os threads, tal como a utilização compartilhada de variáveis e outras estruturas de dados.

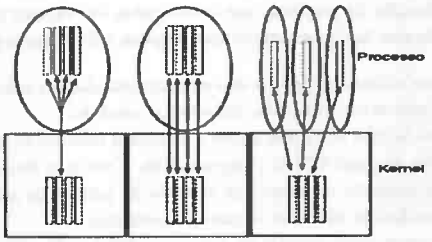

Figura 16: Tipos de implementação de processos e threads

Os threads do Mach são chamados "heavyweight", ou seja os threads são implementados e gerenciadas pelo microkernel. No Mach existem três formas de implementar os threads (figura 16).

. A primeira é utilizar um único thread para cada processo, ou seja existirão vários threads de um processo sendo executados unicamente por um thread do microkernel. Esta implementação é interessante especialmente na fase de "debug" de processos, pois o processo pode ser controlado durante sua execução.

. A segunda forma, e mais usual de implementar os threas no Mach é a utilização de um thread do microkernel para cada thread do processo. Isto faz com que a execução dos threads seja mais eficiente em ambientes multiprocessados.

. A terceira forma de implementar threads é associar um único thread do microkernel a cada processo, com o detalhe de que cada processo contém somente um único thread. Esta implementaçao não é geralmente utilizada, sendo sua utilizaçao mais ligada a processos que necessitam de recursos exclusivos e utilizam grande quantidade de memória.

5 Gerência de memória

O sistema de gerência de memória no Mach é extremamente versátil e poderoso. O mesmo possuí um mecanismo de endereçamento para memória virtual de 32 bits baseado em paginação.

A característica mais importante do sistema de gerenciamento de memória do Mach é a separação dos elementos dependentes da máquina dos elementos independentes da máquina. Esta separação torna o sistema mais portavél para outras plataformas. Por um lado toda a interface dependente de hardware é facilmente identificável e modificável. Por outro lado a implementação dos módulos independentes da máquina facilita a comunicação entre o módulo de gerenência de memória e o sistema de communicação.

Feita a caracterização anterior, podemos dividir o código do sistema de gerência de memória em 3 partes:

. A primeira chamada *pmap*, que é executada no kernel e está relacionada a gerência do Memory Management Unit(MMU), ou seja alocação de páginas e características específicas do hardware.

. A segunda parte é independente da máquina e trata de forma simples o controle de paginação, a substituição de páginas e também trata dos page faults.

. A terceira e última parte, consiste dos *memory managers*, que implementam uma dada política de alocação e substituição de páginas, são executados em espaço usuário, e tem a responsabilidade de manter e manipular todas as estruturas lógicas referentes a paginação.

Como vemos, as duas primeiras partes são implementadas no microkernel do sistema operacional e a última parte corresponde a um processo em espaço usuário.

O protocolo de comunicação utilizado entre o processo usuário de gerência de memória e o microkernel é bem definido através de uma API de programação. Com isto tem-se a possibilidade de implementar diversos mecanismos de gerencia de memória através de processos gerenciadores de memória externos ao kernel, que são chamados de *external memory managers*.

Como já vimos, o compartilhamento de memória feito pelos threads é altamente eficiente, tanto para arquiteturas uni- como multiprocessadoras. Isto permite a sua utilização para a implementação de gerênciadores de memória em espaço usuário. Poderia se ter, por exemplo, um tread cuidando da gerência de memória para cada processo.

As primitivas básicas para o uso do sistema de gerência de memória são:

Allocate e Deallocate para pedir a utilização de uma determinada quantidade de memória e devolve-la apos sua utilização.

Map para obter informações sobre a memória alocada.

Copy para copiar um trecho de memória de uma posição para outra.

Inherit quando um "memory object" herda características de outro "memory object".

Read e Write funções obvias de leitura e escrita em memória.

6 Comunicação

O Mach possui vários mecanismos de comunicação, tais como envio assíncrono de mensagens, o *Remote Procedure Call (RPC)*, o *byte stream* e outros.

O mecanismo de comunicação entre processos (*Inter Process Communication*) atualmente utilizado, tem origem nos sistemas RIG e Accent, sendo portanto orientada à comunicação local e não remota. A comunicação remota é feita com o auxílio de um processo especializado em comunicação remota, chamado *Network Message Server (NMS)*, que executa as funções de um agente proxy, ou seja, que para cada porta remota, cria uma porta auxiliar local e faz o redirecionamento apropriado das mensagens da porta auxiliar para a porta remota de destino.

A seguir veremos cenarios de comunicação entre processos com o objetivo de mostrar a funcionalidade dos elementos do sistema de comunicação.

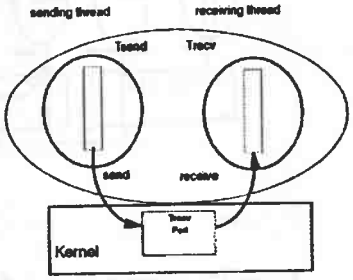

Figura 17: Envio e sincronismo de mensagenes entre threads

Observe a figura 17 e suponha que um thread Tsend deseje enviar uma mensagem para um Thread Trecv. Durante a execução do fluxo de processamento de Tsend é executada a função **send** de uma mensagem. Esta mensagem é escrita no port de recepção de mensagens do thread Trecv. Após este instante, o fluxo de execução do thread Trecv realiza uma chamada **receive** e lê a mensagem que está disponível em seu port de recepção. Note que existe um port de envio e outro de recepção de mensagens. Note também que o caso descrito serve como exemplo tanto para threads que estão em um mesmo processo, quanto para threads localizados em processos diferentes. A única restrição é que a execução da função **receive** seja posterior a execução da função **send**.

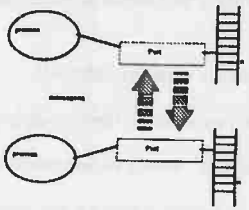

Figura 18: Fila de mensagens

Podemos estender este exemplo para um caso onde exista muita comunicação entre threads (figura 18). Neste caso o microkernel garante o envio da mensagem até o port de destino. O armazenamento de várias mensagens é feito através de uma fila de tamanho parametrizavél, na qual as mensagens que chegam são armazenadas seqüencialemente na ordem de chegada (figura 18).

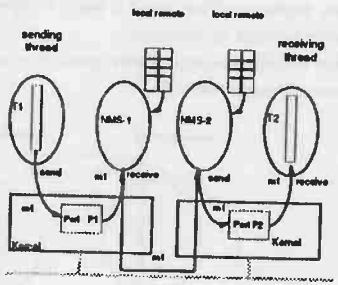

Figura 19: "Network Message Server"

Vamos agora analisar o caso de um thread de um processo T1 que deseja enviar, através de seu port p1, uma mensagem para a porta p2 de um thread T2 em um processo localizado fisicamente em outra máquina(figura 19). Inicialmente o T1 executa uma chamada **send** para envio da mensagem m1 para p2 em T2. O microkernel verifica que p2 em T2 não está sendo executado em seus processadores, e assim passa a mensagem m1, juntamente com a indicação de seu destino (p2; T2), para o network message server NMS-1. O NMS mantém uma tabela de ports remotos de comunicação, juntamente com a informação de qual é o NMS responsável. Assim o NMS-1 é capaz de localizar o NMS-2 da maquina na qual T2 está sendo processado. Em seguida, o NMS-1 passa a mensagem m1 para o NMS-2. O NMS-2, por sua vez, consulta uma tabela com informação sobre os ports e processos locais, envia a mensagem m1 para o port p2 de T2. O mecanismo de confirmação de T2 para T1 segue um caminho análogo através do NMS.

7 Interfaces Orientadas para Objeto

A utilização de metodologias e conceitos baseados em orientação a objetos no desenvolvimento de sistemas operacionais é relativamente recente. Neste contexto, ela representa uma tendência de separar claramente os módulos utilizando interfaces encapsuladas bem definidas.

No desenvolvimento do sistema operacional Mach os conceitos de orientação a objeto foram aplicados de forma a definir os mecanismos e a estrutura da interação entre os clientes do sistema operacional e o sistema propriamente dito.

As interfaces de programação e os módulos executáveis foram implementados utilizando orientação a objetos nos três níves da arquitetura do Mach: no microkernel, nos servidores de sistema e nas bibliotecas de emulação. O objetivo da implementação baseada em objetos é facilitar a criação, o desenvolvimento de novas interfaces e a extensão das interfaces atuais. Este desenvolvimento é facilitado especialmente pela utilização de duas propriedades de linguagens orientadas a objeto: a herança e o polimorfismo. A seguir vamos explicar os dois conceitos e definir encapsulamento e classe.

Encapsulamento é uma forma de esconder a implementação de um conjunto de funções e variáveis de uma biblioteca. Somente as funcões e variáveis públicas estão disponíveis para o usuário de biblioteca.

Classe é a implementação de uma biblioteca encapsulada. Somente variáveis e funções públicas estão disponíveis para serem utilizadas.

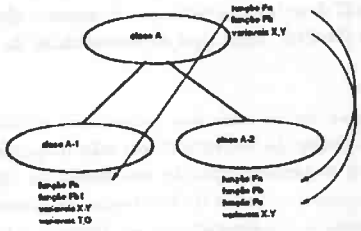

Figura 20: Herança

Herança é um conceito de orientação a objetos que permite a criação de sub-classes que herdam características de um outra classe (figura 20). Suponha uma classe A que implementa as funções Fa e Fb, e que possue as variáveis X e Y. Agora imagine uma sub- classe A2 que herda as características de A. A2 herda a função Fa e a função Fb já implementadas em A, portanto não é necessários que sejam implementadas novamente. Da mesma forma as variáveis X e Y estão disponíveis para uso na classe A2, porém foram declaradas em A. Para que A2 fique pronta, falta somente a implementação da funcionalidade que A não possuia, por exemplo a implementação da função Fc. Suponha agora uma sub-classe A1 que possui as mesmas características de A, a menos de alguns detalhes a mais na função Fb e de duas variáveis. Para isto A1 herda as características de A, declara as variáveis T e Q e cria a função B1 que é uma especialização da função B.

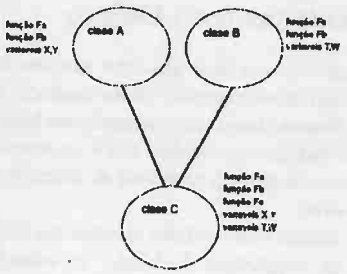

Figura 21: Polimorfismo

Polimorfismo ocorre quando uma sub-classe herda características de duas classes e estas classes implementam uma função com o mesmo nome (figura 21). Suponha a classe A com as funções Fa e Fb e variáveis X e Y. Suponha agora a classe B com as funcoes Fc e Fb e variáveis T e W. Agora considere a classe C que herda características de A e B. Obviamente as variáveis X, Y, T e W estão disponíveis para a classe C. Da mesma forma as funções Fa e Fc também. Porém a função Fb na classe C deve ter sua herança claramente identificada, ou seja, Fb vem de A (A::Fb) ou se Fb vem de B (B::Fb). Este tipo de necessidade da declaração explícita pode ser chamada de polimorfismo.

Claramente a utilização destes conceitos facilita em muito a criação de novos módulos e novas interfaces, pois muito do código já implementado esta disponível em bibliotecas e possivel de ser re-utilizado. Por exemplo a implementação de estruturas de sistemas de arquivo, os mecanismos de leitura e gravação em dispositivos novos de hardware, os gerenciadores de memória, o mecanismo de escalonamento de processos, a implementação de threads e outros por estarem implementados em bibliotecas orientada a objeto, podem ser extendidos ou implementados de acordo com as necessidades do usuário.

8 UNIX em espaço usuário

A implementação do UNIX 4.3 BSD baseado no Mach traz várias vantagens:

Modularidade pode-se trocar facilmente partes do UNIX.

Portabilidade como o UNIX está implementado sobre o microkernel, torna-se fácil a sua adaptação para outras plataformas de hardware, para as quais haja uma implementação do microkernel Mach .

Network Access os mecanismos de acesso a rede podem ser compartilhados de forma mais eficiente por todos os processos que estão executando sobre o Mach.

Real time políticas preemptivas e não preemptivas podem ser implementadas para cada módulo do UNIX, facilitando o desenvolvimento de aplicações que utilizam conceitos de tempo real.

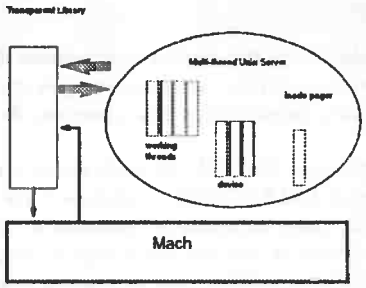

Figura 22: Implementação de UNIX em espaço usuário

A implementação do UNIX em espaço usuário é realizada através de dois módulos (figura 22): A *Transparent Library* e o *UNIX Server*. A Transparent Library é uma biblioteca de emulação de chamadas de sistema operacional, que traduz as chamadas normais do UNIX ao sistema operacional em serviços oferecidos pelo microkernel do Mach. A segunda parte é composta pelo UNIX Server, que na verdade é um conjunto de servidores que fornecem os serviços do UNIX, tais como serviços de arquivos, serviços de diretório, acesso a controladores de periféricos e outros.

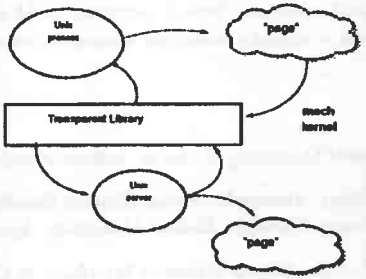

Figura 23: "Transparent Library"

Assim a implementação pode ser realizada em um processo, onde cada um dos threads executa uma determinada função do sistema operacional UNIX. Por exemplo, quando um *thread UNIX* necessita uma página de memória que esta faltando, o kernel intercepta este page fault e repassa o pedido de página, através da biblioteca de emulação, para um outro thread UNIX que esta executando a politica de paginação. Este thread retorna a página ao thread de trabalho e sinaliza ao kernel que terminou a operação de tratamento do page fault (figura 23).

9 Conclusão

O Mach é sem dúvida nenhuma um dos sistemas operacionais distribuídos mais importantes, não só pela flexibilidade e eficiência que oferece, como também pela grande aceitação que obteve. Em [1] [6] são apresentadas introduções à arquitetura e funcionamento do sistema Mach muito interessantes e didáticas.

A situação atual do protótipo do Mach 3.0 multi-server system inclui o servidor de nomes e a biblioteca de emulação para o UNIX 4.3 BSD escritos em C++. Mesmo incompleta, a biblioteca de emulação do UNIX 4.3 BSD, provê as principais chamadas ao sistema e em muitos casos implementa as funções de forma mais eficiente do que em outras implementações do UNIX.

A análise de performance comparativa entre os sistemas implementados no Mach, no Mach com UNIX Server e o SunOS 4.0, descritas em [2], demontram que muitas rotinas internas do Mach já atingiram um alto grau de otimização, visto o excelente desempenho relativo. Porém existe ainda muito espaço para melhoria em outras, pois o desempenho destas ainda é equivalente ou inferior a implementações tradicionais.

O status do projeto Mach em 1990 inclui a implementações de um kernel Mach puro em multi-processadores e diversos uni-processadores, como por exemplo os da linha Vax, DecStation, Sun Microsystems, Intel e Motorola 68K. Existem trabalhos em andamento para permitir o port do Mach para outras arquiteturas e para emular outros sistemas operacionais, como o MacOS e outros. Além disto, existem grupos trabalhando em outras implementações de Unix baseadas em Mach servers e outros trabalhos visando acrescentar funcionalidade (i.e. segurança) ao Mach.

A arquitetura modular implementação do Mach permite uma reconfiguração do sistema de forma simples e rápida. Isto é devido principalmente a sua implementação orientada a objeto. Suas bibliotecas permitem uma rápida substituição dos diversos componentes e a integração de novos módulos facilitando assim a evolução do sistema. Tem-se portanto no Mach um sistema extremamente flexível tanto em termos de interfaces e módulos como em termos de arquitetura.

Referências

[1] Tanenbaum, A Distributed Operating Systems. *editora Prantice Hall 1992.*

[2] David Golub, Randall Dean, Alessandro Forin, Richard Raschid UNIX as an application program. *School of Computer Science, Carnegie Mallon University.* April 1990.

[3] Paulo Guedes, Daniel P. Julin Object Oriented Interfaces in the Mach 3.0 Multi-Server System.

[4] Accetta, M.J., Baron, R.V., Bolosky, W., Golub, D.B., Rashid, R.F., Tevanian, A. and Young Mach: A new Kernel Foundation for UNIX Development In *Proceedings of Summer Useniz.* July, 1986.

[5] Coopers, E. and Daves, R. C Threads. *Technical Report CMU-CS-88-154, Computer Science Department - Carnegie Mellon University.* June, 1988.

[6] Raschid, R., Baron, R., Forin, A., Golub, D., Jones, M., Julin, D., Orr, D., Sanzi, R Mach: A foundation for Operationg Systems; a Position Paper. In *Proceedings of the 2nd Workshop on Workstation Operating Systems.* IEEE, September, 1989.

[7] Silberschatz, A., Galvin, P.B. Operating Systems Concepts Addison-Wesley, 1994.

Movendo o Administrador de Memória para Fora do Núcleo do MACH 3.0

Fabio Kon

1 Introdução

[1] descreve a implementação de uma nova versão do sistema operacional distribuído MACH 3.0 na qual toda a tarefa de paginação de memória foi extraída do núcleo do sistema e transferida para espaço usuário.

Para tanto, foi necessário resolver uma série de problemas que descreveremos a seguir. Historicamente, o fato de o sistema de administração de memória ter que possuir privilégios na alocação de memória sobre os demais componentes do sistema tem feito com que ele sempre fosse implementado no núcleo do sistema. Ao implementá-lo no espaço usuário é preciso manter estes privilégios a fim de evitar possíveis deadlocks. Além disso, ao retirar a administração da memória do núcleo do sistema, é preciso tomar cuidado com a possível perda de eficiência que este ato pode trazer.

O administrador de memória *default* é um componente essencial do sistema MACH 3.0. Ele administra o armazenamento de objetos temporários em memória. Os principais componentes vitais do sistema operacional – como, por exemplo, o sistema de arquivos – são seus clientes. Desta forma, o administrador de memória *default* não pode depender de nenhum destes componentes vitais pois, caso contrário, correria o risco de produzir deadlocks.

O administrador de memória *default* foi incorporado à rotina de inicialização do sistema, dividindo com ela, a mesma tarefa (*task*). Esta tarefa reutiliza parte do código do sistema de arquivos a fim de acessar arquivos em disco sem depender da tarefa responsável pelo sistema de arquivos.

2 Restrições ao Administrador de Memória *Default*

As principais atribuições de um administrador de memória em um sistema operacional moderno são as seguintes:

- Alocação de memória temporária para serviços do sistema e para programas dos usuários.
- Mapear arquivos executáveis e arquivos de dados para o espaço de endereçamento de tarefas.
- Mapear dispositivos de entrada e saída como, por exemplo, monitores de vídeo, linhas seriais.
- Compartilhar arquivos ou memória temporária entre tarefas.

Devido ao seu papel essencial no sistema MACH, a implementação do administrador de memória *default* deve obedecer a uma série de restrições enumeradas a seguir:

1. O administrador de memória *default* deve estar sempre residente na memória, nenhum outro processo pode retirar as suas páginas da memória principal.

2. O administrador de memória *default* não pode ser bloqueado a fim de esperar por mais memória física uma vez que para liberar memória física é necessário que o seu conteúdo seja copiado para o disco (através do administrador de memória *default*).

3. O administrador de memória *default* não pode depender de nenhum outro recurso vital do sistema que seja seu cliente.

A fim de cumprir as restrições acima, tomaram-se as seguintes providências:

- Ao administrador de memória *default* sempre é permitida a alocação de novas págninas de memória mesmo quando o sistema possui pouco espaço disponível. Os *threads* do administrador de memória *default* recebem uma marca especial denominada `vm_privileged` que lhes permitem a alocação de memória em situações que threads comuns iriam ser bloqueados.
- Estes *threads* especias são também marcados como *unswappable* garantindo que as suas páginas sempre estarão na memória principal evitando, assim, possíveis *deadlocks*.
- O administrador de memória *default* utiliza trechos do código do sistema de arquivos do UNIX 4.2 BSD para administrar o armazenamento de págnias em disco.

3 Movendo para Fora do Núcleo

Ao mover o administrador para fora núcleo foi preciso manter os privilégios especias que ele possía quando era executado dentro do núcleo e substituir mecanismos disponíveis apenas para o núcleo por ferramentas acessíveis para processos em nível usuário. Este trabalho foi facilitado pois o administrador de memória do MACH já utilizava uma interface baseada em portas (*ports*) para tratar as solicitações de operações de paginação. Deste modo, não foi necessário alterar esta interface uma vez que processos no nível usuário também podem utilizar portas para se communicarem com outras entidades do sistema.

Apenas algumas operações do núcleo tiveram que ser acessadas diretamente, entre elas, a alocação de espaço em disco, a sincronização de *threads*, o bloqueio (*lock*) de páginas de memória e a prevenção de que certas páginas fossem *swapped out*. Estas operações ou foram substituídas por rotinas equivalentes em espaço usuário ou foram acrescentadas à interface do núcleo.

Duas *system calls* foram acrescentadas: `vm_wire` oferece a possibilidade de bloquear e desbloquear um trecho da memória virtual para acesso restrito por uma determinada tarefa. `thread_wire` faz com que as solicitações por memória física de um certo thread sejam sempre atendidas.

A *system call* `vm_set_default_memory_manager` é utilizada para especificar a porta onde o núcleo deve enviar as solicitações de alocação de memória.

O acesso ao Disco

O problema do acesso ao disco poderia ter sido resolvido fazendo-se uma cópia do código do sistema de arquivos UNIX 4.2 BSD junto ao código do administrador de memória. No entanto, isto representaria um gasto extra de memória física muito grande.

A solução adotada foi reservar um conjunto fixo de blocos do disco para ser usado pela memória virtual. Deste modo, apenas o código para leitura e escrita de blocos e o código para leitura de um arquivo específico chamado `/mach_servers/paging_file` (que especifica quais blocos serão utilizados pela memória virtual) são necessários.

O problema desta solução é a falta de flexibilidade. O MACH 2.5 podia utilizar uma série de estratégias para alocação de espaço em disco: partições reservadas, arquivos pré-alocados ou arquivos

dinamicamente expandíveis. No novo MACH 3.0, o administrador de memória *default* não é capaz de criar novos arquivos ou de alocar mais espaço em disco para um arquivo antigo impossibilitando uma série de técnicas de alocação de espaço em disco.

A alocação de espaço em disco tem que ser feita por outro instrumento que, através do arquivo `/mach_servers/paging_file`, diz ao administrador de memória quais blocos do disco ele deve utilizar.

4 Resultados

A mudança do administrador de memória *default* do núcleo do sistema para espaço usuário mostra que esta é uma solução possível e que oferece maior flexibilidade ao sistema. O administrador de memória pode ser modificado ou reconfigurado sem a necessidade de alterar o núcleo do sistema. Dependendo da situação, seria possível até trocar o administrador de memória *default* sem a necessidade de interromper o funcionamento da máquina.

Análises do desempenho do novo MACH 3.0 comparando-o à versão antiga (com o administrador de memória *default* no núcleo) mostram uma perda em velocidade quase despresível desde que haja uma quantidade suficientemente grande de memória principal.

Inicialmente, utilizou-se uma máquina com 6,5Mb de memória física disponível. Neste equipamento, executou-se um *benchmark* que fazia uso maciço de 8Mb de memória total. O desempenho do novo sistema foi apenas 0,6% abaixo do desempenho do sistema convencional (com o administrador implementado no núcleo).

Por outro lado, executando-se um *benchmark* que exigia 4Mb de memória total em uma máquina com apenas 2,5Mb o desempenho do sistema com o administrador em nível usuário chegou a ser 5,8% pior do que o sistema convencional.

Dois fatores principais são os responsáveis por esta queda no desempenho. O primeiro é o aumento no número de trocas de contexto entre espaço usuário e espaço do núcleo. O segundo é a necessidade de utilizar um grande número de bloqueios (*locks*) ao ler blocos do disco em espaço usuário. Estes bloqueios são necessários apenas na implementação em espaço usuário pois os threads em espaço usuário são preemptíveis ao contrário dos threads em espaço do núcleo que são não preemptíveis.

Outra desvantagem da implementação em espaço usuário é um gasto extra de 112Kb de memória física. Esta desvantagem, no entanto, não é tão relevante e tende a perder importância com a diminuição do custo da memória principal.

5 Conclusão

Golub e Draves mostram ser possível a implementação de um administrador de memória virtual em espaço usuário. Tal implementação vai de encontro à tendência atual de *MicroKernel* que consiste em retirar o maior número possível de serviços do núcleo do sistema operacional de modo a aumentar a sua flexibilidade e reconfigurabilidade. Testes mostraram que ainda há uma perda significativa no desempenho da implementação em espaço usuário em relação à implementação convencional principalmente quando a quantidade de memória física é relativamente pequena.

Referências

[1] Golub, David B. e Draves, Richard P. "Moving the Default Memory Manager out of the Mach Kernel". *Proc. USENIX Mach Symposium*, November, 1991.

Um Servidor de Memória Distribuída Compartilhada para o MACH

Fabio Kon

1 Introdução

Sistemas com memória compartilhada têm se tornado mais freqüêntes ultimamente graças ao desenvolvimento dos multiprocessadores. No entanto, a sua utilização em sistemas distribuídos tem sido limitada pela falta de implementações disponíveis. Sistemas de memória distribuída compartilhada poderiam ser muito úteis para o desenvolvimento de aplicações tão diversas como sistemas de arquivos, bancos de dados, implementação da migração de processos e sistemas distribuídos e paralelos em geral. A nova tecnologia de fibra óptica aumentou enormemente a velocidade das redes de interligação favorecendo a implementação de memória distribuída.

2 Funcionamento

O MACH possibilita que as solicitações por memória de um determinado conjunto de processos sejam tratadas por um paginador, isto é, um determinado processo em espaço usuário que atenderia às solicitações de memória utilizando um algoritmo próprio. Este paginador oferece aos usuários a função *create_memory_object(initial size)* que aloca um porção lógica da memória que pode ser acessada por uma determinada porta. O usuário pode utilizar a *system call vm_map()* para mapear este objeto de memória no espaço de endereçamento de um processo (indicando a porta do objeto a ser compartilhado).

As portas do MACH podem ser acessadas de qualquer máquina da rede. Como o acesso aos objetos de memória se dá, inicialmente, por portas, este acesso pode ser feito não apenas a partir da mesma máquina que alocou a memória mas também por processos remotos. No entanto, quando um objeto é mapeado em diferentes máquinas, o paginador precisa administrar várias cópias do objeto de maneira coerente. Estas cópias dos objetos funcionam de maneira análoga aos caches de memória.

No instante em que solicita a criação de um novo objeto de memória, o processo do usuário determina o algoritmo de administração de memória associada àquele objeto; alternativamente, é possível confiar na política default. Analisemos os dois principais algoritmos, um centralizado e um distribuído.

2.1 O Algortimo Centralizado

Este algoritmo oferece acesso coerente a objetos de memória com tamanho de página fixo e é implementado por uma tarefa denominada scheduler que é executada em um servidor. As demais máquinas da rede agem como clientes. A cada instante, cada objeto possui apenas um "dono" que é o cliente que está realizando escritas no objeto ou o próprio *scheduler* caso nenhum cliente possua acesso para escrita em um certo instante. O algoritmo do *scheduler* pode ser representado por um autômato com quatro estados:

- *Read*: Nenhum cliente possui acesso para escrita ao objeto. Possivelmente, várias máquinas possuem uma cópia só para leitura do objeto. O servidor possui uma cópia válida do objeto. Este é o estado inicial.

- *Write*: Um cliente possui acesso para escrita, mas nenhum outro cliente possui acesso para leitura e nenhum outro cliente solicitou acesso. O servidor não possui uma cópia válida do objeto.
- *ReadWait*: Um cliente possui acesso para escrita, outros clientes estão enfileirados esperando pelo acesso para leitura. O servidor não possui uma cópia válida mas já a solicitou à máquina que possui acesso para escrita.
- *WriteWait*: Um cliente possui acesso para escrita, outros clientes estão enfileirados esperando pelo acesso para escrita. Possivelmente, alguns clientes estão esperando pelo acesso para leitura. O servidor não possui uma cópia válida mas já a solicitou à máquina que possui acesso para escrita.

As transições entre os estados são determinadas pelas solicitações de acesso dos clientes. Existem três possíveis solicitações: *read_fault()*, i.e., solicitação de leitura, write_fault(), solicitação de escrita e pageout(), envio de uma página modificada para o servidor.

Uma variante deste algoritmo foi implementada possibilitando a alocação de objetos de vários tamanhos o que tem muita utilidade em ambientes onde as máquinas possuem diferentes tamanhos de página. A solução adotada foi estabelecer um tamanho fixo para as páginas do *scheduler*. Solicitações de páginas menores do que as páginas do *scheduler* eram simplesmente arredondadas para uma página do *scheduler*. Solicitações de páginas maiores eram atendidas através da alocação de quantas páginas do *scheduler* fossem necessárias e do envio desta série de páginas, todas de uma vez, para os clientes.

2.2 O Algoritmo Distribuído

O algoritmo centralizado possui desvantagens claras. A mais aparente é a sua falta de escalabilidade. Outra desvantagem menos visível é o excesso de mensagens envolvidas na transferência de propriedade de um objeto de um cliente para outro, isto é, quando um cliente passa a ser o novo dono de um objeto. São trocadas duas mensagens entre o cliente solicitante e o servidor e mais duas mensagens entre o antigo dono e o servidor. Se o cliente solicitante se comunicasse diretamente com o antigo dono, apenas duas mensagens seriam suficientes como mostra a figura 24.

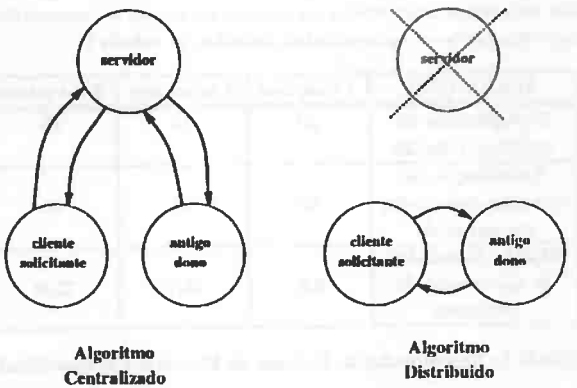

Figura 24: Trocas de mensagens cliente/servidor

A tática utilizada foi tratar cada cliente como um servidor-irmão. Neste algoritmo, o paginador dos clientes aceita solicitações de outros clientes executando um *pageout()* que envia a página em questão

para o cliente que a solicitou. Já os *read_fault()* podem ser tratados por qualquer servidor-irmão que possuir uma cópia atualizada do objeto. Os *write_fault()* são enviados para o servidor-irmão que for o dono momentâneo do objeto. Quando um objeto muda de dono, este fato é comunicado a todos os seus leitores.

Eventualmente, a informação sobre quem é o dono atual de um determinado objeto pode estar desatualizada. Este problema é resolvido através de *forwarding*, i.e., o servidor-irmão que recebeu a solicitação de acesso à página remete esta solicitação para um terceiro servidor-irmão que, segundo o seu ponto de vista é o atual dono do objeto pretendido.

Se houvesse excesso de *forwarding*, o desempenho do algoritmo distribuído poderia sofrer uma perda significativa. No entanto, os testes realizados mostram que, apesar da ineficiência do *forwarding* o desempenho do algoritmo distribuído ainda é melhor do que o do algoritmo centralizado.

3 Tolerância a Falhas

Partições na rede e quedas de máquinas podem prejudicar o funcionamento do sistema de memória dcompartilhada distribuída. A implementação centralizada utiliza o seguinte mecanismo para tolerar quedas de clientes e particões na rede.

Se um cliente que não era dono de um objeto de memória sofre uma queda, então não há problema nenhum e o sistema pode continuar a funcionar normalmente. No entanto, se um dono de um objeto de memória cai, o que o sistema faz é assumir como válida a cópia anterior daquele objeto, ou seja, a cópia que foi enviada ao cliente antes da sua máquina sofrer uma queda. Ou seja, é como se a queda tivesse ocorrido antes da escrita.

A implementação do algoritmo distribuído ainda não possui mecanismos de tolerância a falhas.

4 Avaliação do Desempenho

Os testes realizados mostraram uma maior eficiência por parte do algoritmo distribuído. Deve-se, agora, trabalhar no sentido de torná-lo tolerante a falhas.

Executando-se três aplicações distribuídas no sistema de memória compartilhada distribuída utilizando o algorito distribuído obteu-se os resultados descritos na tabela 1.

APLICAÇÃO	1 máquina	2 máquinas	3 máquinas
Multiplicação de matrizes 128x128	29	15	10
Localização do caminho mais curto em um grafo	60	60	40
Mp3d - Simulador do movimento de partículas	8,6	16,1	23,0

Tabela 1: Desempenho do Sistema de Memória Compartilhada

A tabela 1 apresenta o tempo, em segundos, para a execução de três aplicações distintas com uma, duas e três máquinas interconectadas através de uma rede em anel. As três aplicações apresentam uma aceleração linear (*linear speedup*) quando executadas em multiprocessadores que compartilham a memória através de um barramento. No entanto, somente a multiplicação de matrizes apresentou uma

aceleração linear. O simulador de partículas, por sua vez, mostrou-se menos eficiente cada vez que uma nova máquina era acrescentada.

Segundo os autores, dois problemas básicos afetam o desempenho do sistema. O primeiro é a baixa velocidade da rede em relação aos demais componentes do sistema. A maior parte do tempo do administrador de memória compartilhada é gasta esperando que as mensagens postas na rede sejam entregues. A utilização de redes de fibra óptica reduziriam este tempo em 90%.

Outro ponto significativo é a escolha do algoritmo de alocação de memória. Testes mostraram que a utilização de um algoritmo baseado no esquema de listas livres de Knuth chega a ser de 20 a 30% mais eficiente do que o algoritmo *FirstFit*, também de Knuth. Uma boa escolha e a paralelização do algoritmo de alocação pode trazer ganhos significativos no desempenho do sistema.

5 Conclusão

[1] apresenta a implementação de um serviço de memória compartilhada distribuída para o sistema MACH. Os autores apresentam-se mais otimistas do que os dados coletados poderiam nos fazer supor. Segundo os autores, "o servidor apresenta um desempenho muito bom em todos os testes" e "os [maus] resultados obtidos com as aplicações são dominados pela latência da rede".

Resta saber até que ponto este argumento da grande latência da rede pode ser aceito. É fato que a velocidade de transmissão das redes tende a crescer muito nos próximos anos mas também é fato que, durante este período, também vão crescer proporcionalmente o poder de computação dos processadores e a velocidade dos barramentos. Será que o aprimoramento de todos estes componentes dos sistemas computacionais não farão com que os sistemas de memória compartilhada distribuída continuem perdendo feio para os multiprocessadores baseados em barramentos?

Na realidade, o artigo aqui descrito foi apresentado em 1988, seria interessante investigar os avanços mais recentes nesta área uma vez que a possibilidade de utilização de memória compartilhada em um sistema distribuído facilitaria muito o desenvolvimento de aplicações distribuídas como bancos de dados, sistemas de arquivos [3] e a implementação de linguagens paralelas (como ADA ou Multilisp) e de transações atômicas distribuídas [2].

Referências

[1] A. Forin, J. Barrera, J. Young, and R. Rashid. Design, Implementation, and Performance Evaluation of a Distributed Shared Memory Server for Mach CMU technical report CS-88-165, 1988.

[2] A. Spector, A. Distributed Transaction Processing and the Camelot System In *Distributed Operating System: Theory and Practice.* Springer-Verlag. 1987

[3] M. Young, et al. The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System in *11th Symposium on Operating Systems Principles.* ACM press. 1987.

Migração no sistema operacional Sprite

Fábio Nakano

1. Introdução

Este trabalho é uma compilação do que [1], [2], [3] trazem sobre migração de processos em Sprite.

Sprite é um sistema operacional de rede desenvolvido na Universidade da Califórnia, em Berkeley, que procura oferecer ao usuário o mesmo comportamento do BSD UNIX (para facilidades existentes neste sistema operacional), com melhorias na performance e modularidade do código do kernel.

Migração é um serviço oferecido pelo kernel de Sprite, que transfere processos ativos entre máquinas e difere da invocação remota onde processos são totalmente executados em máquinas remotas, sem a possibilidade de voltar para a máquina de origem.

2. Objetivo

O principal objetivo da migração é aproveitar melhor a capacidade de processamento da rede, distribuindo processos entre máquinas ociosas de forma transparente aos processos e aos usuários, de acordo com as características do ambiente.

3. Características de uso

As principais características de uso são:

3.1 Grande quantidade de máquinas ociosas

Em estudos de carga de máquinas, constatou-se que neste ambiente entre 66 e 78% das máquinas, em média, estavam sem nenhuma carga, já que cada usuário dispunha de duas máquinas. Em ambientes similares, entre 1/7 e 1/3 das máquinas ficava ociosa, isto sugere que algoritmos simples de seleção de máquinas podem ser usados para migração.

3.2 As estações "pertencem" aos usuários

O usuário de uma máquina tem direito a todos os recursos da máquina e não pode ter recursos tomados por processos executando remotamente. Logo há necessidade de se retirar todos os processos remotos quando o usuário retorna à estação. Nestes momentos usa-se migração para remover o processo sem perda do processamento já efetuado.

4. Características de Sprite

As principais características de Sprite são:

4.1 *Sprite acessa kernel*

Pelo sistema ser baseado em chamadas do kernel (system calls) e não em passagem de mensagens, algumas chamadas têm que ser escritas de forma a se comportar diferentemente quando são executadas localmente ou remotamente. Isto não ocorre em sistemas baseados em passagem de mensagens, pois o redirecionamento de mensagens resolve muitos aspectos de migração. No entanto, como Sprite trata muitas chamadas de kernel localmente, sem o overhead da troca de mensagens via rede, esta implementação oferece melhor performance na maioria dos casos.

4.2 *Sprite provê RPC e sistema de arquivos eficientes*

Sprite oferece Remote Procedure Call (RPC) eficiente e um file system único para a rede. Estas características simplificaram a implementação de migração nos aspectos: kernel calls e tranferência de descritores de arquivos.

5. Características desejadas do serviço de migração

5.1 *Transparência para o usuário*

Como citado anteriormente, o usuário não deve perceber perda de desempenho significativa em sua estação. Logo, um processo remoto que esteja sendo executado em sua máquina deve ser removido quando o usuário necessita dela.

5.2 *Transparência de execução*

A transparência de execução implica em resultados iguais para os processos executando remotamente ou localmente, mesmo que exista dependência do ambiente. Por exemplo, uma chamada de gettime tem que retornar o valor da hora na máquina de origem, que não é necessariamente a máquina em que o processo está executando neste caso a chamada tem que ser direcionada para a máquina de origem. No entanto, um pedido de alocação de memória não faz sentido se não for feita à máquina em que o processo está sendo executado.

5.3 *Performance*

O maior ganho de velocidade possível é linear em função do número de máquinas. Usando duas máquinas o processo é executado em metade do tempo que levaria se executado em apenas uma máquina. Usando três máquinas demora-se um terço do tempo. Apesar de Sprite ter RPC e file system eficientes, devido a outros fatores (vide benchmarks), tal aceleração não é obtida.

6. Dependência de outros serviços

A migração depende fortemente do sistema de arquivos, no aspecto de transferência de arquivos (ou mais precisamente, de seus descritores) e do comportamento do sistema com relação à memória virtual, já que as páginas de memória dos processos são guardadas em arquivos normais. É beneficiada pelo sistema garantir uma visão unificada do sistema de arquivos (ao contrário do NFS), independente da máquina onde o processo é executado, facilitando o processo de localização de arquivos dentro da rede, mas é prejudicada pela distribuição dos descritores dos arquivos dentro do sistema.

A maneira com que migração é implementada depende muito de como a comunicação entre processos é executada. Em Sprite, IPC é feita via File System. Outra característica importante é a abstração de arquivos e dispositivos físicos em objetos do File System, diminuindo, ou até eliminando a dependência entre o processo e o dispositivo físico.

O estados de um processo consiste basicamente em sua memória virtual, descritores de arquivos e do process control block. A transferência destas informações é feita ou controlada por RPC, que em Sprite é feita de maneira eficiente, aumentando a eficiência da migração.

7. Mecanismos

7.1 Transferência do estado do processo

O maior problema de migração é a transferência do estado do processo, que é constituído por:

7.1.1 Memória virtual

Em Sprite, as páginas de memória são tratadas como arquivos e o File System é único, logo, facilitando a transferência do conteúdo da memória utilizada pelo processo. O mecanismo é o seguinte: o processo é congelado e as páginas em uso são escritas no sistema de arquivos e removidas da memória da fonte. No destino, quando o processo é iniciado, as páginas são recuperadas à medida que se tornam necessárias. Geralmente isto não implica em acesso a disco, já que o servidor tem um cache onde, na maior parte dos casos, as páginas de memória do processo vão estar. Isto tem a vantagem de não gastar tempo copiando páginas que nunca foram usadas e as páginas não ficam espalhadas pela memória de várias máquinas (figura 1d).

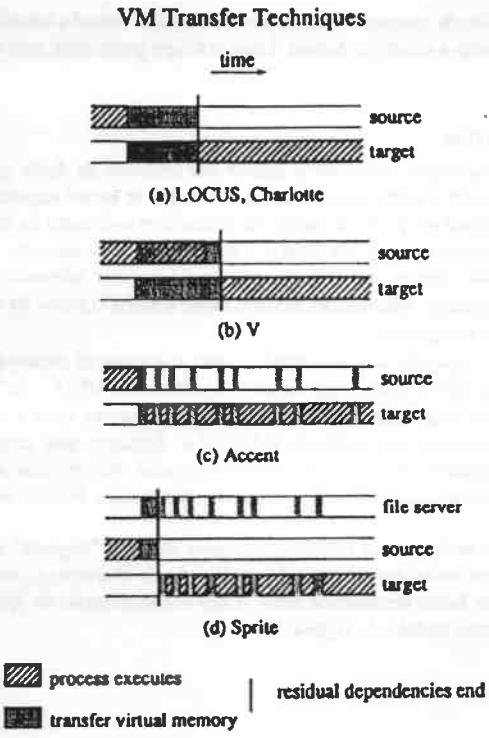

fig 1 - Técnicas de transferência de memória virtual

Nesta figura temos cartas de tempo de técnicas de transferência de memória virtual utilizadas por diversos sistemas operacionais.

em (a) a técnica utilizada em LOCUS e Charlotte, em que o processo é congelado na fonte, as páginas são todas transferidas e o processo é disparado na máquina destino.

em (b) a técnica utilizada em V, em que a execução do processo e a transferência se dão simultaneamente até que todas as páginas sejam transferidas. Então o processo é congelado e apenas as páginas modificadas durante o processo de transferência são copiadas.

em (c) a técnica utilizada em Accent, na qual o processo é disparado na máquina destino e as páginas são recuperadas de acordo com a necessidade do processo.

em (d) a maneira como Sprite transfere memória virtuaL: o processo na máquina fonte congela e envia todas as páginas modificadas para o servidor. Na máquina destino as páginas são recuperadas à medida que são requisitadas.

No caso de pedido de execução remota, não há memória virtual a transferir e o espaço de endereçamento do processo é criado no destino. Logo, o tempo gasto neste caso é muito pequeno.

7.1.2 Arquivos abertos

Em Sprite optou-se por transferir o estado dos arquivos da fonte para o destino. A alternativa seria enviar para o kernel fonte todas as chamadas de kernel responsáveis por acesso a arquivos, mas isto implica em perda de performance muito grande tanto na fonte, que tem que responder, quanto no destino, que tem que esperar o envio da resposta via rede.

O estado de um arquivo tem três partes principais: uma referência que serve para localizar o arquivo no sistema, informações sobre o cache e sobre o ponto de acesso. Cada um oferece um problema para migração.

Se um arquivo é apagado enquanto ainda aberto, o sistema de arquivos se encarrega de atrasar esta operação até que o arquivo seja fechado (semântica do UNIX). Em certas situações, como a descrita a seguir, simplesmente fechar o arquivo na máquina fonte e abri-lo no destino para migrar um processo tem conseqüências indesejadas. Suponha dois processos, P1 e P2, utilizando um mesmo arquivo. P1 apaga o arquivo e termina, P2 tem que migrar, mas se o arquivo for fechado na migração de P2, ele é removido do sistema e P2 não vai completar suas tarefas.

Com relação ao cache, se um arquivo aberto para escrita é "migrado" e o file server for notificado do uso do arquivo no destino antes da notificação da liberação do arquivo na fonte, o cache é desabilitado sem haver necessidade disto. Nesta implementação de Sprite, se o cache é desabilitado, ele não é mais reabilitado (figura 2).

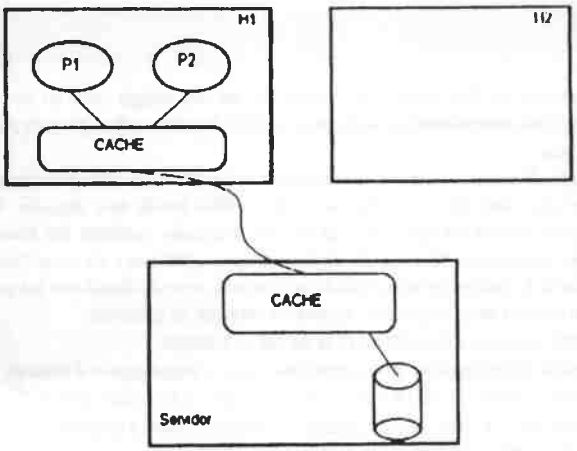

Figura 2a Os processos em H1 compartilham um arquivo para leitura e escrita.

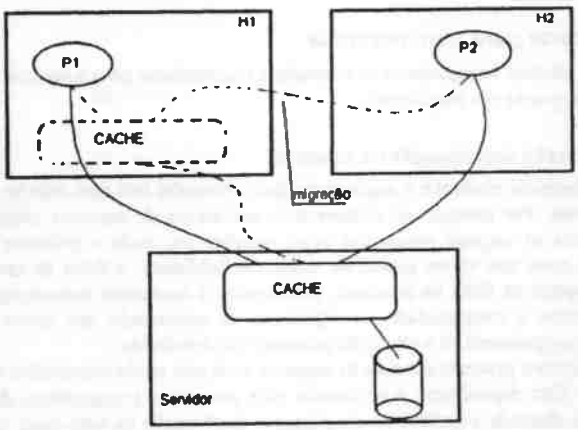

Figura 2b P2 é migrado para H2, isto implica em desabilitar o cache em H1.

Para resolver os dois problemas acima, foi escrito código especial para migração, em que a operação é feita atomicamente, logo, se o arquivo é usado apenas por um processo, o cache continua habilitado.

A posição de acesso a um arquivo pode ser compartilhada entre processos, por exemplo, se um processo abre um arquivo e faz um fork, o filho herda este arquivo. A princípio, os processos estão na mesma máquina, mas se ocorrer migração, nenhum dos hosts pode guardar esta informação, pois como no caso do cache, haveria problemas de consistência. A solução adotada é similar à do cache. Se uma posição de acesso é compartilhada por máquinas diferentes, esta informação passa a estar disponível apenas no servidor de arquivos.

Para cada arquivo, a transferência se dá em três etapas:

A máquina fonte transfere as informações sobre o arquivo para o destino;

O destino informa ao servidor que o arquivo passa a ser usado nele e

O servidor pede à fonte as informações atualizadas sobre o arquivo.

isto garante sincronismo entre as três máquinas envolvidas.

7.1.3 Process control block

Quando um processo é "migrado", existem dois PCB, um na máquina origem, que marca onde o processo foi criado, e outro na máquina destino, que guarda o estado do fluxo. Como não há compartilhamento destas informações e o seu volume não é muito grande, a transferência é feita simplesmente copiando os campos necessários e reiniciando o processo.

7.2 *Suporte para transparência*

Para oferecer transparência de execução e transparência para o usuário, certos pontos do processo de migração são fragilizados.

7.2.1 Migração tem depedência residual

Dependência residual é a dependência que o processo tem com relação às máquinas por onde ele passou. Por exemplo no sistema de transferência de memória virtual em Accent, o algoritmo deixa as páginas espalhadas pelas estações por onde o processo passou. É uma característica ruim sob vários pontos de vista: confiabilidade: a falha de uma das máquinas envolvidas implica na falha do processo; desempenho: é necessária comunicação via rede para manter os dados; e complexidade: os algoritmos de manutenção dos dados tornam-se mais complexos, principalmente se o estado do processo fica distribuído.

Em Sprite o processo depende da máquina onde está sendo executado e da máquina onde foi chamado. Esta dependência é necessária para permitir a transparência de execução. Por exemplo, uma chamada a gettime de um processo local resulta na hora local, uma chamada de um processo remoto causa um pedido de hora, via rede, para a máquina origem.

Infelizmente esta dependência impede que processos sejam "migrados" devido a falha das máquinas onde eles são criados ou executados, o que seria uma característica interessante. Este comportamento seria obtido com uma variação de migração em que a transparência é perdida. Portanto em Sprite, migração serve apenas para balanceamento de carga.

7.2.2 A política de migração implementada

Optou-se por uma política semi-automática em que o SO escolhe em que máquina irá executar o processo se o usuário ou a aplicação assim requisitar. Isto simplifica o serviço pois elimina a necessidade de informações adicionais como quanto tempo a aplicação necessita, se ela é I/O bound ou CPU bound e outras.

O sistema que decide que máquina vai ser usada é composto por uma base de dados centralizada na memória de um servidor e de daemons que verificam a carga de cada estação, informando ao servidor há quanto tempo a máquina está livre. O servidor seleciona a máquina que está há mais tempo livre como receptora de processos.

8. Benchmarks

Três aspectos foram observados:

8.1 Custo de cada etapa da migração

O custo é dado na tabela abaixo:

Selecionar e liberar host	36ms
Migrar o processo "null"	76ms
Transferir informação sobre arquivos abertos	9.4ms/arquivo
"flush" blocos de arquivo	480kbytes/s
"flush" páginas de memória	660kbytes/s
transferir argumentos de execução	480kbytes/s
"fork", executar processo "null" com migração e esperar o processo filho finalizar	81ms
"fork", executar processo "null" localmente e esperar o processo filho finalizar	46ms

Comparando os resultados obtidos nas diversas implementações [1] e [3] nota-se uma pequena melhora no desempenho geral, mas provavelmente o hardware é diferente. Logo, não podemos afirmar se o código foi melhorado ou se a melhora se deve a um hardware melhor.

8.2 Ganho de tempo e aceleração

8.2.1 Descrição dos processos utilizados como benchmarks

PMake é um make que faz uso da facilidade de migração distribuindo arquivos, entre várias máquinas, para serem compilados. A verificação de dependências e a linkagem é feita na máquina onde PMake foi chamado.

LATEX é um compilador de textos com características semelhantes às de PMake.

rcp faz a tranferência, via rede, de um arquivo de 1Mbyte.

8.2.2 Análise dos dados obtidos

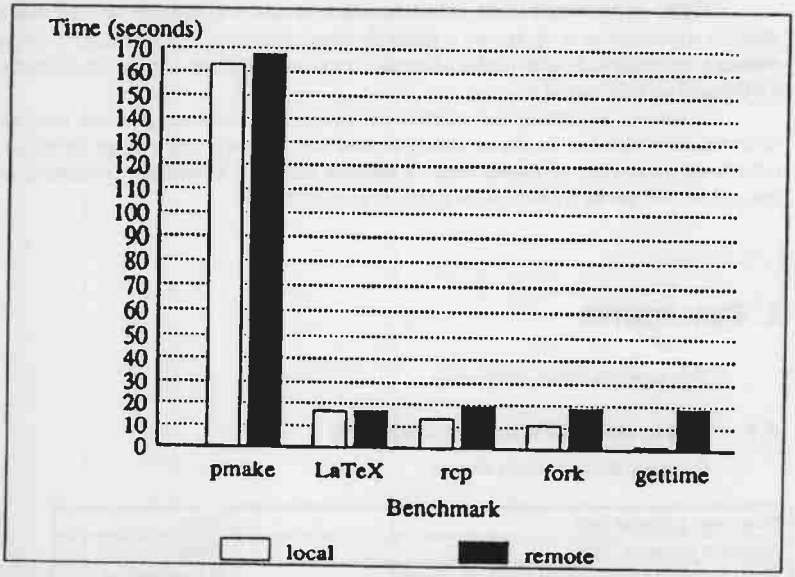

Figure 3: *Comparison between local and remote execution of programs.* The elapsed time to execute CPU-intensive and file-intensive applications such as *pmake* and LaTeX showed negligible effects from remote execution (3% and 1% degradation, respectively). Other applications suffered performance penalties ranging from 42% (*rcp*), to 73% (*fork*), to 3200% (*gettime*).

Name	Description
pmake	recompile *pmake* source sequentially using *pmake*
LaTeX	run LaTeX on a draft of this article
rcp	copy a 1 Mbyte file to another host using TCP
fork	fork and wait for child, 1000 times
gettime	get the time of day 10000 times

Table 2: *Workload for comparisons between local and remote execution.*

Comparando o tempo gasto para executar o processo localmente com o tempo gasto para executá-lo remotamente (sempre em uma máquina), como podemos observar na figura 3, percebe-se que a diferença de desempenho é pequena quando se trata de processos CPU-bound, como pMake e LATEX. Para rcp, em que o uso da rede é necessário, a diferença não é grande e deve ser causada pelo overhead da migração e não por algum gargalo no sistema. Mas para gettime, que é uma chamada de função de kernel dependente de localização (é enviada para a máquina onde o processo foi criado) a diferença é, em média, de 32 vezes.

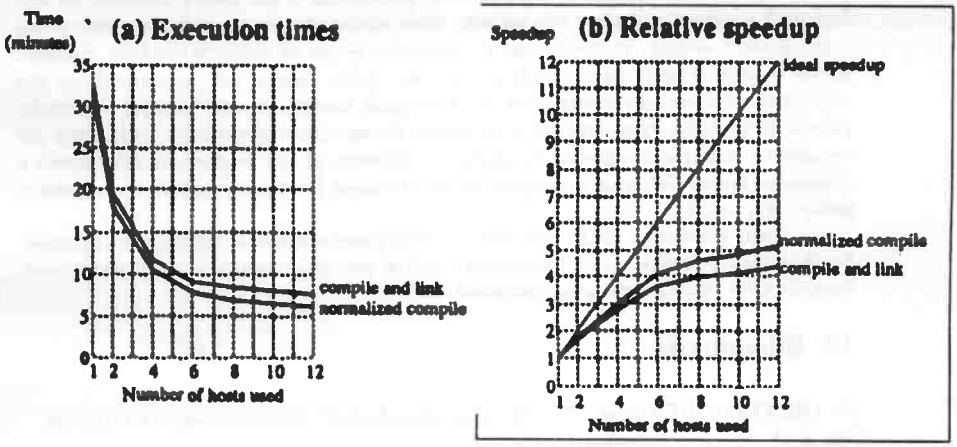

fig. 4 - Benchmarks de pMake
(a) Tempo de execução em função do número de hosts;
(b) Aceleração do processo em função do número de hosts.

Na figura 4, podemos verificar a aceleração obtida no uso de PMake.

O benchmark usado é a compilação e linkagem dos 276 arquivos-fonte de Sprite. Os valores são dados em função da quantidade de máquinas utilizadas.

Como poderia se esperar, a aceleração não é linear, pois existe um overhead crescente em função do número de máquinas utilizadas. Temos como gargalos o file server e a máquina em que PMake foi chamado, que tem que atender a chamadas de kernel dependentes de localização e receber os resultados dos processos remotos. Isto sugere que deve haver um limite para a paralelização de processos, a partir do qual não é mais vantajoso utilizar mais máquinas.

8.3 Uso de migração no ambiente

No ambiente em questão, migração foi mais usada para executar processos remotamente, 86% dos casos. Constatou-se que a migração de processos devido a volta de usuário à estação é muito rara pois, segundo [3], a política de seleção de máquina é boa neste sentido. No entanto o ambiente é peculiar, já que durante a maior parte do tempo as máquinas estão ociosas pois cada usuário tem uma SUN e uma DECStation e dificilmente usa as duas simultaneamente. Dada a constatação acima, é discutível a necessidade de migração no ambiente considerado, já que migração é muito mais complexa e custosa que invocação remota e em 86% dos casos invocação remota seria suficiente.

9. Conclusão

Na opinião dos criadores de Sprite, migração é um recurso valioso, que faz parte de Sprite, e que outros deviam implementar.

No entanto, isto deve ser visto com cuidado, pois o custo de implementação deste serviço é alto, principalmente quando se trata recursos distribuidos, o que parece inevitável no File System. E o sistema avaliado é bem peculiar, tendo muitas estações ociosas, o que para certos sistemas não é verdade. Também existe a constatação de que no ambiente estudado, migração devido à volta de usuários à estação é algo raro. Logo, mesmo com as ineficiências dos mecanismos de invocação remota (rsh), isto (invocação remota), ou uma variação melhorada, poderia ser vantajoso comparado com o mecanismo de migração implementado. Ainda temos que considerar a natureza das tarefas executadas no ambiente, já que o ganho de desempenho é grande para tarefas CPU-bound, mas para tarefas I/O-bound, ou com dependência de máquina, o ganho é bem menor.

Sobre tolerância a falhas, uma menção é feita quando se trata de dependências residuais. Supõe se que o ambiente seja suficientemente confiável para que "crashes" sejam bastante raros, logo, não é um aspecto muito citado nos textos lidos.

10. Bibliografia

[1] **OUSTERHOUT J. et al** - "The Sprite Operating System" *IEEE Computer* - 21(2) 23-26 Feb. 1988

[2] **DOUGLIS F.** - "Experience with Process Migration in Sprite" *Workshop on experience with building distributed (and multiprocessor) systems* - 59-72 Oct 1989

[3] **DOUGLIS F., OUSTERHOUT J.** "Transparent Process Migration: Design Alternatives ant the Sprite Implementation" Resumo de tese - U. C. Berkeley Sep. 1990

O Sistema Chorus

Vanderlei da Silva Porcel

1. INTRODUÇÃO

O projeto Chorus nasceu em 1979 no INRI (Institut National de Recherche en Informatique e Automatique) na França, de uma junção do Projeto Cyclades [Pouz82] e do projeto Esope [Bet70]. Em 1984 experiências UNIX foram trazidas para o ambiente Chorus, pela integração do Projeto Sol ao Projeto Chorus.

O Chorus foi um projeto de pesquisas em Sistemas Operacionais Distribuídos. Três versões foram desenvolvidas e batizadas como *Chorus-V.0*, *Chorus-V.1* e *Chorus-V.2*, baseadas em um Micro-kernel orientado para a comunicação [Zimm81, Guil82, Zimm84,Rozi87]. *Chorus-V.3* é a versão atual desenvolvida pela Chorus Systèmes e integra muitos conceitos superiores de sistemas já conhecidos:

- o envio de mensagens do kernel do Chorus-V.3 é comparável ao V-System da Universidade de Stanford;
- sua memória virtual distribuída e threads são similares àqueles do Mach [Acce86] da Universidade de Carnegie Mellon;
- seu endereçamento de rede incorpora idéias do Amoeba [Mull87] da Universidade de Amesterdan;
- sua forma de nomeação uniforme de arquivos é semelhante ao esquema utilizado pelo Unix dos Laboratórios Bell [Pres86, Wein] (9ª edição).

Inicialmente as pesquisas foram feitas em rede tipo anel, em processadores 8086. Posteriormente utilizou-se processadores Motorola 68000 e depois 68020, interconectados pôr uma Ethernet. A versão atual é escrita em linguagem C++, pôr causa da larga aceitação dessa linguagem na industria de Software e pôr prover as facilidades de programação voltadas para objeto, como classes e heranças.

2. A ARQUITETURA CHORUS

O sistema é composto de um pequeno kernel e um conjunto de sistemas, os quais trabalham num contexto de subsistemas (Fig 1).

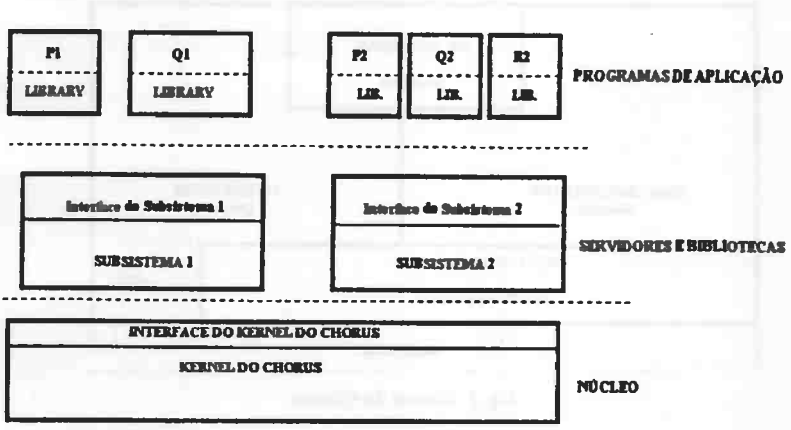

Fig. 1 Arquitetura Chorus

Optou-se pôr estruturar o sistema em 2 níveis, com um núcleo genérico no nível mais baixo e subsistemas quase autônomos, proporcionando aplicações com tarefas tradicionais de sistema operacional. Por essa razão o núcleo não é o coração do sistema operacional específico, mas fornece ferramentas

genéricas projetadas para suportar uma variedade de subsistemas hospedeiros, que podem coexistir no topo desse núcleo.

A proposta central do Chorus consiste de separar as funções do Sistema Operacional em conjuntos de serviços realizados pôr servidores autônomos. Essa separação de funções aumenta a modularidade e a portabilidade geral do sistema.

2.1 O NÚCLEO DO CHORUS

No mais baixo nível o núcleo gerencia os recursos físicos locais de um site (O conceito de site, no sistema Chorus, refere-se a um agrupamento de recursos físicos fortemente acoplados controlados pôr um simples núcleo. A estes recursos físicos inclui-se um ou mais processadores, memória central e dispositivos fixos de I/O.). No nível mais alto ele proporciona um mecanismo de IPC transparente de localização básica dos processos.

O kernel é composto de quatro componentes principais que fornecem serviços locais e globais (Fig. 2):

- Supervisor Chorus que é responsável pelo tratamento básico de interrupções feitas pelo hardware;
- O Executivo de tempo real controla a alocação de processadores, a sincronização e o escalonamento preemptivo baseado em prioridade.;
- O Gerenciador de Memória Virtual é responsável pela manipulação de memória virtual do hardware e recursos locais de memória;
- O Gerenciador de IPC proporciona trocas de mensagens assíncronas e facilidades de RPC independente da localização.

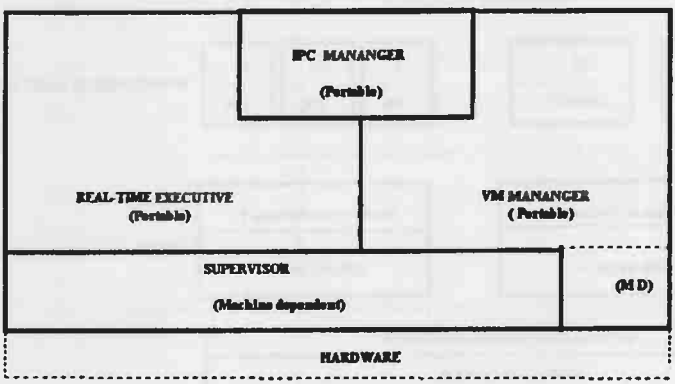

Fig. 2 Kernel do Chorus

Não há interdependência entre esses quatro componentes. Assim, toda a execução do Kernel é de localização transparente. Serviços locais operam com recursos locais e na maioria das vezes podem ser controlados usando apenas informações locais. Serviços globais exigem a cooperação entre os Kernels que no entanto é transparente para os demais subsistemas e aplicações.

O mecanismo padrão do Chorus IPC é a principal forma usada para comunicar com os gerenciadores num sistema Chorus. Pôr exemplo, o gerenciador de memória virtual usa o IPC para requisitar dados remotos para atender uma solicitação de "page fault".

2.2 CONCEITOS BÁSICOS IMPLEMENTADOS PELO KERNEL DO CHORUS

As abstrações básicas implementadas e gerenciadas pelo núcleo são apresentadas na tabela abaixo:

IDENTIFICADOR UNICO	-	NOME GLOBAL
PROCESSO	-	UNID DE ALOC DE RECURSOS
THREAD	-	UNID SEQ DE EXECUÇÃO
MENSAGEM	-	UNIDADE DE COMUNICAÇÃO
PORTA - GR DE PORTAS	-	UNID DE ENDER
REGIÃO	-	UNID DE ESTR. DE ESPAÇO DE END.

Fig 3 Abstrações Gerenciadas pelo Kernel

Todas estas abstrações correspondem a classes de objetos que são particulares do Núcleo Chorus. Tanto a representação e as operações dos objetos são gerenciadas pelo Núcleo.

2.2.1 IDENTIFICADORES ÚNICOS

Todos os objetos Chorus como processos, portas e segmentos são referenciados num estilo global utilizando Identificadores Únicos.

O kernel implementa localizadores de Identificadores Únicos permitindo que entidades do sistema usem esses identificadores para referenciar objetos Chorus sem precisar tomar conhecimento de sua localização. O Kernel garante que um Identificador Único existirá em no máximo um lugar e nunca será reutilizado. O Identificador Único é uma estrutura de 128 Bits, cuja exclusividade é garantida pôr uma escolha randômica no imenso espaço de possíveis identificadores.

Identificadores Locais, são usados dentro do contexto de um servidor para identificar recursos associados a esse servidor. Os Identificadores Locais são inteiros gerados pelo servidor local. Podem ser transmitidos entre entidades dentro do domínio Chorus, mas apenas tem sentido quando referenciando recursos do servidor que o criou.

2.2.2 PROCESSOS

No sistema chorus, é definido o conceito de Actor, que equivale ao conceito de processo. Um processo define um espaço de endereçamento protegido suportando a execução de threads que compartilham os recursos do processo. Um espaço de endereçamento é dividido em espaço de endereçamento do usuário e espaço de endereçamento do sistema. Em um dado site, todos os processos tem espaço de endereçamento de sistema idêntico e seu acesso fica restringido aos níveis privilegiados de execução (Fig. 4) .

O Chorus define três tipos de processos : processos usuários, processos do sistema e processos de supervisão. Estas definições são baseadas nos conceitos de confiabilidade e privilégio de execução. Um processo é confiável se o kernel reconhece seus direitos de executar operações sensíveis do kernel. O privilégio de execução refere-se a habilitação de um processo para executar instruções privilegiadas, o que é controlado pelo registrador de estatus da CPU.

- Processos Usuários não são confiáveis e nem tem privilégio de execução;
- Processos do Sistema são confiáveis mas não tem privilégios;
- Processos Supervisores são confiáveis e tem o privilégio de execução.

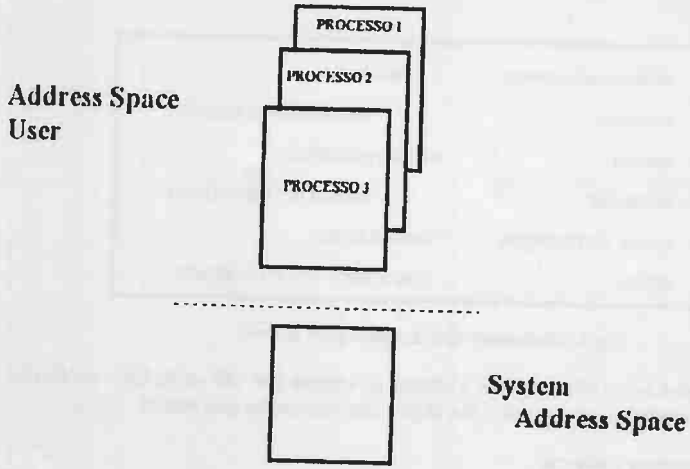

Fig. 4- Espaços de Endereçamento de Processos

Pelo fato de cada processo estar vinculado a um site, todos os threads executam neste site e o contexto fica precisamente definido. O estado de um processo pode ser facilmente determinado e decisões dependentes do contexto podem ser tomadas rapidamente. Uma falha em um site causa a quebra total de seus processos. Não há a possibilidade de uma quebra parcial de um processo.

2.2.3 THREADS

Um thread é a unidade de execução no Sistema Chorus. Um thread é sempre parte de exatamente um processo, que determina o desenvolvimento da execução de um thread. Dentro de um processo podem ser criados muitos theads. Quando um site suporta múltiplos processadores, os threads de um processo podem rodar em paralelo em diferentes processadores.

Threads se comunicam e se sincronizam trocando mensagens usando o mecanismo de IPC do Chorus. Entretanto como todos os threads compartilham o mesmo espaço de endereçamento, mecanismos de comunicação e sincronização baseados em memórias compartilhadas podem ser utilizados.

3. ENTIDADES DE COMUNICAÇÃO

3.1 MENSAGENS

Enquanto os threads de um mesmo processo são capazes de fazer a comunicação usando primitivas de memória compartilhada, o mecanismo básico de comunicação entre threads de processos distintos (possivelmente em sites diferentes), é a troca de mensagens usando portas.

Usando a união entre o gerenciamento virtual e o IPC, longas mensagens podem ser transferidas de maneira eficiente usando técnicas copy-on-write ou eventualmente pela simples movimentação de páginas descritoras.

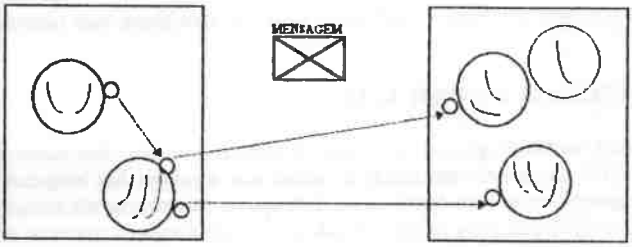

Fig. 5 Mensagens

3.2 PORTAS

Uma porta representa um endereço ao qual mensagens podem ser mandadas e uma coleção de mensagens pode ser armazenadas. Ao ser criada, cada porta é associada a um único processo, sendo que somente threads deste processo podem receber mensagens nessa porta. Uma porta pode migrar de um processo para outro. Essa migração pode ser aplicada também para as mensagens acumuladas nessa porta.

Quando uma porta é criada, o Kernel retorna um Identificador Local e um Identificador Único para designar essa porta. O Identificador Local pode ser usado apenas dentro do contexto do processo ao qual estiver associado. Mensagens sempre carregam o Identificador Único da porta ou grupo de portas para onde estão sendo mandadas.

3.3 AGRUPAMENTO DE PORTAS

As portas podem ser agrupadas em Grupo de Portas (Fig. 6). A abstração grupo de portas é ortogonal ao conceito de processo e amplia a semântica de transmissão de mensagens entre threads, permitindo que as mensagens sejam direcionadas a um grupo inteiro de threads.

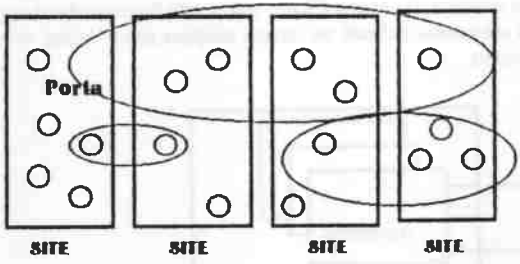

Fig. 6 Agrupamento de Portas

Operações executadas em grupos, como inserção e deleção de portas devem ser controladas pelo kernel para que seja garantida a segurança. O kernel fornece ao criador de um grupo uma chave

manipuladora de grupo. Esta chave precisa ser fornecida para modificar o conteúdo do grupo e pode ser livremente transmitida entre processos autorizados a manipular um grupo.

O Identificador Único de um grupo e a chave manipuladora fornecida tem a seguinte relação

Identificador Único do Grupo = f (CHAVE); onde f é uma função não reversível conhecida pelo kernel.

3.4 SEMÂNTICA DE COMUNICAÇÃO

O IPC no Chorus permite aos threads, se comunicarem remetendo mensagens assíncronas ou utilizando RPC. Quando enviam mensagens assíncronas, o emissor fica bloqueado apenas durante o tempo do processamento local da mensagem. Este tipo de comunicação não assegura que a mensagem tenha sido recebida pela porta destino. Quando a porta não é válida o remetente não é notificado e a mensagem é descartada.

Pôr outro lado, o protocolo de RPC permite construção de tarefas usando o modelo "cliente-servidor". O RPC assegura que a resposta recebida pelo cliente corresponde ao que foi solicitado pelo cliente. O RPC permite saber se o servidor "quebrou" depois de emitir a resposta, ou se o caminho foi interrompido.

Quando remetidas mensagens a um grupo de portas vários modelos de endereçamento são possíveis:

- broadcast para todas as portas no grupo;
- envio para qualquer porta do grupo;
- envio para uma porta do grupo, localizada em um site específico;
- envio para uma porta no grupo, localizada no mesmo site de um dado Identificador Único.

4. GERENCIAMENTO DE MEMÓRIA VIRTUAL.

4.1 REGIÕES

O espaço de endereçamento de um processo é dividido em regiões. Regiões dentro do espaço total de endereçamento de sistema podem ser manipuladas somente pôr threads credenciados. Threads que lêem, escrevem e executam regiões dentro do espaço de endereçamento do sistema evitam overhead de troca de contexto. O sistema Chorus utiliza esta funcionabilidade para otimizar o IPC entre processos de um subsistema rodando na mesma máquina (site). Assim, evita-se também overhead de cópias de mensagens.

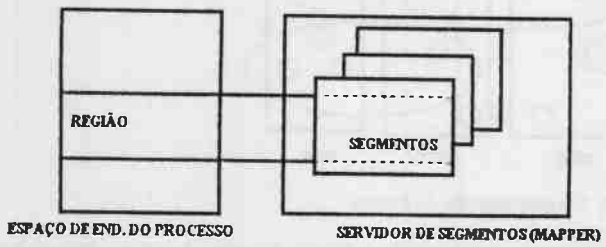

Fig. 7 Regiões e Segmentos

4.2 SEGMENTOS

A unidade de informação trocada entre a memória virtual do sistema e os fornecedores de dados é o segmento. Segmentos são globais e identificados pôr uma capability. Segmentos são gerenciados pôr processo de sistema chamados mappers.

O kernel administra um cache local das páginas físicas pôr segmento. Este cache contém páginas obtidas dos mapeadores que podem ser usadas para preencher futuras solicitações do mesmo segmento de dados, que sejam originadas de chamadas explícitas ou como resultado de "page fault".

Para um site, a consistência de um segmento compartilhado de leitura de segmentos (primitivas sgRead), entre regiões em diferentes processos é assegurada, pois eles compartilham o mesmo cache de memória física (Fig. 8). Algoritmos para lidar com estes problemas de coerência de memória compartilhada são propostas em [Lin86].

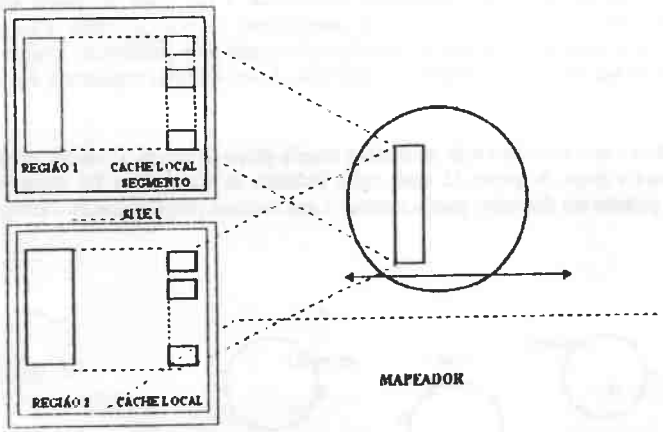

Fig. 8 Representação do Segmento no Kernel

5. SUPORTE PARA RECONFIGURAÇÃO DINÂMICA

5.1 RECONFIGURAÇÃO USANDO MIGRAÇÃO DE PORTAS

A figura 9 ilustra a migração de uma porta do servidor 1 para o servidor 2 sem interrupção dos serviços. Migrações permitem, pôr exemplo, que uma nova versão de um serviço seja dinamicamente instalado e ativado.

Quando uma porta é migrada, qualquer mensagem não concluída associada a essa porta irá migrar com ela. Dessa forma o cliente de um serviço pode continuar operando sem perturbação durante a migração, e a comunicação não é interrompida.

Migrações de portas requerem participação ativa dos servidores envolvidos. Se houver quebra de um servidor, sua porta não será migrada.

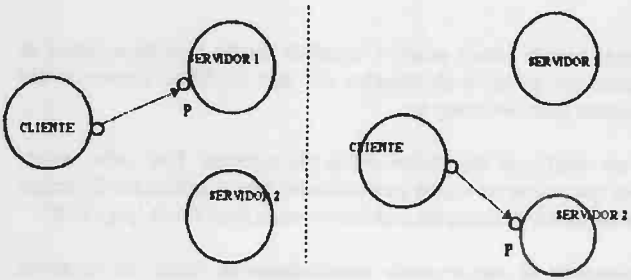

Fig. 9 Reconfiguração usando Migração de Portas

A abstração Grupo de Portas, pôr outro lado, oferece uma estrutura para reconfigurar um serviço passivamente. Mensagens que são diretamente relacionadas a um grupo de portas serão automaticamente desviadas para a porta que é capaz de proporcionar o serviço desejado. Caso um membro do grupo torna-se inválido. Neste caso a reconfiguração requer uma participação mínima de clientes ou servidores de um serviço, permitindo a recuperação de um distúrbio temporário durante uma comunicação.

A figura 10 ilustra essa reconfiguração automática usando grupo de portas. O cliente envia o pedido diretamente para o grupo de portas G, onde estão incluídas as portas P1 e P2. Quando o servidor 1 quebra, os pedidos são desviados para o servidor 2 que continua proporcionando o serviço.

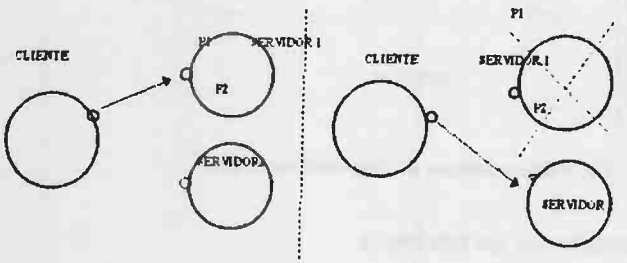

Fig. 10 Reconfiguração usando Grupo de Portas

6. SEGURANÇA

Duas outras abstrações, mostradas na tabela abaixo, são gerenciadas pelo kernel e pelos servidores de subsistemas de forma conjunta.

CAPABILITY	**Unidade de controle de acesso**
IDENTIFICADOR DE PROTEÇÃO	**Unidade de Autenticação**

6.1 CAPABILITIES

Alguns objetos não são diretamente pelo Kernel Chorus, mas pôr serviços externos. Estes objetos são conhecidos globalmente como Capabilities. Uma capability é composta pôr um Identificador Único e uma Chave. O Identificador Único nomeia o servidor que gerencia o objeto. A chave representa um instrumento específico do servidor que identifica o objeto e os direitos de acesso associados com a requisição (Fig. 11). A estrutura e a semântica de uma chave é definida pôr seu servidor.

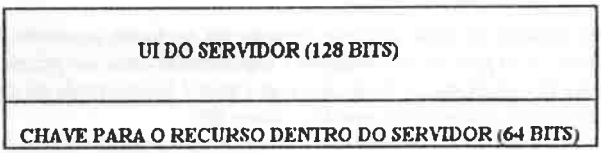

Fig. 11 Capability

6.2 AUTENTICAÇÃO

Dois mecanismos são fornecidos em Chorus:

- Os recursos podem ser identificados dentro de seus servidores pôr uma chave do servidor dependente. Esta chave, quando usada em conjunto com a porta (Identificador Único) do servidor, cria uma capability que pode ser usada para referenciar seguramente um recurso.

- Processos e portas recebem Identificadores de Proteção mandadas junto com as mensagens. Ao receber mensagens os processos podem utilizar estes identificadores para autenticação. Identificadores de Proteção podem ser alterados apenas pôr threads com confiabilidade.

7 OUTROS

7.1 GERENCIAMENTO DE REDES

O gerenciador de rede proporciona facilidades de alto nível para a comunicação usando Chorus. Esse gerenciador, é construído pôr três módulos independentes :

- Interface de Alto Nível (High Interface) - implementa o Chorus IPC remoto, RPC e manuseio de erros;
- Modulo de Comunicação (Comunicação Case) - implementa protocolo padrão de comunicação e serviços como o protocolo OSI;
- Interfaces de Baixo Nível (Low Interface) - gerencia requisições para dispositivos de gerenciamento de redes que implementam de drivers de dispositivos de rede.

7.2 SUBSISTEMAS

Subsistemas são conjuntos de processos que trabalham juntos para exportar uma interface de programação de aplicações. O subsistema mais estudado no Chorus é o Chorus/Mix que trouxe o

ambiente Unix para dentro do ambiente Chorus.exporta abstrações de alto nível do Sistema Operacional; tais como objetos, modelos de proteção e objetos representando dados. Para construir essas abstrações de alto nível, são usadas primitivas Kernel Chorus.

Subsistemas podem exportar estas abstrações fornecendo acesso a elas através de Traps. Código e dados para implementar essas abstrações podem estar carregados no espaço sistema para melhorar a performance no acesso. Processos de subsistemas comunicam-se um com o outro pôr meio do IPC do Chorus.

No exemplo da figura 12, vários processos usuários são mostrados acessando facilidades em um subsistemas Chorus. Uma parte de um subsistema é implementada como um processo do sistema, executando em espaço de endereçamento do sistema, e uma parte é implementada em espaço usuário. Os servidores de subsistemas se comunicam usando o Chorus IPC.

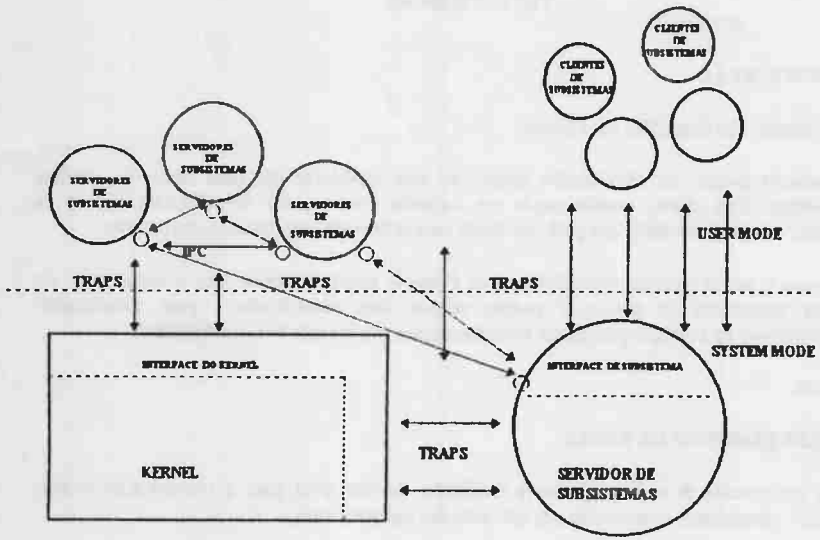

Fig 12 Estrutura de um Subsistema

8. CONCLUSÃO

O Chorus, sem dúvida alguma, foi concebido para ser Sistema Operacional largamente comercializado, tendo em vista enorme capacidade de seu kernel aceitar em seu topo os mais variados Sistemas Operacionais disponíveis no mercado de software. Testes com Unix já foram concluídos e testes com o OS2 da IBM já tiveram início.

É um sistema que ao invés da riqueza de seu design deu preferência a performance. Características como mecanismos rigorosos de segurança, protocolos de aplicações orientadas e estratégias de tolerância a falhas, não aparecem no núcleo do Chorus.

Minha opinião é que o Sistema Operacional Chorus, dada sua possibilidade de permtir a coexistência de outros sistemas operacionais sobre o topo de seu núcleo, pode ser escolhido como o Sistema ideal para a migração, "sem trauma", de um Sistema Operacional Normal para um Sistema Operacional Distribuido.

9 BIBLIOGRAFIA CHORUS

9.1 CHORUS - V0

[Guil82] Marc Guillemont, "Intégration du Système Réparti CHORUS dans le Langage de Haut Niveau Pascal, "Thèse de Docteur Ingénieur, Université Scientifique et Médicale, Grenoble, france, (Mars 1982), 162 p.

[Zimm81] Hubert Zimmermann, Jean Serge Banino, Alain Caristan, Marc Guillermont, and Gérard Morisset, "Basic Caoncepts for the Support of distributed Systems: the CHORUS Approach, "in IEEE 2nd. International Conference on Distributed Computing System Proc., Versailles, France, (April 1981), pp. 60-66.

9.2 CHORUS - V1

[Zimm84] Hubert Zimmermann, Marc Guillermont, Gérard Morisset, and Jean-Serge Banino, "CHORUS : a Communication and Processing Architeture for Distributes Systems," Research Report, INRIA, Recquencourt, France, (September 1984).

9.3 CHORUS - V2

[Rozi87] Marc Rozier and José Legatheoux- Martins, "The CHORUS Distributed Operating System: Some Design Issues, " in Distributed Operating Systems, Theory and Pratice, Yakup Paker, Jean-Pierre Banâtre and Muslim Bozyigit ed., NATO ASI Series, vol. F28, Springer Verlag, Berlin, (1987), pp. 261-287.

10. REFERÊNCIAS

[Acce86] Mike Acceta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young, "Mach: A New Kernel Foundation for UNIX Development, " in *USENIX Summer'86 Conference* Proc., Atlanta, GA, (9-13 June 1986), pp. 93-112.

[Bét70] Claude Bétourné, Jacques Boulenger, Jacques Ferrié, Claude Kaiser, Sacha Krakowiak, and Jacques Mossière, "Process Manangement and Resource Sharing in the Multiaccess System ESOPE," Comunications of the ACM, vol. 13, no. 12, (December 1970).

[Li86] Kai Li, 'Shared Virtual Memory on Loosely Coupled Multiprocessors," PhD, Thesis, Yale University, New-Haven, CT, (September 1986).

[Mull87] Sape J. Mullender et al., *The Amoeba Distributed Operating System: Selected Papers* 1984-1987, CWI Tract No. 41, Amsterdam, Netherlands, (1987), 309 p.

[Pouz82] Louis Pouzin et al., The CYCLADES *Computer Network - Towards Layered Network Architectures*, Monograph Series of the ICCC, 2, Elsevier Publishing Company, Inc, New-York, NY, (1982), 387 p. ISBN 0-444-86482-2.

[Pres86] David L. Presotto, "The Eight Edition UNIX Connection Service," in EUUG *Spring '86 Conference Proc.*, Florence, Italy, (21-24 April 1986), 10 p.

[Wein86] Peter J. Weinberger, "The Eight Edition Remote Filesystem," in EUUG *Spring '86 Conference*, Florence, Italy, (21-24 April 1986), 1 p.

O Sistema ISIS

Jorge Nakahara Jr.

1 Introdução

O sistema ISIS foi desenvolvido inicialmente na **Cornell University**, por **Kenneth P. Birman e Thomas A. Joseph**, entre outros, no período de 1985 a 1989.

ISIS consiste de um conjunto de ferramentas para processamento distribuído e confiável. O objetivo é permitir o desenvolvimento de software distribuído usando o conceito de comunicação de grupo (de processos) e ferramentas de programação de grupo.

O sistema pode ser utilizado com interfaces em C, C++, SmallTalk, ADA, Fortran, Modula 2, entre outras linguagens.

A partir de 1989, ISIS foi estendido e reformulado como um produto comercial com o mesmo nome, mas com performance muito melhor, interfaces mais genéricas, mais confiável e escalável. Um grande número de aplicações de usuário foram desenvolvidas a partir do sistema, e ele foi portado para SunOS, assim como para a maioria das plataformas UNIX, VMS, sistemas operacionais para PC's (NT, Windows), e outros ambientes. O sistema resultante é comercializado pela "ISIS Distributed Systems" uma empresa fundada pelos seus autores.

ISIS está disponível, para o ambiente acadêmico, é usado para uma ampla gama de aplicações comerciais, e continua sendo estendido e melhorado. Um sistema sucessor chamado "Horus", que implementa o modelo de comunicação de grupo em microkernel, já está disponível. (Segundo a mitologia egípcia, Horus é o filho de ISIS).

Duas idéias principais permeiam ISIS : **grupos de processos síncronos virtuais**, e **broadcast atômico**, realizado de diversas formas.

Antes de examinar as várias formas de sincronismo que ISIS distingue, vamos definir alguns conceitos.

Um processo é dito **defeituoso** se seu comportamento não corresponder ao descrito pelo seu algoritmo durante a execução; caso contrário, será um **processo correto**.

Um **broadcast confiável**[4] garante 3 propriedades:

- validade: se um processo correto envia uma mensagem (broadcast) M, então, todos os processos corretos eventualmente recebem M, ie. toda mensagem tem remetente (mensagens não aparecem espontâneamente);
- concordância: se um processo correto recebe uma mensagem M, então, todos os demais processos corretos eventualmente recebem M, ie, a semântica de grupo é respeitada;
- integridade: para toda mensagem M, todo processo correto recebe M no máximo uma vez, e somente se algum processo correto enviou M.

Entretanto, um broadcast confiável não impõe restrições quanto a ordem de recebimento das mensagens. Quando a ordem de recebimento de mensagens enviadas por um mesmo remetente deve obedecer a mesma ordem de envio, define-se um broadcast mais restrito, chamado **broadcast de fila (FIFO)**.

Um **broadcast causal** é um broadcast de fila no qual se exige que as mensagens sejam recebidas de acordo com uma **relação de precedência causal**, ou ordem causal conforme definida em [3]. Porém, um broadcast causal permite que mensagens **não causalmente relacionadas** sejam recebidas em qualquer ordem. Isto é evitado, utilizando-se um **broadcast atômico** que exige que todos os processos corretos recebam todas as mensagens na mesma ordem. Esta ordenação total no recebimento de mensagens assegura que todos os processos corretos têm a mesma "visão" do sistema, de modo que podem agir consistentemente sem necessidade de comunicação adicional.

Formalmente, um broadcast atômico é um broadcast confiável que satisfaz a seguinte propriedade de **ordem total**: se dois processos corretos p e q recebem mensagens M e M', então: p recebe M antes de M' se e somente se q recebe M antes de M'. Um **broadcast atômico em fila** combina as caracteristicas de um broadcast em fila e um broadcast atômico, sendo portanto mais restritivo que ambos.

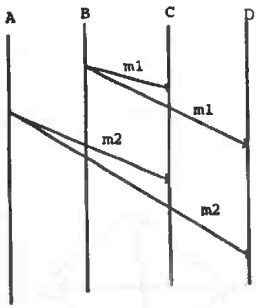

Figura 25: m1 e m2 são mensagens broadcast atômicas em fila

Um **broadcast atômico causal** é um broadcast confiável que satisfaz as exigências de ordem causal e total, sendo mais forte que o broadcast atômico em fila e o broadcast causal.

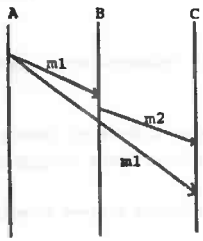

Figura 26: m1 e m2 são mensagens broadcast atômicas causais

ISIS implementa primitivas de comunicação que constituem diferentes formas de broadcast atômico causal, onde a ordem causal das mensagens é utilizada como método de sincronismo entre membros de grupo. Mais adiante veremos detalhes de implementação destas primitivas.

2 Grupos de Processos

Dois tipos de grupos são possíveis:

- grupos anônimos: são semelhantes a grupos de interesse do tipo USENET; um processo publica informações a respeito de algum assunto específico e os demais processos que se interessarem no assunto se subscrevem ao grupo; seus membros não se conhecem e agem de forma relativamente independente.

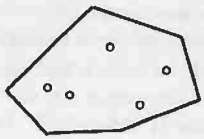

Figura 27: grupo anônimo

- grupos explícitos: nestes grupos todos se conhecem e cooperam entre si e administram a inclusão/exclusão de membros do grupo.

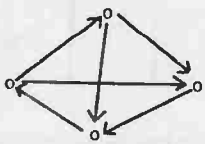

Figura 28: grupo explícito

Entre os problemas que devem ser resolvidos a fim de implementar software distribuído para grupos de processos podemos citar :

- suporte para comunicação de grupo: endereçamento, atomicidade de falha, e ordem de entrega de mensagens.
- sincronização: é necessário haver uma ordem na execução de tarefas, principalmente quando membros de um grupo agem independentemente de acordo com informações compartilhadas.

Veremos que ISIS resolve estes dois problemas através de primitivas de comunicação e um ambiente síncrono virual.

3 Modelo do Sistema

Métodos convencionais de comunicação entre processos como: datagramas, RPC e fluxos de dados não fornecem garantias de comunicação confiável, permitindo perda de mensagens, eventual duplicação, alteração de ordem, etc.

Cada um deles, porém, possui uma relação custo/benefício e pressupõe algumas características da rede. Vamos fazer também algumas considerações relativas ao tipo de ambiente de rede adequado ao uso de ISIS.

Vamos supor que o sistema é constituido de uma WAN, composta de várias LANs interconectadas, sendo cada LAN composta por uma coleção de máquinas, conectadas através de dispositivos com alta taxa de comunicação e baixa latência. Os relógios nas diversas máquinas não precisam estar estreitamente sincronizados.

Em uma LAN assumimos que mensagens poderão eventualmente ser perdidas, duplicadas, chegar fora de ordem, ou serem descartadas por falta de buffer.

Também se supõe que partições na rede são raras, e que os níveis mais baixos do sistema são responsáveis pelo fluxo de controle e tratamento de perda e restauração da ordem de mensagens.

4 Modelo de Falha

Será assumido que se processos ou processadores falharem, então eles serão reinicializados, isto é, não há ações corretivas. Porém, há o caso de problemas transientes como falta de energia ou ausência de resposta que se assemelham a falhas. Como um objetivo de ISIS é garantir progresso na comunicação mesmo na presença de falhas, vamos integrar a camada de transporte de comunicação com a tarefa de detecção de falha usando um modelo **fail-stop**. Isto significa que nos casos transientes o processo será considerado falho.

Para isto ISIS utiliza um protocolo para manter uma lista de pertinência de cada grupo. Apenas processos citados na lista são considerados vivos pelo sistema, e processos que não respondem ou quebrem momentaneamente são excluídos da lista. Tal lista é gerenciada pelo **run-time system**. Se um processo é excluído e posteriormente retorna a comunicar-se com o grupo, este é forçado a se reinicializar ou a executar um protocolo de reconexão. A camada de transporte é responsável tanto pela exclusão de um membro de grupo, como pelo restabelecimento de comunicação mantendo a consistência do modelo.

5 Endereçamento de Grupos

Consideremos o problema de associar endereços de grupos com listas de pertinência em uma aplicação onde os membros de grupo variam dinamicamente. Uma solução é manter um **serviço de pertinência** que mantém uma tabela de membros de grupos e seus respectivos endereços.

Um processo poderia, assim, transmitir uma mensagem para grupos via um **servidor de pertinência**, ou, obter o endereço do grupo através do servidor e sempre utilizar tal endereço para se comunicar com o grupo. A primeira alternativa é melhor para interações ocasionais, a segunda opção é preferível para comunicações frequentes.

Ainda há o problema de manter sincronização entre composição de grupo, e a comunicação em si, ie. quando uma mensagem multicast é enviada espera-se que todos os seus membros a recebam, mas a composição de um grupo varia dinamicamente.

Uma solução é considerar a tabela de pertinência de grupo como um **dado compartilhado** pelo remetente de uma mensagem **multicast** e o algoritmo usado para gerenciar os membros de um grupo (o dado seria um tipo de semáforo).

Um multicast obtém acesso exclusivo (de leitura) à lista de membros de grupo, até que a mensagem tenha sido recebida por todos os seus membros. Uma inclusão na lista de pertinência corresponderia a uma escrita no dado, que só poderia ser executada se o dado não estiver sendo acessado. Dessa forma mensagens são recebidas por grupos apenas quando não há alteração nos seus membros. O problema é o custo de tal implementação.

ISIS usa estas idéias, porém, não emprega acesso exclusivo a dados compartilhados e nem um servidor de pertinência, mas dissemina a informação de pertinência entre os membros de forma integrada

com o envio de mensagens a fim de garantir consistência de composição de grupo a todos os seus membros, ie. todos os membros têm a mesma informação de quais processos pertencem e quais não (mais) pertencem ao grupo, de modo que uma mensagem é recebida apenas pelos processos pertencentes ao grupo no momento instantâneo do seu recebimento pelo grupo (atomicidade de falha) .

6 Sincronia Virtual

Em um **sistema síncrono** os eventos ocorrem de forma estritamente sequencial com cada evento ocorrendo teoricamente de forma **instantânea** i.e. em tempo nulo. Por exemplo, um processo A envia uma mensagem para os processos B e C, (figura abaixo) que, se supõe, chega instantaneamente a todos eles.

Sistemas síncronos são impossíveis na prática, ie. o máximo que podemos obter é um sistema menos rígido em relação ao tempo de comunicação.

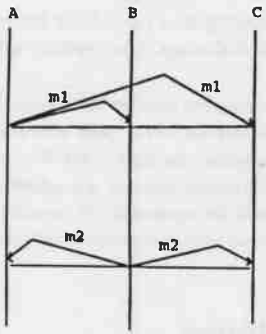

Figura 29: um sistema síncrono

Um sistema fracamente síncrono é um sistema síncrono no qual os eventos requerem um tempo finito não nulo, mas os eventos ocorrem na mesma ordem em todos os destinos. Tais sistemas são factíveis.

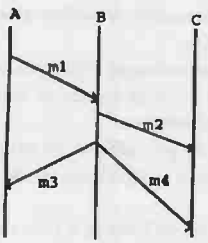

Figura 30: um sistema fracamente síncrono

Um sistema síncrono virtual é um sistema fracamente síncrono no qual as restrições de ordem foram relaxadas.

Em um sistema distribuído, dois eventos são ditos **causalmente relacionados**[7] se o comportamento do segundo evento pode ter sido influenciado de alguma forma pelo primeiro. Assim, se A envia uma mensagem para B, que o inspeciona, e então, B envia uma mensagem para C, a segunda mensagem está causalmente relacionada à primeira, pois, seu conteúdo pode ter sido influenciado em parte pela primeira mensagem. Se tal influência é verdadeira ou não, é irrelevante. A relação é válida se é possível ter havido alguma influência.

Dois eventos **não relacionados** são ditos **concorrentes**. Se, por exemplo, A envia uma mensagem para B, e, quase simultaneamente C envia uma mensagem para B, estes dois eventos são concorrentes, pois, nenhuma mensagem pode ter influenciado a outra. O significado do **sincronismo virtual** é que se duas mensagens são causalmente relacionadas, então, todos os processos envolvidos devem recebê-las na mesma ordem, e na ordem correta. Se porém, as mensagens forem concorrentes, nenhuma garantia é fornecida de que o sistema irá entregá-las na mesma ordem.

Quando dissemos anteriormente que "todos os membros de um grupo são atualizados instantaneamente", significa que os eventos de recebimento de mensagem são concorrentes e totalmente ordenados com respeito aos eventos de alteração na composição de grupo.

7 Primitivas ISIS

As principais primitivas de comunicação em ISIS são:

ABCAST: oferece sincronismo fraco de comunicação e é usado para transmitir dados para membros de um grupo;

CBCAST: oferece sincronismo de comunicação virtual e também é usado para transmissão de dados para grupos;

GBCAST: é semelhante a ABCAST, porém, é usado para gerenciamento de membros de um grupo.

Originalmente [T92], a primitiva ABCAST utilizava um protocolo de 2 fases: um remetente A unia um **timestamp** (na verdade um número sequencial) a uma mensagem antes de enviá-la para cada membro de um grupo. Cada um deles selecionava um **timestamp** maior do que qualquer outro timestamp já utilizado, e o enviava de volta para A. Por sua vez, A selecionava o maior timestamp recebido, e enviava uma mensagem de **confirmação** com este maior timestamp de volta a todos os membros do grupo em ordem crescente dos timestamps recebidos. Só depois de receber a confirmação, os demais processos passam a mensagem multicast para o processo de aplicação. Pode-se provar que este protocolo garante que todas as mensagens serão entregue a todos os processos na mesma ordem.

Considere um sistema com dois processos p1 e p2, enviando mensagens para um grupo G com membros g_1 e g_2. Suponha que p_1 envia m_1 para G e, concorrentemente, p_2 envia m_2. Se, por exemplo, m_1 e m_2 atualizam dados em estruturas de dados replicadas no grupo, uma comunicação de grupo que assegure que as cópias permaneçam idênticas em ambas as estruturas satisfaz uma **ordenação de recebimento atômica**. ABCAST é uma primitiva que satisfaz esta propriedade, mas possui custo elevado, pois uma mensagem ABCAST só pode ser recebida quando se tem certeza que nenhuma mensagem ABCAST anterior deixou de ser entregue. Isto, eventualmente, aumenta a latência de comunicação.

Devido ao custo elevado, foi desenvolvida a primitiva CBCAST que apenas garante uma ordem consistente de entrega para mensagens que são *causalmente relacionadas*. Se um grupo possui n membros, cada processo mantém um vetor com n componentes, um por membro. O i-ésimo componente do vetor é o número da última mensagem recebida sequencialmente do processo i. Os vetores são gerenciados pelo **run time system** e não pelos processos, e são inicializados com 0 na criação do grupo.

Quando um processo A deseja enviar uma mensagem m_1, ele incrementa sua própria posição no vetor, e envia o vetor como parte da mensagem. Quando m_1 chega a um membro B do grupo, verifica-

se se m_1 depende de alguma mensagem que B ainda não recebeu. Se a primeira posição do vetor na mensagem é um a mais que o valor do vetor de B, o que é esperado (e exigido) para uma mensagem de A, então a mensagem é aceita e repassada para B.

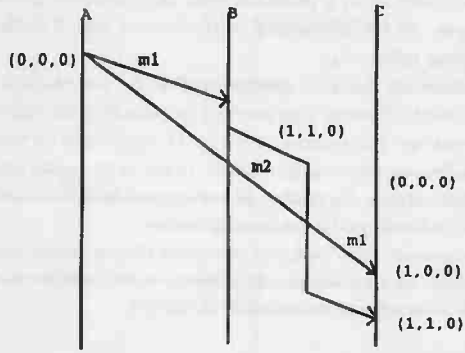

Figura 31: m2 não pode ser recebida enquanto m1 não chegar

Agora, B envia uma mensagem m_2 para C, que chega antes de m_1. Pelo vetor, C observa que B recebeu uma mensagem de A antes de m_2 ter sido enviada, e, como C não recebeu nada de A, m_2 é armazenada até a chegada de m_1.

O algoritmo geral para decidir se uma mensagem deve ser entregue a um processo, ou armazenado, é o seguinte: seja v_i o i-ésimo componente do vetor de **timestamps** correspondente a mensagem recebida, e l_i o i-ésimo componente do vetor armazenado no destinatário da mensagem. Suponha que a mensagem foi enviada pelo processo j.

A primeira condição para entrega imediata é $v_j = l_j + 1$. Isto significa que esta é a mensagem seguinte, na ordem, vinda de j, i.e. não existem mensagens intermediárias (mensagens do mesmo remetente são sempre causalmente relacionadas).

A segunda condição para aceitação é $v_i \leq l_i$ para $i \neq j$. Isto significa que o remetente não recebeu qualquer mensagem que o destinatário ainda não tenha recebido. Se ambas as condições forem satisfeitas o **runtime system** pode entregar a mensagem sem atraso, caso contrário, a mensagem deve esperar todas as demais mensagens causalmente relacionadas a ela.

Basicamente, CBCAST pode ser usada quando multicasts conflitantes são univocamente ordenados por uma única sequência causal. Neste caso, CBCAST garante que todos os multicasts conflitantes serão "vistos" na mesma ordem por todos os destinatários: ie. na ordem causal. Tal sistema é virtualmente síncrono, pois, o resultado da execução é o mesmo caso fosse empregada uma ordem de recebimento atômica.

8 Grupos Preferenciais

A fim de obter um compromisso entre simplicidade e garantia de informações precisas para gerenciamento de grupos, ISIS oferece quatro tipos de grupos de processos que diferem no tipo de interação dos processos com os grupos.

. grupos homogêneos: nestes grupos, os membros cooperam entre si ativamente, e todos os membros são iguais entre si;

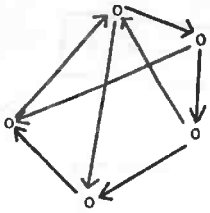

Figura 32: grupo homogêneo

. grupos cliente-servidor: em ISIS, qualquer processo pode se comunicar com qualquer grupo desde que tenha o endereço do grupo e a permissão para acesso. Porém, se um processo fora do grupo se comunica intensamente com o grupo, a comunicação seria otimizada se o processo se cadastrasse como cliente do grupo (ie. otimizando o protocolo de comunicação);

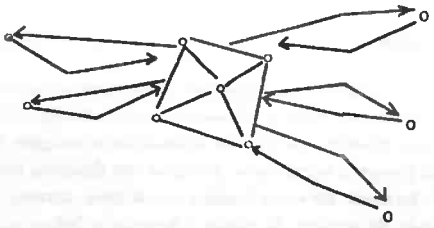

Figura 33: grupo cliente-servidor

. grupos de difusão: é um tipo de grupo cliente-servidor no qual os clientes se registram, porém, os membros do grupo enviam mensagens para todos os clientes, e estes apenas recebem as mensagens (i.e. não enviam qualquer mensagem);

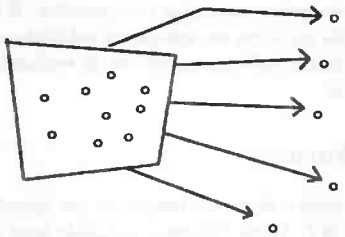

Figura 34: grupo de difusão

. grupos hierárquicos: é uma estrutura constituída por um membro principal (raiz) e subcomponentes de mesmo tipo, acessados a partir da raiz. Sua vantagem é a escalabilidade.

Não se exige que membros de um grupo sejam idênticos, ou mesmo implementados na mesma linguagem de programação, ou que sejam executados na mesma máquina. Além disso, processos podem pertencer a vários grupos.

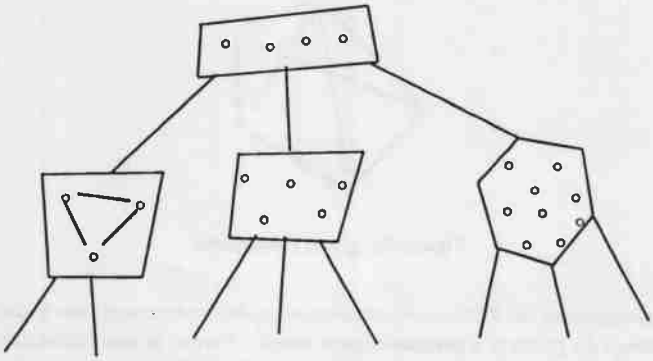

Figura 35: grupo hierárquico

9 Aplicações em ISIS

Um exemplo de aplicação real em ISIS é em sistemas financeiros como o utilizado para comunicação entre acionistas/corretores e bolsas de valores. A arquitetura é do tipo cliente-servidor.

Tais sistemas integram um grande número de aplicações e exigem tempo real na aquisição de informações sobre cotação de preços e negociação de ações em diversos mercados.

O objetivo do sistema é fornecer serviços básicos confiáveis, através de grupos de processos replicando informações de estado do serviço de modo tolerante a falhas, e impedindo que informações inconsistentes sejam fornecidas a clientes distintos, de modo a garantir segurança e rapidez na tomada de decisões.

As informações recebidas neste caso são cotações de compra/venda de ações, e as decisões de corretores baseadas nestas informações envolvem eventualmente grandes somas de dinheiro, daí a importância de obtenção correta, em ordem consistente e rápida destes dados.

Outras aplicações de ISIS são bancos de dados distribuídos. A utilização típica é na replicação de dados para tolerância a falhas e suporte a consultas concorrentes. É um grupo do tipo homogêneo.

Além disso, ISIS também já foi utilizada em aplicações telefônicas, sistemas militares para rastreamento de aeronaves, sistemas médicos, aplicações gráficas de realidade virtual, automação industrial e previsão de tempo em larga escala.

10 Implementações Futuras

Modificações futuras em ISIS incluem o desenvolvimento de um microkernel adequado a sua integração com sistemas operacionais como NT, Mach, Chorus, incluindo uma arquitetura mais segura e comunicação em tempo real.

ISIS não foi o primeiro sistema a utilizar comunicação de grupo, pois, outros sistemas como sistema V, Amoeba, Chorus, Psync, AAS (IBM) e Transis já possuíam estas caracteristicas.

Porém, ISIS foi o primeiro a propor um modelo síncrono virtual com primitivas simples e soluções genéricas e confiáveis.

Referências

[1] Birman, Kenneth P. The Process Group Approach to reliable distributed computing Technical Report 1216, Cornell University. ftp.cs.cornell.edu:/pub/isis

[2] Birman, Kenneth P. and Cooper, Robert The ISIS Project: Real experience with a fault tolerant programming system Operating Systems Review - Abril 1990 - vol. 25 - no. 2, pp. 103-107

[3] Lamport, L. Time, clocks, and the orderings of events in a distributed system CACM vol. 21(7), Julho 1978, p. 558-565

[4] Mullender, Sape. Distributed Systems, 2nd ed., Addison Wesley, 1993

[5] Renesse, Robert van. Causal Controversy at Le mont St.-Michel Operating Systems Review, vol. 27(2), Abril 1993, pp 44-53

[6] Stephenson, Pat and Birman, Kenneth. Fast Causal Multicast - Operating Systems Review, Abril 1990, vol 25(2), pp. 75-79

[7] Tanembaum, Andrew S. Modern Operating Systems, Prentice Hall, 1992

Sistema Horus

Eugênio Akihiro Nassu

1 Introdução

O Horus é um sistema desenvolvido para comunicação de grupo. A comunicação de grupo se faz presente em vários serviços e algoritmos, especialmente os algoritmos distribuídos, que fazem uso freqüente de broadcasts e multicasts. Como exemplos destes serviços, podemos citar os algoritmos de eleição e atualização de servidores replicados em sistemas tolerantes a falha, a distribuição de tarefas e dados em sistemas paralelos, implementação de cache de disco, memória virtual compartilhada, escalonadores distribuídos, exclusão mútua, diversos algoritmos de sincronização, etc.

Alguns sistemas operacionais distribuídos suprem esta demanda incorporando a comunicação de grupo diretamente no kernel, no subsistema de comunicação. Como exemplos deste tipo de sistemas temos o Amoeba, Chorus e V System. Esta forma de implementação traz um desempenho melhor, porém aumenta a complexidade do kernel do sistema operacional o que dificulta o desenvolvimento e a manutenção do sistema.

Para diminuir a complexidade do subsistema de comunicação ou fornecer comunicação de grupo a sistemas que simplesmente não o implementam foi desenvolvido o sistema ISIS[1] na Universidade de Cornell [7], que implementa em espaço usuário a comunicação de grupo. Com o tempo, notou-se que o ISIS um número excessivo de funções visando servir às mais diversas aplicações e que o desenvolvimento de aplicações era bem custoso assim como a análise dos protocolos a fim de melhorar o desempenho. A partir disto, inspirado nas arquiteturas microkernel, surgiu a idéia do Horus: Um sistema de comunicação menor, portável, e altamente configurável. Os testes iniciais em protótipos do Horus indicavam uma performance melhor que o ISIS. As primeiras versões estavam previstas para o início de 1994, sendo portanto, um trabalho relativamente novo.

Neste trabalho vamos descrever a arquitetura do Horus, o modelo de grupo utilizado, seus módulos principais, a descrição de alguns de seus protocolos e algumas conclusões.

2 Arquitetura do sistema

O Horus se utiliza de uma arquitetura baseada na arquitetura microkernel. Ele foi concebido para residir tanto em espaço kernel como em espaço usuário, ou mesmo com uma parte no kernel e outra em espaço usuário. Há um módulo básico, o MUTS, correspondente ao microkernel, que será descrito adiante, que oferece passagem de mensagens assíncrona, multicast, com threads preemptivos, com suporte a timers e múltiplos espaços de endereçamento. O MUTS pode suportar vários protocolos de transporte, confiáveis ou não, multicast ou ponto a ponto, e é facilmente portável para outros ambientes. Todo o resto do

[1] Horus, na mitologia grega, é filho de ISIS

sistema, faz uso dos recursos do MUTS dando a portabilidade ao Horus. Deve-se ressaltar também que todas as máquinas do grupo devem executar o Horus. Não está definido se todas as máquinas do grupo devem estar executando o mesmo sistema operacional, mesmo que neste caso pode-se criar um protocolo que possibilite a comunicação entre sistemas diferentes.

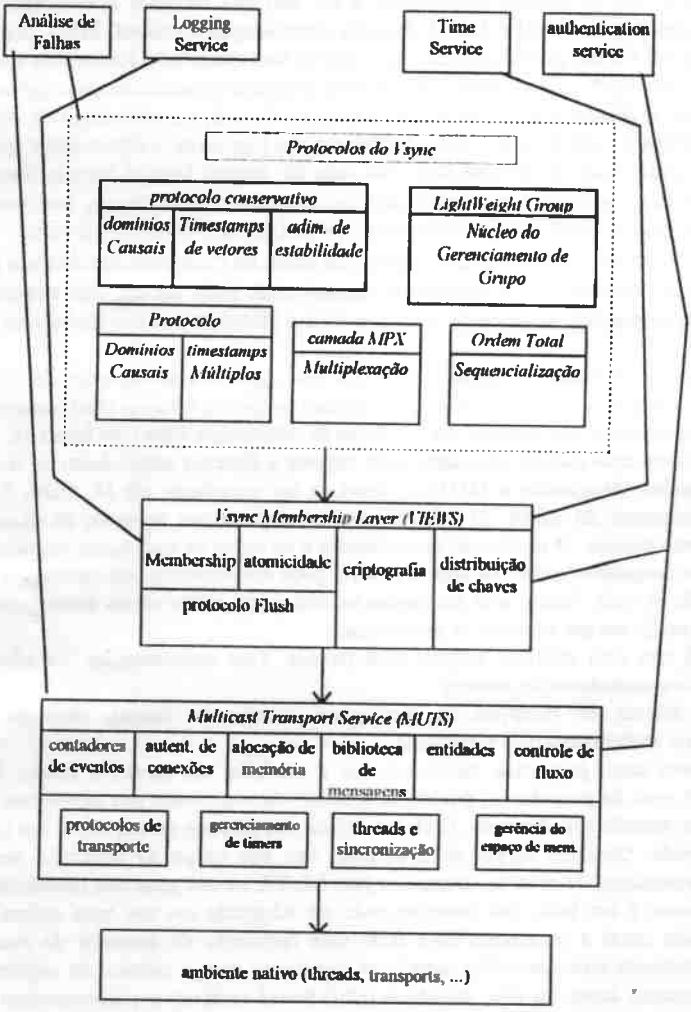

Figura 1: A arquitetura do Horus, as camadas acima do MUTS não precisam estar necessariamente presentes, como será mostrado adiante. Outros módulos podem também ser encaixados.

3 MUTS (Muticast Transport Service)

O MUTS é um dos módulos principais do Horus. Ele implementa todas as funções necessárias para a sua execução, que o sistema não ofereça diretamente a nível de hardware ou a nível do kernel do sistema operacional, e faz uso dos recursos disponíveis (por exemplo, threads). No início [2] o MUTS fornecia comunicação confiável, FIFO, ponto a ponto ou multicast. Porém, posteriormente [1], o MUTS foi reprojetado, fornecendo apenas os mecanismos básicos de comunicação e deixando os protocolos para as camadas que serão descritas adiante. O MUTS oferece seu próprio pacote de threads, gerenciamento de timers, e seu próprio protocolo de multicast. Estas facilidades são primitivas, e foram feitas apenas para suprir as deficiências do sistema host. No caso do sistema host já possuir algumas destas características, o Horus deverá fazer uso dos recursos nativos, porém, isto ainda é uma questão em aberto. O MUTS foi escrito em C, para efeito de máxima portabilidade.

Os usuários podem, em cada grupo, definir que forma de transporte eles desejam (um formato default é oferecido). A integridade da comunicação pode ser mantida através de autenticação de mensagens, se desejado. A criptografia é oferecida por um módulo em um nível mais alto.

Ao contrário de outros sistemas, o MUTS não quebra uma conexão de forma nenhuma, a não ser que seja instruído para tal. A análise de falha é feita em nível usuário ou por uma camada superior, que execute um protocolo de tolerância a falha (ver figura 1).

O MUTS tem uma camada destinada a dar suporte a diversos adaptadores de rede e diversos protocolos. Atualmente o MUTS é capaz de ser executado sob IP, UDP, TCP, Ethernet e mensagens do Mach. O MUTS oferece também um controle de fluxo e congestionamento básicos. O modelo de comunicação é de envio de mensagens assíncrona. No caso de um congestionamento na rede o MUTS pode empacotar várias mensagens em uma transmissão na rede. Assim, uma mensagem na rede pode conter várias mensagens do MUTS, o que resulta em um aumento de performance.

O MUTS tem uma interface simples para threads. Para sincronização, ele oferece locks, semáforos e contadores de eventos.

Há três formas de recepção de mensagens: criação de thread, chamada de procedimento, ou enfileiramento de mensagens.

No primeiro caso, para cada mensagem que é recebida, um thread é criado. Isto permite um alto grau de paralelismo, porém, se a ordem de tratamento das mensagens for importante, este método não funciona. O Horus coloca um número de seqüência em cada mensagem enviada. Qualquer forma de ordenação fica por cargo da aplicação sendo executada. Os contadores de eventos, oferecidos pelo MUTS, servem para esta finalidade. O contador de evento é um lock, que somente pode ser adquirido em um certa ordem. O thread que deseja tratar a mensagem deve fazer uma requisição do contador de evento (acquire), que somente será concedido quando as mensagens com o número de seqüência menor forem tratados antes, ou seja, quando o outro thread terminar o processamento da mensagem (release). Isto força uma sincronização entre os threads, nas quais as mensagens são tratadas somente na ordem correta. A tabela de funções de manipulação dos contadores de eventos e um exemplo de uso são mostrados a seguir:

Função	Argumentos	Resultado
ec_create	-	contador de evento
ec_acquire	contador de evento, núm. de seqüência	-
ec_release	contador de evento	-
ec_destroy	contador de evento	-

tabela 1: a interface dos contadores de eventos.

```
upcall( msg, SeqNum ){
  processamento inicial na msg
  ec.acquire( event counter, SeqNum )
  processamento pricipal de msg
  ec.release( event counter, sequence number )
  continuação do processamento de msg...
}
```

Figura 2: Exemplo de uso dos contadores de eventos. Vários threads iguais a este podem ser executados em paralelo, mas a parte principal deve ocorrer em uma ordem estrita.

4 O modelo de grupo do Horus

Um importante conceito a ser apresentado é o de view. View é o estado instantâneo de um grupo, ou seja, os elementos que estão no grupo num determinado momento. As views são comunicadas ao grupo concorrentemente e assincronamente. Portanto, em um determinado instante os membros de um grupo podem ter diferentes views de um grupo. Os processos mandam suas mensagens através do view correntemente instalado em sua máquina. Os protocolos do Horus procuram informar a mesma seqüência de views para cada membro do grupo, e, se bem sucedido, garante que cada membro recebe o mesmo conjunto de mensagens entre as views. Para mandar uma mensagem para o grupo, o processo precisa entrar no grupo para obter a lista de membros, caracterizando o modelo de grupo fechado.

A comunicação para um grupo é feita por um multicast de grupo. Se não houver falhas, um multicast para um grupo é enviado para todos os membros do view do remetente. Quando há falhas, há uma modificação nessa regra: se uma mensagem for recebida por um membro atingível, ela será recebida por todos os membros atingíveis. Isto é implementado através do protocolo de Flush, que será descrito adiante. O Horus pode executar diversos modelos de comunicação de grupo, de acordo com a necessidade, incluindo o de sincronização virtual do Isis.

Para enviar uma mensagem multicast, a aplicação passa como parâmetro uma lista de entidades. Uma entidade é o representante de um membro de um grupo, que pode ser tanto um processo como uma porta ou uma máquina. A lista de entidades é atualizada periodicamente e corresponde às views.

O Horus pode ser configurado para continuar trabalhando mesmo com falhas e partição da rede, que era uma das deficiências do ISIS, usando adaptações dos protocolos propostos pelo Transis [8]. Quando há uma partição na rede, um grupo pode ser dividido em vários subgrupos: um primário e outros não primários. Os membros de um grupo vão ver seqüências diferentes de views. Quando a partição terminar um subgrupo não primário

pode se juntar à partição primária. No Isis, apenas a partição primária pode continuar a execução. O Horus continuamente tenta encontrar e juntar partições. Estes eventos são informados aos membros do grupo, e os membros rodam um algoritmo para atualizar o novo estado.

5 Design do Horus

O projeto do Horus é altamente modular. O Horus pode ser comparado a uma série de módulos, que podem ser encaixados um sobre o outro, como blocos de LEGO (figura 3). Em [2], algumas destas camadas eram agrupadas como um módulo maior, o Vsync, que implementava alguns dos protocolos descritos adiante. Como os diversos protocolos podiam ser escolhidos, de acordo com a necessidade, o Vsync foi dividido em diversos módulos, chegando ao resultado atual.

A idéia deste design veio do x-kernel, onde há também vários módulos, que desempenham funções diferentes, e os módulos escolhidos são ligados na forma de um grafo.

Módulos podem ser acoplados independentemente, e pode-se até mesmo acoplar um módulo várias vezes. Não há uma ordem global para o acoplamento dos módulos, porém algumas camadas podem depender da semântica de camadas inferiores, o que impõe uma ordenação parcial dos módulos.

A interface entre os módulos é através de uma série de chamadas, para cima ou para baixo (upcalls e downcalls), que são camadas de *Uniform Group Interface* (UGI). A tabela com as camadas é mostrada nas tabelas 2 e 3. A UGI é muito importante, pois com o seu uso se consegue a grande flexibilidade do Horus. Cada módulo, então, apresenta a mesma interface mas implementa um protocolo diferente.

A exceção se faz na camada inferior e superior, que no caso, podem ser camadas que tenha uma interface UNIX BSD. Ou seja, o usuário pode acessar os serviços de comunicação de grupo através de chamadas UNIX *read* e *write*. É possível misturar descritores de arquivo do UNIX, descritores de soquete do TCP, e descritores de soquete do Horus e aplicar uma camada *select*. A chamada *ioctl* fornece controle sobre as propriedades do grupo, como ordenação de mensagens e de como alguns eventos como novos membros no grupo serão informados a aplicação final. O processo, portanto, escolhe para quais membros do grupo quer mandar a mensagem, através de uma chamada *cast*, ou executa um multicast, que manda a mensagem para todos os membros do grupo. Os membros são continuamente informados sobre os membros do grupo, através da chamada *join_request*.

A seguir, vamos descrever algumas camadas e suas funções:

5.1 COM (comunicação básica)

Esta camada não executa nenhum protocolo. A função desta camada é fornecer a interface UGI para outras interfaces de comunicação de baixo nível. Por enquanto, esta camada oferece suporte para IP, UDP, as extensões multicast de Deering de IP e UDP, ATM, mensagens do MACH, as interfaces do x-kernel, e um simulador de rede desenvolvido pelos autores.

downcall	argumentos	descrição
endpoint	protocol stack,lower endpoint	create a comm. endpoint
join	endpoint+group address	join group and return handle
join_denied	join request	deny join request
join_granted	join request	grant join request
view	group handle, list of members	install a group view
merge	view contact	merge with other view
cast	message	multicast a message
send	message + subset of members	send message to a subset
ack	message	acknowledge a message
stable	message	message is stable
leave	group handle	leave group
flush	list of failed members	remove members and flush
flush_ok	group handle	go along with flush
destroy	endpoint	clean up endpoint
focus	identifier	focus on layer, return handle
dump	group handle	dump layer information

tabela 2: Os downcalls da UGI do Horus

5.2 NAK (comunicação FIFO)

Esta camada fornece comunicação ponto a ponto e multicast confiáveis sobre outros mecanismos de comunicação não confiáveis. Este protocolo, não usa o protocolo baseado em janelas. Ou seja, ele não confirma as mensagens recebidas, apenas enviando confirmações negativas, quando são detectados erros (quebra na seqüência de mensagens recebidas).

Se a camada inferior já possuir comunicação confiável, esta camada não precisa ser utilizada.

5.3 MBSHIP (atomicidade e membership)

A parte principal desta camada é o protocolo de flush. O protocolo de flush é executado em caso de falha ou quando diferentes views se juntam. A intenção do protocolo é de finalizar a view, de forma a atingir o sincronismo virtual (ou seja, fazer com que todos os membros do grupo que restam recebam o mesmo conjunto de mensagens). Quando é detectado uma queda em um dos membros de um grupo, o membro mais velho do grupo[2] é eleito o coordenador. O coordenador então executa um broadcast a todos os sobreviventes do grupo, enviando uma mensagem FLUSH. Todos os membros retornam mensagens não estáveis(ver próxima seção) vindas de membros quebrados, seguido de uma mensagem FLUSH_OK. Os membros então ignoram as mensagens vindas de membros supostamente quebrados, esperando por um instalação de uma nova view. Depois de receber todas as mensagens FLUSH_OK, o coordenador envia todas as mensagens vindas de membros quebrados que ainda estão instáveis. Quando todas as mensagens estão estáveis, o flush está

[2] Todos os processos conhecem a mesma seqüência de Views, o que permite que todos concordem quanto ao coordenador sem troca de mensagens.

upcall	argumentos	descrição
JOIN_REQUEST	source	request to join
JOIN_FAILED		request failed
JOIN_DENIED	why	request denied
FLUSH	list of failed members	view flush started
FLUSH_OK		flush completed
VIEW	list of members	view installation
CAST	message + source	received multicast message
SEND	message + source	received subset message
LEAVE	member id	member leaves
DESTROY		endpoint destroyed
LOST_MESSAGE		message was lost
STABLE	stability matrix	stability update
PROBLEM	member id	communication problem
SYSTEM_ERROR	reason	system error report
EXIT		close down event

Tabela 3: Os upcalls da UGI do Horus

completo. Se algum dos processos falhar durante o processo, um novo protocolo flush é iniciado imediatamente. O protocolo de flush está ilustrado na figura 2.

5.4 STABLE (estabilidade)

Uma mensagem é considerada estável quando todos os membros processaram a mensagem.

Para acompanhar a estabilidade das mensagens, o módulo periodicamente faz um broadcast que contém o número das últimas mensagens que foram confirmadas. Para guardar a informação sobre a estabilidade das mensagens é guardada na matriz de estabilidade. Quando multicasts são enviados, alguns dos protocolos do Horus precisam conhecer a k-estabilidade (quando k membros viram a mensagem, onde k pode ser menor que o conjunto de todos os membros do grupo). A matriz de estabilidade é local e para cada posição *ij* guarda o número de mensagens enviada por *i* que foram confirmadas por *j*. A matriz de estabilidade é compartilhada pelos usuários e pela camada, e as atualizações da matriz são executadas por upcalls a camada STABLE.

A matriz não é infinita: A medida que o processamento prossegue, as partes da matriz totalmente confirmadas podem ser descartadas.

5.5 FC (controle de fluxo)

Esta camada é a única que implementa o controle de fluxo no Horus. Ele usa um mecanismo de crédito. O crédito é baseado, na informação fornecida pela camada STABLE. Quando o número de mensagens instáveis ultrapassa um certo patamar, o módulo começa a atrasar as mensagens. Quando o número de mensagens instáveis cai abaixo desse valor, as mensagens guardadas no buffer são colocadas em uma única mensagem. O valores de referência para o tamanho do buffer e do número de mensagens instáveis é ajustável durante a execução. No futuro é planejado que os valores possam ser ajustados automaticamente durante a execução baseado em estatísticas coletadas durante a execução.

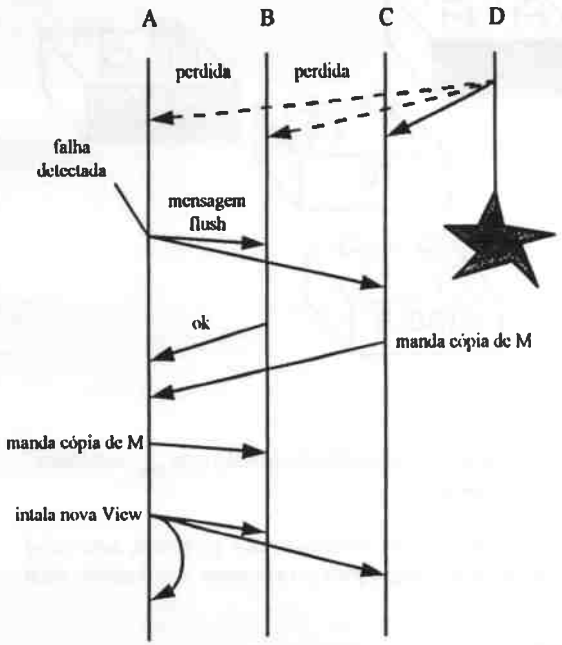

Figura 2: O protocolo de flush. D envia uma mensagem, e quebra antes que a mensagem seja recebida por A e B. A é eleito o coordenador e inicia o processo.

5.6 CAUSAL (envio ordenado por causalidade)

Esta camada acompanha as relações causais entre as mensagens usando relógios. O envio ordenado é tratado pela camada ORDER, descrita a seguir.

5.7 ORDER

Esta camada, além de ordenar o envio de mensagens pela causalidade, pode ser usada para fornecer um envio de mensagens seguro. (Mensagens que estão totalmente estáveis, se executado sobre a camada STABLE).

Em [1], as camadas TOTAL e CASUAL não tinham sido testadas. Nos testes iniciais, foi observado que a camada ORDER raramente executa algum trabalho, a não ser que os processos consumam mensagens em taxas muito diferentes, o que não ocorre comumente na prática, pelo menos nos sistemas onde o Horus estava sendo testado.

5.8 TOTAL (envio totalmente ordenado)

Esta camada implementa o envio totalmente ordenado através de um token. Para enviar uma mensagem, o membro tem que possuir o token. O membro coloca um número de seqüência nas mensagens, e mensagens que chegam fora da ordem são atrasadas até que possam ser enviadas em ordem. O token não é fixo em um servidor, nem é passado por todos os membros. Ao invés disto, o token é passado entre todos os remetentes correntes. Isto funciona bem se os membros tem um fluxo uniforme e alto. Se o tráfego é baixo, o

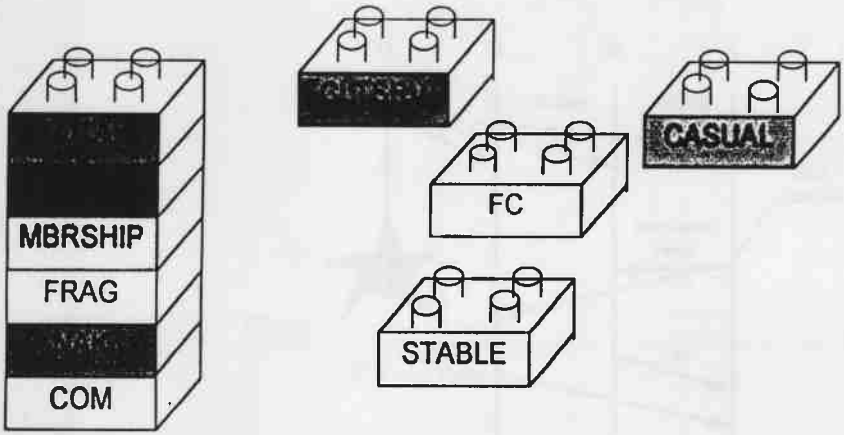

Figura 3: Os módulos do Horus podem ser encaixados de acordo com a necessidade da aplicação

overhead e a latência ficam relativamente altos. Para contornar este problema, uma nova camada está sendo projetada, que usa outra estratégia para providenciar a ordenação total das mensagens.

5.9 XFER (estado de transferência)

A camada XFER providencia informação a um processo quando ele entra em um grupo, isto é, quando ele precisa ser atualizado sobre o estado atual da aplicação. Também, quando duas partições são juntadas, elas precisam concordar sobre um estado comum antes das operações normais possam prosseguir. Esta camada adiciona uma nova etapa na mudança do grupo, em que os processos podem trocar informação sobre o estado.

5.10 CLTSVR (cliente/servidor)

Quando os grupos começam a ter milhares de membros, o protocolo de flush começa a apresentar sinais de degradação de performance muito grandes. Para contornar esta situação, este módulo implementa um protocolo de agrupamento que organiza os elementos de um grupo em uma estrutura hierárquica de clientes e servidores. Os servidores rodam a camada MBRSHIP. Cada servidor é responsável por um conjunto de clientes, e devolve os eventos que recebe. Os clientes, em situação normal, se comunicam com o seu servidor. No caso de quebra de um servidor, outro servidor adota os clientes órfãos. Desta forma, o protocolo simula o modelo da camada MBRSHIP, porém com melhor escalabilidade, ao custo de uma latência dos clientes, devido ao fato dos clientes terem que se comunicar com o servidor, que repassa a transmissão para o(s) destinatário(s). Por exemplo, para se comunicar com um único membro, serão necessárias pelo menos duas mensagens na rede, uma do remetente para o servidor, e uma do servidor para o destinatário. Pode-se criar uma hierarquia maior empilhando-se a camada CLTSVR várias vezes, criando assim uma hierarquia de vários níveis. Um aspecto interessante é o de que as propriedades de ordenação dos servidores são herdados pelos clientes. Por exemplo, se os servidores usarem

a camada TOTAL, as mensagens serão recebidas com ordenação total tanto nos servidores como nos clientes.

5.11 LWG (grupos peso-leve)

Esta camada multiplexa os grupos em um pequeno conjunto de grupos leves. Este método diminui consideravelmente os custos de tratamento de falha, além de outros aspectos de comunicação de grupo. Uma discussão mais detalhada sobre este assunto pode ser encontrada em [4].

5.12 FAST (aceleração de mensagens)

Este módulo usa o mesmo protocolo do módulo NAK, porém as mensagens deixam de passar pela maioria dos módulos. Quando ocorrem erros, o módulo volta ao modo de envio confiável e retransmite mensagens instáveis. Quando as condições anormais acabam, o módulo volta ao modo acelerado.

5.13 Outras camadas

Além destas camadas disponíveis outras camadas podem ser implementadas. Além disto, outras estão sendo planejadas, como uma unidade de controle de fluxo baseado em médias, um módulo para sincronização de relógios e timestamping, um com suporte para RPC, um módulo de logging de mensagens, um módulo de criptografia, e um módulo que simplifica o tratamento de partições na rede.

6 Performance e conclusão

Os testes inicias realizados com versões preliminares do Horus, com o MUTS e Vsync sendo executados em espaço usuário mostraram desempenho ligeiramente superiores ao Isis. O próximo passo será o de colocar o MUTS e Vsync em espaço kernel, o que deve gerar melhor performance.

O Horus é um sistema projetado a partir dos pontos fracos de seu antecessor, o Isis. A portabilidade e escalabilidade são seus pontos fortes, mas ainda é um projeto em andamento. As possíveis críticas ao sistema são contornadas com a desculpa de que o sistema é provisório, e assim devemos esperar o futuro para um conclusão definitiva.

Notamos também que como em outros trabalhos, a um projeto de um sistema muito complexo (ISIS), seguiu-se um projeto mais simples e flexível. Isto reafirma a tendência atual das arquiteturas microkernel, de onde foram extraídas algumas das idéias no projeto do sistema Horus.

7 Bibliografia

[1] Robbert van Renesse, Takako M. Hickey, e Kenneth P. Birman. Design and Performance of Horus: A Lightweight Group Communications System.

[2] Robert van Renesse, Ken Birman, Robert Cooper, Brad Glade, Patrik Stephenson. The Horus System. Julho de 1993.
[3] Robert van Renesse. A MUTS Tutorial.
[4] Bradford B. Glade, Kenneth P. Birman, Robert C. B. Cooper, Robert van Renesse. Light-Weight Process Groups in the ISIS System.
[5] The Isis Group. The Restructuring of Isis for Modern Distributed Operating Systems (Draft Copy). Setembro de 1991.
[6] Bradford B. Glade. Fault-Tolerant Programming with Horus: Overview. Março de 1991.
[7] Kenneth P. Birman. The Process Group Approach to reliable distributed computing. *Communications of the ACM*, 36(12):37-53, dezembro de 1993
[8] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication subsystem for high availability. Em *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pp. 76-84, Boston, MA, Julho 1992. IEEE.

O kernel Synthesis

Rodrigo de Salvo Braz

1 Introdução

Os engenheiros de software têm sempre se defrontado com um conflito aparentemente inevitável entre duas necessidades básicas: a de garantir um bom desempenho e a de projetar sistemas modulares bem organizados, flexíveis e portáteis. A prioridade da primeira necessidade é encontrada geralmente em sistemas comerciais, enquanto a segunda parece mais valorizada em projetos científicos e acadêmicos. O equilíbrio ideal entre as duas é difícil de se determinar; simplesmente evitar o conflito parece impossível. Um sistema operacional não foge a esta regra mas, pelo contrário, apresenta-a de forma ainda mais acentuada. Afinal, de seu desempenho depende o desempenho de todas as outras aplicações. Por outro lado, a complexidade, a necessidade de flexibilidade e de portabilidade a diversos tipos de ambientes demandam uma modularidade que dificulta otimizações e aumenta o custo devido a sistemas de comunicação mais complexos. O kernel Synthesis foi pesquisado por Calton Pu e Henri Massalin [1] na Universidade de Columbia de 1987 a 1992, quando Pu deixou a Universidade e prosseguiu com suas pesquisas através do projeto Synthetix, no Oregon Graduate Institute. O kernel Synthesis trazia a proposta inovadora de solucionar este conflito através de determinadas técnicas, que apresentam um potencial tão grande quanto a complexidade de implementá-las e explorá-las ao máximo. O hardware utilizado foi uma máquina experimental baseada no processador 68020 a 20 MHz com um barramento de 16 bits. Duas características se destacam neste sistema: a *sintetização de código* e o conceito de *máquina sintética*. A primeira trata de incorporar ao sistema a capacidade de gerar código especializado em determinadas situações, enquanto a segunda busca oferecer uma interface de alto nível baseada em quatro tipos de objetos que podem ser manipulados por apenas seis operações ortogonais. Segundo os autores, existe um fator sinergético entre os dois conceitos, pois sem o desempenho incrementado pela sintetização de código, uma interface de nível tão alto seria custosa demais. Não contar com uma interface de alto nível, entretanto, significaria ter de adicionar mais camadas de software para utilizar o sistemas, criando um overhead de comunicação entre elas e prejudicando o desempenho. Examinaremos aqui estes conceitos em maiores detalhes e veremos como eles vêm oferecer alternativas ao conflito mencionado acima.

2 Sintetização de código

A *sintetização de código* no kernel Synthesis é a produção de código especializado para tratamento de estados freqüentes do sistema. Os autores propõem três tipos de sintetização de código: a *Fatoração de Invariantes*, a *Fusão de Camadas* e as *Estruturas de Dados Executáveis*. Entretanto, estes são métodos determinados heuristicamente, baseados em observações sobre a forma de trabalho dos sistemas, e por isso acreditamos que possa haver ainda outros métodos distintos de sintetização de código.

2.1 *Fatoração de Invariantes*

Quando um determinado trecho de código é escrito, deve estar apto a tratar corretamente todas as situações possíveis em sua execução. Isto significa validar diversos valores de entrada e determinar qual de muitos casos teóricos está ocorrendo. Por exemplo, no caso de um pedido de leitura de arquivo em um servidor de arquivos, deve haver uma validação, determinando se o pedido por parte do cliente é válido, se o arquivo existe e qual a sua posição no disco etc. Em sistemas tradicionais, o mesmo trecho de código trata os pedidos de leitura de todos os processos que estejam utilizando o sistema de arquivos, realizando sempre as mesmas validações. Entretanto, do ponto de vista de cada processo, há algumas invariantes envolvidas, já que depois da abertura do arquivo já se sabe que ele existe, onde está armazenado etc. A *Fatoração de Invariantes* procura explorar este aspecto, gerando código em tempo de execução que leve em consideração as invariantes existentes. De modo mais formal, temos que

$$[F_{fat}(p_1)](p_2, p_3, p_4, ..., p_n) = F(p_1, p_2, p_3, ..., p_n),$$

onde F é a função calculada pelo sistema e F_{fat} é uma função que, baseada no valor de p_1, retorna outra função. Esta outra função, $F_{fat}(p_1)$, é equivalente a $F(p_1, p_2, p_3, ..., p_n)$ para quaisquer $p_2, p_3, ..., p_n$. Assim, dado um parâmetro invariante p_1, basta calcular $F_{fat}(p_1)$ uma única vez e utilizá-la para o cálculo mais simples de $[F_{fat}(p_1)](p_2, p_3, p_4, ..., p_n)$. Evidentemente, tais resultados são válidos somente se F não depende de dados globais e só compensam se o custo de cálculo de $F_{fat}(p_1)$ é menor do que o que se ganha com o cálculo mais simples de $[F_{fat}(p_1)](p_2, p_3, p_4, ..., p_n)$, ou seja, se

$$Custo(F_{fat}(p_1)) < m((Custo(F(p_1, p_2, ..., p_n)) - Custo([F_{fat}(p_1)](p_2, ..., p_n))),$$

onde m é o número esperado de chamadas à função F com parâmetro p_1. Note ainda que este método pode ser aplicado a mais de um dos parâmetros de F, sucessivamente ou não (ou seja, calculando F_{fat} em relação a um conjunto de parâmetros ou a cada um deles sucessivamente).

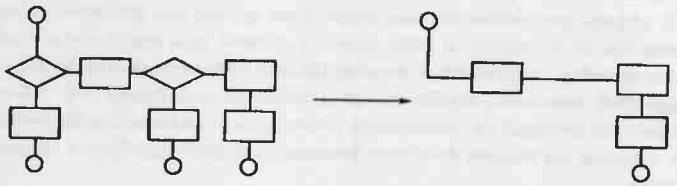

Figura 36: O prévio conhecimento de algumas condições permite a *Fatoração de Invariantes*.

Assim, ainda no exemplo do servidor de arquivos, teríamos na abertura de um arquivo a sintetização de código especializado para este arquivo, levando em conta todas as invariantes existentes durante sua utilização (como sua existência, localização e direitos de acesso), e que é devolvido ao cliente. Este código, do qual foram excluídas várias checagens e cálculos, será utilizado pelo cliente em cada novo acesso ao servidor, otimizando o processo.

2.2 *Fusão de Camadas*

Tempo de execução também é desperdiçado durante a execução de sistemas estruturados em várias camadas. Trata-se de uma divisão conceitual fundamental para a análise e desenvolvimento de programas, mas que implica num considerável custo envolvido em chamadas de procedimentos e eventuais

trocas de contexto. A *Fusão de Camadas* é um método de sintetização de código que elimina as chamadas de procedimentos através da expansão in-line do código das funções dentro do código daquelas que as chamam. Isto poupa chamadas de procedimento (e mesmo trocas de contexto) mas pode aumentar em muito o tamanho dos programas, principalmente se a função chamada for grande ou invocada a partir de vários pontos do código, visto que isso causa a repetição de código em diversas partes do programa. É uma técnica prevista na linguagem C++, na qual se pode declarar funções in-line. Os problemas, vantagens e desvantagens encontrados em ambos os campos são similares.

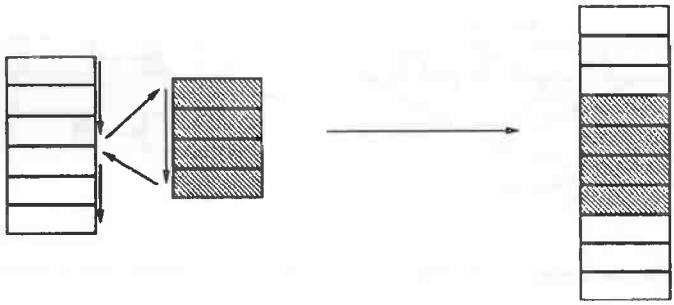

Figura 37: A *Fusão de Camadas* insere o código de uma função no corpo de um programa.

Um exemplo de software dividido em várias camadas é o modelo de protocolos OSI, da ISO. Ele é composto por sete camadas de software pelas quais passam todos os pacotes processados. Embora conceitualmente completo e flexível, este modelo oferece um overhead que muitas vezes impede sua utilização em redes locais das quais se espera um melhor desempenho. Se F é uma função de alto nível no modelo OSI, então vale:

$F(p_1, p_2, ..., p_n) = F_{\text{aplicação}}(p_1, F_{\text{apresentação}}(p_2, F_{\text{sessão}}(p_3, ... F_{\text{físico}}(p_n)...)))$,
onde F_{xxx} é uma função da camada xxx.

A expansão in-line do código referente à camada de apresentação dentro da camada de aplicação gera uma função que poderíamos chamar de F_{plana} e que seria utilizada da seguinte forma:

$$F_{\text{aplicação}}(p_1, F_{\text{apresentação}}(p_2, F_{\text{sessão}}(p_3, ... F_{\text{físico}}(p_n)...))) = F_{\text{plana}}(p_1, p_2, F_{\text{sessão}}(p_3, ... F_{\text{físico}}(p_n)...))$$

Assim como na *Fatoração de Invariantes*, o método pode ser aplicado sucessivamente, neste caso através de todas as camadas. Uma espécie de vantagem colateral da *Fusão de Camadas* é a possibilidade de otimizações adicionais quando se coloca todo o código no mesmo nível. Por exemplo, se uma função tem como parâmetro uma variável passada por referência, no código expandido a variável passada à função não se distinguirá do parâmetro, e portanto informações sobre esta variável podem ser usadas para evitar operações inúteis sobre ela. Isto não poderia ser feito sobre a função original, que é chamada a partir de diversos pontos do programa sob diferentes condições e não pode contar com informações deste tipo.

2.3 *Estruturas de Dados Executáveis*

Um elemento bastante presente em sistemas de software são estruturas de dados percorridas de maneira uniforme por uma rotina que centraliza esta tarefa. Um exemplo clássico, dado pelos autores do artigo, é o de uma fila circular de processos executados um a um durante um determinado quantum de tempo (round robin), onde uma rotina pode ser responsável pela troca de um processo pelo outro. Entretanto, por vezes é possível embutir na própria estrutura a forma de percorrê-la, dispensando a rotina centralizadora e conseqüentes trocas de contexto.

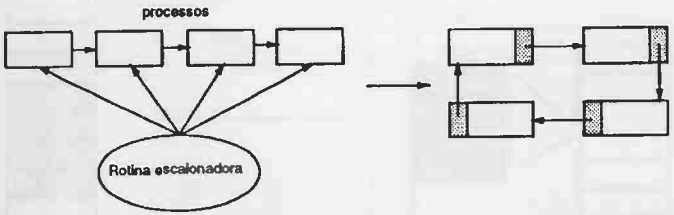

Figura 38: As *Estruturas de Dados Executáveis* contêm o código que as manipula.

No caso da fila de processos, por exemplo, pode-se dotar suas entradas com o próprio código para iniciar e finalizar sua fatia de tempo (rotinas chamadas pelos autores de startjob e stopjob). Quando o quantum do processo corrente se esgota, ocorre uma interrupção e o controle passa à rotina stopjob do processo corrente. Esta salva os registradores e conseqüentemente o estado do processo e passa o controle à rotina startjob do processo seguinte (que ela sabe onde está devido à estrutura da própria fila). A rotina startjob do processo seguinte restaura seu estado e prossegue em sua execução. Como se pode ver, a rotina escalonadora não está mais envolvida, restando-lhe apenas o trabalho de inserir e remover processos da fila. Outro exemplo que podemos imaginar de *Estruturas de Dados Executáveis* é o de uma estrutura em rede cujo processo de localização de um determinado elemento depende de um cálculo ou busca não trivial. Para cada navegação, entretanto, o cliente poderia receber, como brinde, um pedaço de código utilizando os dados do nó corrente que facilitaria a próxima navegação.

2.4 Algumas observações

Alguns problemas e considerações são apontados pelos autores a respeito da sintetização de código. Um dos problemas mais importantes é relativo à redundância de código e o conseqüente tamanho excessivo do kernel, já que as funções geradas tanto pelo método de *Fatoração de Invariantes* quanto pelo de *Fusão de Camadas* podem ser bastante parecidas e estão escritas separadamente. Por isso, certos critérios devem ser adotados na decisão de quando usar estas técnicas. Basicamente, as funções mais freqüentemente utilizadas (e não muito grandes) são expandidas em um espaço próprio do usuário, enquanto as mais raramente usadas ficam disponíveis em páginas de memória compartilhadas por todos os usuários do sistema. Outra importante preocupação a respeito da implementação do Synthesis é referente à segurança do kernel quanto à utilização do código sintetizado. Como evitar que um usuário mal intencionado substitua o código sintetizado por outro de sua autoria, ou que simplesmente o altere? Para prevenir isto, o código sintetizado é armazenado em páginas de memória protegida, inacessíveis aos programas do usuário. Para evitar substituições, tais páginas são acessadas através de um índice de uma tabela interna do kernel que indica sua localização. O programa de usuário possui apenas este índice e não tem acesso à tabela.

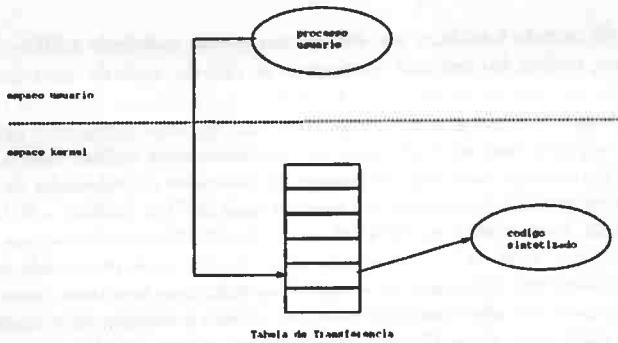

Figura 39: O acesso ao código sintetizado é feito de maneira indireta.

3 Máquinas sintéticas

Ao lado das técnicas de sintetização de código está o conceito de máquina sintética como principal característica do kernel Synthesis. Trata-se de uma unidade fechada dentro do sistema. Seus componentes internos são livremente acessíveis dentro da máquina, mas aos elementos externos só é permitido o acesso através de uma interface bem definida. A máquina sintética é um dos tipos de objetos presentes no Synthesis. Os outros tipos são a CPU, memória e I/O sintéticos (denominados respectivamente SCPU, SMEM e SIO, ao lado de SMACH para máquina sintética). Sobre estes objetos existem seis tipos de operações: create, terminate, reconfigure, query, open e read, embora somente os objetos de I/O processem os dois últimos. O encapsulamento, combinado com as interfaces uniformes aplicadas aos diferentes tipos de objetos, caracteriza o Synthesis como um sistema orientado a objetos (embora o importante conceito de herança não tenha sido citado, bem como se é possível definir novos tipos de objetos e mensagens). A orientação a objetos é desejável em um sistema operacional porque normalmente implica em interfaces de alto nível e, portanto, em grande flexibilidade e facilidade de programação e gerenciamento. Entretanto, uma implementação pouco cuidadosa pode implicar num desempenho bastante fraco. Os autores observam que os sistemas orientados a objeto pioneiros apresentaram desempenho aceitável, e esperam que o seu próprio inove quanto a esta característica graças à sintetização de código.

3.1 Máquinas, componentes e interface

Uma máquina sintética é um objeto encapsulado, acessível apenas através de interface restrita e que pode ser composta por vários tipos de objetos:

SCPUs, que representam fluxos de controle na execução de código; SMEMs, espaços de endereçamento distintos para armazenar dados e código; SIOs, canais de entrada e saída para trocas de dados.

Sobre estes objetos estão definidas as operações create, terminate, query, reconfigure, read e write. A operação create cria e inicializa um objeto, retornando o código para a realização das demais (chamadas operações executivas). Terminate, por sua vez, finaliza e destrói o objeto. A operação query foi idealizada com o objetivo de informar sobre o estado do objeto, ou seja, é uma leitura de suas propriedades. Estas dependerão intrinsecamente do tipo de objeto (prioridade de uma SMACH, tipo de um SIO, tamanho de uma SMEM são alguns exemplos). A operação reconfigure pode alterar tais propriedades, ou seja, sua configuração. Note que a sintetização de código para estas operações pode ser utilizada para definir uma reconfigure de acordo com os direitos de quem a requisitou. Read e write

são as operações de entrada e saída, e por isso mesmo apenas aplicáveis a SIOs. Elas funcionam de acordo com seu tipo, recebendo e enviando mensagens, no caso de canais de comunicação, ou escrevendo e lendo de arquivos, no caso de dispositivos. A interface das máquinas sintéticas e seus componentes foi idealizada de forma a ser ortogonal, ou seja, cada uma de suas operações é independente das outras, ainda que o conjunto como um todo possa ser combinado para realizar tudo o que for necessário. Tal minimalidade permite que estes objetos possam ser utilizados e combinados de diversas maneiras. Assim, uma máquina sintética é composta por uma ou mais SCPUs, SMEMs e SIOs, combinados com razoável flexibilidade. Pode-se ter uma SMACH com várias SCPUs trabalhando em pararelo enquanto outra trabalha com uma única SCPU, simulando uma máquina monoprocessada simples. Também é possível o compartilhamento de componentes entre duas máquinas sintéticas, como por exemplo uma página de memória para trabalho cooperativo em um mesmo problema, ou o estabelecimento de canais de entrada e saída entre duas SMACHs. Pode-se até mesmo aninhar estas máquinas, ou seja, ter máquinas sintéticas rodando e escalonando sub-máquinas com o algoritmo de escalonamento que for mais conveniente. Neste exemplo em particular, o overhead no Synthesis seria mínimo graças à técnica de *Fusão de Camadas*. Quanto ao compartilhamento de componentes, tais como uma SMEM para trabalho conjunto ou uma SIO para troca de mensagens (inclusive em rede), surge o problema da segurança e proteção, pois trata-se de elementos que estão além dos limites das máquinas sintéticas. O compartilhamento é realizado da seguinte forma: uma das máquinas cria o componente a ser compartilhado, recebendo o código sintetizado para acessá-lo. Este código (na verdade os índices que permitem acessá-lo) são enviados à máquina parceira através de um canal SIO protegido. Note que os direitos de acesso obtidos pela segunda máquina dependem do código para ela enviado. Este esquema de proteção baseado no código recebido é mais poderoso do que o usual sistema de capabilities, pois pode implementar qualquer critério. Além disso, permite que o criador do recurso compartilhado revogue os direitos concedidos através da reconfiguração do objeto. A chave do mecanismo de proteção no compartilhamento é, na verdade, o canal de SIO protegido. Os autores não comentam como a máquina sintética pode obter este componente, sendo provavelmente um elemento oferecido pelo kernel para tarefas especiais como esta.

4 Enquanto isso, no mundo real

A implementação do sistema Synthesis ofereceu muitas observações de nível prático que confirmaram algumas suposições mas apresentaram novos problemas. A primeira versão completa deste kernel foi desenvolvida em assembly devido à falta de linguagens de alto nível com suporte a sintetização de código (o assembler usado pela equipe suporta chamadas a ele mesmo para a montagem de templates de código). Além disso, foi considerada fundamental a questão de desempenho e, portanto, a portabilidade foi sacrificada em favor do código específico para cada hardware. Foi mantida apenas, em relação à portabilidade, a estrita compatibilidade com a interface das máquinas sintéticas. Na primeira versão do Synthesis as técnicas apresentadas são utilizadas da forma mais típica: a *Fatoração de Invariantes* nas rotinas de entrada e saída e as *Estruturas de Dados Executáveis* no algoritmo de escalonamento. O sistema de mensagens foi desenvolvido para o suporte ao processamento distribuído, contando com o método de *Fusão de Camadas*. Apenas como exemplo do desempenho do Synthesis podemos apresentar uma tabela com o tempo de execução de dois programas neste e em outros sistemas. O programa 1 lê 100.000 bytes da memória, enquanto o programa 2 lê 10.000 bytes de um arquivo:

Programas	Synthesis	HP	SUN-3	Masscomp
progr. 1	1,2 s	77 s	48 s	29 s
progr. 2	2,0 s	15 s	4,9 s	4,5 s

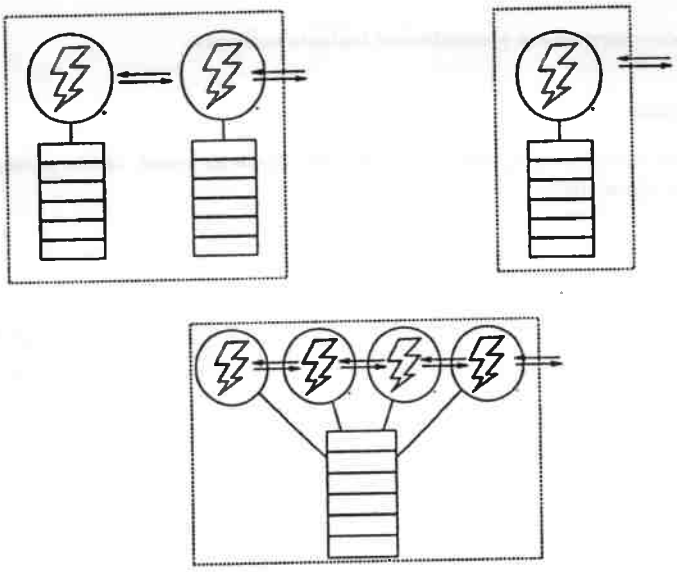

Figura 40: Diversas configurações possíveis de máquinas sintéticas

Aparentemente, a leitura da memória beneficiou-se com as técnicas do Synthesis em um grau maior do que a de disco, pois é mais rápida do que a segunda no Synthesis enquanto é mais lenta nos demais sistemas. Uma possível explicação para o fato é a de que a leitura de disco encontra seu maior limitante na velocidade deste componente do hardware, enquanto a leitura de memória (implementada no Unix como um dispositivo, e portanto não acessada diretamente) encontra sua limitação no próprio esquema de implementação, que pode ser otimizado pelas técnicas de sintetização de código. Na experiência prática, foi verificado que o SIO foi o que mais se beneficiou com a sintetização de código. O sistema conta com um sistema de arquivos similar ao do Unix, mas vários deles coexistirão no Synthesis, como por exemplo os baseados em servidores e um incorporado ao próprio kernel. A equipe trabalhou ainda em uma variante de C chamada Lambda-C, com o objetivo de ter suporte à sintetização de código em alto nível. Este trabalho foi motivado pela idéia de que a sintetização de código é útil não apenas a sistemas operacionais, mas a aplicações em geral. Além disso, a linguagem permitiria que se portasse aplicações que utilizassem sintetização de código para diferentes plataformas com maior facilidade

5 Conclusão

A sintetização de código representa uma forma de lidar com a contradição entre flexibilidade/modularidade e desempenho apontada na introdução. Trata-se de uma solução aplicável a todo tipo de programação, mas tratada aqui especificamente em relação aos sistemas operacionais. O sistema Synthesis aplica bem a idéia da sintetização de código conjugando-a com a interface de alto nível da máquina sintética, pois é no ganho de uma interface de alto nível a baixo custo que a sintetização de código mais se destaca. O paper, entretanto, deixa alguns pontos obscuros. O principal deles, na minha opinião, é o silêncio a respeito da orientação a objetos, que poderia ser estendida de forma a maximizar os ganhos com a interface de alto nível e apresentar oportunidades de criação de novas técnicas de sintetização

de código. Isto, entretanto, é provavelmente bastante complexo.

Referências

[1] Calton Pu, Henri Massalin, and J. Ioannidis. The Synthesis kernel. In *Computing Systems*, 1(1):11-32. IEE, October 1993.

O Distributed Computing Environment da OSF

Alfredo Goldman

1 Introdução

Abordaremos a seguir as características do Distributed Computing Environment da OSF (Open Software Foundation). Este não é um sistema operacional distribuído, mas uma outra alternativa para a implementação de aplicações distribuídas em redes heterogêneas.

O DCE (Distributed Computing Environment) é um ambiente para o desenvolvimento de aplicações distribuídas baseadas no modelo cliente-servidor. Os seus principais objetivos são:

- Utilização em uma rede heterogênea.

  O DCE deve ser instalado, em cada máquina, sobre o sistema operacional já existente (figura 41). Uma exigência é que as máquinas da rede devem possuir pelo menos um protocolo de comunicação em comum.

- Suporte a RPC.

  Com o DCE a programação de um RPC é similar a de um procedimento comum. Todos os detalhes que não estão ligados diretamente a elaboração da chamada remota ficam a cargo do DCE. Existe um compilador específico para gerar os códigos objetos dos *stubs* do cliente e do servidor. Além disto, existem vários serviços para monitorar e resolver os possíveis problemas em tempo de execução.

- Oferecer a possibilidade de programação utilizando *Threads*.

  O DCE implementa threads, até em sistemas operacionais que originariamente não possuem este recurso.

- Aumentar a disponibilidade.

  O DCE dá o suporte necessário para a implementação de serviços distribuídos e replicados, desta forma é possível ter vários servidores para um mesmo serviço.

- Dar uma visão global dos arquivos em uma rede.

  O DCE possui um gerenciador próprio para arquivos distribuídos, que opera de maneira transparente ao usuário.

- Oferecer mecanismos de sincronização.

  O DCE cuida da sincronização do tempo da rede, garantindo assim uma ordenação global de eventos.

- Oferecer mecanismos de proteção.

  O serviço de segurança trata da autenticação das máquinas, processos e usuários do sistema distribuído.

O DCE é um conjunto de serviços e ferramentas que permitem a criação, o uso e a manutenção de aplicações distribuídas em um ambiente de computação heterogêneo.

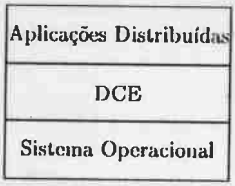

Figura 41: Diagrama da posição do DCE.

2 A arquitetura do DCE

O DCE é formado pelos seguintes componentes (figura 42):

DCE Threads - Mecanismos para a criação, gerenciamento e sincronização de múltiplas threads. Caso o sistema operacional implemente threads, estas podem ser utilizadas.

DCE RPC - Ferramentas para o desenvolvimento dos stubs cliente e servidor e serviços gerais para o tempo de execução. As chamadas de procedimento remotas e threads são serviços básicos do DCE que estão disponíveis em toda máquina na qual o DCE é instalado.

DCE Directory Server - Armazenamento central das informações sobre os recursos espalhados no sistema distribuído.

DCE Distributed Time Server - Sincronização do conjunto de computadores que fazem parte do DCE.

DCE Security Service - Ferramentas e primitivas para autenticação, comunicação segura e autorização de utilização de recursos.

DCE Distributed File Server - Distribuição e acesso aos arquivos espalhados pela rede. Este serviço é similar a uma aplicação que utiliza os serviços básicos (localizados abaixo e dos lados, na figura 42) para a distribuição de arquivos pelo sistema.

DCE Diskless Support Service - Permite que máquinas sem disco possam: carregar o sistema operacional, obter informações de configuração, conectar com o DFS e efetuar o swap remoto.

DCE Management - Intersecção de partes dos vários componentes anteriores. Cada um dos componentes acima possui procedimentos dedicados a configuração e manutenção, não existe um componente dedicado ao gerenciamento. Está em desenvolvimento um componente dedicado a administração de todos os outros componentes, através de uma única interface.

Observe que as aplicações escritas para o DCE não precisam necessariamente utilizar todos os componentes localizados entre elas e o sistema operacional original.

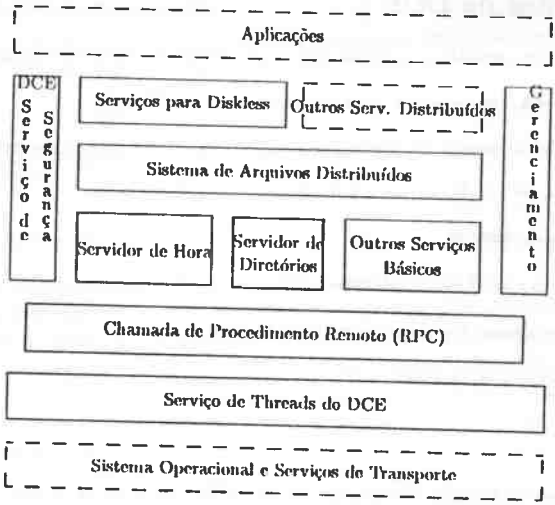

Figura 42: Arquitetura do DCE.

3 Organização do DCE

Um grupo de máquinas DCE que funcionam em cooperação e são administradas como uma só unidade é denominado *célula*. Um ambiente DCE é um grupo de uma ou mais células que podem se comunicar. Uma célula é composta de:

- Máquinas de usuários (máquinas clientes de uso geral)
- Máquinas de administração (para o gerenciamento remoto dos serviços DCE).
- Máquinas servidoras (máquinas que oferecem serviços essenciais). Os seguintes servidores DCE devem obrigatoriamente estar presentes em cada célula:
  - Servidor de diretório;
  - Servidor de segurança;
  - Servidor de hora distribuída.

  Os servidores padrão que podem estar opcionalmente em uma célula são:
  - Agente de diretório global;
  - Servidor de arquivos distribuídos;
  - Servidor para máquinas sem disco.

4 Requisitos do DCE

Temos os seguintes requisitos mínimos para que o DCE possa ser instalado em uma rede:

- Uma camada com serviços de transporte (UDP,TCP, etc);
- Um mesmo protocolo de rede (X.25, IP, ...) em todos os nós da rede;
- Existência, ou condições para a implementação de threads;
- Existência de timers;
- Comunicação local entre processos (IPC);
- Operações básicas com o sistema de arquivos;
- Gerenciamento de memória.

5 RPC

A implementação do modelo cliente-servidor do DCE é através da chamada de procedimentos (RPC). Para o programador uma chamada remota de procedimento é quase igual a uma chamada local. Isto é devido a existência de vários componentes da RPC do DCE que funcionam conjuntamente. São eles:

- A linguagem de definição de interface (IDL) e o seu compilador.

  A IDL é muito similar, semântica e sintaxe, com a linguagem C com a adição de atributos que ajudam a execução de aplicações distribuídas na rede. O código objeto de um programa nesta linguagem ficam em duas partes, uma no lado do cliente e outra no lado do servidor.

  O compilador gera automaticamente a maior parte do código que gerencia a comunicação entre cliente e servidor. Entre outras funções temos seleção e uso dos transportes da rede, *marshaling* e a preparação para a transferência.

- Biblioteca Runtime RPC.

  Consiste de um conjunto de rotinas que são ligadas com os lados servidor e cliente de uma aplicação, e que implementa a comunicação entre eles. As funções da biblioteca incluem a busca do servidor, o envio de mensagens, o gerenciamento dos estados entre as chamadas e o processamento de eventuais erros.

- Autenticação de RPC.

  Este é um serviço integrado com o serviço de segurança e possibilita uma comunicação segura entre clientes e servidores, permitindo a autenticação de clientes e o controle de acesso a serviços.

- Servidor de nomes.

  Serviço integrado com o servidor de diretórios para permitir a localização dos servidores de RPC por seus clientes.

- RPC Daemon.

  É um programa que é executado em todas as máquinas DCE. É um servidor de nomes específico para RPC, gerência o banco de dados que relaciona os servidores de RPC com os seus endereços de recebimento de mensagens (portas).

- Programas de controle de RPC e administração.

  Gerência do RPC daemon e recursos para o administrador do sistema. Exceto em casos especiais, RPC necessita de pouca, ou nenhuma administração.

- Recursos de identificação única de RPC's.

  Existe um gerador (função `uuidgen()`) de identificadores universais únicos (UUID), que são adicionados à IDL de um RPC.

5.1 Programação utilizando o RPC

Apresentaremos nesta seção os principais passos para escrever uma RPC no DCE.

Inicialmente o programador cria um arquivo com a definição da interface e os tipos associados com o RPC utilizando a linguagem de definição de interface (IDL). Esta interface consiste do conjunto de operações que o servidor poderá efetuar e contém os tipos de argumentos e de resultados de cada operação. Esta interface é similar às definições de protótipo em linguagem C. Após a compilação deste arquivo teremos os códigos stubs do cliente, do servidor e um *header file*. Estes stubs são responsáveis pela invocação remota da operação, são eles que tratam de problemas como o empacotamento e o marshaling.

O próximo passo é elaborar o programa do lado do cliente, o qual será ligado com o código do stub do cliente. Em seguida elaboram-se as rotinas que implementam as operações definidas na IDL para o servidor, as quais serão ligadas com o código do stub do servidor (figura 43).

5.2 Funcionamento

Para efetuar um RPC, o cliente, tem que ser capaz de localizar o servidor deste RPC (figura 43). Uma forma é armazenar o endereço do servidor no próprio cliente, mas esta forma é muito inflexível. Uma maneira melhor é que o cliente consulte o servidor de diretórios para descobrir a localização do(s) servidor(es) que pode(m) executar esta função remota. Para isto, o servidor precisa se incluir no espaço de nomes do servidor de diretórios, com o tipo de interfaces que implementa, o protocolo que usa para a comunicação e a sua localização. Assim, é possível que existam vários servidores para o mesmo serviço. O programador de RPC's não precisa se preocupar com qual servidor utilizar, pois há uma camada (layer) específica do stub cliente denominada Serviço Independente de Nome (NSI).

Além de encontrar a localização do servidor existe o problema de encontrar o endereço do processo (a porta de acesso no Unix). Como este endereço pode mudar a cada vez que o processo servidor é inicializado, uma outra solução é adotada. Quando um servidor começa a funcionar, um processo denominado RPC daemon (rpcd), é executado (sempre na mesma porta), ele registra o endereço de seus processos no rpcd. O lado do servidor é responsável pela colocação dos seus dados no espaço de nomes do servidor de diretórios.

6 Threads

As threads no DCE são implementadas em espaço usuário e são baseadas na interface *pthreads* especificada por POSIX no padrão 1003.4. As threads do DCE podem ser utilizadas diretamente ou, caso o sistema já forneça esta opção, podem ser mapeadas para a implementação existente.

O DCE oferece mecanismos para a criação, destruição e gerenciamento de threads. O gerenciamento das threads é construído sobre dois mecanismos interativos: prioridade e política de gerenciamento.

Cada thread possui a sua própria prioridade. Threads com prioridade maior tem preferência sobre os com prioridade menor. Existem três políticas de gerenciamento

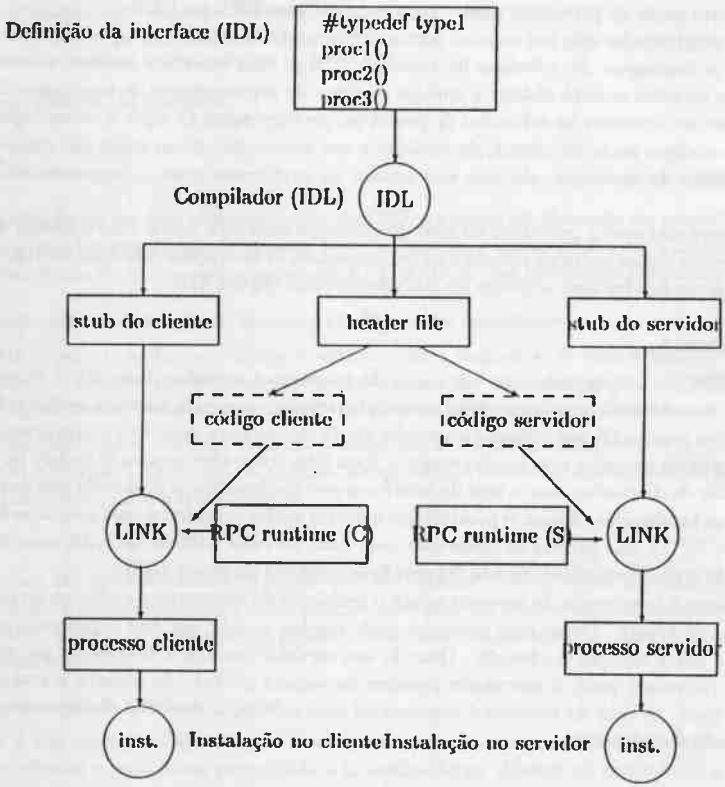

Figura 43: Desenvolvimento de um RPC.

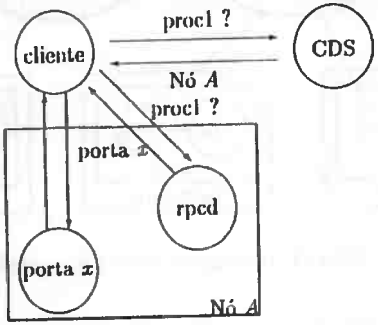

Figura 44: Procura do servidor pelo cliente.

- First-in, First-out (FIFO). O thread de maior prioridade que está esperando a mais tempo é executado primeiro, até ser bloqueado, ou terminar. Sendo que um thread com prioridade ainda maior bloqueia o thread sendo executado. Para prioridades iguais é utilizada uma FIFO.

- Round-Robin (RR). Todos os threads com maior prioridade são executados cada um por um tempo determinado. O processo é repetido até que todos os threads com esta prioridade terminem, ou estejam bloqueados. Então os threads no nível de prioridade imediatamente inferior são executados por Round-Robin.

- Default. Todos threads são executados, cada um por um tempo determinado. Threads com maior prioridade são executados por mais tempo. Desta forma mesmo o thread com menor a prioridade será executado em algum momento.

A comunicação entre threads é feita através de variáveis compartilhadas. Para resolver problemas de sincronismo, são fornecidos três recursos:

- Variáveis de exclusão mútua (mutex). Garantem que apenas uma tarefa acessa o recurso (pode ser uma variável compartilhada) relacionada ao mutex por vez.

- Variáveis de condição. Neste caso utilizam-se as primitivas wait e signal.

- Rotina join. Permite que um thread seja bloqueado até que outro thread especificado tenha terminado a sua execução.

7 DCE Serviço de Diretórios

O serviço de diretórios é o mais básico de todos os serviços distribuídos e é utilizado para encontrar informações necessárias para o acesso a outros serviços. O serviço de diretórios do DCE é composto por três componentes: O serviço de diretório da célula, o serviço de diretório global e o serviço agente de diretório global (figura 45).

Para permitir uma maior flexibilidade o DCE permite também o uso do padrão Domain Name Service (DNS, utilizado na internet). O DCE pode trabalhar com dois tipos de nomes de arquivos: tipado (sintaxe X.500) ou não tipado (sintaxe DNS). O DCE ainda dispõe de dois serviços em um

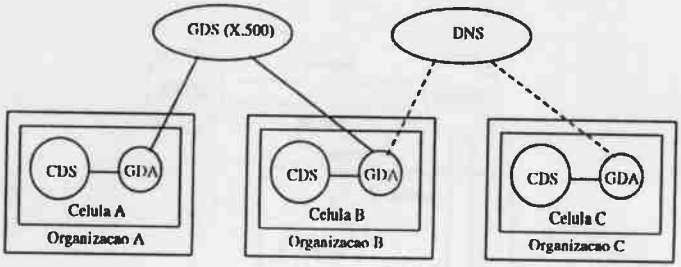

Figura 45: Interligação entre várias células.

servidor de diretórios: o primeiro é o serviço de registro, para garantir um nome único para cada recurso. Isto é feito utilizando o conceito de identificadores únicos universais (UUID). Outro serviço é relacionado com a possibilidade de especificar caminhos (paths) para a busca de nomes. Desta forma, os arquivos executáveis mais comuns que estiverem em diretórios diferentes podem ser chamados diretamente, como se estivessem no diretório corrente.

7.1 Serviço de diretório da célula

Este serviço armazena os nomes e atributos dos recursos localizados em uma célula do DCE. Este serviço está otimizado para o acesso local pois a maioria de pedidos de informação são da própria célula do originador do pedido. Deve existir no mínimo um servidor de diretório da célula em cada célula, mas pode existir mais do que um, para aumentar a disponibilidade. O serviço de diretório da célula (CDS), por sua vez, é composto por três componentes:

Servidor CDS. É executado em um nó contendo um banco de dados com informações sobre os diretórios. É responsável por atender os clientes que acessam o banco de dados da célula (que é denominado *clearinghouse*).

Empregado CDS. O empregado é executado nos nós clientes e servidores e funciona como intermediário entre as aplicações dos clientes e os servidores CDS. O cache no cliente é gerenciado por este processo, que é periodicamente gravado no disco local evitando assim problemas de *crash*, ou término do empregado CDS.

Administrador do CDS. É composto por dois programas, o *Browser* do espaço de nomes e o programa de controle do servidor de diretório da célula.

7.1.1 Banco de dados

As informações do banco de dados do CDS são de três tipos:

Registro do Diretório. Consiste de um nome e seus atributos.

Diretórios. Grupamento lógico das informações do CDS, que é composto por registros. O diretório é a unidade administrativa para replicação. Podem existir uma, ou mais cópias de um dado diretório.

Clearinghouses. Este é o banco de dados físico do CDS, que tipicamente é uma coleção de réplicas de diretórios.

7.1.2 Replicação e consistência de dados

Como muitos outros serviços dependem do servidor de diretórios, é essencial este serviço tenha uma grande disponibilidade, assim como um pequeno tempo de resposta. O CDS atinge estes dois objetivos com a replicação de diretórios e com o cacheamento dos registros de diretório.

Existem dois tipos de réplicas no CDS, a réplica principal, e as réplicas apenas para leitura. Só há uma réplica principal de um dado diretório, e nela pode ser realizado qualquer tipo de operação. Enquanto isto podem existir várias réplicas para leitura, as quais não permitem operações de atualização.

O CDS fornece dois métodos para consistência na replicação de diretórios:

Propagação Imediata. Garante a consistência o tempo todo, através da atualização imediata de todas as réplicas.

Skulking. Quando não há a necessidade de atualização imediata as réplicas dos diretórios são atualizadas periodicamente (por exemplo a cada 24 horas).

Com relação ao cacheamento, o CDS tem primitivas para aplicações clientes que ignoram a existência do cache, acessando diretamente a réplica principal. Isto é feito quando uma aplicação precisa necessariamente possuir a informação mais atualizada.

7.2 Servidor de diretório global

O Servidor de Diretório Global (GDS) do DCE tem implementação baseada no padrão internacional CCITT X.500/ISO 9594. O GDS tem duas funções básicas: Possibilitar a conexão com outras células e pode ser usado como um servidor de diretório da célula.

Vários componentes trabalham em conjunto no GDS.

- Agente Sistema de Diretório (DSA).

  Este processo é executado nas máquinas servidoras de GDS e gerencia o seu banco de dados. Para permitir várias consultas simultâneas, o Servidor de GDS pode rodar diversos processos DSA.

- Agente Usuário de Diretório (DUA).

  É uma biblioteca que implementa o cliente GDS, esta biblioteca está presente em todas as máquinas clientes GDS.

- Cache do agente usuário de diretório.

  Este processo mantém o cache com informações obtidas do DSA. Um processo pode passar por cima deste cache para obter a informação mais atualizada.

- Atualização de réplicas e cache.

  São processos executados apenas quando existe a necessidade, e não continuamente como os anteriores.

Cada máquina GDS pode ser organizada somente como cliente, ou como cliente/servidor.

7.2.1 Agente de diretório global

O Agente de Diretório Global (GDA, figura 45) funciona como um intermediário entre o servidor de diretórios e o servidor de diretórios global. Em particular, o GDA trata as chamadas do DCE dirigidas a outras células.

8 DCE Serviço de Hora Distribuída (DTS)

O DTS tem como função a manutenção da consistência do tempo nos relógios dos diversos componentes do sistema. O problema é tratado de duas formas:

1. Métodos para sincronizar, periodicamente, os relógios nos diferentes nós de um sistema distribuído.

2. Métodos para acertar a hora do sistema distribuído com a hora *correta* (Coordinated Universal Time - UTC).

Desta maneira os nós podem ter a mesma noção de hora, ou ter a hora correta.

Como relógios não podem ser sincronizados continuamente, existe sempre alguma diferença nos intervalos das sincronizações. Além disto existem vários problemas em relação ao envio e recebimento de mensagens de sincronização. Devido a estes problemas, o DTS armazena a hora, não como um ponto (número exato), e sim como um intervalo. Note que este intervalo deve conter a hora certa.

8.1 Organização

O DTS é composto de vários componentes:

- Time Clerk
- Servidor de hora
  - Servidor de hora local
  - Servidor de hora global
  - Servidor de hora courier

e possui duas interfaces:

- Interface para Aplicações (API)
- Interface para Fornecedor de Hora (TPI)
- Representação da Hora

Os componentes ativos são o Time Clerk e os diferentes tipos de Servidor de hora. Existem duas interfaces, uma para programação (API) e outra para um fornecedor de hora externo. Por fim, o DTS define também uma nova forma de expressar a hora.

Time Clerk. Representa o lado cliente do DTS. No momento da instalação o DCE permite que se escolha a configuração da máquina como clerk, ou como servidor.

Com uma periodicidade configurável de acordo com a imprecisão do relógio da máquina onde o Time Clerk está instalado, é requisitada a sincronização. O cliente pede a vários servidores de hora (figura 46) a hora certa e seu intervalo de imprecisão. Baseado nas respostas recebidas, a hora local é atualizada (figura 47). A atualização da hora local do cliente pode ser gradual, ou abrupta.

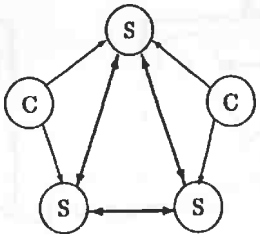

Figura 46: Rede com dois Time Clerks (C) e três Servidores de hora (S). Cada cliente consulta dois servidores de hora para sincronização. Os servidores de hora se consultam entre si.

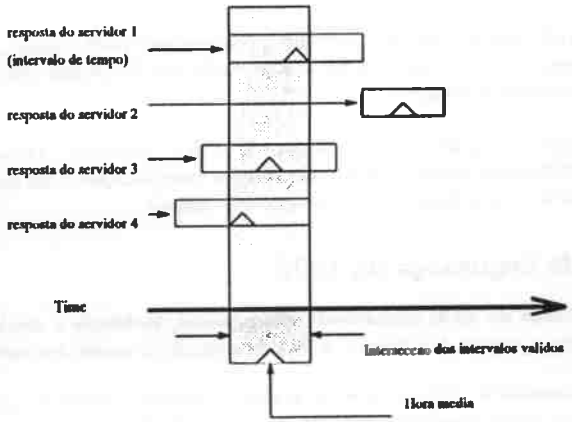

Figura 47: Cálculo da hora local. Neste caso os dados do servidor de hora 2 são ignorados.

Servidor de Hora. Um servidor de hora é projetado para responder consultas sobre o horário. Um número típico é de três servidores de hora por LAN. Os servidores de hora se consultam entre si para a atualização de horário. Um ou mais servidores de hora podem ser conectados a um fornecedor de hora externo.

Os servidores de hora locais e globais, são respectivamente os servidores localizados dentro, ou fora, de uma dada LAN. Um cliente pode precisar consultar um servidor de hora global se necessita da resposta de três servidores, quando apenas dois estão disponíveis na LAN.

Um servidor de hora courier é um servidor local que se sincroniza com um servidor de hora global.

Interface para Fornecedor de Hora. O fornecedor externo de hora pode ser automático, ou pode ser o administrador do sistema.

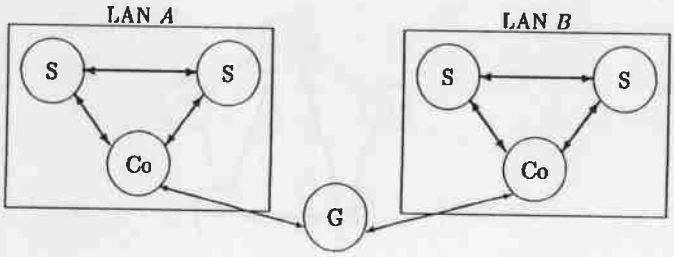

Figura 48: Dadas duas LAN's (A e B) e seus servidores de hora (S). Em cada LAN, um dos servidores de hora funciona como um servidor de hora courier (Co), que consulta um servidor de hora Global (G) (isto é, um servidor de hora que não está nem em A, e nem em B).

Interface para Aplicações. Com esta interface o programador pode acessar diretamente a hora distribuída. Como a representação inclui a imprecisão, são fornecidas novas rotinas para a comparação e conversão de tempos.

A resposta para a questão "Qual o tempo anterior, A, ou B ?" pode ser "Não sei.", logo as aplicações devem estar preparadas para situações deste tipo. Nesses casos as ações mais conservadoras devem ser tomadas (Ex: forçar a recompilação no caso do comando **make**.).

9 Serviço de Segurança do DCE

O Serviço de Segurança do DCE compreende várias partes, incluindo o serviço de autenticação, o serviço de privilégios, o serviço de registros, a lista de controle de acesso e o serviço de login.

- Serviço de autenticação. Permite que um *principal* (usuário, cliente ou servidor) confirme a sua identidade para outro principal. Permite, também que dois processos em máquinas distintas estejam certos da identidade, um do outro. Para isto cada principal do DCE possui uma chave secreta conhecida apenas pelo próprio e pelo servidor de autenticação.

- Serviço de privilégios. Permite a um principal determinar os direitos de outros principais, da mesma célula, através do nome e dos grupos que faz parte. Assim que um servidor verificou a identidade de um usuário que está solicitando um serviço, é necessário ainda determinar se este usuário deve ser autorizado, ou não a utilizar este serviço. O servidor de privilégios transmite, de uma forma segura, as informações que o servidor precisa saber para determinar quais são as permissões do usuário.

- Serviço de registros. É um serviço replicado que gerencia o banco de dados de segurança da célula. O banco de dados de segurança contém entradas para as entidades de segurança (usuários e servidores). O banco de dados também contém as informações associadas a cada entidade.

- Lista de controle de acesso. São listas de usuários que estão autorizados a ter acesso a um dado recurso.

- Servidor de login. Autentica o usuário ao serviço de segurança. O serviço de segurança retorna as credenciais, que serão utilizadas para a autenticação do usuário.

Exemplo de operações envolvendo o serviço de segurança.

1. O administrador de segurança do DCE utiliza a função `rgy-edit()` para registrar cada principal no Banco de Dados. A informação armazenada inclui a chave secreta e os atributos de privilégio.
2. Quando um usuário entra no sistema recebe um *ticket* necessário para o acesso a outros servidores.
3. O servidor de privilégios acessa o banco de dados de segurança para a construção das credenciais necessárias ao acesso a outros servidores (PAC).
4. Quando o usuário inicia uma aplicação cliente, este recebe como herança a identidade e os atributos de privilégio. A primeira RPC do cliente ao servidor inclui a identidade e os atributos de privilégio os quais o servidor pode autenticar novamente.

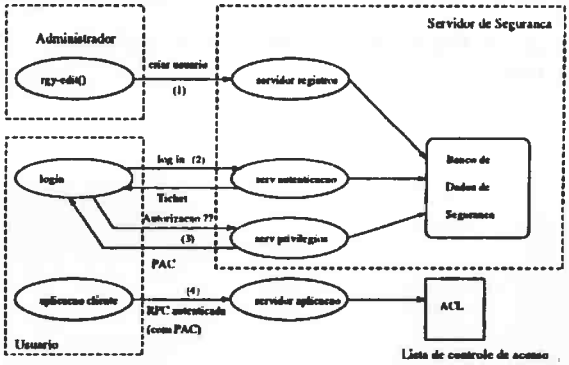

Figura 49: O ticket é codificado usando a password do usuário. PAC - informações das autorizações específicas ao usuário.

10 Conclusão

Com o apanhado anterior tivemos uma visão do funcionamento e dos componentes do DCE. Foram enfocadas as várias vantagens oferecidas, mas devido a falta de informação não foram apresentadas as possíveis dificuldades de implementação em uma rede (real) com máquinas e sistemas operacionais distintos.

Não podemos deixar de notar que existe uma tendência à criação de ambientes padrão para o desenvolvimento de softwares distribuídos sobre sistemas operacionais tradicionais. Além do DCE também podemos citar o ANSIware que possui esta mesma filosofia.

Como vantagens/desvantagens de ambientes como o DCE e sistemas operacionais distribuídos baseados em *Micro Kernel* temos:

- vantagens
  - transforma uma rede de computadores distintos, e/ou com sistemas operacionais distintos, em um único sistema de computação;
  - serviços básicos (sincronização, boot remoto, segurança, etc) já estão implementados;

- permite e dá suporte à diferentes tipos de representação de dados nas chamadas de procedimento remotas.

- desvantagens
  - tamanho (em cada servidor ficam, além do sistema operacional original componentes do DCE);
  - performance (por serem desenvolvidos já com o intuito de serem sistemas distribuídos os ambientes micro kernel tendem a ser mais rápidos).

Referências

[1] W. Rosenberry, D. Kenney and G. Fisher. *Understanding DCE*, O'Reilly & Associates, Inc., 1992.

[2] J. Shirley, W. Hu and D. Magid. *Guide to Writing DCE Applications*, O'Reilly & Associates, Inc., 2nd. Edition, 1994.

RELATÓRIOS TÉCNICOS

DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Instituto de Matemática e Estatística da USP

A listagem contendo os relatórios técnicos anteriores a 1993 poderá ser consultada ou solicitada à Secretaria do Departamento, pessoalmente, por carta ou e-mail(mac@ime.usp.br).

Imre Simon
THE PRODUCT OF RATIONAL LANGUAGES
RT-MAC-9301, Maio 1993, 18 pp.

Flávio Soares C. da Silva
AUTOMATED REASONING WITH UNCERTAINTIES
RT-MAC-9302, Maio 1993, 25 pp.

Flávio Soares C. da Silva
ON PROOF-AND MODEL-BASED TECHNIQUES FOR REASONING WITH UNCERTAINTY
RT-MAC-9303, Maio 1993, 11 pp.

Carlos Humes Jr., Leônidas de O.Brandão, Manuel Pera Garcia
A MIXED DYNAMICS APPROACH FOR LINEAR CORRIDOR POLICIES (A REVISITATION OF DYNAMIC SETUP SCHEDULING AND FLOW CONTROL IN MANUFACTURING SYSTEMS)
RT-MAC-9304, Junho 1993, 25 pp.

Ana Flora P.C.Humes e Carlos Humes Jr.
STABILITY OF CLEARING OPEN LOOP POLICIES IN MANUFACTURING SYSTEMS (Revised Version)
RT-MAC-9305, Julho 1993, 31 pp.

Maria Angela M.C. Gurgel e Yoshiko Wakabayashi
THE COMPLETE PRE-ORDER POLYTOPE: FACETS AND SEPARATION PROBLEM
RT-MAC-9306, Julho 1993, 29 pp.

Tito Homem de Mello e Carlos Humes Jr.
SOME STABILITY CONDITIONS FOR FLEXIBLE MANUFACTURING SYSTEMS WITH NO SET-UP TIMES
RT-MAC-9307, Julho de 1993, 26 pp.

Carlos Humes Jr. e Tito Homem de Mello
A NECESSARY AND SUFFICIENT CONDITION FOR THE EXISTENCE OF ANALYTIC CENTERS IN PATH FOLLOWING METHODS FOR LINEAR PROGRAMMING
RT-MAC-9308, Agosto de 1993

Flavio S. Corrêa da Silva
AN ALGEBRAIC VIEW OF COMBINATION RULES
RT-MAC-9401, Janeiro de 1994, 10 pp.

Flavio S. Corrêa da Silva e Junior Barrera
AUTOMATING THE GENERATION OF PROCEDURES TO ANALYSE BINARY IMAGES
RT-MAC-9402, Janeiro de 1994, 13 pp.

Junior Barrera, Gerald Jean Francis Banon e Roberto de Alencar Lotufo
A MATHEMATICAL MORPHOLOGY TOOLBOX FOR THE KHOROS SYSTEM
RT-MAC-9403, Janeiro de 1994, 28 pp.

Flavio S. Corrêa da Silva
ON THE RELATIONS BETWEEN INCIDENCE CALCULUS AND FAGIN-HALPERN STRUCTURES
RT-MAC-9404, abril de 1994, 11 pp.

Junior Barrera; Flávio Soares Corrêa da Silva e Gerald Jean Francis Banon
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES
RT-MAC-9405, abril de 1994, 15 pp.

Valdemar W. Setzer; Cristina G. Fernandes; Wania Gomes Pedrosa e Flavio Hirata
UM GERADOR DE ANALISADORES SINTÁTICOS PARA GRAFOS SINTÁTICOS SIMPLES
RT-MAC-9406, abril de 1994, 16 pp.

Siang W. Song
TOWARDS A SIMPLE CONSTRUCTION METHOD FOR HAMILTONIAN DECOMPOSITION OF THE HYPERCUBE
RT-MAC-9407, maio de 1994, 13 pp.

Julio M. Stern
MODELOS MATEMATICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9408, maio de 1994, 50 pp.

Imre Simon
STRING MATCHING ALGORITHMS AND AUTOMATA
RT-MAC-9409, maio de 1994, 14 pp.

Valdemar W. Setzer e Andrea Zisman
*CONCURRENCY CONTROL FOR ACCESSING AND COMPACTING B-TREES**
RT-MAC-9410, junho de 1994, 21 pp.

Renata Wassermann e Flávio S. Corrêa da Silva
TOWARDS EFFICIENT MODELLING OF DISTRIBUTED KNOWLEDGE USING EQUATIONAL AND ORDER-SORTED LOGIC
RT-MAC-9411, junho de 1994, 15 pp.

Jair M. Abe, Flávio S. Corrêa da Silva e Marcio Rillo
PARACONSISTENT LOGICS IN ARTIFICIAL INTELLIGENCE AND ROBOTICS.
RT-MAC-9412, junho de 1994, 14 pp.

Flávio S. Corrêa da Silva, Daniela V. Carbogim
A SYSTEM FOR REASONING WITH FUZZY PREDICATES
RT-MAC-9413, junho de 1994, 22 pp.

Flávio S. Corrêa da Silva, Jair M. Abe, Marcio Rillo
MODELING PARACONSISTENT KNOWLEDGE IN DISTRIBUTED SYSTEMS
RT-MAC-9414, julho de 1994, 12 pp.

Nami Kobayashi
THE CLOSURE UNDER DIVISION AND A CHARACTERIZATION OF THE RECOGNIZABLE Z-SUBSETS
RT-MAC-9415, julho de 1994, 29pp.

Flávio K. Miyazawa e Yoshiko Wakabayashi
AN ALGORITHM FOR THE THREE-DIMENSIONAL PACKING PROBLEM WITH ASYMPTOTIC PERFORMANCE ANALYSIS
RT-MAC-9416, novembro de 1994, 30 pp.

Thomaz I. Seidman e Carlos Humes Jr.
SOME KANBAN-CONTROLLED MANUFACTURING SYSTEMS: A FIRST STABILITY ANALYSIS
RT-MAC-9501, janeiro de 1995, 19 pp.

C.Humes Jr. and A.F.P.C. Humes
STABILIZATION IN FMS BY QUASI- PERIODIC POLICIES
RT-MAC-9502, março de 1995, 31 pp.

Fabio Kon e Arnaldo Mandel
SODA: A LEASE-BASED CONSISTENT DISTRIBUTED FILE SYSTEM
RT-MAC-9503, março de 1995, 18 pp.

Junior Barrera, Nina Sumiko Tomita, Flávio Soares C. Silva, Routo Terada
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY PAC LEARNING
RT-MAC-9504, abril de 1995,16 pp.

Flávio S. Corrêa da Silva e Fabio Kon
CATEGORIAL GRAMMAR AND HARMONIC ANALYSIS
RT-MAC-9505, junho de 1995,17 pp.

Henrique Mongelli e Routo Terada
ALGORITMOS PARALELOS PARA SOLUÇÃO DE SISTEMAS LINEARES
RT-MAC-9506, junho de 1995, 158 pp.

Kunio Okuda
PARALELIZAÇÃO DE LAÇOS UNIFORMES POR REDUCÃO DE DEPENDÊNCIA
RT-MAC-9507, julho de 1995, 27 pp.

Valdemar W. Setzer e Lowell Monke
COMPUTERS IN EDUCATION: WHY, WHEN, HOW
RT-MAC-9508, julho de 1995, 21 pp.

Flávio S. Corrêa da Silva
REASONING WITH LOCAL AND GLOBAL INCONSISTENCIES
RT-MAC-9509, julho de 1995, 16 pp.

Julio M. Stern
MODELOS MATEMÁTICOS PARA FORMAÇÃO DE PORTFÓLIOS
RT-MAC-9510, julho de 1995, 43 pp.

Fernando Iazzetta e Fabio Kon
A DETAILED DESCRIPTION OF MAXANNEALING
RT-MAC-9511, agosto de 1995, 22 pp.

Flávio Keidi Miyazawa e Yoshiko Wakabayashi
POLYNOMIAL APPROXIMATION ALGORITHMS FOR THE ORTHOGONAL Z-ORIENTED 3-D PACKING PROBLEM
RT-MAC-9512, agosto de 1995, pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON COLLECTIONS OF CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MORPHOLOGICAL MACHINES
RT-MAC-9513, agosto de 1995, 84 pp.

Marco Dimas Gubitoso e Jörg Cordsen
PERFORMANCE CONSIDERATIONS IN VOTE FOR PEACE
RT-MAC-9514, novembro de 1995, 18pp.

Carlos Eduardo Ferreira e Yoshiko Wakabayashi
ANAIS DA I OFICINA NACIONAL EM PROBLEMAS COMBINATÓRIOS: TEORIA, ALGORITMOS E APLICAÇÕES
RT-MAC-9515, novembro de 1995, 45 pp.

Markus Endler and Anil D'Souza
SUPPORTING DISTRIBUTED APPLICATION MANAGEMENT IN SAMPA
RT-MAC-9516, novembro de 1995, 22 pp.

Junior Barrera, Routo Terada,
Flávio Corrêa da Silva and Nina Sumiko Tomita
*AUTOMATIC PROGRAMMING OF MMACH'S FOR OCR**
RT-MAC-9517, dezembro de 1995, 14 pp.

Junior Barrera, Guillermo Pablo Salas and Ronaldo Fumio Hashimoto
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMING OF MMACH'S
RT-MAC-9518, dezembro de 1995, 14 pp.

Daniela V. Carbogim and Flávio S. Corrêa da Silva
FACTS, ANNOTATIONS, ARGUMENTS AND REASONING
RT-MAC-9601, janeiro de 1996, 22 pp.

Kunio Okuda
REDUÇÃO DE DEPENDÊNCIA PARCIAL E REDUÇÃO DE DEPENDÊNCIA GENERALIZADA
RT-MAC-9602, fevereiro de 1996, 20 pp.

Junior Barrera, Edward R. Dougherty and Nina Sumiko Tomita
AUTOMATIC PROGRAMMING OF BINARY MORPHOLOGICAL MACHINES BY DESIGN OF STATISTICALLY OPTIMAL OPERATORS IN THE CONTEXT OF COMPUTATIONAL LEARNING THEORY.
RT-MAC-9603, abril de 1996, 48 pp.

Junior Barrera e Guillermo Pablo Salas
SET OPERATIONS ON CLOSED INTERVALS AND THEIR APPLICATIONS TO THE AUTOMATIC PROGRAMMINIG OF MMACH'S
RT-MAC-9604, abril de 1995, 66 pp.

Kunio Okuda
CYCLE SHRINKING BY DEPENDENCE REDUCTION
RT-MAC-9605, maio de 1996, 25 pp.

Julio Stern, Fabio Nakano e Marcelo Lauretto
REAL: REAL-VALUED ATTRIBUTE CLASSIFICATION - TREE LEARNING ALGORITHM
RT-MAC-9606, agosto de 1996, pp.

Markus Endler
SISTEMAS OPERACIONAIS DISTRIBUÍDOS: CONCEITOS, EXEMPLOS E TENDÊNCIAS
RT-MAC-9607, agosto de 1996, 120 pp.