

3

Processos

Neste capítulo, examinaremos mais de perto como os diferentes tipos de processos desempenham papel crucial em sistemas distribuídos. O conceito de um processo é originário do campo de sistemas operacionais no qual, em geral, ele é definido como um programa em execução. Da perspectiva de sistema operacional, o gerenciamento e o escalonamento de processos são talvez as questões mais importantes a discutir. Contudo, quando se trata de sistemas distribuídos, outras questões mostram ser de igual ou maior importância.

Por exemplo, para organizar sistemas cliente-servidor com eficiência, muitas vezes é conveniente usar técnicas de multithreading. Como discutimos anteriormente, uma das principais contribuições de threads em sistemas distribuídos é que eles permitem que clientes e servidores sejam construídos de modo tal que comunicação e processamento local possam se sobrepor, o que resulta em alto nível de desempenho.

Nos últimos anos, o conceito de virtualização ganhou popularidade. A virtualização permite que uma aplicação, e possivelmente também seu ambiente completo, incluindo o sistema operacional, execute concorrentemente com outras aplicações, mas com alto grau de independência em relação ao hardware e plataformas subjacentes, o que resulta em alto grau de portabilidade. Além do mais, a virtualização ajuda a isolar falhas causadas por erros ou problemas de segurança. É um conceito importante para sistemas distribuídos, por isso vamos abordar o assunto mais adiante.

Como discutimos no Capítulo 2, organizações cliente-servidor são importantes em sistemas distribuídos. Neste capítulo, examinaremos mais de perto organizações típicas de clientes e de servidores. Também damos atenção a questões gerais de projeto para servidores.

Uma questão importante, em especial em sistemas distribuídos de longa distância, é movimentar processos entre máquinas diferentes. Migração de processo, ou, mais especificamente, migração de código, pode ajudar a conseguir escalabilidade, mas também pode ajudar a configurar dinamicamente clientes e servidores. Neste capítulo, também discutiremos o que realmente quer dizer migração de código e quais são suas implicações.

3.1 Threads

Embora processos formem um bloco de construção em sistemas distribuídos, a prática indica que a granularidade de processos proporcionada pelos sistemas operacionais sobre os quais os sistemas distribuídos são construídos não é suficiente. Em vez disso, observa-se que ter granularidade mais fina sob a forma de múltiplos threads de controle por processo facilita muito a construção de aplicações distribuídas e a obtenção de melhor desempenho. Nesta seção, examinaremos mais de perto o papel de threads em sistemas distribuídos e explicamos por que eles são tão importantes. Se quiser saber mais sobre threads e sobre como eles podem ser usados, consulte Lewis e Berg (1998) e Stevens (1999).

3.1.1 Introdução a threads

Para entender o papel dos threads em sistemas distribuídos, é importante entender o que é um processo e como processos e threads se relacionam. Para executar um programa, um sistema operacional cria vários processadores virtuais, cada um para executar um programa diferente. Para monitorar esses processadores virtuais, o sistema operacional tem uma **tabela de processos** que contém entradas para armazenar valores de registradores de CPU, mapas de memória, arquivos abertos, informações de contabilidade, privilégios e assim por diante. Um **processo** costuma ser definido como um programa em execução, isto é, um programa que está sendo executado em um dos processadores virtuais do sistema operacional no momento em questão.

Um aspecto importante é que o sistema operacional toma grande cuidado para assegurar que processos independentes não possam afetar, de modo intencional ou malicioso, ou mesmo por acidente, a correção do comportamento um do outro. Em outras palavras, o fato de que vários processos podem compartilhar concorrentemente a mesma CPU e outros recursos de hardware torna-se transparente. Normalmente, o sistema operacional requer suporte de hardware para impor essa separação.

Essa transparência de concorrência tem preço relativamente alto. Por exemplo, cada vez que um processo é criado, o sistema operacional deve criar um espaço de endereços completo independente. Alocação pode significar iniciar segmentos de memória: por exemplo, zerando um segmento de dados, copiando o programa associado para um segmento de texto e estabelecendo uma pilha para dados temporários.

Da mesma maneira, chavear a CPU entre dois processos pode ser igualmente caro. Além de salvar o contexto da CPU (que consiste em valores de registradores, contador de programa, ponteiro de pilha e assim por diante), o sistema operacional também terá de modificar registradores da unidade de gerenciamento de memória (Memory Management Unit — MMU) e invalidar caches de tradução de endereços como no buffer lateral de tradução (Translation Lookaside Buffer — TLB). Ademais, se o sistema operacional suportar mais processos do que pode conter simultaneamente a memória principal, ele talvez tenha de efetuar troca dinâmica de processos entre a memória principal e o disco antes que o chaveamento propriamente dito possa ocorrer.

Assim como um processo, um thread executa sua própria porção de código, independentemente de outros threads. Todavia, ao contrário dos processos, nenhuma tentativa é feita para conseguir alto grau de transparência de concorrência se isso resultar em degradação do desempenho. Por conseguinte, um sistema de threads em geral mantém a mínima informação que permita à CPU ser compartilhada por vários threads. Em particular, um **contexto de thread** freqüentemente consiste em nada mais que o contexto da CPU, junto com algumas outras informações para gerenciamento de threads. Por exemplo, um sistema de threads pode monitorar o fato de um thread estar bloqueado em uma variável de mútua exclusão em dado instante, de modo a não selecioná-lo para execução.

Informações que não são estritamente necessárias para gerenciar múltiplos threads em geral são ignoradas. Por essa razão, proteger dados contra acesso inadequado por threads dentro de um único processo fica inteiramente a cargo dos desenvolvedores da aplicação.

Há duas implicações importantes nessa abordagem. Antes de qualquer coisa, o desempenho de uma aplicação multithread não precisa ser necessariamente pior do que o de sua contraparte monothread. Na verdade, em muitos casos o multithreading resulta em ganho de desempenho. Em segundo lugar, como threads não são automaticamente protegidos uns contra os outros, como acontece com os processos, o desenvolvimento de aplicações multithread requer esforço intelectual adicional. Elaborar adequadamente o projeto e manter as coisas simples, como sempre, ajuda muito. Infelizmente, a prática corrente não demonstra que esse princípio é igualmente bem entendido.

Utilização de threads em sistemas não distribuídos

Antes de discutir o papel dos threads em sistemas distribuídos, vamos considerar sua utilização em sistemas não distribuídos tradicionais. Há vários benefícios proporcionados por processos multithread que aumentaram a popularidade da utilização de sistemas de threads.

O benefício mais importante vem do fato de que, em um processo monothread, sempre que for executada uma chamada bloqueadora de sistema, o processo é bloqueado como um todo. Para ilustrar, imagine uma aplicação como um programa de planilha e considere que um usuário quer alterar valores de maneira contínua e interativa. Uma propriedade importante de um programa de planilha é que ele mantém a dependência funcional entre diferentes células, muitas vezes em planilhas diferentes. Por conseguinte, sempre que uma célula for modificada, todas as células dependentes serão automaticamente atualizadas.

Quando um usuário altera o valor de uma única célula, essa modificação pode disparar uma grande série de cálculos. Se houver só um thread de controle, o cálculo não pode prosseguir enquanto o programa estiver esperando uma entrada. Da mesma maneira, não é fácil fornecer entrada enquanto as dependências estiverem sendo calculadas. A solução mais fácil é ter no mínimo dois threads de controle: um para manipular a interação com o usuário e outro para atualizar a planilha. No meio-tempo, um terceiro thread poderia ser usado para fazer uma cópia de segurança da planilha em disco enquanto os outros dois estivessem fazendo seu trabalho.

Uma outra vantagem do multithreading é que se torna possível explorar paralelismo ao executar o programa em um sistema multiprocessador. Nesse caso, cada thread é designado a uma CPU diferente, enquanto dados compartilhados são armazenados em memória principal compartilhada. Quando projetado adequadamente, tal paralelismo pode ser transparente: o processo executará igualmente bem em um sistema monoprocessador, se bem que com mais lentidão. O multithreading para paralelismo está se tornando cada vez mais importante com a disponibilidade de estações de trabalho multiprocessadoras relativamente baratas. Tais sistemas de computação normalmente são usados para rodar servidores em aplicações cliente-servidor.

O multithreading também é útil no contexto de grandes aplicações. Essas aplicações costumam ser desenvolvidas como um conjunto de programas cooperativos, cada qual executado por um processo separado. Essa abordagem é típica para um ambiente Unix. A cooperação entre programas é implementada por meio de mecanismos de comunicação entre processos (Interprocess Communication — IPC). Para sistemas Unix, esses mecanismos normalmente incluem pipes (nomeados), filas de mensagens e segmentos de memória compartilhados [veja também Stevens e Rago (2005)]. A principal desvantagem de

todos os mecanismos IPC é que a comunicação muitas vezes requer extensivo chaveamento de contexto, mostrado em três pontos diferentes na Figura 3.1.

Como IPC requer intervenção do núcleo, em geral um processo terá de chavear primeiro de modo usuário para modo núcleo, mostrado como S1 na Figura 3.1. Isso requer trocar o mapa de memória na MMU, bem como descarregar o TLB. Dentro do núcleo ocorre um chaveamento de contexto de processo (S2 na figura), após o qual a outra parte pode ser ativada pelo chaveamento de modo núcleo para modo usuário novamente (S3 na Figura 3.1). O último chaveamento requer, mais uma vez, trocar o mapa da MMU e descarregar o TLB.

Em vez de usar processos, também se pode construir uma aplicação tal que as diferentes partes sejam executadas por threads separados. A comunicação entre essas partes é inteiramente realizada com a utilização de dados compartilhados. O chaveamento de thread às vezes pode ser feito inteiramente em espaço de usuário, embora em outras implementações o núcleo esteja ciente dos threads e os escalone. O efeito pode ser uma drástica melhoria em desempenho.

Por fim, há também uma razão de pura engenharia de software para usar threads: muitas aplicações são simplesmente mais fáceis de estruturar como um conjunto de threads cooperativos. Imagine aplicações que precisem realizar várias tarefas, mais ou menos independentes. No caso de um processador de texto, por exemplo, podem ser usados threads separados para manipular entrada de usuário, verificação de ortografia e gramática, apresentação do documento, geração de índice e assim por diante.

Implementação de thread

Threads muitas vezes são fornecidos na forma de um pacote de threads. Tal pacote contém operações para criar e terminar threads, bem como operações sobre variáveis de sincronização como de mútua exclusão e variáveis de condição. Há, basicamente, duas abordagens para a implementação de um pacote de threads. A primeira é construir uma biblioteca de threads que é executada inteiramente em modo usuário. A segunda é fazer com que o núcleo fique ciente dos threads e os escalone.

Uma biblioteca de threads de nível de usuário tem várias vantagens. A primeira é que criar e terminar threads é barato. Como toda a administração de threads é mantida no espaço de endereço do usuário, o preço da criação de um thread é primariamente determinado pelo custo de alocar memória para estabelecer uma pilha de threads. De maneira análoga, terminar um thread envolve principalmente liberar para a pilha a memória que já não está mais sendo usada. Ambas as operações são baratas.

Uma segunda vantagem de threads de nível de usuário é que o chaveamento de contexto de thread pode ser feito em apenas algumas instruções. Basicamente, somente os valores dos registradores de CPU precisam ser armazenados e, na sequência, recarregados com os valores previamente carregados do thread para o qual está sendo chaveado. Não há nenhuma necessidade de mudar mapas de memória, descarregar o TLB, fazer contabilidade de CPU e assim por diante. O chaveamento de contexto de thread é feito quando dois threads precisam entrar em sincronia — por exemplo, ao entrar em uma seção de dados compartilhados.

Todavia, uma importante desvantagem de threads de nível de usuário é que a invocação de uma chamada bloqueadora de sistema imediatamente bloqueará todo o processo ao qual o thread pertence, e, assim, também todos os outros threads naquele processo. Como já explicamos, threads são particularmente úteis para estruturar grandes aplicações em partes que poderiam ser logicamente executadas ao mesmo tempo. Nesse caso, o bloqueio de E/S não deveria impedir que outras partes sejam executadas nesse meio-tempo. Para essas aplicações, threads de nível de usuário em nada ajudam.

Esses problemas podem ser contornados, em grande parte, pela implementação de threads no núcleo do sistema operacional. Infelizmente, o preço a pagar é alto: toda operação de thread — criação, encerramento, sincronização etc. — terá de ser executada pelo núcleo, o que requer uma chamada de sistema. Por isso, chavear contextos de thread pode ficar tão caro quanto chavear contextos de processo. O resultado é que, portanto, a maioria dos benefícios de desempenho proporcionada pela utilização de threads, em vez de processos, desaparece.

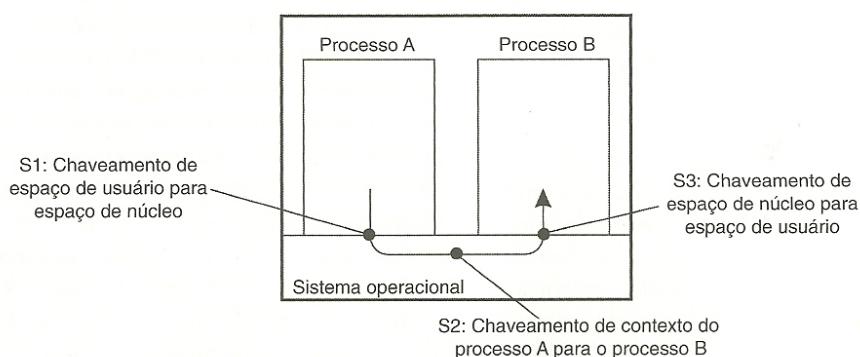


Figura 3.1 Chaveamento de contexto como resultado de IPC.

Uma solução se encontra em uma forma híbrida de *threads de nível de usuário e nível de núcleo, em geral* denominados **processos leves (Lightweight Processes — LWP)**. Um LWP executa no contexto de um único processo (pesado) e pode haver vários LWPs por processo. Além de ter LWPs, um sistema ainda fornece um pacote de threads de nível de usuário, o qual oferece às aplicações as operações usuais de criação e encerramento de threads. Além disso, o pacote provê facilidades para sincronização de threads, como de mútua exclusão e variáveis de condição. A questão importante é que o pacote de threads é implementado inteiramente em espaço de usuário. Em outras palavras, todas as operações em threads são realizadas sem intervenção do núcleo.

O pacote de threads pode ser compartilhado por vários LWPs, como mostra a Figura 3.2. Isso significa que cada LWP pode rodar seu próprio thread (de nível de usuário). Aplicações multithread são construídas com a criação de threads e, na seqüência, com a designação de cada thread a um LWP. A designação de um thread a um LWP normalmente é implícita e oculta do programador.

A combinação de threads (de nível de usuário) e LWPs funciona da seguinte maneira. Um pacote de threads tem uma única rotina para escalar o próximo thread. Ao criar um LWP (o que é feito por meio de uma chamada de sistema), este recebe sua própria pilha e é instruído para executar a rotina de escalonamento em busca de um thread para rodar. Se houver vários LWPs, cada um deles executa o escalonador. Assim, a tabela de threads, que é usada para monitorar o conjunto corrente de threads, é compartilhada pelos LWPs. A proteção dessa tabela para garantir o acesso mutuamente exclusivo é feita por meio de mútua exclusão, que é implementada inteiramente em espaço de usuário. Em outras palavras, a sincronização entre LWPs não requer nenhum suporte do núcleo.

Quando um LWP encontra um thread executável, chaveia o contexto para aquele thread. Enquanto isso, outros LWPs também podem estar procurando outros threads executáveis. Se um thread precisar bloquear por meio de mútua exclusão ou variável de condição, ele faz a administração necessária e, no devido tempo, chama a rotina de escalonamento. Quando for encontrado um

outro thread executável, é feito um chaveamento de contexto para aquele thread. O bom de tudo isso é que o LWP que está executando o thread não precisa ser informado: o chaveamento de contexto é completamente implementado em espaço de usuário e aparece para o LWP como código normal de programa.

Agora vamos ver o que acontece quando um thread faz uma chamada bloqueadora de sistema. Nesse caso, a execução muda de modo de usuário para modo de núcleo, mas ainda continua no contexto do LWP corrente. No ponto em que o LWP corrente não puder mais continuar, o sistema operacional pode decidir chavear contexto para um outro LWP, o que também implica que é feito um chaveamento de volta ao modo usuário. O LWP selecionado simplesmente continuará de onde tinha parado antes.

Há várias vantagens em utilizar LWPs em combinação com um pacote de threads de nível de usuário. A primeira é que criar, destruir e sincronizar threads é relativamente barato e não envolve absolutamente nenhuma intervenção do núcleo. A segunda é que, contanto que um processo tenha LWPs suficientes, uma chamada bloqueadora de sistema não suspenderá o processo inteiro. A terceira é que não há necessidade nenhuma de a aplicação ter conhecimento dos LWPs. Tudo que ela vê são threads de nível de usuário. A quarta é que LWPs podem ser usados com facilidade em ambientes de multiprocessamento, pela execução de diferentes LWPs em diferentes CPUs. Esse multiprocessamento pode ser inteiramente oculto da aplicação. A única desvantagem de processos leves combinados com threads de nível de usuário é que ainda precisamos criar e destruir LWPs, o que é exatamente tão caro quanto fazer o mesmo com threads de nível de núcleo. Contudo, a criação e a destruição de LWPs só precisam ser feitas ocasionalmente, e muitas vezes são totalmente controladas pelo sistema operacional.

Uma abordagem alternativa, mas similar, para processos leves é utilizar **ativações de escalonador** (Anderson et al., 1991). A diferença mais essencial entre ativações de escalonador e LWPs é que, quando um thread bloqueia em uma chamada de sistema, o núcleo faz uma *upcall* para o pacote de threads, o que equivale a chamar a rotina de escalonador para selecionar o próximo

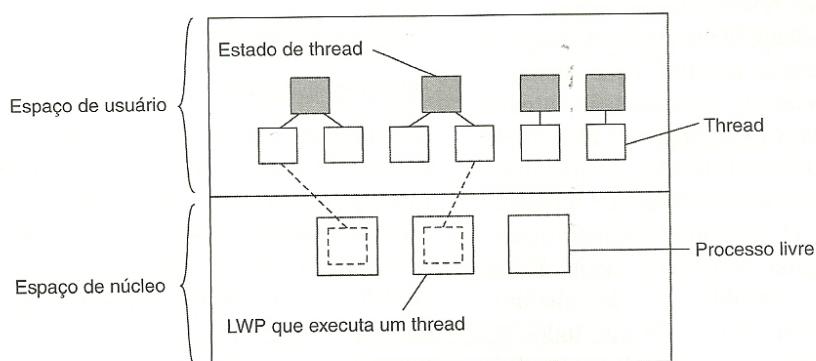


Figura 3.2 Combinação de processos leves de nível de núcleo e threads de nível de usuário.

thread executável. O mesmo procedimento é repetido quando um thread é desbloqueado. A vantagem dessa abordagem é que ela poupa gerenciamento de LWP's pelo núcleo. Contudo, a utilização de upcalls é considerada menos elegante porque viola a estrutura de sistemas em camadas, nos quais somente são permitidas chamadas a uma camada de nível mais baixo.

3.1.2 Threads em sistemas distribuídos

Uma importante propriedade de threads é que eles podem proporcionar um meio conveniente de permitir chamadas bloqueadoras de sistema sem bloquear o processo inteiro no qual o thread está executando. Essa propriedade torna os threads particularmente atraentes para utilização em sistemas distribuídos, uma vez que facilitam muito expressar comunicação na forma de manter múltiplas conexões lógicas ao mesmo tempo. Ilustramos esse ponto examinando mais de perto clientes e servidores multithread, respectivamente.

Cientes multithread

Para estabelecer um alto grau de transparência de distribuição, sistemas distribuídos que operam em redes de longa distância podem precisar esconder longos tempos de propagação de mensagens entre processos. O atraso de viagem de ida e volta em uma rede de longa distância pode ser facilmente da ordem de centenas de milissegundos ou, às vezes, até de segundos.

A maneira usual de ocultar latências de comunicação é iniciar a comunicação e imediatamente prosseguir com alguma outra coisa. Um exemplo típico onde isso acontece é em browsers Web. Em muitos casos, um documento Web consiste em um arquivo HTML que contém texto comum acompanhado de um conjunto de imagens, ícones e assim por diante. Para buscar cada elemento de um documento Web, o browser tem de estabelecer uma conexão TCP/IP, ler os dados que entram e passá-los ao componente de exibição — um visor. O estabelecimento de uma conexão bem como a leitura de dados que chegam são, inherentemente, operações bloqueadoras. Quando estivermos lidando com comunicação de longa distância, também temos a desvantagem de que o tempo para concluir cada operação possa ser relativamente longo.

Um browser Web freqüentemente inicia buscando a página HTML e, na seqüência, a exibe. Para ocultar o máximo possível as latências de comunicação, alguns browsers começam a exibir dados enquanto eles ainda estão entrando. Enquanto o texto está sendo disponibilizado para o usuário, incluindo as facilidades de rolagem e assemelhados, o browser continua buscando outros arquivos para compor a página, como as imagens. Essas últimas são apresentadas à medida que são trazidas. Assim, o usuário não precisa esperar até que todos os componentes da página inteira sejam buscados antes de a página ser disponibilizada.

Na verdade, verifica-se que o browser Web está executando várias tarefas simultaneamente. Ocorre que desenvolver o browser como um cliente multithread simplifica consideravelmente as coisas. Tão logo o principal arquivo HTML tenha sido buscado, threads separados podem ser ativados para se encarregar de buscar as outras partes. Cada thread estabelece uma conexão separada com o servidor e traz os dados.

O estabelecimento de uma conexão e a leitura de dados do servidor podem ser programados com a utilização das chamadas (bloqueadoras) de sistema padronizadas, considerando que uma chamada bloqueadora não suspende o processo inteiro. Como também é ilustrado em Stevens (1998), o código para cada thread é o mesmo e, acima de tudo, é simples. Enquanto isso, o usuário observa atrasos apenas na exibição de imagens e similares, porém, quanto ao mais, pode consultar o documento.

Há um outro benefício importante na utilização de browsers Web multithread no qual várias conexões podem ser abertas simultaneamente. No exemplo anterior, várias conexões foram estabelecidas com o mesmo servidor. Se a carga desse servidor estiver pesada, ou se ele for simplesmente lento, nenhuma melhoria real de desempenho será notada em comparação com trazer os arquivos que compõem a página estritamente um depois do outro.

Contudo, em muitos casos, servidores Web foram replicados em várias máquinas nas quais cada servidor fornece exatamente o mesmo conjunto de documentos. Os servidores replicados estão localizados no mesmo site e são conhecidos pelo mesmo nome. Quando entra uma requisição para uma página Web, ela é repassada para um dos servidores, freqüentemente por meio da utilização de uma estratégia de alternância cíclica ou alguma outra técnica de balanceamento de carga (Katz et al., 1994).

Ao ser usado um cliente multithread, podem ser estabelecidas conexões com diferentes réplicas, o que permite aos dados serem transferidos em paralelo. Por sua vez, isso determina efetivamente que o documento Web inteiro seja totalmente exibido em tempo muito menor do que com um servidor não replicado. Essa abordagem só é possível se o cliente puder manipular fluxos de dados de entrada verdadeiramente paralelos. Threads são ideais para essa finalidade.

Servidores multithread

Embora, como vimos, clientes multithread ofereçam benefícios importantes, a principal utilização de multithreading em sistemas distribuídos é encontrada no lado do servidor. A prática mostra que o multithreading não somente simplifica consideravelmente o código do servidor, mas também facilita muito o desenvolvimento de servidores que exploram paralelismo para obter alto desempenho, até mesmo em sistemas monoprocessadores. Contudo, agora que computadores multiprocessadores estão amplamente

disponíveis como estações de trabalho de uso geral, o multithreading para paralelismo é ainda mais útil.

Para entender os benefícios de threads para escrever código de servidor, considere a organização de um servidor de arquivos que ocasionalmente tenha de bloquear à espera do disco. O servidor de arquivos normalmente espera pela entrada de uma requisição para uma operação de arquivo e, na seqüência, executa a requisição e então devolve a resposta. Uma organização possível e particularmente popular é a mostrada na Figura 3.3. Nesse caso, um thread, o **despachante**, lê requisições que entram para uma operação de arquivo. As requisições são enviadas por clientes para uma porta bem conhecida para esse servidor. Após examinar a requisição, o servidor escolhe um **thread operário** ocioso (isto é, bloqueado) e lhe entrega a requisição.

O operário prossegue realizando uma leitura bloqueadora no sistema de arquivo *local*, que pode fazer com que o thread fique suspenso até que os dados sejam buscados no disco. Se o thread estiver suspenso, um outro thread é selecionado para ser executado. Por exemplo, o despachante pode ser selecionado para adquirir mais trabalho. Alternativamente, pode ser selecionado um outro thread operário que esteja pronto para executar no momento em questão.

Agora considere como o servidor de arquivos poderia ter sido escrito na ausência de threads. Uma possibilidade é fazer com que ele funcione como um único thread. O laço principal do servidor de arquivos obtém uma requisição, examina-a e a executa até a conclusão antes de obter a próxima. Enquanto espera pelo disco, o servidor fica ocioso e não processa nenhuma outra requisição. Por consequência, requisições de outros clientes não podem ser manipuladas. Ademais, se o servidor de arquivos estiver executando em uma máquina dedicada, como costuma ser o caso, a CPU fica naturalmente ociosa enquanto o servidor de arquivos estiver esperando pelo disco. O resultado líquido é que um número muito menor de requisições por segundo pode ser processado. Assim, threads resultam em considerável ganho de desempenho, mas cada thread é programado seqüencialmente, da maneira usual.

Até aqui, vimos dois projetos possíveis: um servidor de arquivos multithread e um servidor de arquivos monothread. Suponha que não haja threads disponíveis, mas os projetistas de sistemas acham inaceitável a perda de desempenho devida a monothread. Uma terceira possibilidade é rodar o servidor como uma grande máquina de estado finito. Quando uma requisição entra, o único e solitário thread a examina. Se a requisição puder ser satisfeita pela cache, tudo bem; porém, se não puder, uma mensagem deve ser enviada ao disco.

Contudo, em vez de bloquear, o thread registra o estado da requisição corrente em uma tabela e então vai obter a mensagem seguinte. Essa mensagem pode ser uma requisição para novo trabalho ou uma resposta do disco sobre uma operação anterior. Se for um novo trabalho, ele é iniciado. Se for uma resposta do disco, a informação relevante é buscada na tabela, a resposta é processada e, na seqüência, enviada ao cliente. Nesse esquema, o servidor terá de utilizar chamadas não bloqueadoras para *send* e *receive*.

Nesse projeto, perde-se o modelo de ‘processo seqüencial’ que tínhamos nos dois primeiros casos. O estado da computação tem de ser explicitamente salvo e restaurado na tabela para toda mensagem enviada e recebida. Na verdade, estamos simulando threads e suas pilhas do modo mais difícil. O processo está se operando como uma máquina de estado finito que obtém um evento e reage a ele, dependendo do que estiver dentro dele.

Nesse momento já deve estar claro o que os threads têm a oferecer. Eles possibilitam reter a idéia de processos seqüenciais que fazem chamadas bloqueadoras de sistema (por exemplo, uma RPC para falar com o disco) e ainda conseguem paralelismo. Chamadas bloqueadoras de sistema facilitam a programação, e paralelismo melhora o desempenho. O servidor monothread conserva a facilidade e a simplicidade de chamadas bloqueadoras de sistema, mas desiste de certa quantidade de desempenho. A abordagem da máquina de estado finito consegue alto desempenho por meio de paralelismo, mas usa chamadas não bloqueadoras, portanto é difícil de programar. Esses modelos estão resumidos na Tabela 3.1.

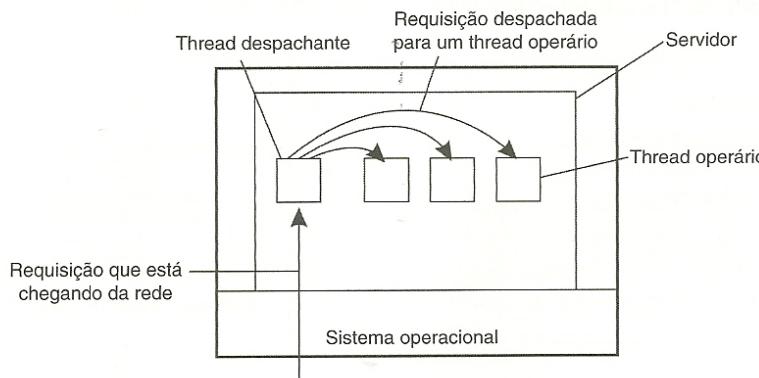


Figura 3.3 Servidor multithread organizado segundo modelo despachante/operário.

Modelo	Características
Threads	Paralelismo, chamadas bloqueadoras de sistema
Processo monothread	Sem paralelismo, chamadas bloqueadoras de sistema
Máquina de estado finito	Paralelismo, chamadas de sistema não bloqueadoras

Tabela 3.1 Três modos de construir um servidor.

3.2 Virtualização

Threads e processos podem ser vistos como um modo de fazer mais coisas ao mesmo tempo. Na verdade, eles nos permitem construir (porções de) programas que parecem ser executados simultaneamente. É claro que em um computador monoprocessador essa execução simultânea é uma ilusão. Como há somente uma única CPU, só uma instrução de um único thread ou processo será executada por vez. A ilusão de paralelismo é criada pelo chaveamento rápido entre threads e processos.

Essa separação entre ter uma única CPU e ser capaz de fingir que há mais delas pode ser estendida também a outros recursos, o que resulta no que denominamos **virtualização de recursos**. Essa virtualização é aplicada há décadas, mas conquistou renovado interesse à medida que sistemas (distribuídos) de computadores se tornaram mais comuns e complexos, o que resultou na situação em que o software de aplicação quase sempre sobrevive a seus sistemas subjacentes de software e hardware. Nesta seção, daremos um pouco de atenção ao papel da virtualização e discutiremos como ela pode ser realizada.

3.2.1 O papel da virtualização em sistemas distribuídos

Na prática, todo sistema (distribuído) de computadores oferece uma interface de programação a software de alto nível, como mostra a Figura 3.4(a). Há vários tipos diferentes de interfaces, que vão desde o conjunto básico de instruções oferecido por uma CPU até o vasto conjunto de interfaces de programação de aplicação que acompanha muitos dos sistemas atuais de middleware. Em sua essência, a virtualização trata de estender ou substituir uma inter-

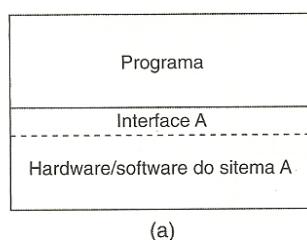
face existente de modo a imitar o comportamento de um outro sistema, como mostra a Figura 3.4(b). Em breve chegaremos à discussão de detalhes técnicos da virtualização, mas antes vamos nos concentrar em saber por que a virtualização é importante para sistemas distribuídos.

Uma das razões mais importantes para introduzir a virtualização na década de 1970 foi permitir que software herdado executasse em caros hardwares de mainframe. O software não somente incluía várias aplicações mas, na verdade, também os sistemas operacionais para os quais tinha sido desenvolvido. Essa abordagem voltada ao suporte de software herdado foi aplicada com sucesso nos mainframes IBM 370 (e seus sucessores), que ofereciam uma máquina virtual para a qual tinham sido transportados diferentes sistemas operacionais.

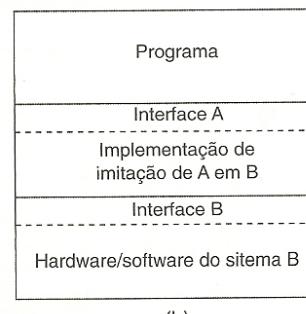
À medida que o hardware ficava mais barato, os computadores ficavam mais potentes e a quantidade de tipos diferentes de sistemas operacionais diminuía, a virtualização deixava de ser um problema tão importante. Todavia, as coisas mudaram desde o final da década de 1990 por várias razões, que discutiremos agora.

Em primeiro lugar, enquanto hardware e sistemas de software de baixo nível mudam com razoável rapidez, softwares em níveis mais altos de abstração — por exemplo, middleware e aplicações — são muito mais estáveis. Em outras palavras, estamos enfrentando a situação em que software herdado não pode ser mantido no mesmo passo que as plataformas de que depende. A virtualização pode ajudar transportando as interfaces herdadas para novas plataformas e, assim, abrindo imediatamente as últimas para grandes classes de programas existentes.

O fato de as redes já estarem onipresentes no ambiente da computação é de igual importância. É difícil imaginar que um computador moderno não esteja conectado a uma rede. Na prática, essa conectividade requer que os administradores de sistemas mantenham um conjunto grande e heterogêneo de computadores servidores, cada um executando aplicações muito diferentes, que podem ser acessadas por clientes. Ao mesmo tempo, os vários recursos devem estar facilmente acessíveis a essas aplicações. A virtualização pode ajudar muito: a diversidade de plataformas e máquinas pode ser reduzida, em essência, deixando que cada aplicação execute em sua



(a)



(b)

Figura 3.4 (a) Organização geral entre programa, interface e sistema. (b) Organização geral da virtualização do sistema A sobre o sistema B.

própria máquina virtual, possivelmente incluindo as bibliotecas e o sistema operacional relacionados que, por sua vez, executam em uma plataforma comum.

Esse último tipo de virtualização proporciona alto grau de portabilidade e flexibilidade. Por exemplo, para realizar redes de entrega de conteúdo que possam suportar com facilidade a replicação de conteúdo dinâmico, Awadallah e Rosenblum (2002) argumentam que o gerenciamento fica muito mais fácil se os servidores de borda suportarem virtualização, permitindo que um site completo, incluindo seu ambiente, seja copiado dinamicamente. Como discutiremos mais adiante, esses argumentos de portabilidade são as razões primordiais que fazem da virtualização um importante mecanismo para sistemas distribuídos.

3.2.2 Arquiteturas de máquinas virtuais

Há variados modos pelos quais a virtualização pode ser realizada na prática. Uma visão geral dessas diferenciadas abordagens é descrita por Smith e Nair (2005). Para entender as diferenças em virtualização, é importante perceber que, em geral, sistemas de computadores oferecem quatro tipos diferentes de interfaces em quatro níveis diferentes:

1. Uma interface entre o hardware e o software, o qual consiste em **instruções de máquina** que possam ser invocadas por qualquer programa.
2. Uma interface entre o hardware e o software, o qual consiste em instruções de máquina que possam ser invocadas somente por programas privilegiados, como um sistema operacional.
3. Uma interface que consiste em **chamadas de sistema** como oferecidas por um sistema operacional.
4. Uma interface que consiste em chamadas de biblioteca que, em geral, formam o que é conhecido como **interface de aplicação de programação (application programming interface — API)**. Em muitos casos, as chamadas de sistema mencionadas anteriormente estão ocultas por uma API.

Os diferentes tipos são mostrados na Figura 3.5. A essência da virtualização é imitar o comportamento dessas interfaces.

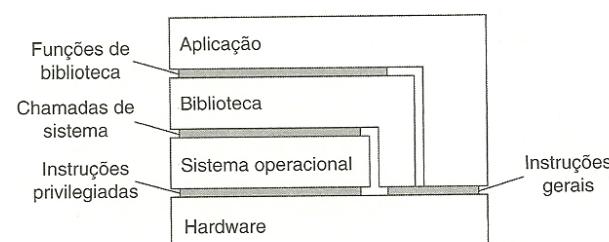


Figura 3.5 Várias interfaces oferecidas por sistemas de computadores.

A virtualização pode ocorrer de dois modos. Primeiro, podemos construir um sistema de execução que, em essência, forneça um conjunto de instruções abstrato que deve ser utilizado para executar aplicações. Instruções podem ser interpretadas (como é o caso do ambiente de execução Java), mas também poderiam ser emuladas, como acontece na execução de aplicações Windows em plataformas Unix. Observe que, no último caso, o emulador também terá de imitar o comportamento de chamadas de sistema, o que, em geral, mostrou estar longe de ser trivial. Esse tipo de virtualização resulta no que Smith e Nair (2005) denominam **máquina virtual de processo**, salientando que essa virtualização é feita, em essência, somente para um único processo.

Uma abordagem alternativa da virtualização é fornecer um sistema que seja essencialmente implementado como uma camada que protege completamente o hardware original, mas que oferece como interface o conjunto de instruções completo do mesmo — ou de outro — hardware. O fato de essa interface poder ser oferecida *simultaneamente* a programas diferentes é crucial. O resultado é que nessa circunstância é possível ter vários sistemas operacionais diferentes executando independente e concorrentemente na mesma plataforma. Em geral, essa camada é denominada **monitor de máquina virtual (Virtual Machine Monitor — VMM)**. Exemplos típicos dessa abordagem são VMware (Sugerman et al., 2001) e Xen (Barham et al., 2003). Essas duas abordagens são mostradas na Figura 3.6.

Como Rosenblum e Garfinkel (2005) argumentaram, VMMs ficarão cada vez mais importantes no contexto de confiabilidade e segurança para sistemas (distribuídos).

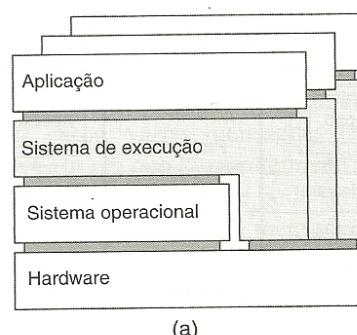
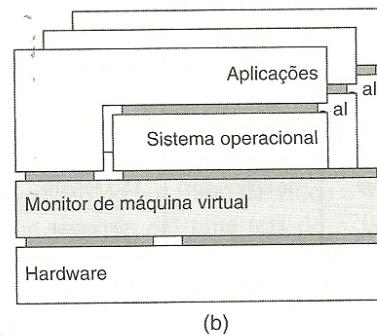


Figura 3.6 (a) Máquina virtual de processo, com várias instâncias de combinações (aplicação, execução).
(b) Monitor de máquina virtual com várias instâncias de combinações (aplicações, sistema operacional).



Como eles permitem o isolamento de uma aplicação completa e seu ambiente, uma falha causada por um erro ou ataque à segurança não precisa mais afetar uma máquina inteira. Ademais, como também mencionamos antes, a portabilidade se aprimora porque os VMMs promovem maior desacoplamento entre hardware e software, o que permite a um ambiente completo ser movido de uma máquina para outra.

3.3 Clientes

Nos capítulos anteriores, discutimos o modelo cliente-servidor, os papéis de clientes e servidores e a maneira como eles interagem. Agora vamos examinar mais de perto a anatomia de clientes e servidores, respectivamente. Nesta seção, começaremos com uma discussão a respeito de clientes. Servidores serão discutidos na próxima seção.

3.3.1 Interfaces de usuário em rede

Uma tarefa importante de máquinas clientes é proporcionar aos usuários meios de interagir com servidores remotos. Há praticamente só dois modos pelos quais essa interação pode ser realizada. Primeiro, para cada serviço remoto, a máquina cliente terá uma contraparte separada que pode contatar o serviço pela rede. Um exemplo típico é uma agenda que executa no PDA de um usuário e que precisa entrar em sincronia com uma agenda remota, possivelmente compartilhada. Nesse caso, um protocolo de nível de aplicação manipulará a sincronização, como mostra a Figura 3.7(a).

Uma segunda solução é fornecer acesso direto a serviços remotos oferecendo apenas uma interface de usuário conveniente. Na verdade, isso significa que a máquina cliente é usada só como um terminal sem nenhuma necessidade de armazenamento local, o que resulta em uma solução independente de aplicação, como mostra a Figura 3.7(b). No caso de interfaces de usuário em rede, tudo é processado e armazenado no servidor. Essa abordagem de **terminais clientes minimizados (thin clients)** está recebendo mais atenção à medida que aumenta a conectividade da Internet e que dispositivos de mão ficam cada vez

mais sofisticados. Como argumentamos no capítulo anterior, as soluções de clientes minimizados também são populares porque facilitam a tarefa de gerenciamento de sistema. Vamos estudar como interfaces de usuário em rede podem ser realizadas.

Exemplo: sistema X Window

Talvez uma das mais antigas e ainda amplamente usadas interfaces de usuário em rede seja o **sistema X Window**. O sistema X Window, em geral denominado simplesmente X, é usado para controlar terminais mapeados em bits, que incluem monitor, teclado e dispositivo de ponteiro, como um mouse. De certo modo, o X pode ser visto como a parte de um sistema operacional que controla o terminal. O cerne do sistema é formado pelo que denominaremos **núcleo X**. Ele contém todos os drivers de dispositivos específicos de terminal e, por isso, é, em geral, altamente dependente de hardware.

O núcleo X oferece uma interface de nível relativamente baixo para controlar a tela e também para capturar eventos do teclado e do mouse. Essa interface é disponibilizada para aplicações como uma biblioteca denominada *Xlib*. Essa organização geral é mostrada na Figura 3.8.

O aspecto interessante do X é que o núcleo X e as aplicações X não precisam necessariamente residir na mesma máquina. Em particular, X fornece o **protocolo X**, que é um protocolo de comunicação de camada de aplicação pelo qual uma instância de *Xlib* pode trocar dados e eventos com o núcleo X. Por exemplo, *Xlib* pode enviar requisições ao núcleo X para criar ou encerrar uma janela, estabelecer cores e definir o tipo de cursor a exibir, entre muitas outras requisições. Por sua vez, o núcleo X reagirá a eventos locais como entrada de teclado e mouse devolvendo pacotes de eventos a *Xlib*.

Várias aplicações podem se comunicar ao mesmo tempo com o núcleo X. Há uma aplicação específica que recebe direitos especiais, conhecida como **gerenciador de janela**. Essa aplicação pode determinar a aparência geral do visor como ele se apresenta ao usuário. Por exemplo, o gerenciador de janela pode prescrever como cada janela é decorada com botões extras, como as janelas devem ser colocadas no visor e assim por diante. Outras aplicações terão de adotar essas regras.

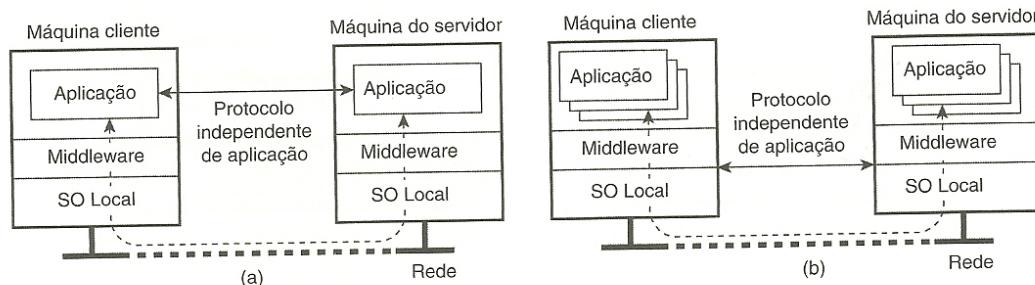


Figura 3.7 (a) Aplicação em rede com seu próprio protocolo. (b) Solução geral para permitir acesso a aplicações remotas.

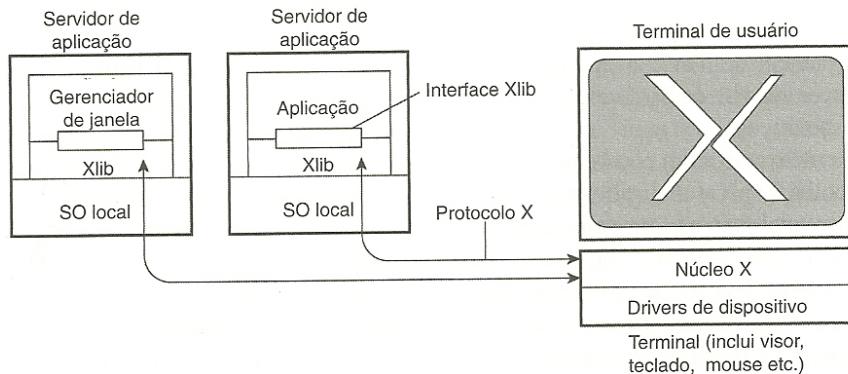


Figura 3.8 Organização básica do sistema X Window.

É interessante notar como o sistema X Window se ajusta à computação cliente–servidor. Pelo que descrevemos até aqui, deve estar claro que o núcleo X recebe requisições para manipular o visor e que recebe essas requisições de aplicações, possivelmente remotas. Nesse sentido, o núcleo X age como um servidor, enquanto as aplicações desempenham o papel de clientes. Essa terminologia foi adotada por X e, embora correta em termos estritos, pode facilmente levar à confusão.

Computação em redes de terminais clientes minimizados

É óbvio que aplicações manipulam um visor usando os comandos específicos de visor como oferecidos por X. Em geral, esses comandos são enviados pela rede em que, na seqüência, são executados pelo núcleo X. Por sua natureza, aplicações escritas para X devem, de preferência, separar a lógica da aplicação dos comandos de interface de usuário. Infelizmente, nem sempre é esse o caso. Como relatado por Lai e Nieh (2002), ocorre que grande parte da lógica da aplicação e da interação do usuário está fortemente acoplada, o que significa que uma aplicação enviará muitas requisições ao núcleo X para as quais esperará uma resposta antes de poder avançar para uma nova etapa. Esse comportamento síncrono pode causar efeitos adversos sobre o desempenho quando em operação por uma rede de longa distância que tenha longas latências.

Há várias soluções para esse problema. Uma é elaborar uma nova engenharia para a implementação do protocolo X, como é feito com NX (Pinzari, 2003). Uma parte importante desse trabalho se concentra na redução da largura de banda pela compressão de mensagens X. Primeiro, considera-se que mensagens consistem em uma parte fixa, que é tratada como identificador, e uma parte variável. Em muitos casos, várias mensagens terão o mesmo identificador, caso em que freqüentemente conterão dados semelhantes. Essa propriedade pode ser usada para enviar somente as diferenças entre mensagens que têm o mesmo identificador.

O lado remetente, bem como o lado destinatário, mantêm uma cache local cujas entradas podem ser consultadas com a utilização do identificador de uma mensagem. Quando uma mensagem é enviada, em primeiro

lugar ela é consultada na cache local. Se for encontrada, isso significa que uma mensagem anterior com o mesmo identificador mas, possivelmente, dados diferentes foi enviada. Nesse caso é utilizada a codificação diferencial para enviar somente as diferenças entre as duas.

No lado destinatário, a mensagem também é consultada na cache local; depois disso, pode ocorrer a decodificação por meio das diferenças. Quando houver ausência da cache, são utilizadas técnicas de compressão padronizadas que, em geral, já resultam em um fator quatro de melhoria na largura de banda. No todo, essa técnica tem apresentado reduções de largura de banda que alcançam um fator de 1.000, o que permite que X também execute por enlaces de largura de banda de apenas 9.600 kbps.

Um efeito colateral importante de armazenar mensagens é que remetente e destinatário têm informações compartilhadas sobre o estado corrente do visor. Por exemplo, a aplicação pode requisitar informações geométricas sobre vários objetos por meio de simples requisições de consulta na cache local. Só o fato de ter essa informação compartilhada já reduz a quantidade de mensagens requeridas para manter sincronizados a aplicação e o visor.

Apesar dessas melhorias, o X ainda requer ter um servidor de exibição em execução. Isso talvez seja exigir demais, em especial se o visor for algo tão simples como um telefone celular. Uma solução para manter muito simples o software embutido no visor é deixar que todo o processamento ocorra no lado da aplicação. Na realidade, isso significa que o visor inteiro é controlado até o nível de pixel no lado da aplicação. Portanto, mudanças no mapa de bits são enviadas pela rede até o visor, onde são imediatamente transferidas para o buffer de quadros local.

Essa abordagem requer sofisticadas técnicas de compressão de modo a impedir que a disponibilidade de largura de banda se torne um problema. Considere a apresentação de um fluxo de vídeo a uma taxa de 30 quadros por segundo em uma tela de 320×240 , por exemplo. Esse tamanho de tela é comum para muitos PDAs. Se cada pixel for codificado por 24 bits, sem compressão precisaríamos de uma largura de banda de aproximadamente 53 Mbps. A compressão é claramente necessária em tal caso e, hoje, muitas técnicas estão sendo disponi-

bilizadas. Entretanto, note que compressão requer descompressão no destinatário, o que, por sua vez, pode ser caro em termos de computação se não houver suporte de hardware. Pode-se fornecer suporte de hardware, mas isso aumenta o custo dos dispositivos.

A desvantagem de enviar dados em pixels brutos, em comparação com protocolos de nível mais alto como o X, é que é impossível fazer qualquer uso da semântica da aplicação, pois ela efetivamente se perde naquele nível. Baratto et al. (2005) propõem uma técnica diferente. Em sua solução, denominada THINC, eles fornecem alguns comandos de alto nível para o visor, que operam no nível dos drivers do dispositivo de vídeo. Assim, esses comandos são dependentes de dispositivo, mas poderosos do que operações em pixels brutos, porém menos poderosos em comparação com o que é oferecido por um protocolo como o X. O resultado é que os servidores de visor podem ser muito mais simples, o que é bom para a utilização da CPU, enquanto, ao mesmo tempo, otimizações dependentes de aplicação podem ser usadas para reduzir largura de banda e sincronização.

Em THINC, requisições de visor vindas da aplicação são interceptadas e traduzidas para os comandos de nível mais baixo. Com a interceptação de requisições de aplicação, o THINC pode fazer uso da semântica da aplicação para decidir qual combinação de comandos de nível mais baixo pode ser mais bem utilizada. Comandos traduzidos não são enviados imediatamente ao visor; em vez disso, são enfileirados. Reunindo vários comandos em lotes, é possível agregar comandos de visor em um único comando, o que resulta em menor número de mensagens. Por exemplo, quando um novo comando para desenhar em determinada região da tela sobrescreve efetivamente o que um comando anterior — e ainda enfileirado — teria estabelecido, esse último não precisa ser enviado ao visor.

Por fim, em vez de deixar o visor solicitar atualizações, o THINC sempre empurra atualizações à medida que elas ficam disponíveis. Essa abordagem de empurrar poupa latência porque não há nenhuma necessidade de o visor enviar uma requisição de atualização.

A abordagem seguida pelo THINC proporciona melhor desempenho global, embora muito próximo daquela mostrado por NX. Detalhes sobre comparação de desempenho podem ser encontrados em Baratto et al. (2005).

Documentos compostos

Modernas interfaces de usuário fazem muito mais do que sistemas como o X ou suas aplicações simples. Em particular, muitas interfaces de usuário permitem que aplicações compartilhem uma única janela gráfica e usem essa janela para trocar dados por meio de ações de usuário. Entre as ações adicionais que podem ser executadas pelo usuário estão as que são denominadas, de modo geral, operações de **arrastar e soltar** e **edição no local**.

Um exemplo típico de funcionalidade arrastar e soltar é mover um ícone que representa um arquivo A até um ícone que representa uma lixeira, o que resulta na remoção do arquivo. Nesse caso, a interface de usuário precisará fazer mais do que apenas arranjar os ícones no visor: ela terá de passar o nome do arquivo A para a aplicação associada com a lixeira tão logo o ícone de A tenha sido deslocado até em cima do ícone da aplicação de lixeira. É fácil lembrar de outros exemplos.

A edição no local pode ser mais bem ilustrada por meio de um documento que contenha texto e gráficos. Imagine que o documento será exibido dentro de um processador de texto padrão. Assim que o usuário colocar o mouse em cima de uma imagem, a interface de usuário passará essa informação para um programa de desenho a fim de permitir que o usuário modifique a imagem. Por exemplo, o usuário pode ter feito uma rotação da imagem, o que pode afetar a disposição da imagem no documento. Por conseguinte, a interface de usuário descobre quais são as novas altura e largura da imagem e passa essas informações ao processador de texto. Este, por sua vez, pode atualizar automaticamente o layout da página do documento.

A idéia fundamental por trás dessas interfaces de usuário é a noção de um **documento composto**, que pode ser definido como um conjunto de documentos, possivelmente de vários tipos bem diferentes (como texto, imagens, planilhas e assim por diante), integrado no nível de interface de usuário sem que se percebam separações. Uma interface de usuário que pode manipular documentos compostos oculta o fato de que diferentes aplicações operam em diferentes partes do documento. Para o usuário, todas as partes estão integradas sem que se percebam separações. Quando a troca de uma parte afeta outras partes, a interface de usuário pode tomar providências adequadas, por exemplo, informar as aplicações relevantes.

De forma semelhante à situação descrita para o sistema X Window, as aplicações associadas com um documento composto não têm de executar na máquina cliente. Contudo, é preciso ficar claro que interfaces de usuário que suportam documentos compostos poderão ter de fazer muito mais processamento do que as que não suportam.

3.3.2 Software do lado cliente para transparência de distribuição

Software cliente comprehende mais do que apenas interfaces de usuário. Em muitos casos, partes do nível de processamento e dados em uma aplicação cliente-servidor são executadas também no lado cliente. Uma classe especial é formada por software cliente embutido, tal como o de caixas automáticos (ATMs), caixas registradoras, leitoras de códigos de barra, receptores de TV e assim por diante. Nesses casos, a interface de usuário é uma parte relativamente pequena do

bilizadas. Entretanto, note que compressão requer descompressão no destinatário, o que, por sua vez, pode ser caro em termos de computação se não houver suporte de hardware. Pode-se fornecer suporte de hardware, mas isso aumenta o custo dos dispositivos.

A desvantagem de enviar dados em pixels brutos, em comparação com protocolos de nível mais alto como o X, é que é impossível fazer qualquer uso da semântica da aplicação, pois ela efetivamente se perde naquele nível. Baratto et al. (2005) propõem uma técnica diferente. Em sua solução, denominada THINC, eles fornecem alguns comandos de alto nível para o visor, que operam no nível dos drivers do dispositivo de vídeo. Assim, esses comandos são dependentes de dispositivo, mas poderosos do que operações em pixels brutos, porém menos poderosos em comparação com o que é oferecido por um protocolo como o X. O resultado é que os servidores de visor podem ser muito mais simples, o que é bom para a utilização da CPU, enquanto, ao mesmo tempo, otimizações dependentes de aplicação podem ser usadas para reduzir largura de banda e sincronização.

Em THINC, requisições de visor vindas da aplicação são interceptadas e traduzidas para os comandos de nível mais baixo. Com a interceptação de requisições de aplicação, o THINC pode fazer uso da semântica da aplicação para decidir qual combinação de comandos de nível mais baixo pode ser mais bem utilizada. Comandos traduzidos não são enviados imediatamente ao visor; em vez disso, são enfileirados. Reunindo vários comandos em lotes, é possível agregar comandos de visor em um único comando, o que resulta em menor número de mensagens. Por exemplo, quando um novo comando para desenhar em determinada região da tela sobrescreve efetivamente o que um comando anterior — e ainda enfileirado — teria estabelecido, esse último não precisa ser enviado ao visor.

Por fim, em vez de deixar o visor solicitar atualizações, o THINC sempre empurra atualizações à medida que elas ficam disponíveis. Essa abordagem de empurrar poupa latência porque não há nenhuma necessidade de o visor enviar uma requisição de atualização.

A abordagem seguida pelo THINC proporciona melhor desempenho global, embora muito próximo daquela mostrado por NX. Detalhes sobre comparação de desempenho podem ser encontrados em Baratto et al. (2005).

Documentos compostos

Modernas interfaces de usuário fazem muito mais do que sistemas como o X ou suas aplicações simples. Em particular, muitas interfaces de usuário permitem que aplicações compartilhem uma única janela gráfica e usem essa janela para trocar dados por meio de ações de usuário. Entre as ações adicionais que podem ser executadas pelo usuário estão as que são denominadas, de modo geral, operações de **arrastar e soltar** e **edição no local**.

Um exemplo típico de funcionalidade arrastar e soltar é mover um ícone que representa um arquivo A até um ícone que representa uma lixeira, o que resulta na remoção do arquivo. Nesse caso, a interface de usuário precisará fazer mais do que apenas arranjar os ícones no visor: ela terá de passar o nome do arquivo A para a aplicação associada com a lixeira tão logo o ícone de A tenha sido deslocado até em cima do ícone da aplicação de lixeira. É fácil lembrar de outros exemplos.

A edição no local pode ser mais bem ilustrada por meio de um documento que contenha texto e gráficos. Imagine que o documento será exibido dentro de um processador de texto padrão. Assim que o usuário colocar o mouse em cima de uma imagem, a interface de usuário passará essa informação para um programa de desenho a fim de permitir que o usuário modifique a imagem. Por exemplo, o usuário pode ter feito uma rotação da imagem, o que pode afetar a posição da imagem no documento. Por conseguinte, a interface de usuário descobre quais são as novas altura e largura da imagem e passa essas informações ao processador de texto. Este, por sua vez, pode atualizar automaticamente o layout da página do documento.

A idéia fundamental por trás dessas interfaces de usuário é a noção de um **documento composto**, que pode ser definido como um conjunto de documentos, possivelmente de vários tipos bem diferentes (como texto, imagens, planilhas e assim por diante), integrado no nível de interface de usuário sem que se percebam separações. Uma interface de usuário que pode manipular documentos compostos oculta o fato de que diferentes aplicações operam em diferentes partes do documento. Para o usuário, todas as partes estão integradas sem que se percebam separações. Quando a troca de uma parte afeta outras partes, a interface de usuário pode tomar providências adequadas, por exemplo, informar as aplicações relevantes.

De forma semelhante à situação descrita para o sistema X Window, as aplicações associadas com um documento composto não têm de executar na máquina cliente. Contudo, é preciso ficar claro que interfaces de usuário que suportam documentos compostos poderão ter de fazer muito mais processamento do que as que não suportam.

3.3.2 Software do lado cliente para transparência de distribuição

Software cliente compreende mais do que apenas interfaces de usuário. Em muitos casos, partes do nível de processamento e dados em uma aplicação cliente-servidor são executadas também no lado cliente. Uma classe especial é formada por software cliente embutido, tal como o de caixas automáticos (ATMs), caixas registradoras, leitoras de códigos de barra, receptores de TV e assim por diante. Nesses casos, a interface de usuário é uma parte relativamente pequena do

software cliente, em contraste com as facilidades locais de processamento e comunicação.

Além da interface de usuário e de outros softwares relacionados com aplicação, o software cliente comprehende componentes para conseguir transparência de distribuição. O ideal seria que um cliente não ficasse ciente de que está se comunicando com processos remotos. Ao contrário, a distribuição muitas vezes é menos transparente para servidores por razões de desempenho e correção. Por exemplo, no Capítulo 6 mostraremos que servidores replicados às vezes precisam se comunicar de modo a determinar que as operações sejam executadas em ordem específica em cada réplica.

Transparência de acesso é, em geral, manipulada por meio da geração de um apêndice de cliente conforme uma definição da interface do que o servidor tem a oferecer. O apêndice fornece a mesma interface que está disponível no servidor, mas oculta as possíveis diferenças em arquiteturas de máquina, bem como a comunicação propriamente dita.

Há vários modos diferentes de manipular transparência de localização, de migração e de relocação. Usar um sistema de nomeação conveniente é crucial, como veremos no próximo capítulo. Em muitos casos, a cooperação com o software do lado cliente também é importante. Por exemplo, quando um cliente já está vinculado a um servidor, ele pode ser informado diretamente quando o servidor mudar de localização. Nesse caso, o middleware do cliente pode ocultar do usuário a corrente localização geográfica do servidor e ainda, se necessário, vincular-se novamente ao servidor de modo transparente. Na pior das hipóteses, a aplicação do cliente pode notar perda de desempenho temporária.

De modo semelhante, muitos sistemas distribuídos implementam transparência de replicação por meio de soluções do lado cliente. Por exemplo, imagine um sistema distribuído com servidores replicados. Tal transparência de replicação pode ser conseguida com o repasse de uma requisição a cada réplica, como mostra a Figura 3.9. O software do lado cliente pode colher todas as respostas e passar uma única resposta à aplicação cliente, conservando a transparência.

Por fim, considere a transparência à falha. O mascaramento de falhas de comunicação com um servidor nor-

malmente é feito por meio de middleware cliente. Por exemplo, middleware cliente pode ser configurado para tentar a conexão com um servidor repetidas vezes, ou talvez tentar um outro servidor após várias tentativas. Também há situações nas quais o middleware cliente retorna dados que tinha guardado em cache durante a sessão anterior, como às vezes é feito por browsers Web que não conseguem se conectar com um servidor.

Transparência de concorrência pode ser manipulada por meio de servidores intermediários especiais, particularmente monitores de transação, e requer menos suporte de software cliente. Da mesma maneira, muitas vezes a transparência de persistência é completamente manipulada no servidor.

3.4 Servidores

Agora vamos examinar mais de perto a organização de servidores. Nas páginas seguintes, primeiro vamos nos concentrar em várias questões gerais de projeto para servidores, seguidas de uma discussão de clusters de servidores.

3.4.1 Questões gerais de projeto

Um servidor é um processo que implementa um serviço específico em nome de um conjunto de clientes. Em essência, cada servidor é organizado do mesmo modo: ele espera por uma requisição que vem de um cliente e, na seqüência, assegura que ela seja atendida, após o que espera pela próxima requisição.

Há vários modos de organizar servidores. No caso de um **servidor iterativo**, é o próprio servidor que manipula a requisição e, se necessário, retorna uma resposta ao cliente requisitante. Um **servidor concorrente** não manipula por si próprio a requisição, mas a passa para um thread separado ou para um outro processo, após o que imediatamente espera pela próxima requisição. Um servidor multithread é um exemplo de servidor concorrente. Uma implementação alternativa de um servidor concorrente é bifurcar um novo processo para cada requisição que chegar. Essa abordagem é adotada em muitos sistemas Unix. O thread ou processo que manipula a requisição é responsável por devolver uma resposta ao cliente requisitante.

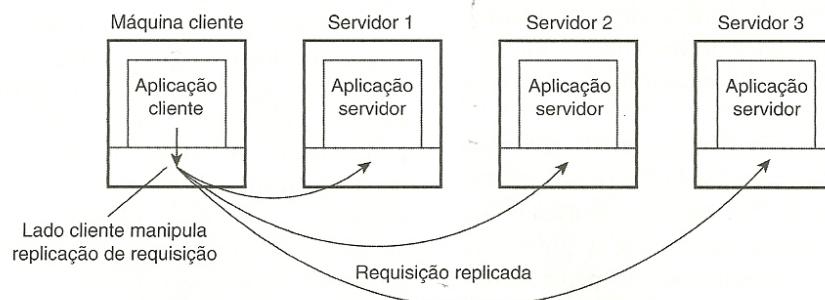


Figura 3.9 Replicação transparente de um servidor usando uma solução do lado cliente.

Uma outra questão é onde os clientes contatam um servidor. Em todos os casos, clientes enviam requisições a um **terminal**, também denominado **porta**, na máquina em que o servidor está executando. Cada servidor ouve uma porta específica. Como os clientes sabem qual é a porta de um serviço? Uma abordagem é designar globalmente portas para serviços bem conhecidos. Por exemplo, servidores que manipulam requisições FTP de Internet sempre ouvem a porta TCP 21. Da mesma maneira, um servidor HTTP para a World Wide Web sempre ouvirá a porta TCP 80.

Essas portas foram designadas pela Autoridade para Atribuição de Números na Internet (Internet Assigned Numbers Authority — Iana) e estão documentadas em Reynolds e Postel (1994). Com portas designadas, o cliente só precisa achar o endereço de rede da máquina em que o servidor está executando. Como explicaremos no próximo capítulo, serviços de nomes podem ser usados para essa finalidade.

Há muitos serviços que não requerem uma porta pré-determinada. Por exemplo, um servidor que informa a hora pode usar uma porta que lhe é dinamicamente designada por seu sistema operacional local. Nesse caso, em primeiro lugar, um cliente tem de consultar a porta. Uma solução é ter um daemon especial que execute em cada máquina que rodar servidores. O daemon monitora a porta corrente de cada serviço implementado por um servidor co-localizado. O próprio daemon ouve uma porta bem conhecida. Um cliente primeiro contatará o daemon, requisitará a porta e então contatará o servidor específico, como mostra a Figura 3.10(a).

É comum associar uma porta com um serviço específico. Contudo, a implementação propriamente dita de cada serviço por meio de um servidor separado pode ser um desperdício de recursos. Por exemplo, em um sistema Unix típico, é comum ter uma grande quantidade de servidores que execute simultaneamente, com a maioria deles esperando passivamente até chegar uma requisição de cliente. Em vez de ter de monitorar uma número tão grande de processos passivos, muitas vezes é mais eficiente ter um único **superservidor** à escuta em cada porta associada com um serviço específico, como mostra a Figura 3.10(b). Essa é a abordagem adotada, por exemplo, para o daemon *inetd* em Unix. O *inetd* ouve uma quantidade de portas bem conhecidas para serviços de Internet. Quando chega uma requisição, o daemon bifurca um processo para continuar a cuidar da requisição. Esse processo sairá após ter concluído.

Uma outra questão que precisa ser levada em conta ao elaborar o projeto de um servidor é se, e como, um servidor pode ser interrompido. Por exemplo, considere um usuário que decidiu transferir um enorme arquivo para um servidor FTP. De repente, entretanto, ao perceber que é o arquivo errado, ele quer interromper o servidor para cancelar a continuação da transmissão de dados.

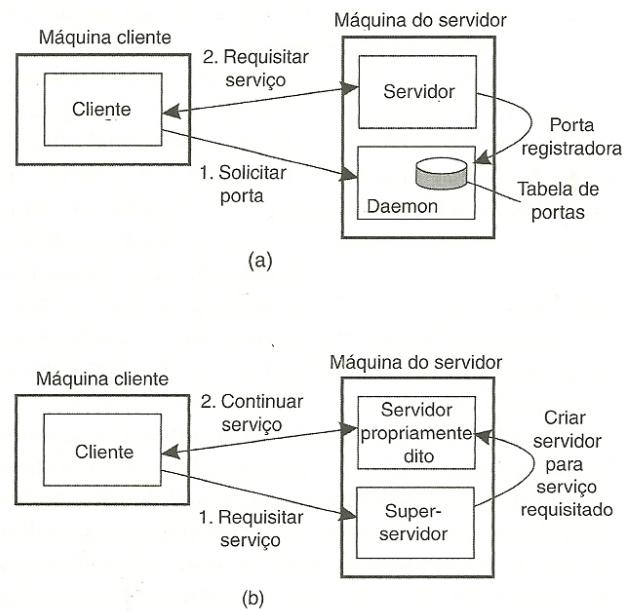


Figura 3.10 (a) Vinculação cliente-a-servidor usando um daemon. (b) Vinculação cliente-a-servidor usando um superservidor.

Há vários modos de fazer isso. Uma abordagem que funciona muitíssimo bem na Internet atual (e às vezes é a única alternativa) é o usuário sair abruptamente da aplicação cliente, o que automaticamente interromperá a conexão com o servidor, reiniciá-la imediatamente e fingir que nada aconteceu. A uma certa altura, o servidor encerrará a conexão antiga, entendendo que o cliente provavelmente falhou.

Uma alternativa muito melhor para manipular interrupções de comunicação é desenvolver o cliente e o servidor de modo tal que seja possível enviar dados **fora da banda**, ou seja, dados que venham a ser processados pelo servidor antes de quaisquer outros dados daquele cliente. Uma solução é deixar o servidor ouvir uma porta de controle separada para a qual o cliente envia dados fora da banda, enquanto, ao mesmo tempo, ouve (com menor prioridade) a porta pela qual passam os dados normais. Uma outra solução é enviar dados fora da banda pela mesma conexão através da qual o cliente está enviando a requisição original. Em TCP, por exemplo, é possível transmitir dados urgentes. Quando dados urgentes são recebidos no servidor, ele é interrompido — por exemplo, por meio de um sinal em sistemas Unix —, após o que pode inspecionar os dados e manipulá-los de acordo.

Uma última questão importante de projeto é se o servidor é sem estado ou não. Um **servidor sem estado** não mantém informações sobre o estado de seus clientes e pode mudar seu próprio estado sem ter de informar a nenhum cliente (Birman, 2005). Um servidor Web, por exemplo, é sem estado. Ele se limita a responder a requisições HTTP que entram, que podem ser para transferir um arquivo para o servidor ou, com mais freqüência, para buscar um arquivo. Após a requisição ser processada, o servidor Web esquece o cliente completamente. Da

mesma maneira, um conjunto de arquivos que um servidor Web gerencia (possivelmente em cooperação com um servidor de arquivos) pode ser mudado sem que os clientes tenham de ser informados.

Observe que, em muitos projetos sem estado, na verdade o servidor mantém informações sobre seus clientes, mas o fato crucial é que, se essas informações forem perdidas, isso não resultará na disruptão do serviço oferecido pelo servidor. Por exemplo, um servidor Web geralmente registra todas as requisições de clientes. Essa informação é útil para decidir se certos documentos devem ser replicados e para onde. Fica claro que não há nenhuma penalidade, exceto, talvez, sob a forma de desempenho abaixo do ótimo, se o registro for perdido.

Um modo particular de um projeto sem estado é aquele em que o servidor mantém o que é conhecido como **estado flexível (soft state)**. Nesse caso, o servidor promete manter estado em nome do cliente, mas apenas por tempo limitado. Após a expiração desse tempo, o servidor volta ao comportamento padrão (default) e, ao fazer isso, descarta quaisquer informações que guardava em nome do cliente associado. Um exemplo desse tipo de estado é um servidor que promete manter um cliente informado sobre atualizações, mas apenas por tempo limitado. Depois disso, o cliente deve selecionar o servidor se quiser atualizações. Abordagens de estado flexível se originam do projeto de protocolo em redes de computadores, mas podem ser igualmente aplicadas ao projeto de servidores (Clark, 1989; Lui et al., 2004).

Ao contrário, um **servidor com estado** em geral mantém informações persistentes sobre seus clientes. Isso significa que as informações precisam ser explicitamente removidas pelo servidor. Um exemplo típico é um servidor de arquivos que permite a um cliente manter cópia local de um arquivo, mesmo após ter realizado operações de atualização. Tal servidor manteria uma tabela que contivesse entradas (*cliente, arquivo*). Essa tabela permite que o servidor monitore qual cliente tem as permissões de atualização em qual arquivo no momento em questão, e assim, possivelmente, também a versão mais recente daquele arquivo.

Essa abordagem pode melhorar o desempenho percebido pelo cliente de operações de leitura e escrita. Melhorar desempenho em servidores sem estado costuma ser um benefício importante de projetos com estado. Contudo, o exemplo também ilustra a grande desvantagem de servidores com estado. Se falhar, o servidor tem de recuperar sua tabela de entradas (*cliente, arquivo*); caso contrário, não poderá garantir que processou as mais recentes atualizações em um arquivo.

Em geral, um servidor com estado precisa recuperar todo o seu estado, tal como era um pouco antes da falha. Como discutiremos no Capítulo 8, possibilitar recuperação pode apresentar considerável complexidade. Em um projeto sem estado, não é preciso tomar absolutamente

nenhuma providência especial para que um servidor que falhou se recupere. Ele simplesmente começa a funcionar novamente e espera pela entrada de requisições de cliente.

Ling et al. (2004) argumentam que, na verdade, deveríamos fazer uma distinção entre **estado de sessão** (temporário) e estado permanente. O exemplo anterior é típico para estado de sessão: ele está associado com uma série de operações por um único usuário e deve ser mantido por algum tempo, mas não indefinidamente. Ocorre que o estado de sessão é freqüentemente mantido em arquiteturas cliente–servidor de três camadas, nas quais o servidor de aplicação realmente precisa acessar um servidor de banco de dados por meio de uma série de consultas antes de poder responder ao cliente requisitante. Aqui, a questão é que nenhum dano é causado se o estado de sessão for perdido, contanto que o cliente possa simplesmente emitir novamente a requisição original. Essa observação permite armazenamento de estado mais simples e menos confiável.

O que resta para estado permanente normalmente são informações mantidas em bancos de dados, como informações de clientes, chaves associadas com software comprado e assim por diante. Contudo, para a maioria dos sistemas distribuídos, manter estado de sessão já implica um projeto com estado que requer medidas especiais quando acontecerem falhas e adotar premissas explícitas sobre a durabilidade de estado armazenada no servidor. Voltaremos a esses assuntos de maneira mais abrangente ao discutirmos tolerância à falha.

Ao elaborar o projeto de um servidor, a opção por um projeto com estado ou sem estado não deve afetar os serviços oferecidos pelo servidor. Por exemplo, se arquivos têm de ser abertos antes de lidos, ou escritos, um servidor sem estado deve imitar esse procedimento, de um jeito ou de outro. Uma solução comum, que discutiremos com mais detalhes no Capítulo 11, é que o servidor responde a uma requisição de escrita ou leitura primeiro abrindo o arquivo referido e em seguida realizando a operação de leitura ou escrita propriamente dita, para depois fechar imediatamente o arquivo de novo.

Em outros casos, um servidor pode querer manter um registro do comportamento de um cliente, de modo que possa atender mais efetivamente às suas requisições. Por exemplo, servidores Web às vezes oferecem a possibilidade de dirigir um cliente imediatamente a suas páginas favoritas. Essa abordagem só é possível se o servidor tiver um histórico de informações sobre esse cliente. Quando o servidor não puder manter estado, uma solução comum é deixar o cliente enviar informações sobre seus acessos anteriores. No caso da Web, essas informações costumam ser armazenadas de modo transparente pelo browser do cliente no que é chamado **cookie**: pequena porção de dados que contém informações específicas do cliente que interessam ao servidor. Cookies nunca são executados por um browser; apenas são armazenados.

Na primeira vez que um cliente acessa um servidor, este manda um cookie junto com as páginas Web requisitadas de volta ao browser, após o que o browser guarda o cookie em segurança. Depois disso, cada vez que o cliente acessar o servidor, seu cookie para aquele servidor é enviado junto com a requisição. Embora em princípio essa abordagem funcione bem, o fato de que cookies são enviados do servidor para serem guardados em segurança pelo browser costuma ficar inteiramente oculto aos usuários. Lá se vai a privacidade! Diferente da maioria dos cookies (biscoitos) da vovó, esses cookies deveriam ficar onde foram produzidos.*

3.4.2 Clusters de servidores

No Capítulo 1 discutimos brevemente cluster de computadores como uma das muitas formas de apresentação de sistemas distribuídos. Agora vamos examinar mais de perto a organização de clusters de servidores com questões relevantes de projeto.

Organização geral

Em palavras simples, um cluster de servidores nada mais é do que um conjunto de máquinas conectadas por uma rede, no qual cada máquina executa um ou mais servidores. Os clusters de servidores que consideraremos aqui são aqueles nos quais as máquinas estão conectadas por uma rede local, que muitas vezes oferece alta largura de banda e baixa latência.

Na maioria dos casos, um cluster de servidores é organizado logicamente em três camadas, como mostra a Figura 3.11. A primeira camada consiste em um comutador (lógico) por meio do qual são roteadas as requisições de clientes. Esse comutador pode variar muito. Por exemplo, comutadores de camada de transporte aceitam requisições de conexão TCP e passam requisições para um dos servidores no cluster, como discutiremos a seguir. Um exemplo completamente diferente é um servidor Web que

aceita requisições HTTP, mas passa parte dessas requisições a servidores de aplicação para posterior processamento somente para mais tarde colher os resultados e retornar uma resposta HTTP.

Como em qualquer arquitetura cliente-servidor multicamadas, muitos clusters de servidores também contêm servidores dedicados a processamento de aplicação. Em clusters de computadores, normalmente eles são servidores que executam em hardware de alto desempenho dedicado a aumentar a capacidade de computação. Todavia, no caso de clusters corporativos, pode ser que as aplicações só precisem rodar em máquinas de tecnologia relativamente baixa porque a capacidade de computação requerida não é o gargalo, mas o acesso ao armazenamento.

Isso nos leva à terceira camada, que consiste em servidores de processamento de dados, especialmente servidores de arquivo e bancos de dados. Mais uma vez, dependendo da utilização do cluster de servidores, esses servidores podem executar em máquinas especializadas, configuradas para acesso de alta velocidade a disco e que têm grandes caches de dados do lado servidor.

Claro que nem todos os clusters de servidores seguirão essa separação estrita. Invariavelmente ocorre que cada máquina está equipada com seu próprio armazenamento local, que muitas vezes integra processamento de aplicação e de dados em um único servidor, o que resulta em uma arquitetura de duas camadas. Por exemplo, quando se trata de fluxos de mídia por meio de um cluster de servidores, é comum disponibilizar uma arquitetura de sistema de duas camadas, na qual cada máquina age como um servidor de mídia dedicado (Steinmetz e Nahrstedt, 2004).

Quando um cluster de servidores oferece vários serviços, pode acontecer que máquinas diferentes executem diferentes servidores de aplicação. Por consequência, o comutador terá de ser capaz de distinguir serviços, senão não poderá repassar requisições para as máquinas adequadas. Acontece que muitas máquinas de segunda camada executam apenas uma única aplicação. Essa

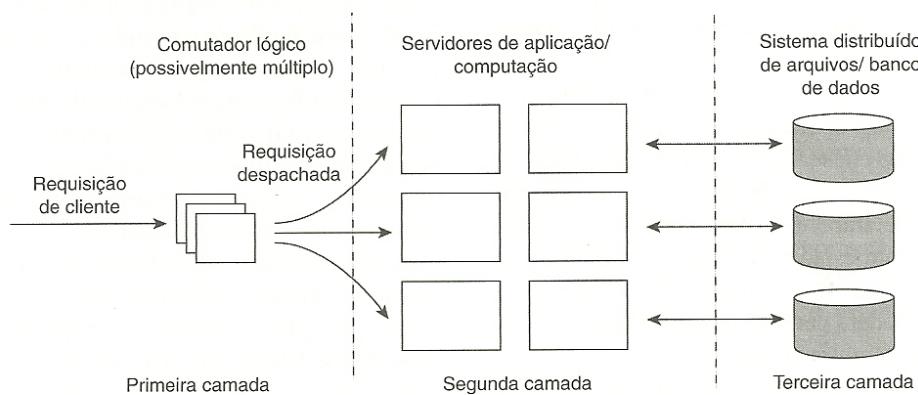


Figura 3.11 Organização geral de um cluster de servidores de três camadas.

* O autor se exprime aqui ironicamente, mas de modo altamente polêmico. A supressão dos cookies iria desabilitar a maioria dos serviços que a Web oferece hoje (N. do R.T.).

limitação vem da dependência de software e hardware disponíveis, mas também do fato de que aplicações diferentes muitas vezes são gerenciadas por administradores diferentes, e estes não gostam de interferir com as máquinas de outros.

Por consequência, podemos descobrir que certas máquinas estão temporariamente ociosas, enquanto outras estão recebendo uma sobrecarga de requisições. Algo útil seria migrar serviços temporariamente para máquinas ociosas. Uma solução proposta em Awadallah e Rosenblum (2004) é usar máquinas virtuais que permitam migração relativamente fácil de código para máquinas reais. Voltaremos à migração de código mais adiante neste capítulo.

Vamos examinar mais de perto a primeira camada, que consiste no comutador. Uma meta importante do projeto de clusters de servidores é ocultar o fato de que há vários servidores. Em outras palavras, aplicações clientes que executam em máquinas remotas nada precisariam saber sobre a organização interna do cluster. Essa transparência de acesso é invariavelmente oferecida por meio de um único ponto de acesso, por sua vez implementado por intermédio de algum tipo de comutador em hardware, tal como uma máquina dedicada.

O comutador forma o ponto de entrada para o cluster de servidores, oferecendo um único endereço de rede. Por questão de escalabilidade e disponibilidade, um cluster de servidores pode ter vários pontos de acesso e, portanto, cada ponto de acesso é realizado por uma máquina dedicada separada. Consideraremos apenas o caso de um único ponto de acesso.

Um modo padrão de acessar um cluster de servidores é estabelecer uma conexão TCP pela qual requisições de nível de aplicação sejam enviadas como parte de uma sessão. Uma sessão termina com o encerramento da conexão. No caso de **comutadores de camada de transporte**, o comutador aceita requisições de conexão TCP que entram e transfere tais conexões a um dos servidores (Hunt et al., 1997; Pai et al., 1998). O princípio de funcionamento do que é comumente conhecido como **transferência TCP (TCP handoff)** é mostrado na Figura 3.12.

Quando recebe uma requisição de conexão TCP, o comutador identifica o melhor servidor para manipular aquela requisição e repassa o pacote de requisição para aquele servidor. Por sua vez, o servidor enviará um reconhecimento de volta ao cliente requisitante, mas insere o endereço IP do comutador como o endereço de fonte no cabeçalho do pacote IP que está transportando o segmento TCP. Note que essa falsificação (*spoofing*) é necessária para que o cliente continue executando o protocolo TCP: ele está esperando uma mensagem de volta do comutador, e não de algum servidor arbitrário do qual nunca ouviu falar. É claro que a implementação da transferência TCP requer modificações no nível do sistema operacional.

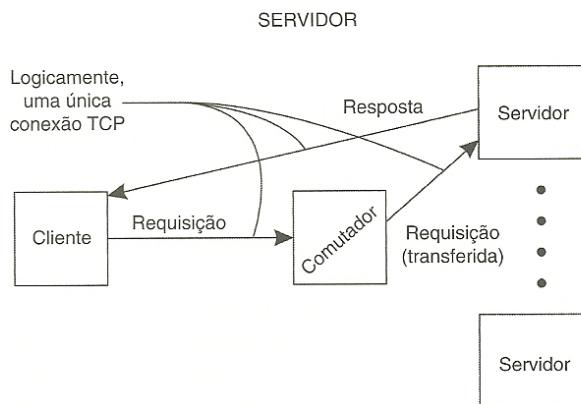


Figura 3.12 Princípio da transferência TCP.

Já podemos ver que o comutador pode desempenhar importante papel na distribuição da carga entre os vários servidores. Por decidir para onde repassar uma requisição, o comutador também decide qual servidor deve manipular o processamento ulterior da requisição. A política de balanceamento de carga mais simples que o comutador pode seguir é a alternância cíclica: cada vez ele escolhe o próximo servidor de sua lista para o qual repassará a requisição.

Critérios mais avançados para seleção de servidor também podem ser disponibilizados. Por exemplo, considere que vários serviços sejam oferecidos pelo cluster de servidores. Se o comutador puder distinguir entre esses serviços quando uma requisição entrar, então pode tomar decisões conscientes sobre para onde repassar a requisição. Essa seleção de servidor pode ainda ocorrer no nível de transporte, contanto que os serviços sejam distinguidos por meio de um número de porta. Um passo adiante é fazer com que o comutador realmente inspecione a carga útil da requisição que está entrando. Esse método pode ser aplicado só quando se sabe qual pode ser a aparência dessa carga útil. Por exemplo, no caso de servidores Web, suponha que o comutador esteja esperando uma requisição HTTP; com base nisso, ele pode decidir quem deve processá-la. Voltaremos a essa **distribuição de requisição dependente de contexto** quando discutirmos sistemas baseados na Web no Capítulo 12.

Servidores distribuídos

Os clusters de servidores que discutimos até aqui em geral são configurados estaticamente, mas do que de qualquer outro modo. Nesses clusters, muitas vezes há uma máquina de administração separada que monitora servidores disponíveis e passa essa informação para outras máquinas conforme adequado, tal como o comutador.

Como mencionamos, a maioria dos clusters de servidores oferece um único ponto de acesso. Quando esse ponto falha, o cluster fica indisponível. Para eliminar esse problema potencial, vários pontos de acesso podem ser fornecidos, cujos endereços são disponibilizados publica-

mente. Por exemplo, o **Sistema de Nomes de Domínio (Domain Name System — DNS)** pode retornar vários endereços, todos pertencentes ao mesmo nome de hospedeiro. Essa abordagem ainda requer que os clientes façam diversas tentativas se um dos endereços falhar. Além do mais, isso não resolve o problema de requerer pontos de acesso estáticos.

Ter estabilidade, como um ponto de acesso de longa permanência, é um aspecto desejável do ponto de vista de um cliente e de um servidor. Por outro lado, também é desejável ter um alto grau de flexibilidade na configuração de um cluster de servidores, incluindo o comutador. Essa observação resultou no projeto de um **servidor distribuído** que nada mais é do que um conjunto de máquinas que possivelmente muda dinamicamente, com vários pontos de acesso também possivelmente variáveis, mas que, quanto ao mais, se apresenta ao mundo externo como uma única e poderosa máquina. O projeto de tal servidor distribuído é dado em Szymaniak et al. (2005). Aqui, nós apenas o descrevemos brevemente.

A idéia básica por trás de um servidor distribuído é que os clientes se beneficiem de um servidor robusto, estável e de alto desempenho. Essas propriedades muitas vezes podem ser fornecidas por mainframes de alta tecnologia, entre os quais alguns alardeiam um tempo médio entre falhas de mais de 40 anos. Contudo, agrupando máquinas mais simples transparentemente em um cluster e sem confiar na disponibilidade de uma única máquina, pode ser possível alcançar um grau melhor de estabilidade do que com cada componente individualmente. Por exemplo, um cluster como esse poderia ser configurado dinamicamente por máquinas de usuário final, como no caso de um sistema distribuído colaborativo.

Vamos nos concentrar em como se pode conseguir um ponto de acesso estável em tal sistema. A idéia principal é fazer uso de serviços de rede disponíveis, em especial suporte de mobilidade para IP versão 6 (MIPv6). Em MIPv6, considera-se que um nó móvel tem uma **rede nativa** em que ele normalmente reside e na qual tem um endereço estável associado, conhecido como seu **endereço nativo (Home Address — HoA)**. Essa rede nativa tem um repassador especial conectado a ela, conhecido como **agente nativo**, que tomará conta do tráfego para o nó móvel quando ele estiver fora.

Com essa finalidade, quando um nó móvel se liga a uma rede externa, ele receberá um **endereço externo (Care-of Address — COA)**, onde poderá ser alcançado. Esse endereço COA é informado ao agente nativo do nó, que então providencia para que todo o tráfego seja repassado para o nó móvel. Observe que aplicações que se comunicam com o nó móvel só verão o endereço associado com a rede nativa do nó. Elas nunca verão o endereço COA.

Esse princípio pode ser usado para oferecer um endereço estável de um servidor distribuído. Nesse caso,

um único **endereço de contato** é inicialmente designado ao cluster de servidores. O endereço de contato será o endereço do servidor durante sua vida útil e será usado em todas as comunicações com o mundo externo. A qualquer instante, um nó no servidor distribuído funcionará como um ponto de acesso usando aquele endereço de contato, mas esse papel pode ser assumido com facilidade por um outro nó. Ocorre que o ponto de acesso registra seu próprio endereço como o endereço COA no agente nativo associado com o servidor distribuído. Nessa circunstância, todo o tráfego será dirigido ao ponto de acesso, que então cuidará da distribuição de requisições entre os nós que estão participando naquele instante. Se o ponto de acesso falhar, entra em cena um simples mecanismo de cobertura de falha, pelo qual um outro ponto de acesso informa um novo endereço COA.

Essa configuração simples transformaria o agente nativo, bem como o ponto de acesso, em potenciais garrafais, porque todo o tráfego fluiria por essas duas máquinas. Essa situação pode ser evitada usando uma característica do MIPv6 conhecida como *otimização de rota*. Otimização de rota funciona da seguinte maneira: sempre que um nó móvel com endereço nativo *HA* informar seu endereço COA corrente, digamos, *CA*, o agente nativo pode repassar *CA* para um cliente. Então, este armazenará o par (*HA*, *CA*) no local. Desse momento em diante, a comunicação será repassada diretamente a *CA*. Embora a aplicação no lado cliente ainda possa usar o endereço nativo, o software básico subjacente para MIPv6 traduzirá aquele endereço para *CA* e o usará no lugar do outro.

Otimização de rota pode ser usada para fazer com que clientes diferentes acreditem que estão se comunicando com um único servidor quando, na verdade, cada cliente está se comunicando com um diferente nó membro do servidor distribuído, como mostra a Figura 3.13. Com essa finalidade, quando um ponto de acesso de um servidor distribuído repassa uma requisição do cliente *C₁* para, digamos, o nó *S₁* (com endereço *CA₁*), ele passa informações suficientes para *S₁* a fim de deixar que este inicie o procedimento de otimização de rota que faz o cliente acreditar que o COA é *CA₁*. Isso permitirá a *C₁* armazenar o par (*HA*, *CA₁*). Durante esse procedimento, o ponto de acesso (bem como o agente nativo) envia grande parte do tráfego entre *C₁* e *S₁* por um túnel. Isso impedirá que o agente nativo acredite que o COA mudou, de modo que ele continuará a se comunicar com o ponto de acesso.

Claro que enquanto esse procedimento de otimização de rota estiver ocorrendo, requisições de outros clientes continuarão a chegar. Elas permanecerão em estado pendente no ponto de acesso até que possam ser repassadas. Portanto, a requisição de um outro cliente *C₂* pode ser repassada ao nó membro *S₂* (com endereço *CA₂*), permitindo que esse último deixe o cliente *C₂* armazenar o par (*HA*, *CA₂*). O resultado é que diferentes clientes estarão se comunicando diretamente com membros diferentes do

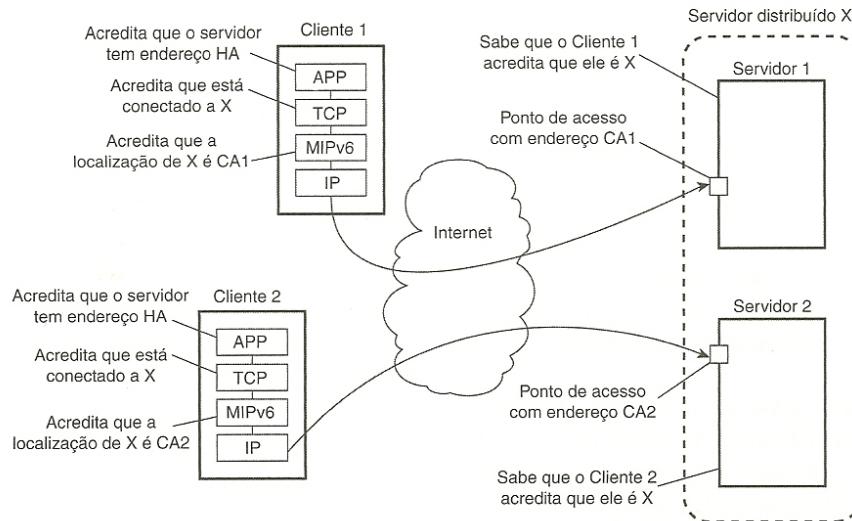


Figura 3.13 Otimização de rota em um servidor distribuído.

servidor distribuído, no qual cada aplicação cliente ainda tem a ilusão de que esse servidor tenha endereço *HA*. O agente nativo continua a se comunicar com o ponto de acesso conversando com o endereço de contato.

3.4.3 Gerenciamento de clusters de servidores

Um cluster de servidores deve se apresentar ao mundo exterior como um único computador, como muitas vezes é, de fato, o caso. Contudo, quando se trata de gerenciar um cluster, a situação muda drasticamente. Várias tentativas foram feitas para facilitar o gerenciamento de clusters de servidores, como discutiremos a seguir.

Abordagens comuns

De longe, a abordagem mais comum para o gerenciamento de um cluster de servidores é estender as tradicionais funções de gerenciamento de um único computador para um cluster. Em sua forma mais primitiva, isso significa que um administrador pode se registrar em um nó com base em um cliente remoto e executar comandos locais de gerenciamento para monitorar, instalar e trocar componentes.

Um pouco mais avançado é ocultar o fato de que você precisa se registrar (login) em um nó e, em vez disso, fornecer uma interface em uma máquina administradora que permita colher informações de um ou mais servidores, atualizar componentes, adicionar e remover nós etc. A principal vantagem da última abordagem é que operações coletivas, que operam sobre um grupo de servidores, podem ser fornecidas com mais facilidade. Esse tipo de gerenciamento de clusters de servidores tem ampla aplicação na prática, exemplificada por software de gerenciamento como o Cluster Systems Management da IBM (Hochstetler e Beringer, 2004).

Contudo, tão logo os clusters cresçam e passem de várias dezenas de nós, esse tipo de gerenciamento não

serve mais. Muitas centrais de dados precisam gerenciar milhares de servidores organizados em muitos clusters, mas todos funcionando de modo colaborativo. Fazer isso por meio de servidores de administração centralizados está simplesmente fora de questão. Além do mais, é fácil de ver que clusters muito grandes precisam de gerenciamento contínuo de falhas (incluindo atualizações). Para simplificar as coisas, se p é a probabilidade de um servidor estar correntemente avariado e admitirmos que falhas são independentes, então, para que um cluster de N servidores opere sem que nenhum servidor esteja avariado, a probabilidade é $(1 - p)^N$. Portanto, com $p = 0,001$ e $N = 1.000$, há somente 36 por cento de probabilidade de que todos os servidores estejam funcionando corretamente.

Ocorre que suporte para clusters de servidores muito grandes é quase sempre ad hoc. Há várias regras práticas que devem ser consideradas (Brewer, 2001), mas não há nenhuma abordagem sistemática para lidar com o gerenciamento de sistemas maciços. O gerenciamento de cluster ainda está em sua infância, embora seja de esperar que chegará a hora em que soluções de autogerenciamento como as discutidas no capítulo anterior avançarão nessa área, após ganharmos mais experiência com elas.

Exemplo: PlanetLab

Agora vamos examinar mais de perto um comportamento de cluster um tanto fora do comum. PlanetLab é um sistema distribuído colaborativo no qual diferentes organizações doam um ou mais computadores, somando um total de até centenas de nós. Juntos, esses computadores formam um cluster de servidores de uma camada, no qual acesso, processamento e armazenamento podem ocorrer em cada nó individualmente. O gerenciamento do PlanetLab é, por necessidade, quase inteiramente distribuído. Antes de explicar seus princípios básicos, primeiro

vamos descrever os principais aspectos arquitetônicos (Peterson et al., 2005).

Em PlanetLab, uma organização doa um ou mais nós, e nesse sistema é mais fácil imaginar cada nó como apenas um único computador, embora também ele possa ser um cluster de máquinas. Cada nó é organizado como mostra a Figura 3.14. Há dois componentes importantes (Bavier et al., 2004). O primeiro é o monitor de máquina virtual (VMM), que é um sistema operacional Linux aprimorado. Os aperfeiçoamentos compreendem principalmente ajustes para suportar o segundo componente, denominado **vservers**.

Um vserver (Linux) pode ser mais bem imaginado como um ambiente separado no qual executa um grupo de processos. Processos de diferentes vservers são *completamente* independentes. Eles não podem compartilhar diretamente nenhum recurso como arquivos, memória principal e conexões de rede, como normalmente acontece com processos que executam sobre sistemas operacionais. Em vez disso, um vserver proporciona um ambiente que consiste em seu próprio conjunto de pacotes de software, programas e facilidades de rede. Por exemplo, um vserver pode fornecer um ambiente no qual um processo notará que pode utilizar o Python 1.5.2 combinado com um servidor Web Apache mais antigo, digamos, *httpd 1.3.1*. Ao contrário, um outro vserver pode suportar as versões mais recentes de Python e *httpd*. Nesse sentido, chamar um vserver de ‘servidor’ é um pouco incongruente porque, na realidade, ele apenas isola grupos de processos um do outro. Mais adiante voltaremos brevemente aos vservers.

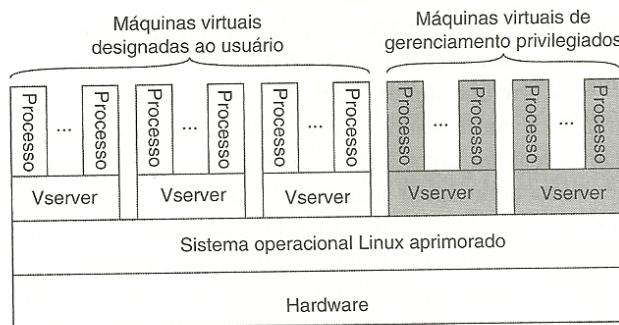


Figura 3.14 Organização básica de um nó PlanetLab.

O VMM Linux assegura que os vservers estão separados: processos em vservers diferentes são executados concorrente e independentemente, cada um utilizando somente os pacotes de software e programas disponíveis em seu próprio ambiente. O isolamento entre processos em vservers diferentes é estrito. Por exemplo, dois processos em vservers diferentes podem ter o mesmo ID de usuário, mas isso não implica que eles se originaram do mesmo usuário. Essa separação facilita consideravelmente o suporte de usuários de diferentes organizações que querem usar o PlanetLab: por exemplo, uma plataforma

de teste para fazer experimentos com sistemas distribuídos e aplicações completamente diferentes.

Para suportar tais experimentos, o PlanetLab introduz a noção de **fatia**, que é um conjunto de vservers no qual cada vserver executa em um nó diferente. Assim, pode-se imaginar uma fatia como um cluster virtual de servidores, implementado por meio de um conjunto de máquinas virtuais. As máquinas virtuais em PlanetLab executam sobre o sistema operacional Linux, que foi estendido com uma quantidade adicional de módulos do núcleo.

Há várias questões que fazem do gerenciamento do PlanetLab um problema especial. Três evidentes são:

1. Nós pertencem a diferentes organizações. Cada nó deve ter capacidade de especificar quem pode ter permissão de executar aplicações em seus nós e restringir adequadamente a utilização de recursos.
2. Há várias ferramentas de monitoração disponíveis, mas todas elas admitem uma combinação muito específica de hardware e software. Além do mais, todas são projetadas para serem usadas dentro de uma única organização.
3. Programas de fatias diferentes, mas que executam no mesmo nó, não devem interferir um com o outro. Esse problema é similar à independência de processo em sistemas operacionais.

Vamos examinar cada uma dessas questões com mais detalhes.

Uma entidade central para o gerenciamento de recursos do PlanetLab é o **gerente de nó**. Cada nó tem um desses gerentes, implementado por meio de um vserver separado, cuja única tarefa é criar outros vservers no nó que gerencia e controlar a alocação de recursos. O gerente de nó não toma nenhuma decisão de regulação; ele é um mero mecanismo que provê os ingredientes essenciais para fazer com que um programa execute em determinado nó.

A monitoração de recursos é feita por meio de uma especificação de recurso ou, abreviadamente, *rspec*. Uma *rspec* especifica um intervalo de tempo durante o qual certos recursos foram alocados. Entre esses recursos estão espaço em disco, descritores de arquivo, largura de banda de entrada e saída de rede, terminais de nível de transporte, memória principal e utilização de CPU. Uma *rspec* é identificada por meio de um identificador de 128 bits globalmente exclusivo, conhecido como capacidade de recurso (*rcap*). Dada uma *rcap*, o gerente do nó pode consultar a *rspec* associada em uma tabela local.

Recursos são vinculados a fatias. Em outras palavras, para utilizar recursos, é necessário criar uma fatia. Cada fatia é associada a um **provedor de serviços**, que pode ser mais bem imaginado como uma entidade que tem uma conta corrente no PlanetLab. Sendo assim, toda fatia pode ser identificada por um par (*principal_id*, *slice_tag*), no qual *principal_id* identifica o provedor, e *slice_tag* é um identificador escolhido pelo provedor.

Para criar uma nova fatia, cada nó executará um **serviço de criação de fatia** (*Slice Creation Service — SCS*) que, por sua vez, pode contatar o gerente de nó e requisitar que ele crie um vserver e aloque recursos. O gerente de nó, em si, não pode ser contatado diretamente por uma rede, o que permite que ele se concentre somente no gerenciamento de recursos locais. Por sua vez, o SCS não aceitará requisições de criação de fatia de qualquer um. Somente **autoridades de fatia** específicas são qualificadas para requisitar a criação de uma fatia. Cada autoridade de fatia terá direitos de acesso a um conjunto de nós. O modelo mais simples é aquele em que há somente uma única autoridade de fatia com permissão para requisitar criação de fatia em todos os nós.

Para completar o quadro, um provedor de serviços contatará uma autoridade de fatia e requisitará que ela crie uma fatia que abranja um conjunto de nós. A autoridade de fatia saberá quem é o provedor de serviços porque ele já tinha sido anteriormente autenticado e subsequentemente registrado como um usuário PlanetLab. Na prática, usuários do PlanetLab contatam uma autoridade de fatia por meio de um serviço baseado na Web. Mais detalhes podem ser encontrados em Chun e Spalink (2003).

Esse procedimento revela que o gerenciamento do PlanetLab é feito por meio de intermediários. Uma importante classe desses intermediários é formada por autoridades de fatia. Tais autoridades obtiveram credenciais em nós para criar fatias. A obtenção dessas credenciais foi realizada fora de banda, em essência, contatando administradores de sistemas em vários sites. É óbvio que esse é um processo demorado que não deve ser executado por usuários finais ou, na terminologia do PlanetLab, por provedores de serviços.

Além das autoridades de fatia, há também autoridades de gerenciamento. Enquanto uma autoridade de fatia se concentra somente no gerenciamento de fatias, uma autoridade de gerenciamento é responsável por vigiar os nós. Em particular, ela assegura que os nós sob seu regime executem o software básico do PlanetLab e obegeçam às regras estabelecidas pelo PlanetLab. Provedores de serviços confiam que uma autoridade de gerenciamento fornece nós que se comportarão adequadamente.

Essa organização resulta na estrutura de gerenciamento mostrada na Figura 3.15, descrita em termos de

relações de confiança em Peterson et al. (2005). As relações são as seguintes:

1. Um proprietário de nó coloca seu nó sob o regime de uma autoridade de gerenciamento, possivelmente restringindo utilização quando adequado.
2. Uma autoridade de gerenciamento fornece o software necessário para adicionar um nó ao PlanetLab.
3. Um provedor de serviços se registra junto a uma autoridade de gerenciamento, confiando que ela forneça nós que se comportem bem.
4. Um provedor de serviços contata uma autoridade de fatia para criar uma fatia em um conjunto de nós.
5. A autoridade de fatia precisa autenticar o provedor de serviços.
6. Um proprietário de nó fornece um serviço de criação de fatia para uma autoridade de fatia criar fatias. Em essência, ele delega o gerenciamento de recursos à autoridade de fatia.
7. Uma autoridade de gerenciamento delega a criação de fatias a uma autoridade de fatia.

Esses relacionamentos abrangem o problema da delegação de nós de modo controlado, de maneira que um proprietário de nó possa confiar em um gerenciamento decente e seguro. A segunda questão que precisa ser tratada é a monitoração. É necessária uma abordagem unificada que permita aos usuários verem como seus programas estão se comportando dentro de uma fatia específica.

O PlanetLab segue uma abordagem simples. Todo nó é equipado com um conjunto de sensores, cada um capaz de dar informações como utilização de CPU, atividade de disco e assim por diante. Sensores podem ser arbitrariamente complexos, mas a questão importante é que eles sempre dão informações por nó. Essa informação é disponibilizada por meio de um servidor Web: todo sensor é acessível por meio de simples requisições HTTP (Bavier et al., 2004).

Considera-se que essa abordagem de monitoração ainda é bastante primitiva, mas é preciso vê-la como base para esquemas avançados de monitoração. Por exemplo, em princípio, não há nenhuma razão por que o Astrolabe, que discutimos no Capítulo 2, não possa ser usado para leituras agregadas de sensores em vários nós.

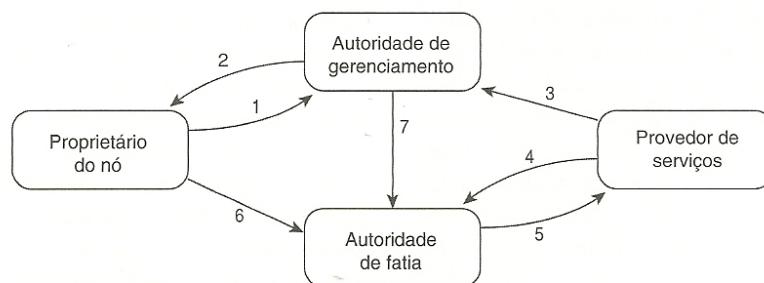


Figura 3.15 Relações de gerenciamento entre várias entidades PlanetLab.

Por fim, para chegar à última questão de gerenciamento, a proteção mútua de programas, o PlanetLab usa servidores virtuais Linux (denominados vservers) para isolar fatias. Como mencionado, a idéia principal de um vserver é executar aplicações em seus próprios ambientes, o que inclui todos os arquivos que são normalmente compartilhados em uma única máquina. Tal separação pode ser conseguida com relativa facilidade por meio de um comando Unix chroot, que efetivamente muda a raiz do sistema de arquivo em que as aplicações procurarão arquivos. Somente o superusuário pode executar chroot.

Claro que é preciso mais. Servidores virtuais Linux não somente separam o sistema de arquivo, mas normalmente também as informações compartilhadas sobre processos, endereços de rede, utilização de memória e assim por diante. Por consequência, uma máquina física é, na verdade, repartida em várias unidades, cada unidade correspondente a um ambiente Linux totalmente desenvolvido e isolado das outras partes. Uma visão geral de servidores virtuais Linux pode ser encontrada em Potzl et al. (2005).

3.5 Migração de Código

Até aqui, nossa principal preocupação foram sistemas distribuídos nos quais a comunicação é limitada à passagem de dados. Contudo, há situações em que passar programas, às vezes até mesmo enquanto estão sendo executados, simplifica o projeto de um sistema distribuído. Nesta seção, estudamos com detalhes o que é realmente migração de código. Começamos considerando diferentes abordagens da migração de código e em seguida discutimos como lidar com os recursos locais que um programa em migração usa. Um problema de particular dificuldade é migrar código em sistemas heterogêneos, o que também discutiremos.

3.5.1 Abordagens para migração de código

Antes de examinar as diferentes formas de migração de código, primeiro vamos considerar por que migrar código pode ser útil.

Razões para migrar código

A migração de código em sistemas distribuídos ocorria tradicionalmente na forma de **migração de processo**, na qual todo o processo era movido de uma máquina para outra (Milojicic et al., 2000). Mover um processo em execução para uma máquina diferente é uma tarefa custosa e complicada, e é bom que haja uma boa razão para fazer isso. Essa razão sempre foi o desempenho. A idéia básica é que o desempenho global do sistema pode ser melhorado se processos forem movidos de máquinas muito carregadas para máquinas com cargas mais leves. A carga costuma ser expressa em termos do comprimento da fila da

CPU ou da utilização da CPU, mas outros indicadores de desempenho também são usados.

Algoritmos de distribuição de carga pelos quais são tomadas decisões concernentes à alocação e redistribuição de tarefas com relação a um conjunto de processadores desempenham importante papel em sistemas de computação intensiva. Todavia, em muitos sistemas distribuídos modernos, otimizar capacidade de computação é uma questão menor do que, por exemplo, tentar minimizar comunicação. Além do mais, devido à heterogeneidade das plataformas e redes de computadores subjacentes, a melhoria do desempenho por meio de migração de código costuma ser baseada em raciocínio qualitativo em vez de em modelos matemáticos.

Considere, como exemplo, o sistema cliente–servidor no qual o servidor gerencia um enorme banco de dados. Se uma aplicação cliente precisar realizar muitas operações de banco de dados que envolvam grandes quantidades de dados, talvez seja melhor despachar parte da aplicação cliente para o servidor e enviar somente os resultados pela rede. Caso contrário, a rede pode ficar sobrecarregada com a transferência de dados do servidor para o cliente. Nesse caso, a migração de código é baseada na premissa de que, em geral, tem sentido processar dados perto de onde esses dados residem.

Essa mesma razão pode ser usada para migrar partes do servidor para o cliente. Por exemplo, em muitas aplicações interativas de banco de dados, clientes precisam preencher formulários que, na seqüência, são traduzidos em uma série de operações de banco de dados. Processar o formulário no lado cliente e enviar somente o formulário completo para o servidor às vezes pode evitar que um número relativamente grande de pequenas mensagens tenha de atravessar a rede. O resultado é que o cliente percebe melhor desempenho, enquanto, ao mesmo tempo, o servidor gasta menos tempo no processamento do formulário e na comunicação.

Suporte para migração de código também pode ajudar a melhorar o desempenho pela exploração do paralelismo, mas sem as usuais complexidades relacionadas à programação paralela. Um exemplo típico é procurar informações na Web. É relativamente simples implementar uma consulta de pesquisa sob a forma de um pequeno programa móvel denominado **agente móvel**, que passa de site para site. Fazendo várias cópias desse programa e enviando cada uma para sites diferentes, talvez seja possível conseguir um aumento linear de velocidade em comparação com utilizar apenas uma instância de programa único.

Além de melhorar o desempenho, também há outras razões para suportar migração de código. A mais importante é a flexibilidade. A abordagem tradicional para construir aplicações distribuídas é repartir a aplicação em porções diferentes e decidir antecipadamente onde cada porção deve ser executada. Essa abordagem resultou, por exemplo, em diferentes aplicações cliente–servidor multcamadas discutidas no Capítulo 2.

Contudo, se pudermos mover código entre máquinas diferentes, torna-se possível configurar dinamicamente sistemas distribuídos. Suponha que um servidor implemente uma interface padronizada para um sistema de arquivo. Para permitir que clientes remotos acessem o sistema de arquivo, o servidor usa um protocolo proprietário. Normalmente, a implementação do lado cliente da interface de sistema de arquivo, que é baseada naquele protocolo, precisaria ser ligada com a aplicação cliente. Essa abordagem requer que o software esteja facilmente disponível para o cliente no momento em que a aplicação cliente for se desenvolvendo.

Uma alternativa é deixar que o servidor forneça a implementação do cliente só quando for estritamente necessário, isto é, quando o cliente se vincular ao servidor. Nesse ponto, o cliente descarrega a implementação dinamicamente, percorre as etapas de inicialização necessárias e, na sequência, invoca o servidor. Esse princípio é mostrado na Figura 3.16. Esse modelo de movimentação dinâmica de código com base em um site remoto realmente requer que o protocolo para descarregar e inicializar código seja padronizado. Além disso, é necessário que o código descarregado possa ser executado na máquina cliente. Diferentes soluções são discutidas logo adiante e em capítulos posteriores.

Uma vantagem importante desse modelo de descarregar dinamicamente software do lado cliente é que os clientes não precisam ter todo o software instalado com antecedência para falar com servidores. Em vez disso, o software pode ser movido conforme necessário e, da mesma maneira, descartado quando não for mais necessário. Uma outra vantagem é que, contanto que as interfaces sejam padronizadas, podemos mudar o protocolo cliente-servidor e sua implementação quantas vezes quisermos. As mudanças não afetarão aplicações clientes existentes que dependam do servidor. É óbvio que também há desvantagens. A mais séria, que discutiremos no Capítulo 9, tem a ver com a segurança. Confiar cegamente que o código descarregado implementa somente a interface anunciada enquanto acessa seu disco rígido desprotegido, sem enviar as melhores partes para saber-se lá quem, nem sempre é uma boa idéia.

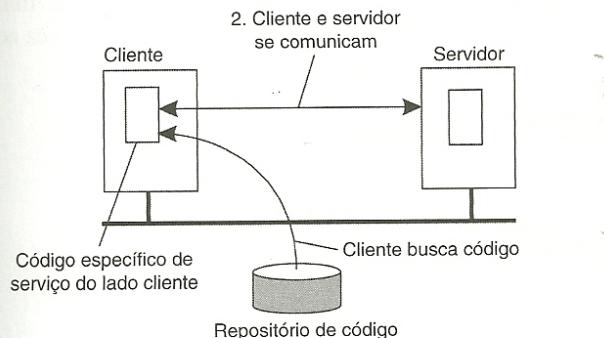


Figura 3.16 Princípio de configuração dinâmica de um cliente para se comunicar com um servidor. Primeiro o cliente busca o software necessário e então invoca o servidor.

Modelos para migração de código

Embora a migração de código sugira que movemos somente código entre máquinas, na verdade o termo abrange uma área muito mais rica. A comunicação em sistemas distribuídos se preocupa tradicionalmente com a troca de dados entre processos. No sentido mais amplo, migração de código trata da movimentação de programas entre máquinas com a intenção de executá-los na máquina-alvo. Em alguns casos, como em migração de processo, o status de execução de um programa, sinais pendentes e outras partes do ambiente também podem ser transferidos.

Para entender melhor os diferentes modelos para migração de código, usamos uma estrutura descrita em Fuggetta et al. (1998). Nessa estrutura, um processo consiste em três segmentos. O *segmento de código* é a parte que contém o conjunto de instruções que compõem o programa que está em execução. O *segmento de recursos* contém referências a recursos externos de que o processo necessita, tal como arquivos, impressoras, dispositivos, outros processos e assim por diante. Por fim, um *segmento de execução* é usado para armazenar o estado de execução de um processo no momento em questão, que consiste em dados privados, pilha e, é claro, o contador de programa.

O mínimo essencial para migração de código é fornecer **mobilidade fraca**. Nesse modelo, é possível transferir somente o segmento de código, talvez junto com alguns dados de inicialização. Um aspecto característico da mobilidade fraca é que um programa transferido é sempre iniciado de acordo com várias posições de partida predefinidas. Isso é o que acontece, por exemplo, com applets Java, que sempre iniciam execução do começo. O benefício dessa abordagem é sua simplicidade. Mobilidade fraca requer somente que a máquina-alvo possa executar aquele código, o que, em essência, se resume a tornar o código portável. Voltaremos a esses assuntos quando discutirmos migração em sistemas heterogêneos.

Ao contrário da mobilidade fraca, em sistemas que suportam **mobilidade forte** o segmento de execução também pode ser transferido. O aspecto característico da mobilidade forte é que um processo em execução pode ser parado e, na sequência, movido para uma outra máquina e então retomar a execução no ponto em que ele a deixou. Claro que mobilidade forte é muito mais geral do que mobilidade fraca, mas também muito mais difícil de implementar.

Independentemente de a mobilidade ser fraca ou forte, há uma outra distinção que pode ser feita entre migração iniciada pelo remetente e iniciada pelo destinatário. Em migração **iniciada pelo remetente**, a migração é iniciada na máquina em que o código está em execução no momento em questão. A migração iniciada pelo remetente normalmente é executada para transferir programas para um servidor de computação. Um outro exemplo é

enviar um programa de busca pela Internet a um servidor de banco de dados Web para realizar pesquisas naquele servidor. Em migração **iniciada pelo destinatário**, a iniciativa da migração de código é tomada pela máquina-alvo. Applets Java são um exemplo dessa abordagem.

Migração iniciada pelo destinatário é mais simples do que migração iniciada pelo remetente. Em muitos casos, a migração de código ocorre entre um cliente e um servidor, quando o cliente toma a iniciativa da migração. Transferir código com segurança para um servidor em migração iniciada pelo remetente muitas vezes requer que o cliente tenha sido previamente registrado e autenticado naquele servidor. Em outras palavras, o servidor tem de saber quem são todos os seus clientes e a razão para isso é que o cliente possivelmente requisitará acesso aos recursos do servidor, como seu disco. Proteger esses recursos é essencial.

Ao contrário, a transferência de código iniciada pelo destinatário muitas vezes pode ser feita no anonimato. Além do mais, em geral o servidor não está interessado nos recursos do cliente — a migração de código para o cliente é feita apenas para melhorar o desempenho do lado cliente. Com essa finalidade, só uma quantidade limitada de recursos precisa ser protegida, como memória e conexões de rede. Voltaremos à migração segura de código no Capítulo 9.

No caso de mobilidade fraca, também faz diferença se o código migrado é executado pelo processo-alvo ou se é iniciado um processo em separado. Por exemplo, applets Java são simplesmente descarregados por um browser Web e executados no espaço de endereço do browser. O benefício dessa abordagem é que não há nenhuma necessidade de iniciar um processo separado, evitando assim comunicação na máquina-alvo. A principal desvantagem é que o processo-alvo precisa ser protegido contra ataque malicioso ou execuções inadvertidas de código. Uma solução simples é deixar o sistema operacional cuidar disso criando um processo separado para executar o código migrado. Note que essa solução não resolve pro-

blemas de acesso a recursos já mencionados. Eles ainda terão de ser tratados.

Em vez de movimentar um processo em execução, operação também denominada *migração de processo*, a mobilidade forte também pode ser suportada por clonagem remota. Em contraste com a migração de processo, a clonagem produz uma cópia exata do processo original, mas agora executando em uma máquina diferente. O processo clonado é executado em paralelo com o processo original. Em sistemas Unix, a clonagem remota ocorre com a bifurcação do processo em um processo-filho e ao deixar que o filho continue em uma máquina remota. O benefício da clonagem é que o modelo é muito parecido com o que já é usado em muitas aplicações. A única diferença é que o processo clonado é executado em uma máquina diferente. Nesse sentido, a migração por clonagem é um modo simples de melhorar a transparência de distribuição.

As várias alternativas para migração de código são resumidas na Figura 3.17.

3.5.2 Migração e recursos locais

Até aqui, apenas a migração do código e do segmento de execução foi levada em conta. O segmento de recurso requer um pouco de atenção especial. O que muitas vezes torna a migração de código tão difícil é que o segmento de recurso nem sempre pode ser simplesmente transferido junto com outros segmentos sem ser trocado. Por exemplo, suponha que um processo detenha uma referência a uma porta TCP específica por meio da qual ele estava se comunicando com outros processos (remotos). Essa referência é mantida em seu segmento de recurso. Quando o processo passa para uma outra localização, terá de devolver a porta e requisitar uma nova no destino. Em outros casos, transferir uma referência não precisa ser um problema. Por exemplo, uma referência a um arquivo por meio de um URL absoluto permanecerá válida independentemente da máquina onde reside o processo que detém o URL.

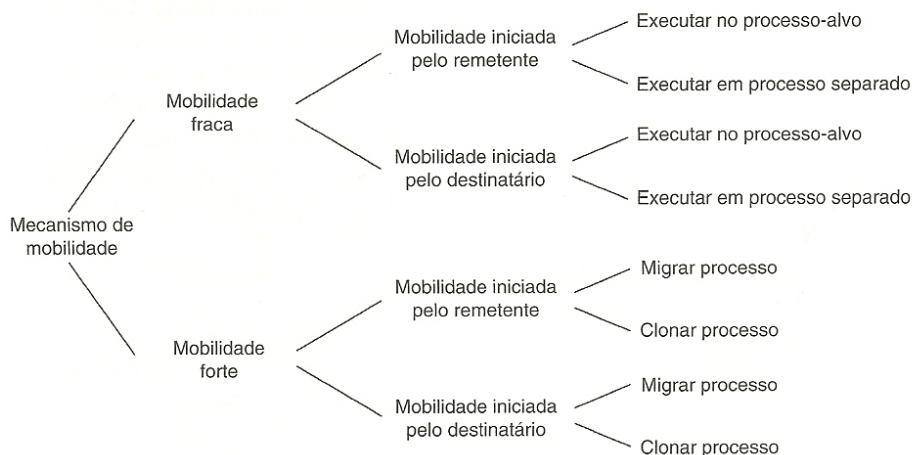


Figura 3.17 Alternativas para migração de código.

Para entender as implicações que a migração de código tem sobre o segmento de recurso, Fuggetta et al. (1998) distinguem três tipos de vinculações processo–recurso. A vinculação mais forte é quando um processo se refere a um recurso por seu identificador. Nesse caso, o processo requer exatamente o recurso referenciado e nada mais. Um exemplo dessa **vinculação por identificador** é o caso de um processo usar um URL para se referir a um site Web específico ou o caso de ele se referir a um servidor FTP por meio do seu endereço de Internet. Na mesma linha de raciocínio, referências a terminais locais de comunicação também resultam em uma vinculação por identificador.

Uma vinculação processo–recurso mais fraca ocorre quando só o valor de um recurso é necessário. Nesse caso, a execução do processo não seria afetada se um outro recurso fornecesse aquele mesmo valor. Um exemplo típico de **vinculação por valor** é o caso de um programa depender de bibliotecas padronizadas, como as de programação em C ou Java. Essas bibliotecas devem sempre estar disponíveis no local em questão, mas sua exata localização no sistema local de arquivo pode ser diferente entre sites. O importante para a adequada execução do processo não é o arquivo específico, é seu conteúdo.

Por fim, a forma de vinculação mais fraca de todas ocorre quando um processo indica que precisa de somente um recurso de um tipo específico. Essa **vinculação por tipo** é exemplificada por referências a dispositivos locais, como monitores, impressoras e assim por diante.

Quando migramos código, muitas vezes precisamos mudar as referências a recursos, mas não podemos afetar o tipo de vinculação processo–recurso. Se uma referência deve ser mudada, e exatamente como, depende de o recurso poder ser movido junto com o código para a máquina-alvo. Em termos mais específicos, precisamos considerar as vinculações recurso–máquina e distinguir os seguintes casos. **Recursos não ligados** podem ser movidos com facilidade entre máquinas diferentes e normalmente são arquivos (de dados) associados somente com o programa que deve ser migrado. Por comparação, mover ou copiar um **recurso amarrado** pode ser possível, mas só a custo relativamente alto. Exemplos típicos de recursos amarrados são bancos de dados locais e sites Web completos.

Embora, em teoria, esses recursos não sejam dependentes da máquina em que estão em determinado momento, muitas vezes é inviável movê-los para um outro ambiente. Por fim, **recursos fixos** estão intimamente vinculados a uma máquina ou ambiente específico, e não podem ser movidos. Recursos fixos freqüentemente são dispositivos locais. Um outro exemplo de um recurso fixo é uma porta de comunicação local.

A combinação desses três tipos de vinculação processo–recurso e três tipos de vinculação recurso–máquina resulta em nove combinações que precisamos considerar na migração de código. Essas nove combinações são mostradas na Tabela 3.2.

Em primeiro lugar vamos considerar as possibilidades quando um processo está vinculado a um recurso por identificador. Quando o recurso é não ligado, em geral é melhor movê-lo junto com o código que está migrando. Contudo, quando o recurso é compartilhado com outros processos, uma alternativa é estabelecer uma referência global, isto é, uma referência que possa atravessar as fronteiras das máquinas. Um exemplo de tal referência é um URL. Quando o recurso é amarrado ou fixo, a melhor solução é também criar uma referência global.

É importante perceber que estabelecer uma referência global pode ser mais do que apenas usar URLs e que o preço da utilização de tal referência às vezes é proibitivo. Considere, por exemplo, um programa que gera imagens de alta qualidade para uma estação de trabalho multimídia dedicada. Produzir imagens de alta qualidade em tempo real é uma tarefa que exige intensa computação, razão por que o programa talvez seja movido para um servidor de computação de alto desempenho. Estabelecer uma referência global à estação de trabalho multimídia significa estabelecer um caminho de comunicação entre o servidor de computação e a estação de trabalho. Ademais, há significativo processamento envolvido em ambos, servidor e estação de trabalho, para atender aos requisitos de largura de banda para a transferência de imagens. O resultado líquido talvez seja que mover o programa para o servidor de computação não é uma idéia tão boa assim, já que o custo da referência global é muito alto.

Um outro exemplo de que estabelecer uma referência global nem sempre é tão fácil ocorre no caso de se fazer

Vinculação recurso–máquina			
Vinculação processo–recurso	Não ligado	Amarrado	Fixo
Por identificador	MV (ou GR)	GR (ou MV)	GR
Por valor	CP (ou MV,GR)	GR (ou CP)	GR
Por tipo	RB (ou MV,CP)	RB (ou GR,CP)	RB (ou GR)

GR Estabelecer referência global no âmbito do sistema
 MV Mover o recurso
 CP Copiar o valor do recurso
 RB Vincular novamente o processo ao recurso disponível no local

Tabela 3.2 Ações a executar no que se refere às referências a recursos locais quando da migração de código para uma outra máquina.

migração de um processo que está usando uma porta de comunicação local. Nessa circunstância, estamos lidando com um recurso fixo ao qual o processo está vinculado pelo identificador. Há basicamente duas soluções. Uma é deixar que o processo estabeleça uma conexão com a máquina-fonte após ter migrado e instalar um processo separado na máquina-fonte que simplesmente repasse todas as mensagens que chegam. A principal desvantagem dessa abordagem é que, sempre que a máquina-fonte funcionar mal, a comunicação com o processo migrado falhará. A solução alternativa é fazer com que todos os processos que se comunicam com o processo que está migrando mudem suas referências globais e enviem mensagens à nova porta de comunicação na máquina-alvo.

A situação é diferente quando se trata de vinculações de valor. Considere, em primeiro lugar, um recurso fixo. A combinação de um recurso fixo e uma vinculação por valor ocorre, por exemplo, quando um processo admite que a memória pode ser compartilhada entre processos. Estabelecer uma referência global nesse caso significaria que precisamos implementar uma forma distribuída de memória compartilhada. Em muitos casos, essa não é realmente uma solução viável ou eficiente.

Recursos amarrados típicos que são referenciados por seu valor são bibliotecas de tempo de execução. Cópias de tais recursos normalmente estão disponíveis de imediato na máquina-alvo e, se não estiverem, devem ser copiadas antes de ocorrer a migração de código. Estabelecer uma referência global é uma alternativa melhor quando enormes quantidades de dados devem ser copiadas, como pode ser o caso de dicionários e tesouros em sistemas de processamento de textos.

O caso mais fácil é lidar com recursos não ligados. A melhor solução é copiar (ou mover) o recurso para o novo destino, a menos que ele seja compartilhado por vários processos. Nesse último caso, estabelecer uma referência global é a única opção.

O último caso trata de vinculações por tipo. Independentemente da vinculação recurso-máquina, a solução óbvia é vincular novamente o processo a um recurso do mesmo tipo disponível no local. Só quando tal recurso não estiver disponível é que precisaremos copiar ou mover o original para o novo destino ou estabelecer uma referência global.

3.5.3 Migração em sistemas heterogêneos

Até aqui, ficou subentendido que o código migrado pode ser executado com facilidade da máquina-alvo. Essa premissa vale quando estamos tratando de sistemas homogêneos. Contudo, em geral, sistemas distribuídos são construídos sobre um conjunto heterogêneo de plataformas, cada um com seu próprio sistema operacional e arquitetura de máquina. Migração em tais sistemas requer que cada plataforma seja suportada, isto é, que o segmento de código possa ser executado em cada plataforma. Além disso,

precisamos assegurar que o segmento de execução pode ser adequadamente representado em cada plataforma.

Os problemas que surgem da heterogeneidade são, sob muitos aspectos, os mesmos da portabilidade. Não é surpresa que as soluções também sejam muito similares. Por exemplo, no final da década de 1970, uma solução simples para amenizar muitos dos problemas para transportar Pascal para máquinas diferentes era gerar um código intermediário independente de máquina para uma máquina virtual abstrata (Barron, 1981). Claro que essa máquina precisaria ser implementada em muitas plataformas, mas então permitiria que programas em Pascal fossem executados em qualquer lugar. Embora essa idéia simples tenha sido muito usada por alguns anos, ela nunca se firmou realmente como solução geral para problemas de portabilidade para outras linguagens, em especial a C.

Aproximadamente 25 anos mais tarde, a migração de código em sistemas heterogêneos está sob ataque por linguagens de ‘scripts’ e linguagens de alta portabilidade como a Java. Em essência, essas soluções adotam a mesma abordagem adotada para transportar Pascal. Todas essas soluções têm em comum o fato de dependerem de uma máquina virtual (de processo) que ou interprete diretamente códigos-fonte (como é o caso de linguagens de ‘script’) ou interprete código intermediário gerado por um compilador (como em Java). Estar no lugar certo na hora certa também é importante para desenvolvedores de linguagem.

Desenvolvimentos recentes começaram a enfraquecer a dependência em relação a linguagens de programação. Em particular, foram propostas soluções não apenas para migrar processos, mas para migrar ambientes de computação inteiros. A idéia básica é compartmentalizar o ambiente global e fornecer a processos que estão na mesma parte sua própria visão de seu ambiente de computação.

Se a compartmentalização for feita adequadamente, torna-se possível desacoplar uma parte do sistema subjacente e realmente migrá-lo para uma outra máquina. Desse modo, a migração realmente proporcionaria uma forma de mobilidade forte para processos porque, então, eles poderiam ser movidos em qualquer ponto durante sua execução e continuar de onde saíram quando a migração estivesse concluída. Além do mais, muitas das complexidades relacionadas com migração de processos enquanto eles ainda têm vinculações com recursos locais podem ser resolvidas porque, em muitos casos, essas vinculações são simplesmente preservadas. Os recursos locais, especificamente, muitas vezes fazem parte do ambiente que está migrando.

Há várias razões para querer migrar ambientes inteiros, mas talvez a mais importante seja que esse tipo de migração permite continuação de operação enquanto uma máquina precisa ser desligada. Por exemplo, em um cluster de servidores, o administrador de sistemas pode decidir desligar ou substituir uma máquina, mas não terá de parar todos os seus processos em execução. Em vez disso,

ele pode congelar temporariamente um ambiente, movê-lo para uma outra máquina (onde ele ficará ao lado de outros ambientes existentes) e simplesmente descongelá-lo novamente. É óbvio que esse é um modo extremamente poderoso de gerenciar ambientes de computação de longo tempo de execução e seus processos.

Vamos considerar um exemplo específico de migração de máquinas virtuais, como discutido em Clark et al. (2005). Nesse caso os autores se concentraram em migração em tempo real de um sistema operacional virtualizado, algo que normalmente seria conveniente em um cluster de servidores em que se consegue um forte acoplamento por meio de uma única rede local compartilhada. Sob essas circunstâncias, a migração envolve dois problemas principais: migrar toda a imagem da memória e migrar vinculações a recursos locais.

Quanto ao primeiro problema há, em princípio, três modos de tratar a migração (que podem ser combinados):

1. Empurrar páginas de memória para a nova máquina e reenviar as que forem modificadas mais tarde durante a migração do processo.
2. Parar a máquina virtual corrente; migrar memória e iniciar a nova máquina virtual.
3. Deixar que a nova máquina virtual puxe novas páginas conforme necessário, isto é, deixar que processos comecem imediatamente na nova máquina virtual e copiar páginas por demanda.

A segunda opção pode resultar em tempo ocioso inaceitável se a máquina virtual que está migrando estiver executando um serviço vivo, isto é, serviço contínuo. Por outro lado, uma abordagem por demanda pura, como representada pela terceira opção, pode prolongar por muito tempo o período de migração, mas também pode resultar em mau desempenho porque transcorre muito tempo antes de o conjunto de trabalho do processo migrado ser movido para a nova máquina.

Como alternativa, Clark et al. (2005) propõem usar uma abordagem de pré-cópia que combina a primeira opção com uma breve fase parar-e-copiar, como representada pela segunda opção. Essa combinação pode resultar em tempos de parada de serviço de 200 ms ou menos.

Em relação a recursos locais, as coisas são simplificadas quando lidamos somente com um cluster de servidores. Em primeiro lugar, como há só uma rede, a única coisa que precisa ser feita é anunciar a nova vinculação rede-endereço MAC, de modo que clientes possam contactar os processos migrados na interface de rede correta. Por fim, se for possível admitir que o armazenamento é fornecido como camada separada (como a que mostramos na Figura 3.11), migrar vinculação a arquivos também será simples.

O efeito global é que, em vez de migrar processos, agora vemos, realmente, que um sistema operacional inteiro pode ser movido entre máquinas.

3.6 Resumo

Processos desempenham um papel fundamental em sistemas distribuídos porque formam uma base para comunicação entre máquinas diferentes. Uma questão importante é como os processos são organizados internamente e, em particular, se suportam ou não vários threads de controle. Threads em sistemas distribuídos são particularmente úteis para continuar usando a CPU quando é realizada uma operação bloqueadora de E/S. Desse modo, torna-se possível construir servidores de alta eficiência que executam vários threads em paralelo, entre os quais diversos podem estar bloqueados à espera da conclusão de E/S de disco ou de comunicação de rede.

A organização de uma aplicação distribuída em termos de clientes e servidores se mostrou útil. Em geral, processos clientes implementam interfaces de usuário, que podem ser desde visores muito simples até interfaces avançadas que podem manipular documentos compostos. Ademais, o software cliente visa a conseguir transparência de distribuição ocultando detalhes referentes à comunicação com servidores, à localização desses servidores no momento em questão e a respeito de se os servidores são ou não replicados. Além disso, o software cliente é parcialmente responsável por ocultar falhas e recuperação de falhas.

Servidores costumam ser mais complicados do que clientes, porém, não obstante, estão sujeitos a um número relativamente pequeno de questões de projeto. Por exemplo, servidores podem ser iterativos ou concorrentes, implementar um ou mais serviços e podem ser sem estado ou com estado. Outras questões de projeto tratam de serviços de endereçamento e mecanismos para interromper um servidor após uma requisição de serviço ter sido emitida e, possivelmente, já estar sendo processada.

É preciso dar especial atenção quanto da organização de servidores em um cluster. Um objetivo comum é ocultar do mundo exterior as partes internas de um cluster. Isso significa que a organização de um cluster deve ficar resguardada das aplicações. Com essa finalidade, a maioria dos clusters usa um único ponto de entrada que pode entregar mensagens a servidores no cluster. Um problema desafiador é substituir transparentemente esse único ponto de entrada por uma solução totalmente distribuída.

Um tópico importante para sistemas distribuídos é a migração de código entre máquinas diferentes. Duas razões importantes para implementar migração de código são aumentar desempenho e flexibilidade. Quando a comunicação é cara, às vezes podemos reduzi-la deslocando computações do servidor para o cliente e deixando o cliente fazer o máximo possível de processamento local. A flexibilidade aumenta se um cliente puder descarregar dinamicamente software necessário para a comunicação com um servidor específico. O

software descarregado pode ser dirigido especificamente àquele servidor, sem forçar o cliente à instalação prévia desse software.

Migração de código vem acompanhada de problemas relacionados à utilização de recursos locais quando essa utilização requer que ou os recursos também sejam migrados e sejam estabelecidas novas vinculações a recursos locais na máquina-alvo, ou quando são utilizadas referências de rede no âmbito do sistema. Um outro problema é que a migração de código requer que levemos a heterogeneidade em conta. A prática corrente indica que a melhor solução para lidar com a heterogeneidade é usar máquinas virtuais. Estas podem assumir a forma de máquinas virtuais de processo como no caso de Java, por exemplo, ou a utilização de monitores de máquina virtual que efetivamente permitem a migração de um conjunto de processos com seu sistema operacional subjacente.

Problemas

1. Nesse problema você deverá fazer uma comparação entre ler um arquivo usando um servidor de arquivos monothread ou um servidor multithread. Obter uma requisição para trabalho, despachá-la e fazer o resto do processamento necessário demora 15 ms, considerando que os dados necessários estejam em uma cache na memória principal. Se for preciso uma operação de disco, como acontece em um terço das vezes, serão necessários mais 75 ms, durante os quais o thread dorme. Quantas requisições por segundo o servidor pode manipular se for monothread? E se for multithread?
2. Teria sentido limitar a quantidade de threads em um processo servidor?
3. Descrevemos no texto um servidor de arquivo multithread mostrando por que ele é melhor do que um servidor monothread e um servidor com máquina de estado finito. Há alguma circunstância na qual um servidor monothread poderia ser melhor? Dê um exemplo.
4. Associar estaticamente somente um thread com um processo leve não é uma idéia assim tão boa. Por quê?
5. Ter só um processo leve por processo nem sempre é uma idéia assim tão boa. Por quê?
6. Descreva um esquema simples no qual há tantos processos leves quanto sejam os threads executáveis.
7. X designa um terminal de usuário como hospedeiro do servidor, enquanto uma aplicação é referida como cliente. Isso tem sentido?
8. O protocolo X sofre problemas de escalabilidade. Como esses problemas poderiam ser atacados?
9. Proxies podem suportar transparência de replicação invocando cada réplica, como explicado no texto. O lado servidor de uma aplicação pode estar sujeito a uma chamada replicada?
10. Construir um servidor concorrente por meio da multiplicação de um processo tem algumas vantagens e desvantagens em comparação com servidores multithread. Cite algumas.
11. Faça um desenho esquemático de um servidor multithread que suporta vários protocolos que usam Sockets como sua interface de nível de transporte para o sistema operacional subjacente.
12. Como podemos impedir que uma aplicação evite um gerenciador de janela e, assim, consiga bagunçar completamente uma tela?
13. Um servidor que mantém uma conexão TCP/IP com um cliente é com estado ou sem estado?
14. Imagine um servidor Web que mantenha uma tabela na qual endereços IP de clientes sejam mapeados para as páginas Web acessadas mais recentemente. Quando um cliente se conecta ao servidor, este consulta o cliente em sua tabela e, caso o encontre, retorna a página registrada. Esse servidor é com estado ou sem estado?
15. Mobilidade forte em sistemas Unix pode ser suportada com a permissão de bifurcação de um processo para um filho em uma máquina remota. Explique como isso funcionaria.
16. A Figura 3.17 sugere que mobilidade forte não pode ser combinada com execução de um código migrado em um processo-alvo. Dê um exemplo que contradiga isso.
17. Considere um processo P que requer acesso ao arquivo F , disponível na máquina em que P está executando no momento em questão. Quando P passa para outra máquina, ele ainda requer acesso a F . Se a vinculação arquivo-máquina for fixa, como poderia ser implementada a referência a F no âmbito do sistema?
18. Descreva com detalhes como pacotes TCP fluem no caso de transferência TCP junto com a informação sobre endereços de fonte e destino nos vários cabeçalhos.