

2

Arquiteturas

Sistemas distribuídos muitas vezes são complexas peças de software cujos componentes estão, por definição, espalhados por várias máquinas. Para controlar sua complexidade, é crucial que esses sistemas sejam organizados adequadamente. Há diferentes modos de ver a organização de um sistema distribuído, mas uma maneira óbvia é fazer uma distinção entre a organização lógica do conjunto de componentes de software e, por outro lado, a realização física propriamente dita.

A organização de sistemas distribuídos trata, em grande parte, dos componentes de software que constituem o sistema. Essas **arquiteturas de software** nos dizem como os vários componentes de software devem ser organizados e como devem interagir. Neste capítulo, em primeiro lugar vamos dar atenção a algumas abordagens comumente aplicadas à organização de sistemas (distribuídos) de computadores.

A realização efetiva de um sistema distribuído implica que especifiquemos e coloquemos componentes de software em máquinas reais. Para fazer isso, há diferentes opções. A especificação final de uma arquitetura de software é também denominada **arquitetura de sistema**. Neste capítulo estudaremos arquiteturas centralizadas tradicionais nas quais um único servidor implementa a maioria dos componentes de software — e, portanto, a funcionalidade — enquanto clientes remotos podem acessar esse servidor usando meios de comunicação simples. Além disso, vamos considerar arquiteturas descentralizadas nas quais as máquinas desempenham papéis mais ou menos iguais, bem como organizações híbridas.

Como explicamos no Capítulo 1, uma meta importante de sistemas distribuídos é separar aplicações das plataformas subjacentes provendo uma camada de middleware. Adotar tal camada é uma decisão arquitetônica importante, e sua finalidade principal é proporcionar transparência de distribuição. Contudo, é preciso fazer compromissos para conseguir transparência, o que resulta em várias técnicas para tornar o middleware adaptativo. Discutiremos algumas das técnicas mais comumente aplicadas neste capítulo porque elas afetam a organização do próprio middleware.

Também pode se conseguir a adaptabilidade em sistemas distribuídos fazendo o sistema monitorar seu próprio comportamento e tomar as providências adequadas quando necessário. Esse modo de ver as coisas resultou em uma classe que agora denominamos **sistemas autônomos**. Esses sistemas distribuídos são freqüentemente organizados sob a forma de realimentações de contato que formam um importante elemento arquitetônico durante o projeto de um sistema. Neste capítulo, dedicamos uma seção a sistemas distribuídos autônomos.

2.1 Estilos Arquitetônicos

Começamos nossa discussão de arquiteturas considerando, em primeiro lugar, a organização lógica de sistemas distribuídos em componentes de software, também denominada arquitetura de software (Bass et al., 2003). A pesquisa de arquiteturas de software teve considerável amadurecimento e atualmente já é comum aceitar que projetar ou adotar uma arquitetura é crucial para o sucesso no desenvolvimento de grandes sistemas.

Para nossa discussão, a noção de um **estilo arquitetônico** é importante. Tal estilo é formulado em termos de componentes, do modo como esses componentes estão conectados uns aos outros, dos dados trocados entre componentes e, por fim, da maneira como esses elementos são configurados em conjunto para formar um sistema. Um **componente** é uma unidade modular com interfaces requeridas e fornecidas bem definidas que é substituível dentro de seu ambiente (OMG, 2004b).

Como discutiremos a seguir, a questão importante sobre um componente para sistemas distribuídos é que ele pode ser substituído, contanto que respeitemos suas interfaces. Um conceito um pouco mais difícil de entender é o de um **conector** que, em geral, é descrito como um mecanismo que serve de mediador da comunicação ou da cooperação entre componentes (Mehta et al., 2000; Shaw e Clements, 1997). Por exemplo, um conector pode ser formado pelas facilidades para chamadas de procedimento (remotas), passagem de mensagem ou fluxos de dados.

Usando componentes e conectores, podemos chegar a várias configurações que, por sua vez, foram classificadas em estilos arquitetônicos. Até agora já foram identificados vários estilos, entre os quais os mais importantes para sistemas distribuídos são:

1. Arquiteturas em camadas
2. Arquiteturas baseadas em objetos
3. Arquiteturas centradas em dados
4. Arquiteturas baseadas em eventos

A idéia básica para o estilo em camadas é simples: os componentes são organizados em **camadas**, e um componente na camada L_i tem permissão de chamar componentes na camada subjacente L_{i-1} , mas não o contrário, como mostra a Figura 2.1(a). Esse modelo tem sido amplamente adotado pela comunidade de redes; faremos uma breve revisão dele no Capítulo 4. Uma observação fundamental é que, em geral, o controle flui de camada para camada: requisições descem pela hierarquia, ao passo que resultados fluem para cima.

Uma organização bem mais solta é seguida nas **arquiteturas baseadas em objetos**, que são ilustradas na Figura 2.1(b). Em essência, cada objeto corresponde ao que definimos como componente, e esses componentes são conectados por meio de uma chamada de procedimento (remota). Não é surpresa que essa arquitetura de software se ajuste à arquitetura de sistema cliente-servidor que descrevemos antes. As arquiteturas em camadas e baseadas em objetos ainda formam os estilos mais importantes para sistemas de software de grande porte (Bass et al., 2003).

Arquiteturas centradas em dados se desenvolvem em torno da idéia de que processos se comunicam por meio de um repositório comum (passivo ou ativo). Pode-se argumentar que, para sistemas distribuídos, essas arquiteturas são tão importantes quanto as arquiteturas em camadas ou baseadas em objetos. Por exemplo, foi desenvolvida uma profusão de aplicações em rede que dependem de um sistema distribuído de arquivos compartilhados no qual praticamente toda a comunicação ocorre por meio de arquivos. Da mesma maneira, sistemas distribuídos baseados na Web, que discutiremos extensivamente no Capítulo 12, são, em grande parte, centrados em dados: processos se comunicam por meio da utilização de serviços de dados baseados na Web.

Em **arquiteturas baseadas em eventos**, processos se comunicam, em essência, por meio da propagação de eventos que, opcionalmente, também transportam dados, como mostra a Figura 2.2(a). No caso de sistemas distribuídos, a propagação de eventos tem sido associada, em geral, com o que denominamos **sistemas publicar/subscrever** (Eugster et al., 2003). A idéia básica é que processos publiquem eventos após os quais o middleware assegura que somente os processos que se subscreveram para esses eventos os receberão. A principal vantagem de sistemas baseados em eventos é que os processos são fracamente acoplados. Em princípio, eles não precisam se referir explicitamente uns aos outros, o que também é conhecido como desacoplados no espaço ou **referencialmente desacoplados**.

Arquiteturas baseadas em eventos podem ser combinadas com arquiteturas centradas em dados, resultando no

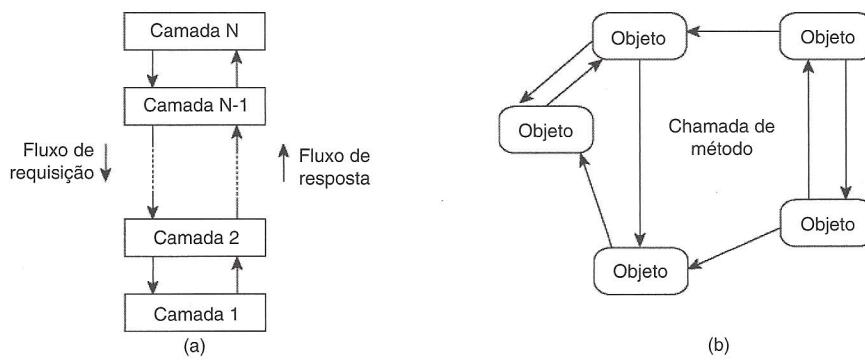


Figura 2.1 Estilo arquitetônico (a) em camadas e (b) baseado em objetos.

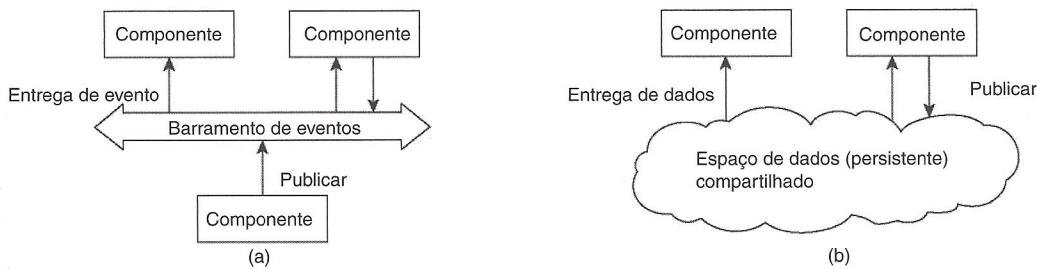


Figura 2.2 Estilo arquitetônico (a) baseado em eventos e (b) de espaço de dados compartilhado.

que também é conhecido como **espaços compartilhados de dados**. A essência de espaços compartilhados de dados é que, agora, os processos também estão desacoplados no tempo: não precisam estar ambos ativos quando ocorre a comunicação. Além do mais, muitos espaços compartilhados de dados usam uma interface semelhante à SQL com o repositório compartilhado, no sentido de que os dados podem ser acessados com a utilização de uma descrição em vez de uma referência explícita, como acontece no caso dos arquivos. Dedicamos o Capítulo 13 a esse estilo arquitetônico.

O que torna essas arquiteturas de software importantes para sistemas distribuídos é que todas elas visam obter transparência de distribuição, em um nível razoável. Todavia, como já argumentamos, transparência de distribuição requer fazer compromissos entre desempenho, tolerância à falha, facilidade de programação e assim por diante. Como não há nenhuma solução única que cumprirá os requisitos para todas as aplicações distribuídas possíveis, os pesquisadores abandonaram a idéia de que um único sistema distribuído pode ser usado para cobrir 90% de todos os casos possíveis.

2.2 Arquiteturas de Sistemas

Agora que já discutimos brevemente alguns estilos arquitetônicos comuns, vamos ver como diversos sistemas distribuídos são realmente organizados, considerando onde são colocados os componentes de software. Decidir a respeito de componentes de software, sua integração e sua colocação leva a um exemplo de uma arquitetura de software também denominada **arquitetura de sistema** (Bass et al., 2003). Discutiremos organizações centralizadas e descentralizadas, bem como várias formas híbridas.

2.2.1 Arquiteturas centralizadas

Apesar da falta de consenso sobre muitas questões de sistemas distribuídos, há uma delas com a qual muitos pesquisadores e praticantes concordam: pensar em termos de *clientes* que requisitam serviços de *servidores* nos ajuda a entender e gerenciar a complexidade de sistemas distribuídos, e isso é bom.

No modelo cliente-servidor básico, processos em um sistema distribuído são divididos em dois grupos, com possível sobreposição. Um **servidor** é um processo que implementa um serviço específico — por exemplo, um serviço de sistema de arquivo ou um serviço de banco de dados. Um **cliente** é um processo que requisita um serviço de um servidor enviando-lhe uma requisição e, na seqüência, esperando pela resposta do servidor. Essa interação cliente-servidor, também conhecida como **comportamento de requisição-resposta**, é mostrada na Figura 2.3.

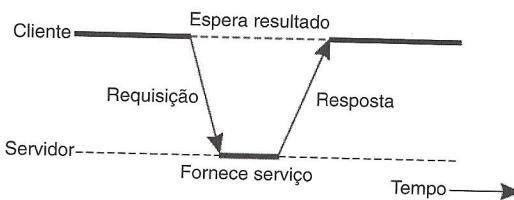


Figura 2.3 Intereração geral entre um cliente e um servidor.

A comunicação entre um cliente e um servidor pode ser implementada por meio de um protocolo simples sem conexão quando a rede subjacente for razoavelmente confiável, como acontece em muitas redes locais. Nesses casos, quando um cliente requisita um serviço, ele simplesmente empacota uma mensagem para o servidor, identificando o serviço que quer, junto com os dados de entrada necessários. Então a mensagem é enviada ao servidor. Por sua vez, o servidor sempre vai esperar pela chegada de uma requisição e, em seguida, a processará e empacotará os resultados em uma mensagem de resposta que é enviada ao cliente.

Usar um protocolo sem conexão tem a óbvia vantagem de ser eficiente. Contanto que as mensagens não se percam nem sejam corrompidas, o protocolo de requisição/resposta que acabamos de esboçar funciona bem. Infelizmente, fazer com que o protocolo seja resistente a ocasionais falhas de transmissão não é trivial. A única coisa que podemos fazer é possivelmente deixar que o cliente reenvie a requisição quando não receber nenhuma mensagem de resposta. Todavia, o problema está no fato de que o cliente não pode detectar se a mensagem de requisição original se perdeu ou se a transmissão da resposta falhou. Se a resposta se perdeu, reenviar uma requisição pode resultar em executar a operação duas vezes. Se a operação for algo como “transfira \$ 10.000 de minha conta”, então é claro que teria sido melhor apenas comunicar que houve um erro.

Por outro lado, se a operação for “informe quanto dinheiro ainda tenho”, seria perfeitamente aceitável reenviar a requisição. Quando uma operação pode ser repetida várias vezes sem causar dano, diz-se que ela é **idempotente**. Visto que algumas requisições são idempotentes e outras não, deve ficar claro que não há nenhuma solução única para tratar mensagens perdidas. Adiaremos uma discussão detalhada sobre o tratamento de falhas de transmissão para o Capítulo 8.

Como alternativa, muitos sistemas cliente-servidor usam um protocolo confiável orientado a conexão. Embora não seja inteiramente adequada para uma rede local devido a seu desempenho relativamente lento, essa solução funciona perfeitamente bem em sistemas de longa distância, nos quais a comunicação é inherentemente não confiável. Praticamente todos os protocolos de aplicação da Internet são baseados em conexões TCP/IP confiáveis. Nesse caso, sempre que um cliente requisita

um serviço, primeiro ele estabelece conexão com o servidor e depois envia a requisição.

Em geral, o servidor usa a mesma conexão para enviar a mensagem de resposta, após o que a conexão é encerrada. O problema é que estabelecer e encerrar uma conexão custa relativamente caro, em especial quando as mensagens de requisição e resposta forem pequenas.

Camadas de aplicação

O modelo cliente-servidor tem sido alvo de muitos debates e controvérsias ao longo dos anos. Uma das questões principais era como estabelecer uma distinção clara entre um cliente e um servidor. Não é surpresa que freqüentemente não haja nenhuma distinção clara. Por exemplo, um servidor para um banco de dados distribuído pode agir continuamente como um cliente porque está repassando requisições para diferentes servidores de arquivo responsáveis pela implementação das tabelas do banco de dados. Nesse caso, em essência, o próprio servidor do banco de dados nada mais faz do que processar consultas.

Contudo, considerando que muitas aplicações cliente-servidor visam a dar suporte ao acesso de usuários a banco de dados, muitas pessoas defendem uma distinção entre os três níveis citados abaixo, seguindo, em essência, o estilo arquitetônico em camadas que discutimos antes:

1. Nível de interface de usuário
2. Nível de processamento
3. Nível de dados

O nível de interface de usuário contém tudo que é necessário para fazer interface diretamente com o usuário, como gerenciamento de exibição. O nível de processamento normalmente contém as aplicações. O nível de dados gerencia os dados propriamente ditos sobre os quais está sendo executada alguma ação.

Em geral, clientes implementam o nível de interface de usuário. Esse nível consiste em programas que permitem aos usuários finais interagir com aplicações. Há considerável diferença entre os níveis de sofisticação de programas de interface de usuário.

O programa de interface de usuário mais simples nada mais é do que uma tela baseada em caracteres. Tal interface normalmente é usada em ambientes de mainframe. Nos casos em que o mainframe controla toda a interação, incluindo teclado e monitor, mal podemos falar em ambiente cliente-servidor. Entretanto, em muitos casos, o terminal de usuário realiza algum processamento local, como ecoar teclas acionadas ou suportar interfaces do tipo formulário nas quais deve-se editar uma entrada completa antes de enviá-la ao computador principal.

Hoje em dia, mesmo em ambientes de mainframe, vemos interfaces de usuário mais avançadas. Normalmente, a máquina cliente oferece no mínimo um visor gráfico no qual são usados menus pop-up ou pull-down com controles

de tela, muitos dos quais manipulados por meio de um mouse em vez de pelo teclado. Entre os exemplos típicos dessas interfaces estão as interfaces X-Windows utilizadas em ambientes Unix e interfaces mais antigas desenvolvidas para PCs MS-DOS e Macintoshes da Apple.

Interfaces de usuário modernas oferecem consideravelmente mais funcionalidade, permitindo que aplicações compartilhem uma única janela gráfica e usem essa janela para permitir dados por meio de ações de usuário. Por exemplo, para apagar um arquivo, em geral é possível arrastar o ícone que representa esse arquivo até um ícone que representa uma lata de lixo. Da mesma maneira, muitos editores de texto permitem a um usuário transferir texto de um documento para uma outra posição usando apenas o mouse. Voltaremos a interfaces de usuário no Capítulo 3.

Muitas aplicações cliente-servidor podem ser construídas de acordo com três partes diferentes: uma parte que manipula a interação com um usuário, uma parte que age sobre um banco de dados ou sistema de arquivo e uma parte intermediária que, em geral, contém a funcionalidade central de uma aplicação. Essa parte intermediária está localizada logicamente no nível de processamento. Ao contrário de interfaces de usuário e bancos de dados, não há muitos aspectos comuns no nível de processamento. Portanto, daremos vários exemplos para esclarecer melhor esse nível.

Como primeiro exemplo, considere um mecanismo de busca da Internet. Ignorando todos os banners animados, imagens e outras extravagâncias das janelas, a interface de usuário de um mecanismo de busca é muito simples: um usuário digita uma seqüência de palavras-chave e, em seguida, aparece na tela uma lista de títulos de páginas Web. A retaguarda de apoio é formada por um enorme banco de dados de páginas Web que foram pesquisadas antecipadamente e indexadas. O núcleo do mecanismo de busca é um programa que transforma a seqüência de palavras-chaves do usuário em uma ou mais consultas a banco de dados. Em seguida, o mecanismo de busca ordena os resultados em uma lista e a transforma em uma série de páginas HTML. Dentro do modelo cliente-servidor, essa parte de recuperação de informações costuma estar localizada no nível de processamento. A Figura 2.4 mostra essa organização.

Como um segundo exemplo, considere um sistema de suporte à decisão para uma corretora de valores. Análogo ao mecanismo de busca, tal sistema pode ser dividido em uma extremidade frontal que implementa a interface de usuário, uma retaguarda de apoio para acessar um banco de dados que contém os dados financeiros e os programas de análise entre essas duas. A análise de dados financeiros pode exigir métodos e técnicas sofisticados de estatística e inteligência artificial. Em alguns casos, o núcleo de um sistema de suporte a decisões financeiras pode até precisar ser executado em computadores de alto desempenho de

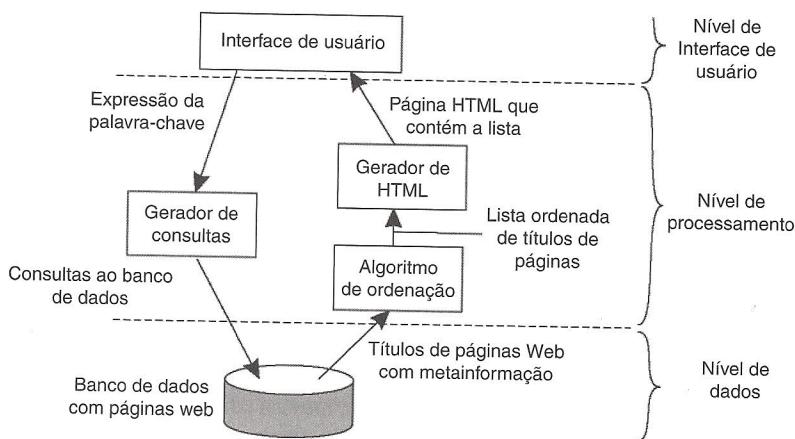


Figura 2.4 Organização simplificada de um mecanismo de busca da Internet em três camadas diferentes.

modo a conseguir a produtividade e a capacidade de resposta esperada por seus usuários.

Como um último exemplo, considere um pacote típico para computadores de mesa composto de um processador, uma aplicação de planilha, facilidades de comunicação e assim por diante. Esses conjuntos ‘de escritório’, em geral, são integrados por meio de uma interface comum de usuário que suporta documentos compostos e age sobre arquivos do diretório particular (*home directory*) do usuário. Em ambientes de escritório, esse diretório particular costuma estar localizado em um servidor de arquivo remoto.

Nesse exemplo, o nível de processamento consiste em um conjunto relativamente grande de programas, cada um com capacidades de processamento bastante simples.

O nível de dados no modelo cliente–servidor contém os programas que mantêm os dados propriamente ditos sobre os quais as aplicações agem em suas operações. Uma importante propriedade desse nível é que os dados costumam ser **persistentes**, isto é, ainda que nenhuma aplicação esteja sendo executada, os dados estarão armazenados em algum lugar para a próxima utilização. Em sua forma mais simples, o nível de dados consiste em um sistema de arquivo, porém é mais comum utilizar um banco de dados plenamente capacitado. No modelo cliente–servidor, o nível de dados normalmente é implementado no lado servidor.

Além do mero armazenamento de dados, em geral o nível de dados também é responsável por manter os dados consistentes nas diferentes aplicações. Quando bancos de dados estão sendo usados, manter consistência significa que os metadados, como descrições de tabelas, restrições de entrada e metadados específicos de aplicação, também são armazenados nesse nível. No caso de um banco, podemos querer gerar um aviso quando o cartão de débito de um cliente chegar a certo valor. Esse tipo de informação pode ser mantido por meio de um mecanismo de disparo (*trigger*) de banco de dados que ativa um manipulador que é acionado no momento adequado.

Na maioria dos ambientes orientados a negócios, o nível de dados é organizado como um banco de dados

relacional. Nesse caso, a independência dos dados é crucial. Os dados são organizados independentemente das aplicações de modo tal que alterações nessa organização não afetam as aplicações, nem as aplicações afetam a organização dos dados. Usar bancos de dados relacionais no modelo cliente–servidor ajuda a separar o nível de processamento do nível de dados, porque processamento e dados são considerados independentes.

Todavia, nem sempre bancos de dados relacionais são a opção ideal. Um aspecto característico de muitas aplicações é que elas operam sobre tipos de dados complexos cuja modelagem é mais fácil em termos de objetos do que em termos de relações. Exemplos desses tipos de dados vão de simples polígonos e círculos até representações de projetos de aeronaves, como é o caso de sistemas de projeto auxiliado por computador (*Computer-aided Design — CAD*).

Nos casos em que é mais fácil expressar operações de dados em termos de manipulações de objetos, faz sentido implementar o nível de dados por meio de um banco de dados orientado a objetos ou de um banco de dados relacional orientado a objetos. Esse último tipo está conquistando notável popularidade porque esses bancos de dados são construídos sobre o modelo de dados relacionais amplamente dispersos e, ao mesmo tempo, oferecem as vantagens que a orientação a objetos proporciona.

Arquiteturas multidivididas

A distinção entre três níveis lógicos, como discutimos até aqui, sugere várias possibilidades para a distribuição física de uma aplicação cliente–servidor por várias máquinas. A organização mais simples é ter só dois tipos de máquinas:

1. Uma máquina cliente que contém apenas os programas que implementam o nível (parte do nível) de interface de usuário.
2. Uma máquina do servidor que contém o resto, ou seja, os programas que implementam o nível de processamento e de dados.

Nessa organização, tudo é manipulado pelo servidor, *ao passo que*, em essência, o cliente nada mais é do que um terminal burro, possivelmente com uma interface gráfica bonitinha. Há muitas outras possibilidades e estudaremos algumas das mais comuns nesta seção.

Uma abordagem para organizar clientes e servidores é distribuir os programas presentes nas camadas de aplicação da seção anterior por máquinas diferentes, como mostra a Figura 2.5 [veja também Umar (1997) e Jing et al. (1999)]. Como primeira etapa, fazemos uma distinção entre dois tipos de máquinas apenas: máquinas clientes e máquinas servidoras, o que resulta em algo também denominado **arquitetura de duas divisões (físicas)**.

Uma possível organização é ter na máquina cliente só a parte da interface de usuário que é dependente de terminal, como mostra a Figura 2.5(a), e dar às aplicações o controle remoto sobre a apresentação de seus dados. Uma alternativa é colocar todo o software de interface de usuário no lado cliente, como mostra a Figura 2.5(b). Nesses casos, em essência, dividimos a aplicação em uma extremidade frontal gráfica, que se comunica com o resto da aplicação — que reside no servidor — por meio de um protocolo específico de aplicação. Nesse modelo, a extremidade frontal — o software cliente — não faz nenhum processamento exceto o necessário para apresentar a interface da aplicação.

Prosseguindo nessa linha de raciocínio, também podemos deslocar parte da aplicação para a extremidade frontal, como mostra a Figura 2.5(c). Um exemplo no qual isso faz sentido é a aplicação utilizar um formulário que precise ser completamente preenchido antes de poder ser processado. Então, a extremidade frontal pode verificar a correção e a consistência do formulário e, quando necessário, interagir com o usuário. Um outro exemplo de organização como a da Figura 2.5(c) é o de um editor de texto no qual as funções básicas de edição são executadas no lado cliente, onde operam sobre dados presentes em caches locais ou em dados da memória, mas no qual as ferramentas avançadas de suporte, como verificação de ortografia e gramática, são executadas no lado servidor.

Em muitos ambientes cliente-servidor, as organizações mostradas nas figuras 2.5(d) e 2.5(e) gozam de particular popularidade. Essas organizações são utilizadas quando a máquina cliente é um PC ou estação de trabalho, conectado por meio de uma rede a um sistema de arquivo distribuído ou a um banco de dados. Em essência, grande parte da aplicação está executando na máquina cliente, mas todas as operações com arquivos ou entradas em banco de dados vão para o servidor. Por exemplo, muitas aplicações bancárias executam na máquina de um usuário final na qual este prepara transações e coisas semelhantes. Uma vez concluída, a aplicação contata o banco de dados no servidor do banco e carrega as transações para processamento ulterior.

A Figura 2.5(e) representa a situação em que o disco local do cliente contém parte dos dados. Ao consultar a Web com seu browser, por exemplo, um cliente pode construir gradativamente uma enorme cache em disco local com as páginas Web mais recentemente consultadas.

Observamos que nos últimos anos tem ocorrido forte tendência para abandonar as configurações mostradas nas figuras 2.5(d) e 2.5(e) nos casos em que o software cliente é colocado em máquinas de usuários finais. Nesses casos, grande parte do processamento e do armazenamento de dados é manipulada no lado do servidor. A razão para isso é simples: embora máquinas clientes façam muito, também são mais problemáticas para gerenciar. Ter mais funcionalidade na máquina cliente torna o software do lado do cliente mais propenso a erros e mais dependente da plataforma subjacente do cliente, isto é, do sistema operacional e respectivos recursos.

Da perspectiva de gerenciamento de sistema, ter o que denominamos **clientes gordos (fat clients)** não é ótimo. Em vez disso, os terminais **clientes magros (thin clients)**, como representados pelas organizações mostradas nas figuras 2.5(a) a 2.5(c), são muito mais fáceis, talvez ao custo de interfaces de usuário menos sofisticadas e do desempenho percebido pelo cliente.

Observe que essa tendência não implica que não precisemos mais de sistemas distribuídos. Ao contrário, o que estamos vendo é que as soluções do lado do servidor

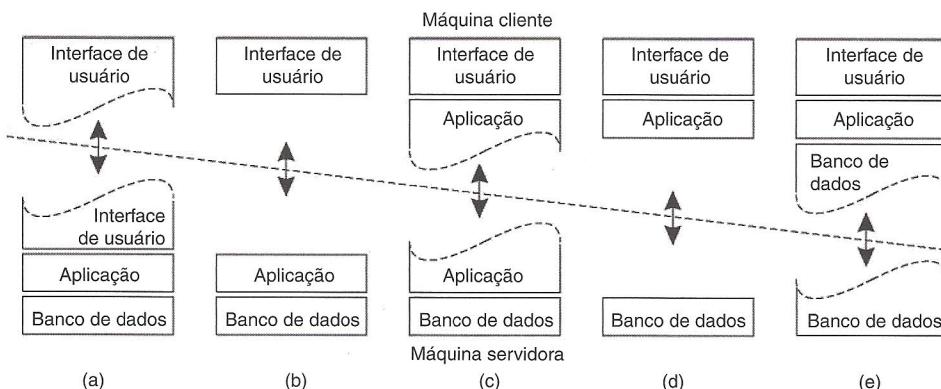


Figura 2.5 Alternativas de organizações cliente-servidor (a)–(e).

estão se tornando cada vez mais distribuídas à medida que um servidor único está sendo substituído por vários servidores que executam em máquinas diferentes. Em particular, quando fizemos distinção somente entre máquinas clientes e máquinas servidoras, como fizemos até aqui, deixamos passar em branco o fato de que um servidor às vezes pode precisar agir como um cliente, como mostra a Figura 2.6, o que resulta em uma **arquitetura de três divisões, em termos físicos**. Nessa arquitetura, programas que formam parte do nível de processamento residem em um servidor separado, mas, além disso, podem ser parcialmente distribuídos pelas máquinas cliente e servidora. Um exemplo típico da utilização de uma arquitetura de três divisões é o processamento de transações.

Como discutimos no Capítulo 1, um processo separado, denominado monitor de processamento de transação, coordena todas as transações em servidores de dados possivelmente diferentes.

Um outro exemplo, porém muito diferente, no qual muitas vezes vemos uma arquitetura de três divisões, é a organização de sites Web. Nesse caso, um servidor Web age como ponto de entrada para um site, passando requisições para um servidor de aplicação no qual ocorre o processamento propriamente dito. Por sua vez, esse servidor de aplicação interage com um servidor de banco de dados. Um servidor de aplicação pode ser responsável por rodar o código para inspecionar o estoque disponível de algumas mercadorias oferecidas por uma livraria eletrônica. Para fazer isso, ele talvez precise interagir com um banco de dados que contém os dados brutos do estoque. Voltaremos à organização de sites Web no Capítulo 12.

2.2.2 Arquiteturas descentralizadas

Arquiteturas cliente-servidor multidivididas são uma consequência direta da divisão de aplicações em uma interface de usuário em componentes de processamento e em um nível de dados. As diferentes divisões correspondem diretamente à organização lógica das aplicações. Em muitos ambientes de negócios, processamento distribuído equivale a organizar uma aplicação cliente-servidor como uma arquitetura multidividida. Esse tipo de distribuição é denominado **distribuição vertical**. O aspecto característico da distribuição vertical é que ela é obtida ao se colocar

componentes *logicamente* diferentes em máquinas diferentes. O termo está relacionado ao conceito de **fragmentação vertical** como utilizada em bancos de dados relacionais distribuídos, nos quais significa que as tabelas são subdivididas em colunas e, na seqüência, distribuídas por várias máquinas (Oszu e Valduriez, 1999).

Mais uma vez, da perspectiva de gerenciamento de sistema, ter uma distribuição vertical pode ajudar: funções são subdivididas lógica e fisicamente por várias máquinas, e cada máquina é projetada para um grupo específico de funções. Contudo, a distribuição vertical é apenas um dos modos de organizar aplicações cliente-servidor. Em arquiteturas modernas, muitas vezes é a distribuição dos clientes e dos servidores que conta, à qual nos referimos como **distribuição horizontal**.

Nesse tipo de distribuição, um cliente ou servidor pode ser fisicamente subdividido em partes logicamente equivalentes, mas cada parte está operando em sua própria porção do conjunto completo de dados, o que equilibra a carga. Nesta seção, examinaremos uma classe moderna de arquiteturas de sistemas que suporta distribuição horizontal, conhecida como **peer-to-peer**.

De uma perspectiva de alto nível, os processos que constituem um sistema peer-to-peer são todos iguais, o que significa que as funções que precisam ser realizadas são representadas por todo processo que constitui o sistema distribuído. Como consequência, grande parte da interação entre processos é simétrica: cada processo agirá como um cliente e um servidor ao mesmo tempo (o que também se denomina agir como **servente**).

Dado esse comportamento simétrico, arquiteturas peer-to-peer se desenvolvem em torno da questão de como organizar os processos em uma **rede de sobreposição**, isto é, uma rede na qual os nós são formados pelos processos e os enlaces representam os canais de comunicação possíveis (que usualmente são realizados como conexões TCP). Em geral, um processo não pode se comunicar diretamente com um outro processo arbitrário, mas deve enviar mensagens por meio dos canais de comunicação disponíveis.

Existem dois tipos de redes de sobreposição: as que são estruturadas e as que não são. Esses dois tipos são estudados extensivamente em Lua et al. (2005), acompan-

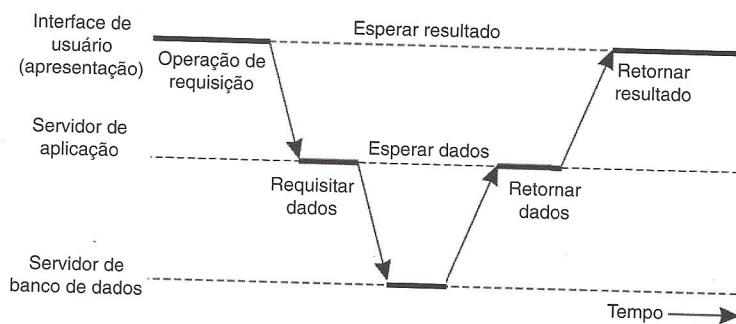


Figura 2.6 Exemplo de um servidor que age como cliente.

nizados de numerosos exemplos. Aberer et al. (2005) dão uma arquitetura de referência que permite uma comparação mais formal entre os diferentes tipos de sistemas peer-to-peer. Androusellis-Theotokis e Spinellis (2004) dão um levantamento realizado com base na perspectiva da distribuição de conteúdo.

Arquiteturas peer-to-peer estruturadas

Em uma arquitetura peer-to-peer estruturada, a rede de sobreposição é construída com a utilização de um procedimento determinístico. O procedimento mais usado é, de longe, organizar os processos por meio de uma **tabela de hash distribuída (Distributed Hash Table — DHT)**.

Em um sistema baseado em DHT, os itens de dados recebem uma chave aleatória, como um identificador de 128 bits ou 160 bits, de um grande espaço de identificadores. Da mesma maneira, os nós do sistema também recebem um número aleatório do mesmo espaço de identificadores. Portanto, o ponto crucial de todo sistema baseado em DHT é implementar um esquema eficiente e determinístico que mapeie exclusivamente a chave de um item de dado para o identificador de um nó tendo como base somente alguma distância métrica (Balakrishnan et al., 2003). O mais importante é que, ao consultar um item de dado, o endereço de rede do nó responsável por aquele item de dado é retornado. Na verdade, consegue-se isso roteando uma requisição para um item de dado até o nó responsável.

No sistema Chord (Stoica et al., 2003), por exemplo, os nós estão logicamente organizados em um anel de modo tal que um item de dado com chave k seja mapeado para o nó que tenha o menor identificador $id \geq k$. Esse nó é denominado *sucessor* da chave k e denotado como $succ(k)$, como mostra a Figura 2.7. Portanto, para consultar o item de dado, uma aplicação que executa em um nó arbitrário teria de chamar a função $LOOKUP(k)$ que, na seqüência, retornaria o endereço de rede de $succ(k)$. Nesse ponto, a aplicação pode contatar o nó para obter uma cópia do item de dado.

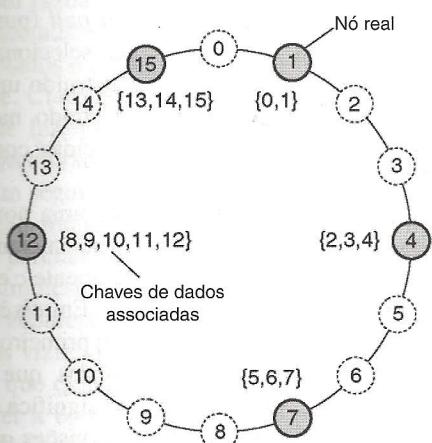


Figura 2.7 Mapeamento de itens de dados para nós em Chord.

Não entraremos no assunto de algoritmos para consultar uma chave agora; adiaremos essa discussão até o Capítulo 5, no qual descreveremos detalhes de vários sistemas de nomeação. Em vez disso, vamos nos concentrar no modo como os nós se organizam em uma rede de sobreposição, em outras palavras, no **gerenciamento de associação ao grupo**. No que for apresentado adiante, é importante entender que consultar uma chave não segue a organização lógica de nós no anel da Figura 2.7. Na verdade, cada nó manterá atalhos para outros nós, de modo tal que, em geral, as consultas possam ser feitas em $O(\log(N))$ números de etapas, onde N é o número de nós que participam da rede de sobreposição.

Agora considere novamente o Chord. Quando um nó quer se juntar ao sistema, ele começa gerando um identificador aleatório id . Observe que, se o espaço do identificador for grande o suficiente, contanto que o gerador de números aleatórios seja de boa qualidade, a probabilidade de gerar um identificador que já esteja designado a um nó real é próxima de zero. Portanto, o nó pode simplesmente fazer uma pesquisa em id , que retornará o endereço de rede de $succ(id)$. Nesse ponto, o nó que está se juntando ao grupo pode simplesmente contatar $succ(id)$ e seu predecessor e se inserir no anel. Claro que esse esquema requer que cada nó também armazene informações sobre seu predecessor. Da inserção ainda decorre que cada item de dado cuja chave esteja agora associada com o nó id seja transferido de $succ(id)$.

Sair também é simples: o nó id informa sua partida a seu predecessor e sucessor e transfere seus itens de dados para $succ(id)$.

Abordagens similares são adotadas em outros sistemas baseados em DHT. Para ilustrar, considere a **rede de conteúdo endereçável (Content Addressable Network — CAN)** descrita em Ratnasamy et al. (2001). A CAN emprega um espaço de coordenadas cartesianas de d dimensões que é completamente particionado entre todos os nós que participam do sistema. Para finalidade de ilustração, vamos considerar apenas o caso de duas dimensões, do qual mostraremos um exemplo na Figura 2.8.

A Figura 2.8(a) mostra como um espaço bidimensional $[0,1] \times [0,1]$ é dividido entre seis nós. Cada nó tem uma região associada. A todo item de dados em CAN será atribuído um único ponto desse espaço, após o que também fica claro qual nó é responsável por aquele dado (ignorando itens de dados que caem na fronteira de várias regiões, para os quais é utilizada uma regra determinística de atribuição).

Quando um nó P quer se juntar a um sistema CAN, ele escolhe um ponto arbitrário do espaço de coordenadas e, na seqüência, pesquisa o nó Q em cuja região o ponto cai. Essa pesquisa é realizada por roteamento baseado em posicionamento, cujos detalhes adiaremos até capítulos posteriores. Então, o nó Q subdivide sua região em duas metades, como mostra a Figura 2.8(b), e uma metade é

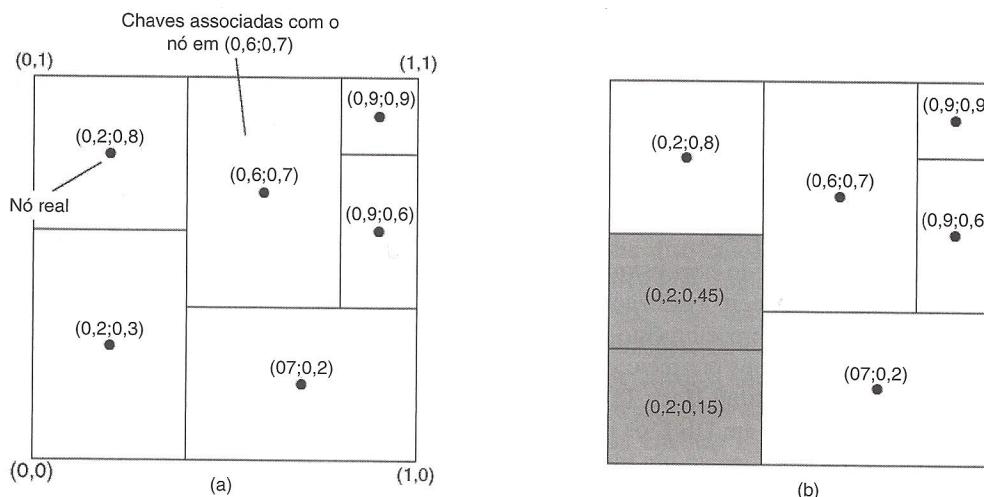


Figura 2.8 (a) Mapeamento de itens de dados para nós em CAN. (b) Subdivisão de uma região quando um nó se junta ao grupo.

designada ao nó P . Os nós monitoram seus vizinhos, isto é, nós responsáveis por regiões adjacentes. Na ocasião de subdivisão de uma região, fica fácil para o nó P , que está se juntando ao grupo, vir a saber quem são seus novos vizinhos perguntando ao nó Q . Como acontece em Chord, os itens de dados pelos quais o nó P agora é responsável são transferidos do nó Q .

Sair da CAN é um pouco mais complicado. Considere que, na Figura 2.8, o nó cuja coordenada é $(0,6;0,7)$ sai. Sua região será designada a um de seus vizinhos, por exemplo, o nó em $(0,9;0,9)$, mas é claro que não se pode simplesmente fundi-la e obter um retângulo. Nesse caso, o nó em $(0,9;0,9)$ simplesmente cuidará daquela região e informará isso aos antigos vizinhos. É óbvio que tal fato pode levar a uma repartição menos simétrica do espaço de coordenadas, razão por que um processo de fundo é iniciado periodicamente para fazer a repartição do espaço inteiro.

Arquiteturas peer-to-peer não estruturadas

Sistemas peer-to-peer não estruturados dependem, em grande parte, de algoritmos aleatórios para construir uma rede de sobreposição. A idéia principal é que cada nó mantenha uma lista de vizinhos, mas que essa lista seja construída de modo mais ou menos aleatório. Da mesma maneira, admite-se que itens de dados sejam colocados aleatoriamente em nós. Por consequência, quando um nó precisa localizar um item de dado específico, a única coisa que ele efetivamente pode fazer é inundar a rede com uma consulta de busca (Risson e Moors, 2006). Voltaremos à busca em redes de sobreposição não estruturadas no Capítulo 5 e, por enquanto, vamos nos concentrar em gerenciamento de associação ao grupo.

Uma das metas de muitos sistemas peer-to-peer não estruturados é construir uma rede de sobreposição parecida com um **gráfico aleatório**. O modelo básico é que cada nó mantenha uma lista de c vizinhos na qual, de pre-

ferência, cada um dos vizinhos represente um nó *vivo* escolhido aleatoriamente no conjunto de nós vigente no momento. A lista de vizinhos também é denominada **visão parcial**. Há muitos modos de construir essa visão parcial. Jelasity et al. (2004, 2005a) desenvolveram uma estrutura que captura muitos algoritmos diferentes para a construção da rede de sobreposição, de modo a permitir avaliações e comparações. Nessa estrutura, a premissa adotada é que nós trocam entradas regularmente de sua visão parcial. Cada entrada identifica um outro nó na rede e tem uma idade associada que indica a antiguidade das referências àquele nó. São usados dois threads, conforme mostra a Figura 2.9.

O thread ativo toma a iniciativa de se comunicar com um outro nó e seleciona esse nó de acordo com sua visão parcial corrente. Considerando que entradas precisam ser *empurradas* (*pushed*) até o par selecionado, o thread continua e constrói um buffer que contém $c/2 + 1$ entradas, entre elas uma entrada que identifica ele próprio. As outras entradas são tiradas da visão parcial vigente naquele momento.

Se o nó também estiver em *modo pull* (puxar), ele vai esperar por uma resposta do par selecionado. No meio-tempo, esse par também terá construído um buffer por intermédio do thread passivo mostrado na Figura 2.9(b), cujas atividades são muito parecidas com as do thread ativo.

O ponto crucial é a construção de uma nova visão parcial. Essa visão, que serve para o nó iniciante, bem como para o nó contatado, conterá exatamente c entradas, parte das quais virá de buffer recebido. Em essência, há dois modos de construir a nova visão. No primeiro, os dois nós podem decidir descartar as entradas que tinham enviado um ao outro. Na realidade, isso significa que eles *trocarão*, dinamicamente, parte de suas visões originais. A segunda abordagem é descartar o maior número possível de entradas *velhas*. Em geral, verificamos que as duas

abordagens são complementares [ver Jelasity et al. (2005a) se quiser detalhes]. Ocorre que muitos protocolos de gerenciamento de associação a um grupo para sobreposições não estruturadas se encaixam nessa estrutura. Há várias observações interessantes a fazer.

Primeiro, vamos considerar que, quando um nó quer se juntar ao grupo, ele contata um outro nó arbitrário, possivelmente de uma lista de pontos de acesso bem conhecidos. Esse ponto de acesso é apenas um membro comum da rede de sobreposição, exceto que podemos considerar que ele tenha alta disponibilidade.

Ações por thread ativo (repetidas periodicamente):
 selecione um par P da visão parcial corrente;
 se PUSH_MODE {
 mybuffer = [[MyAddress, 0]];
 permuta visão parcial;
 move H entradas mais velhas para o fim;
 anexe primeiras c/2 entradas a mybuffer;
 envie mybuffer a P;
 } else {
 envie gatilho a P;
 }
 se PULL_MODE {
 receba buffer de P;
 }
 construa uma nova visão parcial com base na corrente e no buffer de P;
 incremente a idade de cada entrada na nova visão parcial;
 (a)

Ações por thread passivo:
 recebe buffer de qualquer processo Q;
 se PULL_MODE {
 mybuffer = [[MyAddress, 0]];
 permuta visão parcial;
 move H entradas mais velhas para o fim;
 anexe primeiras c/2 entradas a mybuffer;
 envie mybuffer a P;
 }
 construa uma nova visão parcial com base na corrente e no buffer de P;
 incremente a idade de cada entrada na nova visão parcial;
 (b)

Figura 2.9 (a) Etapas seguidas pelo thread ativo. (b) Etapas seguidas pelo thread passivo.

Nesse caso, ocorre que protocolos que usam somente *modo pull* ou somente *modo push* podem resultar, com razoável facilidade, em redes de sobreposição desconectadas. Em outras palavras, grupos de nós ficam isolados e nunca poderão alcançar nenhum outro nó na rede. Claro que esse é um aspecto indesejável, razão por que faz mais sentido deixar que os nós realmente troquem entradas.

Em segundo lugar, abandonar a rede vem a ser uma operação muito simples contanto que os nós troquem visões parciais a intervalos regulares. Nesse caso, um nó pode simplesmente sair sem informar qualquer outro nó. O que acontecerá é que, quando um nó P selecionar um de seus vizinhos aparentes, por exemplo, o nó Q, e descobrir que Q não está mais respondendo, ele apenas vai remover a entrada de sua visão parcial para selecionar um outro par. Ocorre que, ao construir uma nova visão parcial, um nó segue a política de descartar o maior

número possível de entradas velhas; portanto, nós que abandonaram a rede serão rapidamente esquecidos. De outro modo: entradas que se referem a nós que saíram da rede serão rápida e automaticamente removidas das visões parciais.

Entretanto, há um preço a pagar quando se adota essa estratégia. Para explicar, considere, para um nó P, o conjunto de nós cuja entrada em suas visões parciais se refere a P. Em termos técnicos, isso é conhecido como o **grau interno** de um nó. Quanto mais alto o grau interno de P, maior a probabilidade de que algum outro nó decida contatar P. Resumindo, há um perigo de P se tornar um nó popular, o que poderia facilmente levá-lo a uma posição de desequilíbrio em relação à carga de trabalho. O descarte sistemático de entradas velhas acaba promovendo nós a nós que tenham alto grau interno.

Há outros compromissos além desses, para os quais damos como referência Jelasity et al. (2005a).

Gerenciamento de topologia de redes de sobreposição

Embora possa parecer que sistemas peer-to-peer estruturados e não estruturados formem classes estritamente independentes, na verdade pode não ser esse o caso [ver também Castro et al. (2005)]. Uma observação fundamental é que, ao trocar e selecionar cuidadosamente as entradas de visões parciais, é possível construir e manter topologias específicas de redes de sobreposição. Esse gerenciamento de topologia é obtido pela adoção de uma abordagem em duas camadas, como mostra a Figura 2.10.

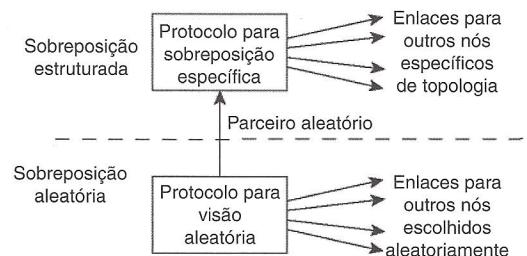


Figura 2.10 Abordagem de duas camadas para construir e manter topologias específicas de sobreposição usando técnicas de sistemas peer-to-peer não estruturadas.

A camada mais baixa constitui um sistema peer-to-peer não estruturado no qual os nós trocam periodicamente entradas de suas visões parciais com o objetivo de manter um gráfico aleatório preciso. Nesse caso, precisão se refere ao fato de que a visão parcial deva ser preenchida com entradas que se refiram a nós vivos selecionados aleatoriamente.

A camada mais baixa passa essa visão parcial para a camada mais alta, em que ocorre uma seleção adicional de entradas. Então, isso resulta em uma segunda lista de vizinhos correspondente à topologia desejada. Jelasity e Babaoglu (2005) propõem usar uma função de ordena-

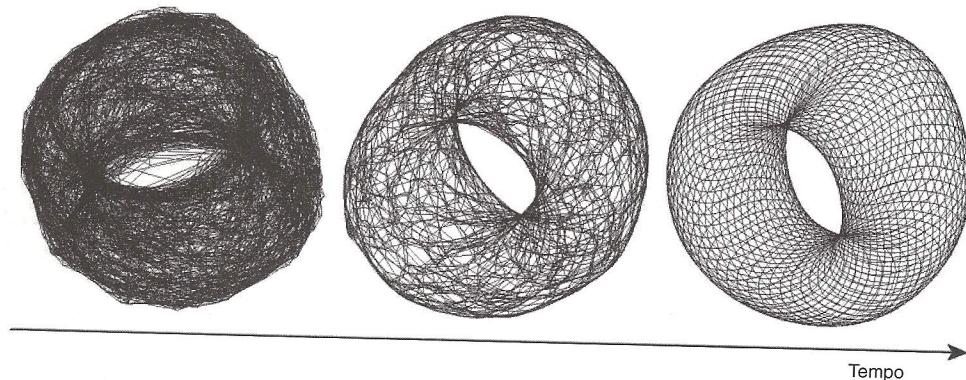


Figura 2.11 Geração de rede de sobreposição específica que utiliza um sistema peer-to-peer não estruturado de duas camadas [adaptado com a permissão de Jelasity e Babaoglu (2005)].

ção pela qual os nós são ordenados de acordo com certo critério em relação a determinado nó. Uma função de ordenação simples é ordenar um conjunto de nós em ordem crescente de distância em relação a um determinado nó P . Nesse caso, o nó P gradativamente montará uma lista de seus vizinhos mais próximos, contanto que a camada mais baixa continue a lhe passar nós selecionados aleatoriamente.

Como ilustração, considere uma grade lógica de tamanho $N \times N$ que tenha um nó colocado em cada ponto da grade. Todo nó deve manter uma lista de c vizinhos mais próximos, na qual a distância entre um nó em (a_1, a_2) e (b_1, b_2) é definida como $d_1 + d_2$, com $d_i = \min(N - |a_i - b_i|, |a_i - b_i|)$. Se a camada mais baixa executar periodicamente o protocolo como esboçado na Figura 2.9, a topologia que se desenvolverá será um toro, mostrado na Figura 2.11.

Naturalmente podem ser usadas funções de ordenação completamente diferentes. Em particular, são interessantes as relacionadas com a captura da **proximidade semântica** de itens de dados como os armazenados em um nó par. Essa proximidade possibilita a construção de **redes semânticas de sobreposição** que permitem algoritmos de busca de alta eficiência em sistemas peer-to-peer não estruturados. Voltaremos a esses sistemas no Capítulo 5 quando discutirmos nomeação baseada em atributo.

Superpares (superpeers)

Deve-se notar que localizar itens de dados relevantes em sistemas peer-to-peer não estruturados pode se tornar problemático à medida que a rede cresce. A razão para esse problema de escalabilidade é simples: como não há nenhum modo determinístico para rotear uma requisição de pesquisa até um item de dado específico, em essência, a única técnica à qual um nó pode recorrer é enviar a requisição a todos os nós. Há vários modos de limitar uma inundação de mensagens, como discutiremos no Capítulo 5, porém, como alternativa, muitos sistemas peer-to-peer

propuseram a utilização de nós especiais que mantenham um índice de itens de dados.

Há outras situações em que é sensato abandonar a natureza simétrica dos sistemas peer-to-peer. Considere uma colaboração de nós que oferecem recursos uns aos outros. Por exemplo, em uma **rede colaborativa de entrega de conteúdo** (**Content Delivery Network — CDN**), os nós podem oferecer armazenamento para hospedar cópias de páginas Web, permitindo que clientes Web acessem páginas próximas e que, por isso, esse acesso seja rápido. Nesse caso, um nó P pode precisar buscar recursos em uma parte específica da rede. Se isso acontecer, usar um intermediário que coleta utilização de recurso para vários nós que estão nas proximidades uns dos outros permitirá a rápida seleção do nó que tenha recursos suficientes.

Nós como os que mantêm um índice, ou agem como intermediários, em geral são denominados **superpares (superpeers)**. Como seu nome sugere, superpares muitas vezes também são organizados em uma rede peer-to-peer, o que resulta em uma organização hierárquica, como explicado em Yang e Garcia-Molina (2003). Um exemplo simples de tal organização é mostrado na Figura 2.12. Nessa organização, todo par comum está conectado como cliente a um superpar. Toda comunicação de e para um par comum ocorre por meio daquele superpar associado ao par.

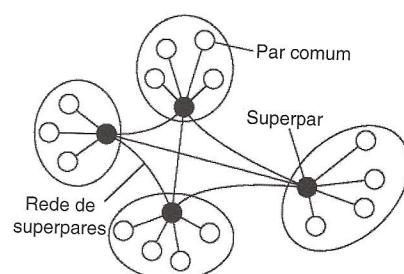


Figura 2.12 Organização hierárquica de nós em uma rede de superpares.

Em muitos casos, a relação cliente-superpar é fixa: sempre que um par comum se juntar à rede, ele se liga a um dos superpares e continua ligado até sair da rede. É óbvio que se espera que superpares sejam processos de longa vida com alta disponibilidade. Para compensar o comportamento potencialmente instável de um superpar, podem-se disponibilizar esquemas de segurança, como montar pares de cada superpar com um outro superpar para todos os superpares e requerer que os clientes se liguem a ambos.

Ter uma associação fixa com um superpar pode não ser sempre a melhor solução. Por exemplo, no caso de redes de compartilhamento de arquivo, talvez seja melhor que um cliente se ligue a um superpar que mantenha um índice de arquivos nos quais o cliente geralmente está interessado. Quando for esse o caso, são maiores as chances de que, no caso de um cliente estar procurando um arquivo específico, seu superpar saberá onde encontrá-lo.

Garbacki et al. (2005) descrevem um esquema relativamente simples no qual a relação cliente-superpar pode mudar à medida que clientes descobrem superpares melhores com os quais se associar. Em particular, um superpar que está retornando o resultado de uma operação de consulta recebe preferência sobre outros superpares.

Como vimos, redes peer-to-peer oferecem um meio flexível para os nós entrarem e saírem da rede. Contudo, as redes de superpares introduzem um novo problema, a saber, como selecionar os nós que estão qualificados para se tornar um superpar. Esse problema guarda estreita relação com o problema da seleção do líder, que discutiremos no Capítulo 6, quando voltarmos à qualificação de superpares em uma rede peer-to-peer.

2.2.3 Arquiteturas híbridas

Até aqui, focalizamos arquiteturas cliente-servidor e várias arquiteturas peer-to-peer. Muitos sistemas distribuídos combinam aspectos arquitetônicos, como já encontramos em redes de superpares. Nesta seção estudaremos algumas classes específicas de sistemas distribuídos nas quais soluções cliente-servidor são combinadas com arquiteturas descentralizadas.

Sistemas de servidor de borda

Uma classe importante de sistemas distribuídos organizada segundo uma arquitetura híbrida é formada por sistemas de servidor de borda. Esses sistemas são disponibilizados na Internet onde servidores são colocados ‘na borda’ da rede. Essa borda é formada pela fronteira entre as redes corporativas e a Internet propriamente dita, por exemplo, como fornecida por um provedor de serviço de Internet (Internet Service Provider — ISP). Da mesma maneira, quando usuários finais que estão em suas casas se conectam com a Internet por meio de seu ISP, pode-se considerar que o ISP reside na borda da Internet. Isso resulta na organização geral mostrada na Figura 2.13.

Usuários finais, ou clientes em geral, se conectam com a Internet por meio de um servidor de borda. A principal finalidade do servidor de borda é servir conteúdo, possivelmente após ter aplicado funções de filtragem e transcodificação. Mais interessante é o fato de que um conjunto de servidores de borda pode ser usado para otimizar distribuição de conteúdo e de aplicação. O modelo básico é aquele em que, para uma organização específica, um único servidor de borda age como um servidor de origem do qual se origina todo o conteúdo. Esse servidor pode usar outros servidores de borda para replicar páginas Web e coisas semelhantes (Leff et al., 2004; Nayate et al., 2004; Rabinovich e Spatscheck, 2002). Voltaremos a sistemas de servidor de borda no Capítulo 12, quando discutirmos soluções baseadas na Web.

Sistemas distribuídos colaborativos

Estruturas híbridas são disponibilizadas notavelmente em sistemas distribuídos colaborativos. A questão principal em muitos desses sistemas é conseguir dar a partida, para o que muitas vezes é disponibilizado um esquema cliente-servidor tradicional. Tão logo um nó se junta ao sistema, ele pode usar um esquema totalmente descentralizado para colaboração.

Para ficarmos no terreno concreto, em primeiro lugar vamos considerar o sistema de compartilhamento de arquivos BitTorrent (Cohen, 2003). O BitTorrent é um

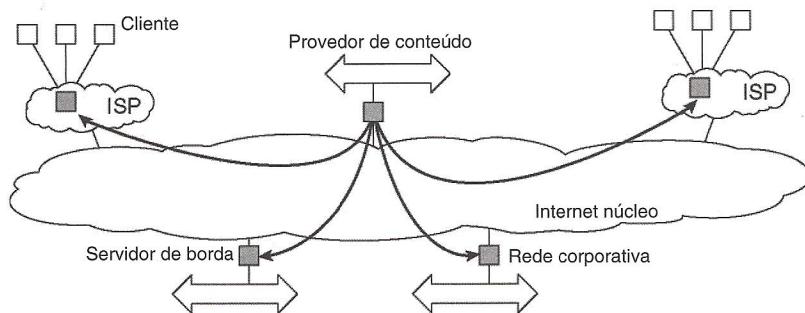


Figura 2.13 Visão da Internet como rede composta por um conjunto de servidores de borda.

sistema peer-to-peer de transferência (*download*) de arquivos. Seu funcionamento principal é mostrado na Figura 2.14. A idéia básica é que, quando um usuário final estiver procurando um arquivo, ele transfira porções do arquivo de outros usuários até que as porções transferidas possam ser montadas em conjunto, resultando no arquivo completo. Uma meta importante de projeto era garantir colaboração. Na maioria dos sistemas de compartilhamento de arquivo, uma fração significativa de participantes se limita a obter arquivos; porém, quanto ao mais, sua contribuição é quase nula (Adar e Huberman, 2000; Saroiu et al., 2003; Yang et al., 2005). Com essa finalidade, um arquivo só pode ser transferido quando o cliente que está transferindo estiver fornecendo conteúdo a mais alguém. Em breve voltaremos a esse comportamento toma-lá-dá-cá.

Para obter um arquivo, um usuário precisa acessar um diretório global, que é apenas um de alguns sites Web bem conhecidos. Tal diretório contém referências ao que denominamos arquivos *.torrent*. Um arquivo *.torrent* contém as informações necessárias para transferir um arquivo específico. Em particular, ele se refere a algo conhecido como **rastreador** — um servidor que está mantendo uma contabilidade precisa de nós *ativos* que têm o arquivo requisitado (porções dele). Um nó ativo é um nó que está transferindo um outro arquivo no momento em questão. É óbvio que haverá muitos rastreadores diferentes, embora, em geral, somente um único por arquivo (ou conjunto de arquivos).

Tão logo os nós tenham identificado de onde as porções podem ser transferidas, o nó que está transferindo se torna efetivamente ativo. Nesse ponto, ele será forçado a auxiliar outros: por exemplo, fornecendo porções do arquivo que está transferindo e que outros ainda não têm. Essa imposição resulta de uma regra muito simples: se o nó *P* perceber que o nó *Q* está enviando mais do que está recebendo, *P* pode decidir reduzir a taxa à qual ele envia dados a *Q*. Esse esquema funciona bem contanto que *P* tenha algo a enviar de *Q*. Por essa razão, muitas vezes os nós são supridos com referências a muitos outros nós, o que os coloca em uma posição melhor para negociar dados.

É claro que o BitTorrent combina soluções centralizadas e descentralizadas. Porém, ocorre que o gargalo do sistema são os rastreadores, o que não é surpresa.

Como outro exemplo, considere a rede colaborativa de distribuição de conteúdo Globule (Pierre e Van Steen, 2006). Essa rede é muito parecida com a arquitetura de servidor de borda que já mencionamos. Nesse caso, em vez de servidores de borda, são usuários finais — mas também organizações — que fornecem voluntariamente servidores Web aprimorados que sejam capazes de colaborar na replicação de páginas Web. Em sua forma mais simples, cada um desses servidores tem os seguintes componentes:

1. Um componente que pode redirecionar requisições de clientes a outros servidores.
2. Um componente para analisar padrões de acesso.
3. Um componente para gerenciar a replicação de páginas Web.

O servidor fornecido por Alice é o servidor Web que normalmente manipula o tráfego para o site Web de Alice. É denominado **servidor de origem** para esse site. Ele colabora com outros servidores, como o fornecido por Bob, para hospedar as páginas do site de Bob. Nesse sentido, a Globule é um sistema distribuído descentralizado. Requisições para o site Web de Alice são inicialmente repassadas a seu servidor, ponto em que elas podem ser redirecionadas para um dos outros servidores. Suporta-se também redirecionamento distribuído.

Todavia, a Globule ainda tem um componente centralizado na forma de seu **agente**. O agente é responsável pelo registro de servidores e por fazer com que a existência desses servidores seja conhecida por outros servidores. Os servidores se comunicam com o agente de modo completamente análogo ao que seria de esperar em um sistema cliente-servidor. Por razão de disponibilidade, o agente pode ser replicado, porém, como veremos mais adiante neste livro, esse tipo de replicação é amplamente aplicado para conseguir computação cliente-servidor confiável.

2.3 Arquiteturas *versus* Middleware

Quando consideramos as questões arquitetônicas que discutimos até aqui, uma pergunta que vem à nossa mente é onde o middleware se encaixa. Como discutimos no Capítulo 1, o middleware forma uma camada entre

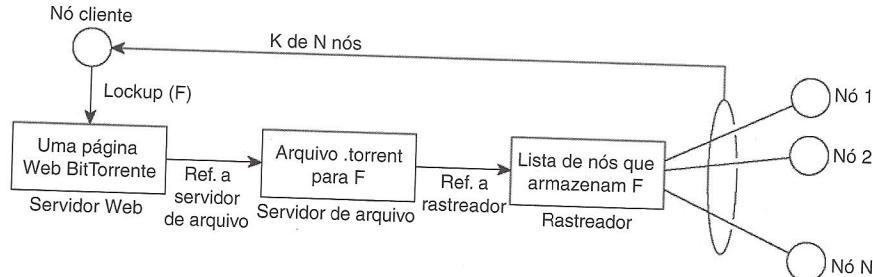


Figura 2.14 Funcionamento principal do BitTorrent [adaptado com permissão de Pouwelse et al. (2004)].

aplicações e plataformas distribuídas, como mostra a Figura 1.1. Uma finalidade importante é proporcionar um grau de transparência de distribuição, isto é, ocultar das aplicações, até certo ponto, a distribuição de dados, processamento e controle.

O que se vê comumente na prática é que, na realidade, sistemas de middleware seguem um estilo arquitetônico específico. Por exemplo, muitas soluções de middleware adotaram um estilo arquitetônico baseado em objetos, como o CORBA (OMG, 2004a). Outros, como o TIB/Rendezvous (TIBCO, 2005), fornecem middleware que segue o estilo arquitetônico baseado em eventos. Em capítulos posteriores encontraremos mais exemplos de estilos arquitetônicos.

Moldar o middleware de acordo com um estilo arquitetônico específico tem como benefício a simplificação do projeto de aplicações. Contudo, uma óbvia desvantagem é que o middleware pode não ser mais o ideal para aquilo que um desenvolvedor de aplicação tinha em mente. No início, CORBA oferecia somente objetos que podiam ser invocados por clientes remotos. Mais tarde, considerou-se que ter somente essa forma de interação era demasiadamente restritivo, portanto foram adicionados outros padrões de interação como passagem de mensagens. É óbvio que acrescentar novas características pode resultar facilmente em soluções de middleware inchadas.

Além disso, embora a intenção do middleware seja proporcionar transparência de distribuição, o pensamento geral é que soluções específicas deveriam ser adaptáveis a requisitos de aplicação. Uma solução para esse problema é fazer várias versões de um sistema de middleware no qual cada versão seja projetada para uma classe específica de aplicação.

Uma abordagem em geral considerada melhor é fazer sistemas de middleware de modo que sejam simples de configurar, adaptar e personalizar conforme necessário para uma aplicação. O resultado é que atualmente são desenvolvidos sistemas nos quais está sendo introduzida uma separação mais estrita entre políticas e mecanismos. Isso resultou em vários mecanismos com os quais pode-se modificar o comportamento do middleware (Sadjadi e McKinley, 2003). Vamos estudar algumas das abordagens comumente adotadas.

2.3.1 Interceptadores

Em termos de conceito, um **interceptador** nada mais é do que um constructo de software que interromperá o fluxo de controle usual e permitirá que seja executado um outro código (específico de aplicação). Fazer com que interceptadores sejam genéricos pode exigir esforço substancial de implementação, como ilustrado em Schmidt et al. (2000), e não fica claro se, em tais casos, a generalidade seria preferível à simplicidade e à aplicabilidade restrita. Além disso, há muitos casos em que ter somente facilidades limitadas de interceptação

melhorará o gerenciamento do software e do sistema distribuído como um todo.

Para ficar no campo concreto, considere a interceptação como suportada em muitos sistemas distribuídos baseados em objetos. A idéia básica é simples: um objeto *A* pode chamar um método que pertence a um objeto *B* enquanto este residir em uma máquina diferente de *A*. Como explicaremos detalhadamente mais adiante neste livro, tal invocação remota a objeto é realizada como uma abordagem de três etapas:

1. É oferecida ao objeto *A* uma interface local que é exatamente a mesma oferecida pelo objeto *B*. A simplesmente chama o método disponível naquela interface.
2. A chamada por *A* é transformada em uma invocação a objeto genérico, possibilitada por meio de uma interface geral de invocação de objeto oferecida pelo middleware na máquina em que *A* reside.
3. Por fim, a invocação a objeto genérico é transformada em uma mensagem que é enviada por meio de uma interface de rede de nível de transporte como oferecida pelo sistema operacional local de *A*.

Esse esquema é mostrado na Figura 2.15.

Após a primeira etapa, a chamada *B.faz_alguma_coisa(valor)* é transformada em uma chamada genérica como *invoke(B, &faz_alguma_coisa, valor)* com uma referência ao método de *B* e os parâmetros que acompanham a chamada. Agora imagine que o objeto *B* é replicado. Nesse caso, cada réplica deveria ser realmente invocada. Esse é claramente um ponto em que a interceptação pode ajudar. O que o **interceptador de nível de requisição** fará é simplesmente chamar *invoke(B, &faz_alguma_coisa, valor)* para cada uma das réplicas. O bom disso tudo é que o objeto *A* não precisa estar ciente da replicação de *B*, mas também o objeto middleware não precisa ter componentes especiais para lidar com essa chamada replicada. Só o interceptador de nível de requisição, que pode ser *adicionado* ao middleware, precisa saber da replicação de *B*.

No final, uma chamada a objeto remoto terá de ser enviada pela rede. Na prática, isso significa que a interface de envio de mensagens como a oferecida pelo sistema operacional local vai precisar ser invocada. Nesse nível, um **interceptador de nível de mensagem** pode ajudar na transferência da invocação ao objeto visado. Por exemplo, imagine que o parâmetro *valor*, na verdade, corresponda a um imenso arranjo de dados. Nesse caso, talvez seja sensato fragmentar os dados em partes menores e montá-los novamente no destino. Tal fragmentação pode melhorar o desempenho ou a confiabilidade. Mais uma vez, o middleware não precisa estar ciente dessa fragmentação; o interceptador de nível mais baixo manipulará transparentemente o resto da comunicação com o sistema operacional local.

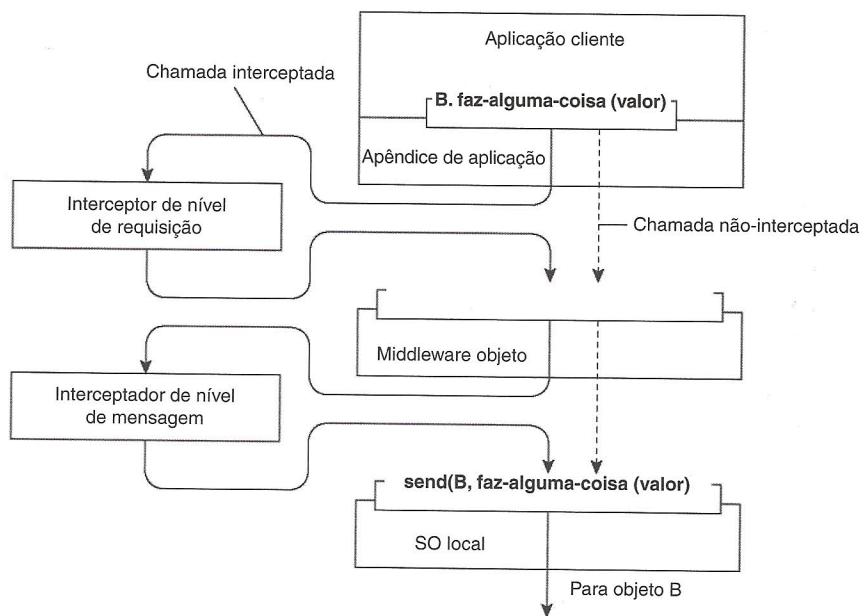


Figura 2.15 Utilização de interceptadores para manipular invocações de objeto remoto.

2.3.2 Abordagens gerais para o software adaptativo

O que os interceptadores realmente oferecem é um meio de adaptar o middleware. A necessidade de adaptação resulta do fato de que o ambiente no qual as aplicações distribuídas são executadas está sempre mudando. As mudanças incluem as resultantes da mobilidade, uma forte variância na qualidade de serviço de redes, hardware defeituoso e esgotamento da bateria, entre outras. Em vez de fazer com que as aplicações sejam responsáveis por reagir à mudança, essa tarefa é colocada no middleware.

Essas fortes influências do ambiente levaram muitos projetistas de middleware a considerar a construção de *software adaptativo*. Contudo, o software adaptativo não alcançou o sucesso que se esperava. Como muitos pesquisadores e desenvolvedores consideram que ele é um aspecto importante de sistemas distribuídos modernos, vamos lhe dar uma ligeira atenção. McKinley et al. (2004) distinguem três técnicas para chegar à adaptação de software:

1. Separação de interesses
2. Reflexão computacional
3. Projeto baseado em componente

A separação de interesses está relacionada com o modo tradicional de modularizar sistemas: separar as partes que implementam funcionalidade das que cuidam de outras coisas — conhecidas como *funcionalidades extras* — como confiabilidade, desempenho, segurança e assim por diante. Poderíamos argumentar que desenvolver middleware para aplicações distribuídas é, em grande parte, manipular funcionalidades extras independentemente de aplicações.

O principal problema é que não é fácil separar essas funcionalidades extras por meio de modularização. Apenas colocar segurança em um módulo separado não vai funcionar. Da mesma maneira, é difícil imaginar como a tolerância à falha pode ser isolada em uma caixa separada e vendida como um serviço independente. Separar e, na seqüência, entrelaçar esses interesses *cruzados* em um sistema (distribuído) é o tema principal abordado pelo **desenvolvimento de software orientado a aspecto** (Filman et al., 2005). Contudo, a orientação a aspecto ainda não foi aplicada com sucesso ao desenvolvimento de sistemas distribuídos de grande escala, e a expectativa é que ainda há um longo caminho a percorrer antes que ela chegue a esse estágio.

Reflexão computacional se refere à capacidade de um programa inspecionar a si mesmo e, se necessário, adaptar seu comportamento (Kon et al., 2002). A reflexão foi embutida em linguagens de programação, entre elas Java, e oferece uma facilidade poderosa para modificações em tempo de execução. Além disso, alguns sistemas de middleware proporcionam os meios para aplicar técnicas refletivas. Contudo, exatamente como no caso da orientação a aspecto, o middleware reflexivo ainda tem de provar que é uma ferramenta poderosa para gerenciar a complexidade de sistemas distribuídos de grande escala. Como mencionado por Blair et al. (2004), a aplicação da reflexão a um extenso domínio de aplicações ainda está por acontecer.

Por fim, o projeto baseado em componente suporta adaptação por meio de composição. Um sistema pode ser configurado estaticamente durante a elaboração do projeto ou dinamicamente em tempo de execução. O último requer suporte para ligação tardia, uma técnica que tem sido aplicada com sucesso em ambientes de linguagem de

programação, mas também em sistemas operacionais nos quais os módulos podem ser carregados e descarregados à vontade. Há pesquisas já bem adiantadas para permitir a seleção automática da melhor implementação de um componente em tempo de execução (Yellin, 2003); porém, mais uma vez, o processo permanece complexo para sistemas distribuídos, em especial quando se considera que a substituição de um componente implica saber que efeito ela terá sobre os outros componentes. Em muitos casos, os componentes são menos independentes do que podemos imaginar.

2.3.3 Discussão

Arquiteturas de software para sistemas distribuídos, encontradas consideravelmente como middleware, são volumosas e complexas. Em grande parte, tal volume e complexidade surgem da necessidade de ser geral no sentido de que é preciso proporcionar transparência de distribuição. Ao mesmo tempo, aplicações têm requisitos extrafuncionais específicos que conflitam com a meta de conseguir totalmente essa transparência. Esses requisitos conflitantes para generalidade e especialização resultaram em soluções de middleware de alta flexibilidade.

Todavia, o preço a pagar é a complexidade. Por exemplo, Zhang e Jacobsen (2004) informam um aumento de 50% no tamanho de determinado produto de software em apenas quatro anos após seu lançamento, enquanto o número de arquivos para aquele produto triplicou durante o mesmo período. É óbvio que essa não é uma direção encorajadora para seguir.

Considerando que, hoje em dia, praticamente todos os grandes sistemas de software têm de executar em um ambiente de rede, podemos nos perguntar se a complexidade de sistemas distribuídos é simplesmente um aspecto inerente da tentativa de fazer com que a distribuição seja transparente. Claro que assuntos como abertura são de igual importância, mas a necessidade de flexibilidade nunca foi tão predominante como no caso do middleware.

Coyler et al. (2003) argumentam que precisamos de um foco mais forte sobre a simplicidade (externa), um modo mais simples de construir middleware por componentes e independência da aplicação. Se qualquer uma das técnicas que já mencionamos é a solução, é algo sujeito a debate. Em particular, nenhuma das técnicas propostas até agora conquistou adoção maciça nem foi aplicada com sucesso a sistemas de grande escala.

A premissa subjacente é que precisamos de *software adaptativo* no sentido de que se deve permitir que o software mude à medida que o ambiente muda. Contudo, devemos questionar se a adaptação a um ambiente em mutação é uma boa razão para adotar a mudança de software. Hardware defeituoso, ataques contra a segurança, consumo de energia etc. parecem ser influências ambientais que podem, e devem, ser antecipadas por software.

O argumento mais forte e por certo o mais válido para suportar software adaptativo é que muitos sistemas distribuídos não podem ser desligados. Essa restrição exige soluções para substituir e atualizar componentes durante o funcionamento do sistema, mas não está claro se qualquer uma das soluções já propostas é a melhor para atacar o problema de manutenção.

Portanto, o que prevalece disso tudo é que sistemas distribuídos devem ser capazes de reagir a mudanças em seu ambiente, como trocar dinamicamente políticas para alocação de recursos. Todos os componentes de software para habilitar tal adaptação já estarão presentes. São os algoritmos contidos nesses componentes e que determinam o comportamento que mudam sua montagem. O desafio é deixar que tal comportamento reativo ocorra sem intervenção humana. Considera-se que essa abordagem funciona melhor quando se discute organização física de sistemas distribuídos ao serem tomadas decisões sobre onde colocar os componentes, por exemplo. Em seguida, discutiremos essas questões de arquitetura de sistemas.

2.4 Autogerenciamento em Sistemas Distribuídos

Sistemas distribuídos — e em especial seu middleware associado — precisam fornecer soluções gerais de blindagem contra aspectos indesejáveis inerentes a redes, de modo que possam suportar o maior número possível de aplicações. Por outro lado, a total transparência de distribuição não é algo que, na verdade, a maioria das aplicações quer, o que resulta em soluções específicas de aplicação que também precisam ser suportadas. Já argumentamos que, por essa razão, sistemas distribuídos deveriam ser adaptativos, mas especificamente quando se tratar de adaptar seu comportamento de execução, e não os componentes de software que eles compreendem.

Quando a adaptação precisa ser automática, observamos um forte intercâmbio entre arquiteturas de sistema e arquiteturas de software. Por um lado, precisamos organizar os componentes de um sistema distribuído de modo tal que seja possível fazer monitoração e ajustes, enquanto, por outro, precisamos decidir onde devem ser executados os processos que manipulam a adaptação.

Nesta seção, damos maior atenção à organização de sistemas distribuídos como sistemas de realimentação de controle de alto nível que permitem adaptação automática a mudanças. Esse fenômeno também é conhecido como **computação autônoma** (Kephart, 2003) ou **sistemas auto*** (Babaoglu et al., 2005). O último nome indica a variedade de modos de adaptação automática que estão sendo englobados: autogerenciador, auto-reparador, auto-configurador, auto-otimizador e assim por diante. Resolvemos utilizar simplesmente o nome autogerenciador como abrangente de suas muitas variantes.

2.4.1 O modelo de realimentação de controle

Há muitas visões diferentes sobre sistemas autogerenciadores, mas o que a maioria deles tem em comum, explícita ou implicitamente, é a premissa de que as adaptações ocorrem por meio de um ou mais **laços de realimentação de controle**. De acordo com esse fato, sistemas organizados por meio desses laços são denominados **sistemas de realimentação de controle**. Há muito que a realimentação de controle é aplicada em vários campos da engenharia, e seus fundamentos matemáticos também estão gradativamente encontrando seu espaço em sistemas de computação (Hellerstein et al., 2004; Diao et al., 2005). No caso de sistemas autogerenciadores, as questões arquitetônicas são, de início, mais interessantes. A idéia básica que fundamenta essa organização é bastante simples, como mostra a Figura 2.16.

O núcleo de um sistema de realimentação de controle é formado pelos componentes que precisam ser gerenciados. Adota-se como premissa que esses componentes sejam guiados por parâmetros de entrada que possam ser controlados, mas seu comportamento pode ser influenciado por todos os tipos de entradas *não controláveis*, também conhecidas como perturbações ou ruídos. Embora as perturbações freqüentemente virão do ambiente no qual um sistema distribuído está executando, pode perfeitamente bem ser o caso de uma interação de componentes não prevista provocar o comportamento inesperado.

Há, em essência, três elementos que formam o laço de realimentação de controle. Primeiro, o sistema em si precisa ser monitorado, o que implica que vários aspectos do sistema precisam ser medidos. Quando se trata de medir comportamento, em muitos casos é mais fácil falar do que fazer. Por exemplo, atrasos de viagens de ida e volta na Internet podem variar enlouquecedoramente e também depender do que, exatamente, está sendo medido. Nesses casos, estimar um atraso com precisão pode ser realmente difícil. As coisas se complicam ainda mais quando um nó A precisa estimar a latência

entre dois nós, B e C, completamente diferentes, sem poder se comportar como intruso em qualquer um dos dois nós. Por razões como essa, um laço de realimentação de controle em geral contém um **componente de estimativa de medição**.

Uma outra parte do laço de realimentação de controle analisa as medições e as compara com valores de referência. Esse **componente de análise de realimentação** forma o cerne do laço de realimentação porque conterá os algoritmos que decidem possíveis adaptações.

O último grupo de componentes consiste em vários mecanismos para influenciar diretamente o comportamento do sistema. Pode haver muitos mecanismos diferentes: colocação de réplicas, mudança de prioridades de escalonamento, troca dinâmica de serviços, movimentação de dados por razões de disponibilidade, redirecionamento de requisições para servidores diferentes etc. O componente de análise vai precisar estar ciente desses mecanismos e de seu efeito (esperado) sobre o comportamento do sistema. Portanto, ele acionará um ou vários mecanismos para, na seqüência, observar o efeito.

Uma observação interessante é que o laço de realimentação de controle também se ajusta ao gerenciamento manual de sistemas. A principal diferença é que o componente de análise é substituído por administradores humanos. Contudo, para gerenciar adequadamente qualquer sistema distribuído, esses administradores precisarão de equipamentos de monitoração decentes, bem como de mecanismos decentes para controlar o comportamento do sistema. Deve ficar claro que a análise adequada de dados medidos e o acionamento das ações corretas é que tornam o desenvolvimento de sistemas autogerenciadores tão difícil.

É preciso salientar que a Figura 2.16 mostra a organização *lógica* de um sistema autogerenciador e, por isso, corresponde ao que vimos quando discutimos arquiteturas de software. Entretanto, a organização *física* pode ser muito diferente. Um exemplo é que o componente de análise pode ser totalmente distribuído pelo sistema. Da mesma maneira, as medições de desempe-

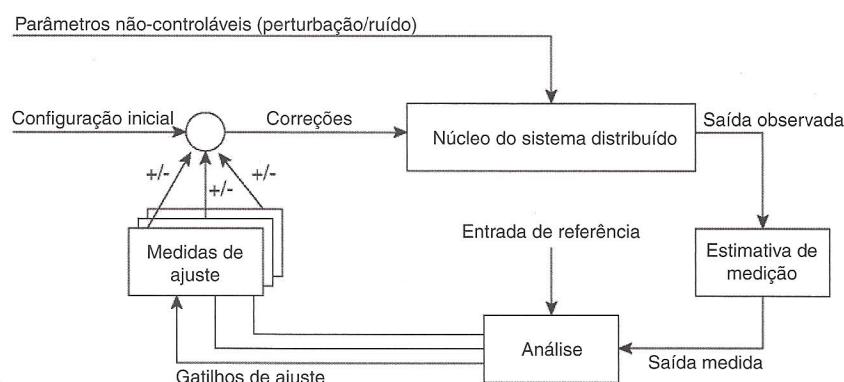


Figura 2.16 Organização lógica de um sistema de realimentação de controle.

não costumam ser realizadas em cada máquina que faz parte do sistema distribuído.

Estudemos então alguns exemplos concretos de como monitorar, analisar e corrigir sistemas distribuídos automaticamente. Esses exemplos também ilustrarão a distinção entre organização lógica e organização física.

2.4.2 Exemplo: monitoração de sistemas com Astrolabe

Como nosso primeiro exemplo, consideramos o Astrolabe (Van Renesse et al., 2003), sistema que pode suportar monitoração geral de sistemas distribuídos muito grandes. No contexto de sistemas autogerenciadores, o Astrolabe deve ser posicionado como uma ferramenta geral para a observação do comportamento de sistemas. Seus resultados podem ser utilizados para alimentar um componente de análise a fim de decidir ações corretivas.

O Astrolabe organiza um grande conjunto de hospedeiros em uma hierarquia de zonas. As zonas de nível mais baixo consistem em apenas um único hospedeiro, que são subsequentemente agrupados em zonas de tamanho crescente. A zona de nível mais alto abrange todos os hospedeiros. Cada hospedeiro executa um processo Astrolabe, denominado *agente*, que colhe informações nas zonas em que aquele hospedeiro está contido. O agente também se comunica com outros agentes com a finalidade de propagar informações de zona por todo o sistema.

Cada hospedeiro mantém um conjunto de *atributos* para colher informações locais. Um hospedeiro, por exemplo, pode monitorar arquivos específicos que armazena, a utilização que faz dos recursos e assim por diante. Somente os atributos mantidos diretamente por hospedeiros, isto é, no nível mais baixo da hierarquia, podem ser escritos. Cada zona também pode ter um conjunto de atributos, mas os valores desses atributos são computados com base em valores das zonas de nível mais baixo.

Considere o seguinte exemplo simples mostrado na Figura 2.17, com três hospedeiros, A, B e C, agrupados em uma zona. Cada máquina monitora o endereço IP, a carga de CPU, a memória livre disponível e a quantidade de processos ativos. Cada um desses atributos pode ser escrito diretamente usando informações locais de cada hospedeiro. No nível de zona, somente podem ser colhidas informações agregadas, como a carga média da CPU ou a quantidade média de processos ativos.

A Figura 2.17 mostra como a informação reunida por cada máquina pode ser vista como um registro em um banco de dados e que esses registros em conjunto formam uma relação (tabela). Essa representação é feita de propósito: é o modo como o Astrolabe vê todos os dados colhidos. Taisvez, informações por zona só podem ser computadas de acordo com registros básicos mantidos por hospedeiros.

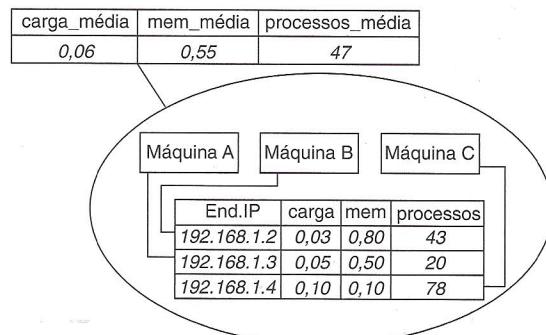


Figura 2.17 Coleta de dados e agregação de informações em Astrolabe.

Informações agregadas são obtidas por funções de agregação programáveis, que são muito semelhantes a funções disponíveis na linguagem de bancos de dados relacionais SQL. Por exemplo, considerando que a informação de hospedeiro da Figura 2.17 seja mantida em uma tabela local denominada *hostinfo*, poderíamos coletar a quantidade média de processos para a zona que contém as máquinas A, B e C por meio da simples consulta SQL

```
SELECT AVG(processos) AS processos_média FROM hostinfo
```

Em combinação com alguns aprimoramentos da SQL, não é difícil imaginar que podem ser formuladas mais consultas informativas.

Consultas como essa são avaliadas continuamente por cada agente que executa em cada hospedeiro. É óbvio que isso só é possível se as informações de zona forem propagadas para todos os nós que abrangem o Astrolabe. Com essa finalidade, um agente que executa em um hospedeiro é responsável por computar partes das tabelas de suas zonas associadas. Registros sobre os quais ele não tem nenhuma responsabilidade computacional lhe são enviados de tempos em tempos por meio de um procedimento de troca simples, porém efetivo, denominado *gossiping (mexerico)*. Protocolos de gossiping serão discutidos detalhadamente no Capítulo 4. Do mesmo modo, um agente também passará resultados computados para outros agentes.

O resultado dessa troca de informações é que, a certa altura, todos os agentes necessários para auxiliar na obtenção de alguma informação agregada verão o mesmo resultado, contanto que não ocorra nenhuma alteração nesse meio-tempo.

2.4.3 Exemplo: diferenciação de estratégias de replicação em Globule

Vamos agora estudar a Globule, rede colaborativa de distribuição de conteúdo (Pierre e Van Steen, 2006). A Globule depende da colocação de servidores de usuário

final na Internet e da colaboração entre esses servidores para otimizar desempenho por meio de replicação de páginas Web. Com essa finalidade, cada servidor de origem — isto é, o servidor responsável pela manipulação das atualizações de um site Web específico — monitora padrões de acesso por página. Padrões de acesso são expressos como operações de leitura e escrita para uma página, sendo que cada operação recebe um selo de tempo e é registrada pelo servidor de origem para aquela página.

Em sua forma mais simples, a Globule parte da premissa de que a Internet pode ser vista como um sistema de servidor de borda, conforme explicamos antes. Em particular, ela admite que as requisições sempre podem ser passadas por meio de um servidor de borda adequado, como mostra a Figura 2.18. Esse modelo simples permite que um servidor de origem veja o que teria acontecido se ele tivesse colocado uma réplica em um servidor de borda específico. Por um lado, colocar uma réplica mais perto dos clientes melhoraria a latência percebida do cliente, mas isso induziria tráfego entre o servidor de origem e o servidor de borda, de modo a manter uma réplica consistente com a página original.

Quando um servidor de origem recebe uma requisição para uma página, ele registra o endereço IP de onde a requisição se originou e consulta o ISP ou a rede corporativa associada com aquela requisição usando o serviço WHOIS de Internet (Deutsch et al., 1995). Então, o servidor de origem procura o servidor de réplica existente que estiver mais próximo e que poderia agir como servidor de borda para aquele cliente; na seqüência, computa a latência até aquele servidor junto com a largura de banda máxima. Em sua configuração mais simples, a Globule adota como premissa que a latência entre o servidor de réplica e a máquina do usuário requisitante é desprezível e, da mesma maneira, que a largura de banda entre os dois é abundante.

Tão logo tenham sido colhidas requisições suficientes para uma página, o servidor de origem realiza uma simples análise do tipo ‘o que aconteceria se’. Tal análise se resume a avaliar diversas políticas de replicação, considerando que uma política descreve para onde uma página especificada é replicada e como essa página é

mantida consistente. Cada política de replicação incorre em um custo que pode ser expresso como uma função linear simples:

$$\text{custo} = (w_1 \times m_1) + (w_2 \times m_2) + \dots + (w_n \times m_n)$$

onde m_k denota uma métrica de desempenho e w_k é o peso que indica a importância dessa métrica. Métricas de desempenho típicas são os atrasos agregados entre um cliente e um servidor de réplica referentes ao retorno de cópias de páginas Web; a largura de banda total consumida entre o servidor de origem e um servidor de réplica para manter uma réplica consistente e a quantidade de cópias velhas que são retornadas (tolera-se que sejam retornadas) a um cliente (Pierre et al., 2002).

Considere, por exemplo, que o atraso típico entre o instante em que o cliente C emite uma requisição e o instante em que aquela página é retornada pelo melhor servidor de réplica é d_C ms. Observe que o melhor servidor de réplica é determinado por uma política de replicação. Vamos denotar por m_1 o atraso agregado durante certo período, isto é, $m_1 = \sum d_C$. Se o servidor de origem quiser otimizar a latência percebida do cliente, escolherá um valor relativamente alto para w_1 . Como consequência, somente as políticas que realmente minimizam m_1 mostrarão que têm custo relativamente baixo.

Em Globule, um servidor de origem avalia periodicamente algumas dezenas de políticas de replicação usando uma simulação guiada por rastreador para cada página Web em separado. Por essas simulações, a melhor política é selecionada e, na seqüência, imposta. Isso pode implicar que novas réplicas sejam instaladas em diferentes servidores de borda ou que é escolhido um modo diferente de manter as réplicas consistentes. A coleta de amostras pelo rastreador, a avaliação de políticas de replicação e a imposição de uma política selecionada são todas feitas automaticamente.

Há várias questões sutis que precisam ser tratadas. Uma razão é que não está claro quantas requisições precisam ser colhidas antes de se poder realizar uma avaliação da política corrente. Para explicar, suponha que no tempo T_i o servidor de origem seleciona a política p para o pró-

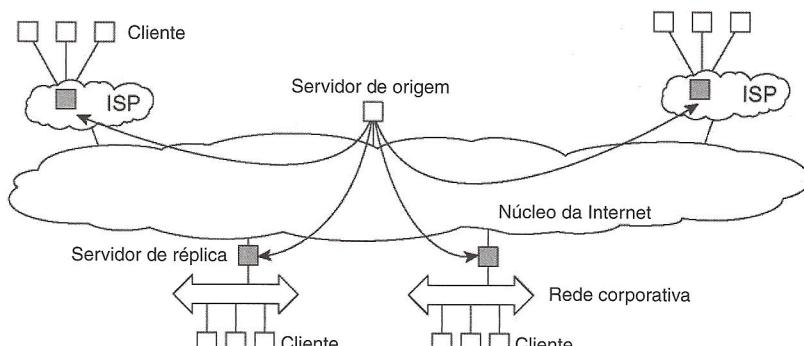


Figura 2.18 Modelo de servidor de borda adotado pela Globule.

ximo período até T_{i+1} . Essa seleção ocorre com base em uma série de requisições passadas que foram emitidas entre T_{i-1} e T_i . Claro que, em retrospectiva no tempo T_{i+1} , o servidor pode chegar à conclusão de que deveria ter selecionado a política p^* dadas as requisições que realmente foram emitidas entre T_i e T_{i+1} . Se p^* for diferente de p , então a seleção de p em T_i estava errada.

Ocorre que a porcentagem de previsões erradas depende dos comprimentos das séries de requisições (denominadas comprimento da amostra) que são usadas para prever e selecionar a próxima política. Essa dependência está esboçada na Figura 2.19. O gráfico mostra que o erro na previsão da melhor política aumenta se a amostragem não for longa o suficiente. Isso é facilmente explicado pelo fato de que precisamos de quantidade suficiente de requisições para fazer uma avaliação adequada.

Contudo, o erro também aumenta se usarmos uma quantidade demasiadamente grande de requisições. A razão para isso é que uma amostragem muito longa captanta mudanças em padrões de acesso que prever a melhor política a seguir fica difícil, se não impossível. Esse fenômeno é bem conhecido e é análogo a tentar prever o tempo que fará amanhã examinando o que aconteceu durante os cem anos imediatamente precedentes. Uma previsão muito melhor pode ser feita apenas com a consulta do passado recente.

Também pode-se achar o comprimento de amostragem ótimo automaticamente. Deixamos como exercício a proposição de uma solução para esse problema.

2.4.4 Exemplo: gerenciamento automático de conserto de componente em Jade

Para fazer a manutenção de clusters de computadores, nos quais cada computador executa servidores sofisticados, torna-se importante amenizar problemas de gerenciamento. Uma abordagem que pode ser aplicada a servidores construídos segundo uma tecnologia baseada em componentes é detectar falhas de componentes e substituí-los automaticamente. O sistema Jade segue essa abordagem (Bouchenak et al., 2005). Nós a descreveremos brevemente nesta seção.

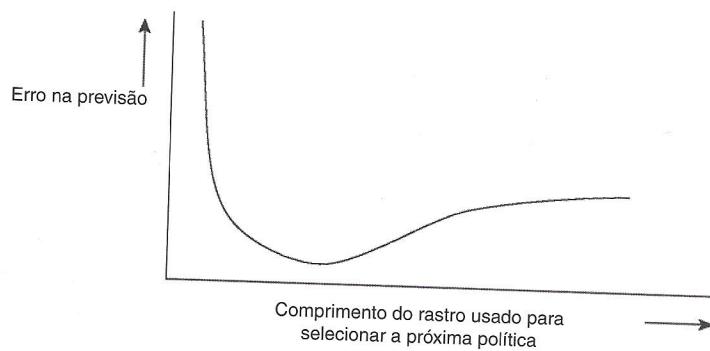


Figure 2.19 A dependência entre precisão da previsão e comprimento do rastro.

O sistema Jade é construído sobre o modelo de componente Fractal, implementação Java de uma estrutura que permite aos componentes serem adicionados e removidos em tempo de execução (Bruneton et al., 2004). Em Fractal, um componente pode ter dois tipos de interfaces. Uma **interface de servidor** é usada para chamar métodos que são implementados por aquele componente. Uma **interface de cliente** é usada por um componente para chamar outros componentes. Os componentes são conectados uns aos outros por interfaces de **vinculação**. Um exemplo é a interface de cliente do componente C_1 , que pode ser vinculada à interface de servidor do componente C_2 . Uma vinculação primitiva significa que uma chamada a uma interface de cliente resulta diretamente na chamada da interface vinculada de servidor.

No caso de vinculação composta, a chamada pode prosseguir por um ou mais componentes; por exemplo, porque as interfaces de cliente e servidor não combinam e é preciso algum tipo de conversão. Uma outra razão pode ser que os componentes conectados estejam em máquinas diferentes.

O Jade usa a noção de **domínio de gerenciamento de conserto**. Tal domínio consiste em uma quantidade de nós na qual cada nó representa um servidor junto com os componentes que são executados por aquele servidor. Há um gerente de nó em separado que é responsável por adicionar e remover nós do domínio. O gerente de nó pode ser replicado para garantir alta disponibilidade.

Cada nó é equipado com detectores de falha que monitoram a saúde de um nó ou de um de seus componentes e informam quaisquer falhas ao gerente de nó. Normalmente esses detectores observam mudanças excepcionais no estado do componente, a utilização de recursos e a falha propriamente dita de um componente. Observe que, na verdade, a última pode significar que a máquina está avariada.

Quando uma falha é detectada, é iniciado um procedimento de conserto. Tal procedimento é guiado por uma política de conserto, parcialmente executada pelo gerente de nó. Políticas são declaradas explicitamente e executadas dependendo da falha detectada. Por exemplo, suponha que uma falha de nó tenha sido detectada. Nesse caso, a

política de conserto pode prescrever que devem ser executadas as seguintes etapas:

1. Encerre todas as vinculações entre um componente que está em um nó sem falha e o componente que está no nó que acabou de falhar.
2. Requisite ao gerente de nó que inicie e adicione um novo nó ao domínio.
3. Configure o novo nó exatamente com os mesmos componentes do nó que falhou.
4. Restabeleça todas as vinculações que foram encerradas anteriormente.

Nesse exemplo, a política de conserto é simples e só funcionará quando nenhum dado crucial tiver se perdido (os componentes avariados são conhecidos como **sem estado**).

A abordagem seguida pelo Jade é um exemplo de autogerenciamento: quando é detectada uma falha, uma política de conserto é automaticamente executada para levar o sistema como um todo a um estado no qual estava antes da avaria. Por ser um sistema baseado em componente, esse conserto automático requer suporte específico para permitir que componentes sejam adicionados e removidos em tempo de execução. Em geral não é possível transformar aplicações herdadas em sistemas de autogerenciamento.

2.5 Resumo

Sistemas distribuídos podem ser organizados de modos diferentes. Podemos fazer uma distinção entre arquitetura de software e arquitetura de sistema. A última considera onde os componentes que constituem um sistema distribuído estão colocados nas várias máquinas. A primeira se preocupa mais com a organização lógica do software: como os componentes interagem, de que modos eles podem ser estruturados, como podem ficar independentes e assim por diante.

Uma idéia fundamental quando falamos sobre arquiteturas é o estilo arquitetônico. Um estilo reflete o princípio básico que é seguido na organização da interação entre os componentes de software que compreendem um sistema distribuído. Entre os estilos importantes estão disposição em camadas, orientação a objetos, orientação a eventos e orientação a espaço de dados.

Há variadas organizações de sistemas distribuídos. Uma classe importante é aquela em que as máquinas são divididas em clientes e servidores. Um cliente envia uma requisição a um servidor, que então produzirá um resultado que é retornado ao cliente. A arquitetura cliente-servidor reflete o modo tradicional de modularização de software pelo qual um módulo chama as funções disponíveis em um outro módulo. Colocando componentes diferentes em máquinas diferentes, obtemos

uma distribuição física natural de funções por um conjunto de máquinas.

Arquiteturas cliente-servidor costumam apresentar alto grau de centralização. Em arquiteturas descentralizadas, freqüentemente vemos um papel igual desempenhado pelos processos que constituem um sistema distribuído, também conhecidos como sistemas peer-to-peer. Em sistemas peer-to-peer, os processos são organizados em uma rede de sobreposição, que é uma rede lógica na qual todo processo tem uma lista local de outros pares com os quais ele pode se comunicar. A rede de sobreposição pode ser estruturada, caso em que são disponibilizados esquemas determinísticos para rotear mensagens entre processos. Em redes não estruturadas, a lista de pares é mais ou menos aleatória, o que implica que é preciso disponibilizar algoritmos de busca para localizar dados ou outros processos.

Como alternativa, foram desenvolvidos sistemas autogerenciadores. Até certo ponto, esses sistemas fundem idéias de arquiteturas de sistema e de software. Sistemas autogerenciadores podem ser organizados, de modo geral, como laços de realimentação de controle. Esses laços contêm um componente de monitoração pelo qual é medido o comportamento de um sistema distribuído, um componente de análise para verificar se alguma coisa precisa ser ajustada e um conjunto de vários instrumentos para mudar o comportamento. Laços de realimentação de controle podem ser integrados a sistemas distribuídos em vários lugares. Ainda é preciso muita pesquisa antes de se chegar a um entendimento comum de como tais laços devem ser desenvolvidos e disponibilizados.

Problemas

1. Se um cliente e um servidor forem colocados longe um do outro, podemos ver a latência de rede dominar o desempenho global. Como podemos atacar esse problema?
2. O que é uma arquitetura cliente-servidor de três divisões?
3. Qual é a diferença entre uma distribuição vertical e uma distribuição horizontal?
4. Considere uma cadeia de processos P_1, P_2, \dots, P_n implementando uma arquitetura cliente-servidor multidivida. O processo P_i é cliente do processo P_{i+1} , e P_i retornará uma resposta a P_{i-1} somente após receber uma resposta de P_{i+1} . Quais são os principais problemas dessa organização quando se examina o desempenho de requisição-resposta no processo P_1 ?
5. Em uma rede de sobreposição estruturada, mensagens são roteadas de acordo com a topologia da sobreposição. Cite uma importante desvantagem dessa abordagem.

5. Considere a rede CAN da Figura 2.8. Como você rotearia uma mensagem do nó cujas coordenadas são $(0,2;0,3)$ até o nó cujas coordenadas são $(0,9;0,6)$?
6. Considerando que um nó em CAN conheça as coordenadas de seus vizinhos imediatos, uma política de roteamento razoável seria repassar uma mensagem ao nó mais próximo na direção do destino. Quão boa é essa política?
7. Considere uma rede de sobreposição não estruturada na qual cada nó escolhe aleatoriamente c vizinhos. Se P e Q forem ambos vizinhos de R , qual é a probabilidade de também serem vizinhos um do outro?
8. Considere, mais uma vez, uma rede de sobreposição não estruturada na qual cada nó escolhe aleatoriamente c vizinhos. Para procurar um arquivo, um nó envia uma requisição para todos os seus vizinhos e requisita que estes repassem a requisição mais uma vez. Quantos nós serão alcançados?
9. Nem todo nó em uma rede peer-to-peer deve se tornar um superpar. Cite requisitos razoáveis que um superpar deve cumprir.
10. Considere um sistema BitTorrent no qual cada nó tem um enlace de saída com uma largura de banda de capacidade B_{out} e um enlace de entrada com uma capacidade B_{in} . Alguns desses nós, denominados ‘sementes’, oferecem voluntariamente arquivos para serem transferidos por outros. Qual é a capacidade máxima de transferência de um cliente BitTorrent se admitirmos que ele pode contatar no máximo uma semente por vez?
11. Dê um argumento técnico interessante para explicar por que a política toma-lá-dá-cá, como usada em BitTorrent, está longe de ser ótima para compartilhamento de arquivos na Internet.
12. Demos dois exemplos de utilização de interceptadores em middleware adaptativo. Cite outros exemplos que lhe venham à mente.
13. Até que ponto interceptadores são dependentes do middleware em que são disponibilizados?
14. Carros modernos estão repletos de dispositivos eletrônicos. Dê alguns exemplos de sistemas de realimentação de controle em carros.
15. Dê um exemplo de um sistema autogerenciador no qual o componente de análise está completamente distribuído ou até mesmo oculto.
16. Proponha uma solução para determinar automaticamente o melhor comprimento de amostragem para prever políticas de replicação em Globule.
17. (Tarefa de laboratório) Usando software existente, projete e implemente um sistema baseado em BitTorrent para distribuir arquivos a muitos clientes com base em um único e poderoso servidor. Simplifique as coisas utilizando um servidor Web padrão que possa funcionar como rastreador.