CAPÍTULO

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 **18** 19 20 21 22 23 24 25 26

Engenharia de software distribuído

Objetivos

O objetivo deste capítulo é introduzir engenharia de sistemas distribuídos e arquiteturas de sistemas distribuídos. Com a leitura deste capítulo, você:

- conhecerá as questões fundamentais que devem ser consideradas ao se projetarem e implementarem sistemas de software distribuídos;
- entenderá o modelo de computação cliente-servidor e a arquitetura em camadas de sistemas cliente-servidor;
- terá sido apresentado aos padrões mais comumente usados em arquiteturas de sistemas distribuídos e conhecerá os tipos de sistema para os quais cada arquitetura é mais aplicável;
- compreenderá o conceito de software como um serviço, fornecendo acesso baseado na Web para sistemas de aplicações implantados remotamente.

Conteúdo

Praticamente todos os grandes sistemas computacionais agora são sistemas distribuídos. Um sistema distribuído é um sistema que envolve vários computadores, em contraste com sistemas centralizados, em que todos os componentes do sistema executam em um único computador. Tanenbaum e Van Steen (2007) definem um sistema distribuído como:

... uma coleção de computadores independentes que aparece para o usuário como um único sistema coerente.

Obviamente, a engenharia de sistemas distribuídos tem muito em comum com a engenharia de qualquer outro software. No entanto, existem questões específicas que precisam ser consideradas ao projetar esse tipo de sistema. Estas surgem porque os componentes do sistema podem ser executados em computadores gerenciados de forma independente e porque eles se comunicam por meio de uma rede.

Coulouris et al. (2005) identificam as seguintes vantagens da utilização de uma abordagem distribuída de desenvolvimento de sistemas:

1. *Compartilhamento de recursos.* Um sistema distribuído permite o compartilhamento de recursos de hardware e software — tais como discos, impressoras, arquivos e compiladores — que estão associados em computadores em uma rede.
2. *Abertura.* Geralmente, os sistemas distribuídos são sistemas abertos, o que significa que são projetados para protocolos-padrão que permitem que os equipamentos e softwares de diferentes fornecedores sejam combinados.

3. *Concorrência.* Em um sistema distribuído, vários processos podem operar simultaneamente em computadores separados, na rede. Esses processos podem (mas não precisam) comunicar-se uns com os outros durante seu funcionamento normal.
4. *Escalabilidade.* Em princípio, pelo menos, os sistemas distribuídos são escaláveis, assim, os recursos do sistema podem ser aumentados pela adição de novos recursos para fazer face às novas exigências do sistema. Na prática, a rede que liga os computadores individuais ao sistema pode limitar a escalabilidade deste.
5. *Tolerância a defeitos.* A disponibilidade de vários computadores e o potencial para replicar as informações significa que os sistemas distribuídos podem ser tolerantes com algumas falhas de hardware e software (veja o Capítulo 13). Na maioria dos sistemas distribuídos, um serviço de má qualidade pode ser fornecido quando ocorrem falhas; a perda total do serviço só ocorre quando há uma falha de rede.

Para sistemas organizacionais de grande porte, essas vantagens significam que os sistemas distribuídos substituíram, em grande medida, os sistemas legados de *mainframe* que foram desenvolvidos na década de 1990. No entanto, existem muitos sistemas computacionais de aplicação pessoal (por exemplo, sistemas de edição de foto) que não são distribuídos e que executam em um único computador. A maioria dos sistemas embutidos também é de sistemas de processador único.

Os sistemas distribuídos são, inerentemente, mais complexos que os sistemas centralizados, o que os torna mais difíceis para projetar, implementar e testar. É mais difícil compreender as propriedades emergentes de sistemas distribuídos por causa da complexidade das interações entre os componentes do sistema e sua infraestrutura. Por exemplo, em vez de o desempenho do sistema depender da velocidade de execução de um processador, ele depende da largura da banda de rede, da carga de rede e da velocidade de todos os computadores que fazem parte do sistema. Mover os recursos de uma parte do sistema para outra pode afetar significativamente o desempenho do sistema.

Além disso, como todos os usuários da internet sabem, sistemas distribuidos têm respostas imprevisíveis. O tempo de resposta depende da carga geral sobre o sistema, sua arquitetura e a carga de rede. Como todos esses podem mudar em um curto período, o tempo necessário para responder a uma solicitação do usuário varia drasticamente de uma solicitação para outra.

O mais importante desenvolvimento que tem afetado os sistemas de software distribuído, nos últimos anos, é a abordagem orientada a serviços. Grande parte deste capítulo se concentra nas questões gerais de sistemas distribuídos, mas discuto a noção de aplicações implantadas como serviços na Seção 18.4. Esse material complementa o material do Capítulo 19, que se concentra em serviços como componentes de uma arquitetura orientada a serviços e questões mais gerais de engenharia de software orientada a serviços.

18.1 Questões sobre sistemas distribuídos

Conforme discutido na introdução deste capítulo, os sistemas distribuídos são mais complexos do que os executados em um único processador. Esta complexidade surge porque é praticamente impossível ter um modelo de controle *top-down* desses sistemas. Muitas vezes, os nós do sistema que fornecem a funcionalidade são sistemas independentes com nenhuma autoridade sobre eles. A rede conectando esses nós é um sistema gerenciado separadamente. Esse é um sistema complexo em si mesmo, que não pode ser controlado pelos proprietários dos sistemas que usam a rede. Portanto, essa é uma imprevisibilidade inerente à operação de sistemas distribuídos, a qual deve ser considerada pelo projetista do sistema.

Algumas das questões mais importantes de projeto, que devem ser consideradas nos sistemas distribuídos, são:

1. *Transparência.* Em que medida o sistema distribuído deve aparecer para o usuário como um único sistema? Quando é útil aos usuários entender que o sistema é distribuído?
2. *Abertura.* Um sistema deveria ser projetado usando protocolos-padrão que ofereçam suporte à interoperabilidade ou devem ser usados protocolos mais especializados que restrinjam a liberdade do projetista?
3. *Escalabilidade.* Como o sistema pode ser construído para que seja escalável? Ou seja, como todo o sistema poderia ser projetado para que sua capacidade possa ser aumentada em resposta às crescentes exigências feitas ao sistema?
4. *Proteção.* Como podem ser definidas e implementadas as políticas de proteção que se aplicam a um conjunto de sistemas gerenciados independentemente?
5. *Qualidade de serviço.* Como a qualidade do serviço que é entregue aos usuários do sistema deve ser especificada e como o sistema deve ser implementado para oferecer uma qualidade aceitável e serviço para todos os usuários?

6. *Gerenciamento de falhas.* Como as falhas do sistema podem ser detectadas, contidas (para que elas tenham efeitos mínimos em outros componentes do sistema) e reparadas?

Em um mundo ideal, o fato de um sistema ser distribuído seria transparente para os usuários. Isso significa que os usuários veriam o sistema como um único sistema cujo comportamento não é afetado pela maneira como o sistema é distribuído. Na prática, isso é impossível de se alcançar. O controle central de um sistema distribuído é impossível, e, como resultado, os computadores individuais em um sistema podem ter comportamentos diferentes em momentos diferentes. Além disso, como sempre leva um tempo determinado para os sinais viajarem através de uma rede, os atrasos na rede são inevitáveis. O comprimento desses atrasos depende da localização dos recursos no sistema, da qualidade da conexão da rede do usuário e da carga de rede.

A abordagem de projeto para atingir a transparência depende de se criarem abstrações dos recursos em um sistema distribuído de modo que a realização física desses recursos possa ser alterada sem a necessidade de alterações no sistema de aplicação. O *middleware* (discutido na Seção 18.1.2) é usado para mapear os recursos lógicos referenciados por um programa para os recursos físicos reais e para gerenciar as interações entre esses recursos.

Na prática, é impossível fazer um sistema completamente transparente e, geralmente, os usuários estão conscientes de que estão lidando com um sistema distribuído. Assim, você pode decidir que é melhor expor a distribuição aos usuários; eles, por sua vez, podem ser preparados para algumas das consequências da distribuição, como atrasos na rede, falhas de nó remoto etc.

Os sistemas distribuídos abertos são sistemas construídos de acordo com normas geralmente aceitas. Isso significa que os componentes de qualquer fornecedor podem ser integrados ao sistema e podem interoperar com outros componentes do sistema. Atualmente, no nível de rede, com sistemas se conformando aos protocolos de Internet, a abertura é tida como confirmada, mas no nível de componente, a abertura ainda não é universal. A abertura implica que os componentes de sistema possam ser desenvolvidos independentemente em qualquer linguagem de programação e, se estas estiverem em conformidade com as normas, funcionarão com outros componentes.

O padrão CORBA (POPE, 1997) desenvolvido na década de 1990, tinha esse objetivo, mas nunca alcançou uma massa crítica de usuários. Em vez disso, muitas empresas optaram por desenvolver sistemas usando padrões proprietários para componentes de empresas, como a Sun e a Microsoft. Estes forneciam melhores implementações e suporte de software, além de melhor suporte de longo prazo para os protocolos industriais.

Os padrões de *web services* (discutidos no Capítulo 19) para arquiteturas orientadas a serviços foram desenvolvidos para serem padrões abertos. No entanto, existe uma significativa resistência a eles por causa de sua ineficiência. Alguns desenvolvedores de sistemas baseados em serviços optaram pelos chamados protocolos RESTful porque estes têm um *overhead* inerentemente mais baixo do que os protocolos de *web services*.

A escalabilidade de um sistema reflete sua capacidade de oferecer um serviço de alta qualidade, uma vez que aumenta a demanda de sistema. Neuman (1994) identifica três dimensões da escalabilidade:

1. *Tamanho.* Deve ser possível adicionar mais recursos a um sistema para lidar com um número crescente de usuários.
2. *Distribuição.* Deve ser possível dispersar geograficamente os componentes de um sistema sem comprometer seu desempenho.
3. *Capacidade de gerenciamento.* É possível gerenciar um sistema à medida que ele aumenta de tamanho, mesmo que partes dele estejam localizadas em organizações independentes.

Em termos de tamanho, existe uma distinção entre escalamento para cima e escalamento para fora. Escalamento para cima significa a substituição de recursos no sistema por recursos mais poderosos. Por exemplo, você pode aumentar a memória em um servidor de 16 GB para 64 GB. Escalamento para fora significa adicionar recursos ao sistema (por exemplo, um servidor Web extra para trabalhar ao lado de um servidor existente). Frequentemente, o escalamento para fora é mais efetivo do que o escalamento para cima, mas, geralmente, significa que o sistema precisa ser projetado, de maneira que o processamento concorrente seja possível.

Na Parte 2 deste livro eu discuti as questões gerais de proteção e questões de engenharia de proteção. No entanto, quando um sistema é distribuído, o número de maneiras pelas quais o sistema pode ser atacado é significativamente maior, em comparação com sistemas centralizados. Se uma parte do sistema é atacada com êxito, o invasor pode ser capaz de usar isso como uma 'porta dos fundos' para outras partes do sistema.

Os tipos de ataques dos quais um sistema distribuído deve se defender são:

1. *Intercepção*, em que as comunicações entre as partes do sistema são interceptadas por um invasor de tal modo que haja uma perda de confidencialidade.
2. *Interrupção*, em que os serviços de sistema são atacados e não podem ser entregues conforme o esperado. Ataques de negação de serviços envolvem bombardear um nó com solicitações de serviço ilegítimas, para que ele não consiga lidar com solicitações válidas.
3. *Modificação*, em que os dados ou serviços no sistema são alterados por um invasor.
4. *Fabricação*, em que um invasor gera informações que não deveriam existir e, em seguida, usa-as para obter alguns privilégios. Por exemplo, um invasor pode gerar uma entrada de senha falsa e usá-la para obter acesso a um sistema.

A grande dificuldade em sistemas distribuídos é estabelecer uma política de proteção que possa ser fielmente aplicada a todos os componentes de um sistema. Conforme discutido no Capítulo 11, uma política de proteção define o nível de proteção a ser alcançado por um sistema. Mecanismos de proteção, como criptografia e autenticação, são usados para reforçar as políticas de proteção. As dificuldades em um sistema distribuído surgem porque diferentes organizações podem possuir partes do sistema. Essas organizações podem ter mecanismos de proteção e políticas de proteção incompatíveis entre si. Compromissos de proteção podem ser necessários para permitir que os sistemas trabalhem juntos.

A qualidade de serviço (QoS, do inglês *quality of service*) oferecida por um sistema distribuído reflete sua capacidade de entregar seus serviços de maneira confiável e com um tempo de resposta e taxa de transferência aceitáveis para seus usuários. Idealmente, os requisitos de QoS devem ser especificados com antecedência e o sistema, criado e configurado para entregar essa QoS. Infelizmente, isso nem sempre é possível, por duas razões:

1. Pode não ser efetivo projetar e configurar o sistema para oferecer uma alta QoS no momento de carga de pico. Isso poderia envolver disponibilizar recursos que não são usados na maior parte do tempo. Um dos principais argumentos para a 'computação em nuvem' é que ela aborda parcialmente esse problema. Usando uma nuvem, é fácil adicionar recursos conforme a demanda aumenta.
2. Os parâmetros de QoS podem ser mutuamente contraditórios. Por exemplo, maior confiabilidade pode significar taxas reduzidas de transferência, uma vez que as verificações de procedimentos sejam introduzidas para garantir que todas as entradas do sistema sejam válidas.

A QoS é particularmente crítica quando o sistema lida com dados críticos de tempo, como fluxos de som ou vídeo. Nessas circunstâncias, se a QoS cai abaixo de um valor-limite, em seguida, o som ou vídeo podem tornar-se tão degradados que se torna impossível compreendê-los. Os sistemas que lidam com som e vídeo devem incluir componentes de negociação e gerenciamento de QoS. Eles devem avaliar os requisitos de QoS comparados aos recursos disponíveis e, se forem insuficientes, negociar por mais recursos ou por um objetivo de QoS reduzido.

Em um sistema distribuído, é inevitável que ocorram falhas, assim, o sistema precisa ser projetado para ser resistente a essas falhas. A falha é tão onipresente que uma definição irreverente de um sistema distribuído sugerido por Leslie Lamport, um proeminente pesquisador de sistemas distribuídos, é:

> *Você sabe que você tem um sistema distribuído quando a parada de um sistema do que você nunca ouviu impede você de realizar qualquer trabalho.*

O gerenciamento de falhas envolve a aplicação de técnicas de tolerância a defeitos discutidas no Capítulo 13. Portanto, os sistemas distribuídos devem incluir mecanismos para descobrir se um componente do sistema falhou, devem continuar a oferecer tantos serviços quanto possível mesmo que falhe e, tanto quanto possível, devem recuperar-se automaticamente de falhas.

18.1.1 Modelos de interação

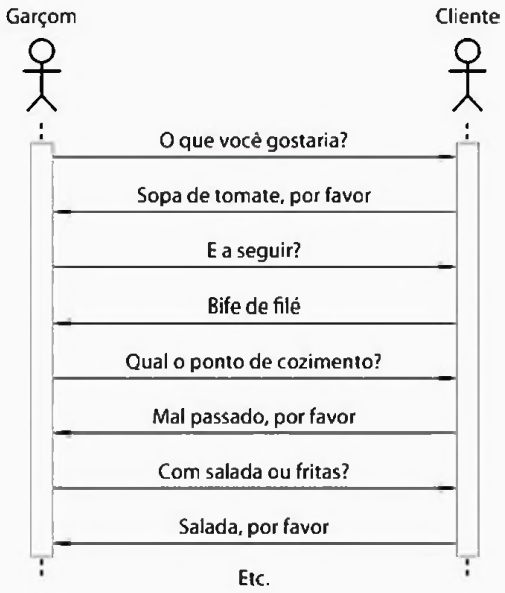
Existem dois tipos fundamentais de interação que podem ocorrer entre os computadores em um sistema de computação distribuído: interação procedural e interação baseada em mensagens. A interação procedural envolve um computador que chama um serviço conhecido oferecido por algum outro computador e (normalmente) esperando que esse serviço seja fornecido. A interação baseada em mensagens envolve o computador 'que envia' que define as informações sobre o que é requerido em uma mensagem, que são, então, enviadas para outro computador. Geralmente, as mensagens transmitem mais informações em uma única interação do que uma chamada de procedimento para outra máquina.

Para ilustrar a diferença entre a interação procedural e a interação baseada em mensagens, considere uma situação na qual você está pedindo uma refeição em um restaurante. Quando você tem uma conversa com o garçom, você está envolvido em uma série de interações síncronas, procedurais que definem seu pedido. Você faz um pedido, o garçom reconhece o pedido; você faz outra solicitação, a qual é reconhecida, e assim por diante. Isso é comparável aos componentes interagindo em um sistema de software em que um componente chama métodos de outros componentes. O garçom anota seu pedido junto com os pedidos de seus acompanhantes. Ele passa esse pedido para a cozinha, que inclui detalhes de tudo o que tenha sido ordenado, para a cozinha preparar a refeição. Essencialmente, o garçom está passando uma mensagem para o pessoal da cozinha definindo a refeição que deve ser preparada. Isso é a interação baseada em mensagens.

A Figura 18.1 ilustra isso. Ela mostra o processo síncrono de pedido como uma série de chamadas, e o Quadro 18.1 mostra uma mensagem XML hipotética que define um pedido feito por uma mesa de três pessoas. A diferença entre essas formas de intercâmbio de informações é clara. O garçom pega o pedido como uma série de interações, com cada interação definindo parte do pedido. No entanto, o garçom tem uma única interação com a cozinha, onde a mensagem define o pedido completo.

Normalmente, em um sistema distribuído, a comunicação procedural é implementada usando-se chamadas de procedimento remoto (RPCs, do inglês *remote procedure calls*). Nas RPCs, um componente chama outro componente, como se fosse um método ou procedimento local. O *middleware* no sistema intercepta essa chamada

Figura 18.1 Interação procedural entre um cliente e um garçom



Quadro 18.1 Interação baseada em mensagens entre um garçom e o pessoal da cozinha

```
<entrada>
      <nome do prato = "sopa" type = "tomate" />
      <nome do prato = "sopa" type = "peixe" />
      <nome do prato = "salada de pombo" />
</entrada>
<curso principal>
      <nome do prato = "bife" type = "lombo" cozinhar = "médio" />
      <nome do prato = "bife" type = "filé" cozinhar = "mal passado" />
      <nome do prato = "robalo">
</principal>
<acompanhamento>
      <nome do prato = "batatas fritas" porções = "2" />
      <nome do prato = "salada" porções = "1" />
</acompanhamento>
```

e transmite-a para um componente remoto. Este realiza o processamento necessário e, via *middleware*, retorna o resultado para o componente chamado. Em Java, as chamadas de método remoto (RMI, do inglês *remote method invocations*) são comparáveis às RPCs, embora não idênticas. O *framework* de RMI trata a chamada de métodos remotos em um programa em Java.

As RPCs exigem um 'stub' para o procedimento chamado ser acessível no computador que está iniciando a chamada. O stub é chamado e converte os parâmetros de procedimento em uma representação-padrão de transmissão para o procedimento remoto. Em seguida, por meio do *middleware*, envia a solicitação para execução do procedimento remoto. O procedimento remoto usa funções de biblioteca para converter os parâmetros para o formato exigido, efetua o processamento e, em seguida, comunica os resultados via 'stub' que representa o chamador.

Normalmente, a interação baseada em mensagens envolve um componente que cria uma mensagem que detalha os serviços necessários de outro componente. Através do *middleware* de sistema, ela é enviada para o componente que recebe a solicitação. O receptor analisa a mensagem, realiza os processamentos e cria uma mensagem para o componente que enviou a solicitação com os resultados desejados. Em seguida, ela é passada para o *middleware* para a transmissão para o componente que enviou a solicitação.

Um problema com a abordagem RPC para a interação é que o chamador e o chamado precisam estar disponíveis no momento da comunicação e devem saber como se referir um ao outro. Em essência, uma RPC tem os mesmos requisitos que uma chamada de método ou procedimento local. Por outro lado, em uma abordagem baseada em mensagens, a indisponibilidade pode ser tolerada, pois a mensagem permanece em uma fila até que o receptor esteja disponível. Além disso, não é necessário que o transmissor e o receptor da mensagem estejam conscientes um do outro. Eles se comunicam com o *middleware*, que é responsável por garantir que as mensagens sejam passadas para o sistema apropriado.

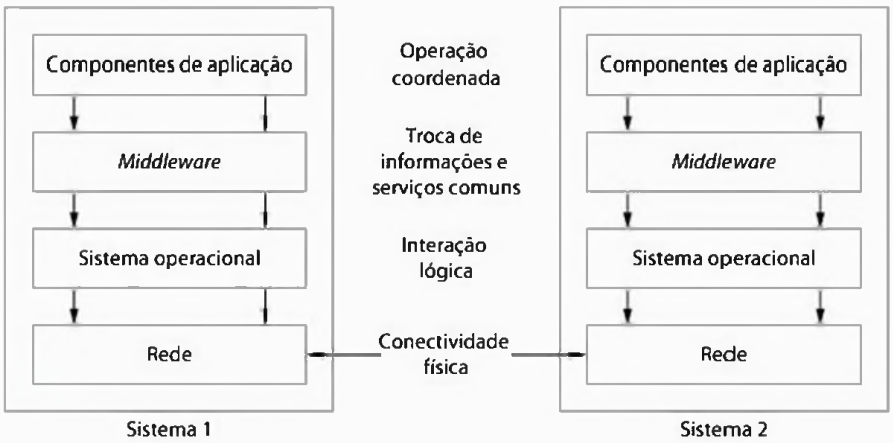
18.1.2 *Middleware*

Os componentes em um sistema distribuído podem ser implementados em diferentes linguagens de programação e podem ser executados em diferentes tipos de processador. Os modelos de dados, representação de informações, protocolos para comunicação podem ser todos diferentes. Um sistema distribuído, portanto, requer um software que possa gerenciar essas diversas partes e assegurar que elas podem se comunicar e trocar dados.

O termo '*middleware*' é usado para se referir a esse software — ele fica no meio, entre os componentes distribuídos do sistema. Isso é ilustrado na Figura 18.2, que mostra que o *middleware* é uma camada entre o sistema operacional e programas de aplicação. Normalmente, o *middleware* é implementado como um conjunto de bibliotecas que é instalado em cada computador distribuído, além de um sistema de *run-time* para gerenciar a comunicação.

Bernstein (1996) descreve os tipos de *middleware* que estão disponíveis para oferecer suporte para a computação distribuída. O *middleware* é um software de uso geral geralmente comprado no mercado e que não é escrito especialmente por desenvolvedores de aplicações. Exemplos de *middleware* incluem o software para gerenciamento de comunicações com bancos de dados, gerenciadores de transações, conversores de dados e controladores de comunicação.

Figura 18.2 O *middleware* em um sistema distribuído



Em um sistema distribuído, o *middleware* costuma fornecer dois tipos distintos de suporte:

1. Suporte a interações, em que o *middleware* coordena as interações entre diferentes componentes do sistema. O *middleware* fornece transparência da localização, assim não é necessário que os componentes saibam os locais físicos dos outros componentes. Ele também pode suportar a conversão de parâmetros se diferentes linguagens de programação forem usadas para implementar componentes, detecção de eventos e comunicação etc.
2. A prestação de serviços comuns, em que o *middleware* fornece implementações reusáveis de serviços que podem ser exigidas por vários componentes do sistema distribuído. Usando esses serviços comuns, os componentes podem, facilmente, interoperar e prestar serviços de usuário de maneira consistente.

Na Seção 18.1.1, dei exemplos do suporte a interações que o *middleware* pode fornecer. Você usa o *middleware* para suporte a chamadas de procedimento remoto e de método remoto, troca de mensagens etc.

Serviços comuns são os serviços que podem ser exigidos por componentes diferentes, independentemente da funcionalidade deles. Conforme discutido no Capítulo 17, eles podem incluir serviços de proteção (autenticação e autorização), serviços de notificação e identificação, serviços de gerenciamento de transações etc. Você pode pensar nesses serviços comuns como sendo fornecidos por um contêiner de *middleware*. Em seguida, pode implantar seu componente nesse contêiner e ele pode acessar e usar esses serviços comuns.

18.2 Computação cliente-servidor

Sistemas distribuídos que são acessados pela Internet normalmente são organizados como sistemas cliente-servidor. Em um sistema cliente-servidor, o usuário interage com um programa em execução em seu computador local (por exemplo, um *browser* de Web ou uma aplicação baseados em telefone). Este interage com outro programa em execução em um computador remoto (por exemplo, um servidor Web). O computador remoto fornece serviços, como o acesso a páginas Web, que estão disponíveis para clientes externos. Esse modelo cliente-servidor, conforme discutido no Capítulo 6, é um modelo de arquiteura geral de uma aplicação. Não está restrito a aplicações distribuídas em várias máquinas. Você também pode usá-lo como um modelo lógico de interação no qual o cliente e o servidor executam no mesmo computador.

Em uma arquitetura cliente-servidor, uma aplicação é modelada como um conjunto de serviços que são fornecidos por servidores. Os clientes podem acessar esses serviços e apresentar os resultados para os usuários finais (ORFALI e HARKEY, 1998). Os clientes precisam estar cientes dos servidores que estão disponíveis, mas não devem saber da existência de outros clientes. Clientes e servidores são processos separados, conforme mostra a Figura 18.3, a qual ilustra uma situação em que existem quatro servidores (s1–s4), que fornecem serviços diferentes. Cada serviço tem um conjunto de clientes associados que acessam esses serviços.

A Figura 18.3 mostra processos entre cliente e servidor, em vez de processadores. É normal que vários processos clientes executem em um único processador. Por exemplo, em seu PC, você pode executar um cliente de correio que transfere mensagens de um servidor de correio remoto. Você também pode executar um *browser* de Web que interage com um servidor Web remoto e um cliente de impressão que envia documentos para uma impressora remota. A Figura 18.4 ilustra a situação em que os 12 clientes lógicos mostrados na Figura 18.3 estão em execução em seis computadores. Os quatro processos servidores são mapeados em dois computadores físicos de servidor.

Figura 18.3 Interação cliente-servidor

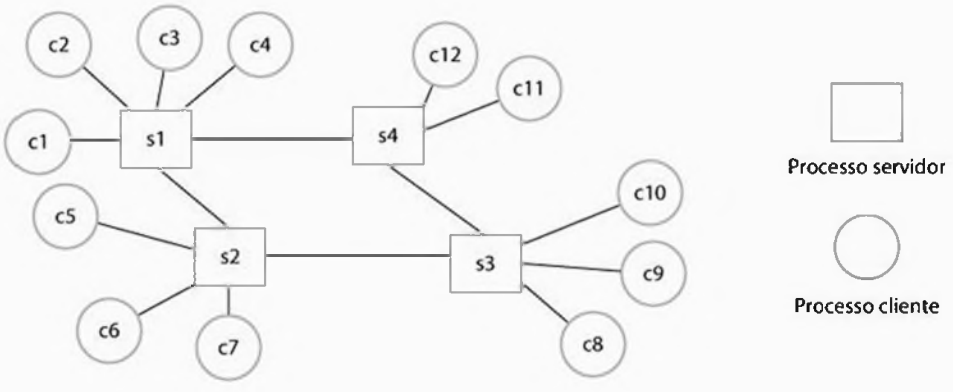
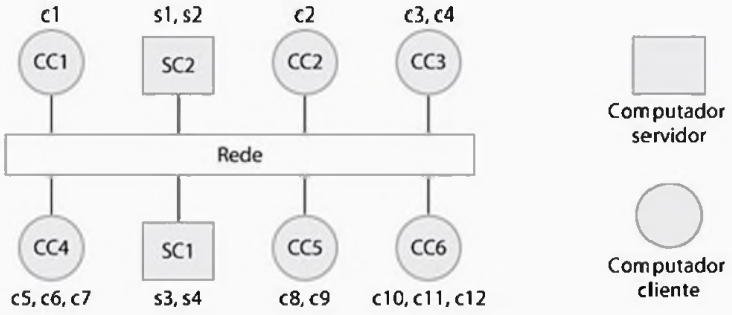


Figura 18.4 Mapeamento de clientes e servidores para computadores em rede



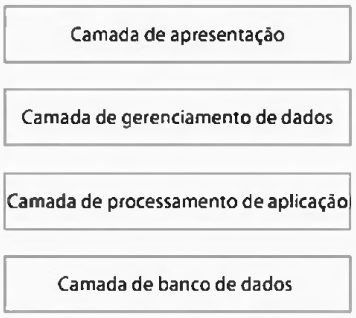
Vários processos servidores diferentes podem ser executados no mesmo processador, mas, muitas vezes, os servidores são implementados como sistemas multiprocessadores em que uma instância separada do processo servidor é executada em cada máquina. O software de balanceamento de carga distribui pedidos de serviço de clientes para servidores diferentes para que cada servidor realize a mesma quantidade de trabalho. Isso permite que um maior volume de transações com clientes seja manipulado, sem degradar a resposta aos clientes individuais.

Os sistemas cliente-servidor dependem de haver uma separação clara entre a apresentação de informações e as computações que criam e processam essas informações. Consequentemente, você deve projetar a arquitetura dos sitemas cliente-servidor distribuídos para que eles sejam estruturados em várias camadas lógicas, com interfaces claras entre essas camadas. Isso permite que cada camada seja distribuída para um computador diferente. A Figura 18.5 ilustra esse modelo, mostrando uma aplicação estruturada em quatro camadas:

- uma camada de apresentação que diz respeito à apresentação de informações para o usuário e gerenciamento de toda a interação com o usuário;
- uma camada de gerenciamento de dados que gerencia os dados que são passados de e para o cliente. Essa camada pode implementar verificações sobre os dados, gerar páginas Web etc.;
- uma camada de processamento de aplicação que está preocupada com a implementação da lógica da aplicação e, assim, fornece a funcionalidade necessária para os usuários finais;
- uma camada de banco de dados que armazena os dados e fornece serviços de gerenciamento de transações etc.

A seção a seguir explica como diferentes arquiteturas cliente-servidor distribuem essas camadas lógicas de maneiras diferentes. O modelo cliente-servidor também serve como base para o conceito de software como serviço (SaaS, do inglês *software as a service*), uma forma cada vez mais importante de implantação e acesso do software através da Internet. Discuto isso na Seção 18.4.

Figura 18.5 Modelo de arquitetura em camadas para aplicações cliente-servidor



18.3 Padrões de arquitetura para sistemas distribuídos

Como expliquei no início deste capítulo, os projetistas de sistemas distribuídos precisam organizar seus projetos de sistema para encontrar um equilíbrio entre desempenho, confiança, proteção e capacidade de gerenciamento do sistema. Não existe um modelo universal de organização de sistema distribuído que seja apropriado para todas as circunstâncias, de modo que surgiram vários estilos de arquitetura. Ao projetar uma aplicação distribuída, você deve escolher um estilo de arquitetura que ofereça suporte aos requisitos não funcionais críticos de seu sistema.

Nesta seção, eu discuto cinco estilos de arquitetura:

1. *Arquitetura de mestre-escravo*, é usada em sistemas de tempo real em que tempos de resposta precisos de interação são requeridos.
2. *Arquitetura cliente-servidor de duas camadas*, é usada para sistemas cliente-servidor simples e em situações nas quais é importante centralizar o sistema por razões de proteção. Nestes casos, a comunicação entre o cliente e o servidor costuma ser criptografada.
3. *Arquitetura cliente-servidor multicamadas*, é usada quando existe um alto volume de transações a serem processadas pelo servidor.
4. *Arquitetura distribuída de componentes*, é usada quando recursos de diferentes sistemas e bancos de dados precisam ser combinados ou é usada como um modelo de implementação para sistemas cliente-servidor em várias camadas.
5. *Arquitetura ponto-a-ponto*, é usada quando os clientes trocam informações localmente armazenadas e o papel do servidor é introduzir clientes uns aos outros. Ela também pode ser usada quando um grande número de computações independentes pode ser feito.

18.3.1 Arquiteturas mestre-escravos

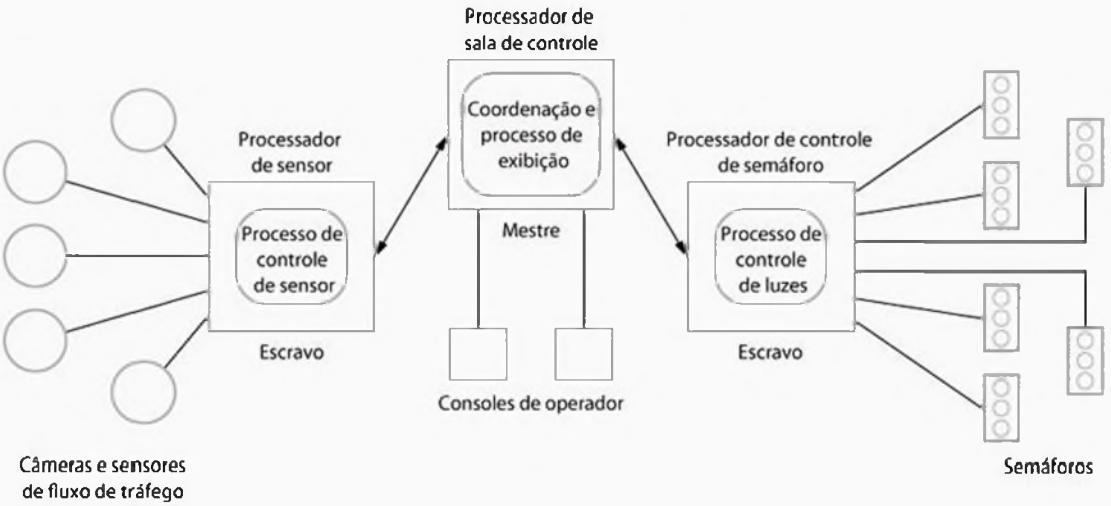
Arquiteturas mestre-escravos para sistemas distribuídos são comumente usadas em sistemas de tempo real em que pode haver processadores separados associados à aquisição de dados do ambiente do sistema, processamento de dados, de gerenciamento de atuadores e computação. Como discutido no Capítulo 20, os atuadores são dispositivos controlados pelo sistema de software que agem para alterar o ambiente do sistema. Por exemplo, um atuador pode controlar uma válvula e alterar seu estado de 'aberta' para 'fechada'. O processo 'mestre' é geralmente responsável pelo processamento, coordenação e comunicações e controla os processos 'escravo'. Processos 'escravo' são dedicados a ações específicas, como a aquisição de dados de um vetor de sensores.

A Figura 18.6 ilustra esse modelo de arquitetura como um modelo de um sistema de controle de tráfego em uma cidade, em que três processos lógicos são executados em processadores separados. O processo mestre é o processo da sala de controle, que se comunica com processos escravos separados e que são responsáveis pela coleta de dados de tráfego e gerência de funcionamento de semáforos.

Um conjunto de sensores distribuídos coleta informações sobre o fluxo de tráfego. O processo de controle de sensores varre-os periodicamente para capturar as informações do fluxo de tráfego e confere essa informação para processamento adicional. O processador de sensor é varrido periodicamente para obtenção de informações por processo mestre que se preocupa com a exibição do *status* de tráfego para os operadores, processamento de sequências de luzes de semáforos e aceitação de comandos de operador para modificar essas sequências. O sistema de sala de controle envia comandos para um processo de controle de semáforo e converte-os em sinais para controlar o hardware das luzes de semáforos. O sistema de sala de controle mestre é organizado como um sistema cliente-servidor, com os processos clientes executando em consoles de operador.

Você pode usar esse modelo de mestre-escravo de um sistema distribuído em situações em que seja possível ou necessário prever o processamento distribuído, bem como em casos nos quais o processamento pode ser facilmente localizado para processadores escravos. Essa situação é comum em sistemas de tempo real, quando é importante cumprir os *deadlines* (prazos) de processamento. Processadores escravos podem ser usados para operações computacionalmente intensivas, como processamento de sinais e o gerenciamento de equipamentos controlados pelo sistema.

Figura 18.6 Um sistema de gerenciamento de tráfego com uma arquitetura mestre-escravo



18.3.2 Arquitetura cliente-servidor de duas camadas

Na Seção 18.2, abordei a forma geral dos sistemas cliente-servidor em que parte do sistema de aplicação é executada no computador do usuário (o cliente) e parte é executada em um computador remoto (o servidor). Apresentei também um modelo de aplicação em camadas (Figura 18.5), em que as diferentes camadas do sistema podem ser executadas em computadores diferentes.

Uma arquitetura cliente-servidor de duas camadas é a forma mais simples da arquitetura cliente-servidor. O sistema é implementado como um único servidor lógico e, também, um número indefinido de clientes que usam esse servidor. A Figura 18.7 mostra duas formas desse modelo de arquitetura:

1. *Modelo cliente-magro*, em que a camada de apresentação é implementada no cliente e todas as outras camadas (gerenciamento de dados, processamento de aplicação e banco de dados) são implementadas em um servidor. O software de cliente pode ser um programa especialmente escrito no cliente para tratar a apresentação. No entanto, é frequente um *browser* de Web no computador cliente ser usado para apresentação dos dados.
2. *Modelo cliente-gordo*, em que parte ou todo o processamento de aplicação é executado no cliente e as funções de banco de dados e gerenciamento são implementadas no servidor.

A vantagem do modelo cliente-magro é a simplicidade em gerenciar os clientes. Isso é um grande problema se houver um grande número de clientes, pois pode ser difícil e caro instalar em novo software em todos eles. Se um *browser* de Web for usado como o cliente, não é necessário instalar qualquer software.

Figura 18.7 Modelos de arquitetura cliente-magro e cliente-gordo

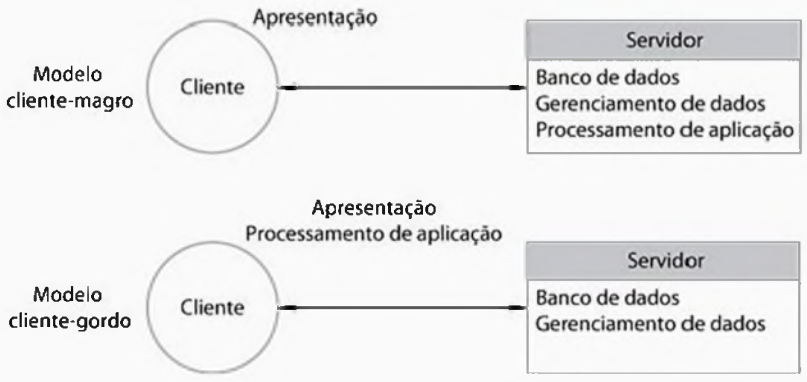
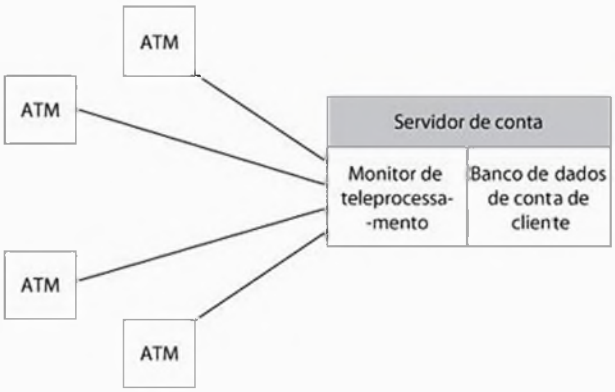


Figura 18.8 Uma arquitetura cliente-gordo para um sistema de ATM



A desvantagem da abordagem cliente-magro, porém, é poder colocar uma carga pesada de processamento no servidor e na rede. O servidor é responsável por toda a computação e isso pode levar à geração de tráfego significativo de rede entre o cliente e o servidor. A implementação de um sistema usando esse modelo, portanto, pode exigir investimentos adicionais na capacidade de rede e de servidor. No entanto, *browsers* podem efetuar algum processamento local executando *scripts* (por exemplo, Javascript) na página Web que é acessada pelo *browser*.

O modelo cliente-gordo faz uso do poder de processamento disponível no computador executando o software cliente e distribui alguns ou todo o processamento de aplicação e a apresentação para o cliente. O servidor é essencialmente um servidor de transação que gerencia todas as transações do banco de dados. O gerenciamento de dados é simples e não é necessário haver interação entre o cliente e o sistema de processamento de aplicação. Certamente, o problema com o modelo cliente-gordo é requerer gerenciamento de sistema adicional para implantar e manter o software no computador cliente.

Um exemplo de uma situação em que a arquitetura cliente-gordo é usada é um sistema de banco ATM, que oferece dinheiro e outros serviços bancários para os usuários. ATM é o computador cliente, e o servidor é, normalmente, um *mainframe* executando o banco de dados de conta de cliente. Um computador *mainframe* é uma máquina poderosa que é projetada para o processamento de transações. Assim, ele pode lidar com o grande volume de transações geradas pelas ATMs e outros sistemas de caixa e bancos on-line. O software na máquina de caixa realiza vários processamentos relacionados ao cliente associado com uma transação.

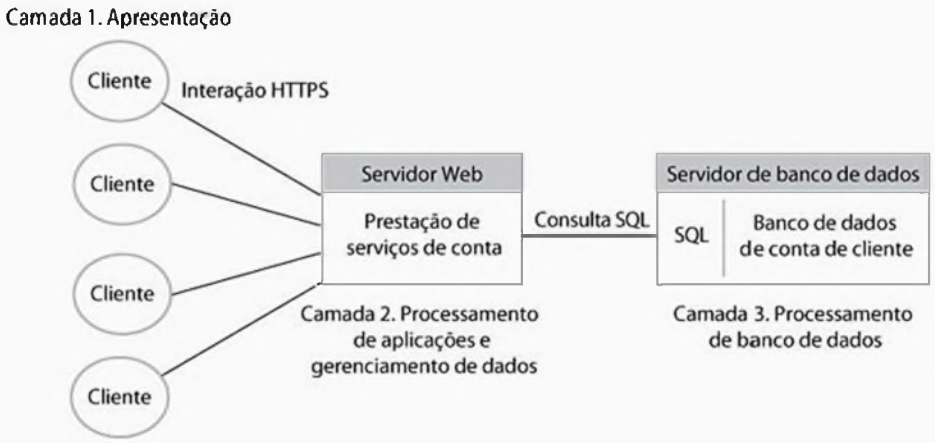
A Figura 18.8 mostra uma versão simplificada da organização do sistema de uma ATM. Observe que as ATMs não estão ligadas diretamente com o banco de dados do cliente, mas sim a um monitor de teleprocessamento (TP). Um monitor de teleprocessamento é um sistema de *middleware* que organiza as comunicações com os clientes remotos e serializa as transações de clientes para o processamento no banco de dados. Isso garante que as transações sejam independentes e não interfiram entre si. Usar transações seriais significa que o sistema pode se recuperar de defeitos sem corromper os dados do sistema.

Considerando que um modelo cliente-gordo distribui o processamento mais eficazmente do que um modelo cliente-magro, o gerenciamento de sistema é mais complexo. A funcionalidade de aplicação é espalhada por muitos computadores. Quando o software de aplicação precisa ser alterado, isso envolve a reinstalação em cada computador cliente, o que pode ter um grande custo se houver centenas de clientes no sistema. O sistema pode ser projetado para oferecer suporte a atualizações de software remoto e pode ser necessário desligar todos os serviços de sistema até que o software de cliente seja substituído.

18.3.3 Arquiteturas cliente-servidor multicamadas

O problema fundamental com uma abordagem cliente-servidor de duas camadas é que as camadas lógicas de sistema — apresentação, processamento de aplicação, gerenciamento de dados e banco de dados — devem ser mapeadas para dois sistemas de computador: o cliente e o servidor. Pode haver problemas com escalabilidade e desempenho se o modelo cliente-magro for escolhido, ou problemas de gerenciamento de sistema se o modelo cliente-gordo for usado. Para evitar esses problemas, uma arquitetura de 'cliente-servidor multicamadas'

Figura 18.9 Arquitetura de três camadas para um sistema de *Internet banking*



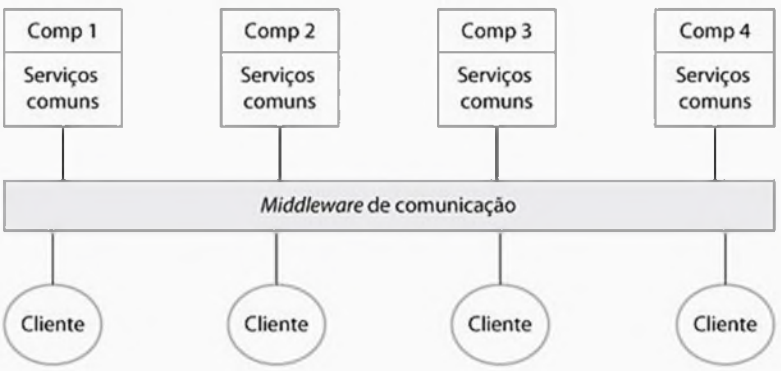
pode ser usada. Nessa arquitetura, as diferentes camadas do sistema, apresentação, gerenciamento de dados, processamento de aplicação e banco de dados, são processos separados que podem ser executados em diferentes processadores.

Um sistema de *Internet banking* (Figura 18.9) é um exemplo da arquitetura cliente-servidor multicamadas, em que existem três camadas no sistema. O banco de dados de clientes do banco (geralmente, hospedado em um computador *mainframe*, como discutido anteriormente) fornece serviços de banco de dados. Um servidor Web fornece serviços de gerenciamento de dados, como a geração de página Web e alguns serviços de aplicação. Os serviços de aplicação, como recursos para transferir dinheiro, gerar extratos, pagar contas, e assim por diante, são implementados no servidor Web como *scripts*, os quais são executados pelo cliente. O computador do usuário com um *browser* de Internet é o cliente. Esse sistema é escalável, pois é relativamente fácil adicionar servidores (escalabilidade para cima), assim como o número de clientes.

Nesse caso, o uso de arquitetura de três camadas permite a transferência de informações entre o servidor Web e o servidor de banco de dados para ser otimizado. As comunicações entre esses sistemas podem usar protocolos de troca de dados rápidos e de baixo nível. Um *middleware* eficiente oferece suporte em consultas de banco de dados em SQL (Structured Query Language), sendo usado para tratar informações de recuperação do banco de dados.

O modelo cliente-servidor de três camadas pode ser estendido para uma variante em multicamadas, na qual os servidores adicionais são adicionados ao sistema. Esse processo envolve o uso de um servidor Web para gerenciamento de dados e de servidores separados para processamento de aplicação e serviços de banco de dados. Sistemas multicamadas também podem ser usados quando aplicações precisam acessar e usar dados de diferentes bancos de dados. Nesse caso, talvez você precise adicionar um servidor de integração ao sistema, o qual atuaria coletando os dados distribuídos e apresentando-os ao servidor de aplicação, como se tratasse de um único banco de dados. Como discuto na seção seguinte, arquiteturas de componentes distribuídos podem ser usadas para implementar sistemas cliente-servidor multicamadas.

Figura 18.10 Uma arquitetura de componentes distribuídos



Os sistemas cliente-servidor multicamadas que distribuem o processamento de aplicação entre vários servidores são inerentemente mais escaláveis do que as arquiteturas de duas camadas. O processamento de aplicação é, muitas vezes, a parte mais volátil do sistema e pode ser facilmente atualizado, pois está centralmente localizado. O processamento, em alguns casos, pode ser distribuído entre os servidores de lógica de aplicação e de gerenciamento de dados, gerando uma resposta mais rápida para as solicitações de clientes.

Os projetistas de arquiteturas cliente-servidor devem considerar vários fatores ao escolher a arquitetura de distribuição mais adequada. A Tabela 18.1 descreve situações em que as arquiteturas cliente-servidor podem ser adequadas.

18.3.4 Arquiteturas de componentes distribuídos

A Figura 18.5 mostra o processamento organizado em camadas, e cada camada de um sistema pode ser implementada como um servidor lógico separado. Esse modelo funciona bem para muitos tipos de aplicação. No entanto, ele limita a flexibilidade de projetistas de sistema, pois é necessário decidir quais serviços devem ser incluídos em cada camada. Na prática, contudo, não é sempre claro se um serviço é um serviço de gerenciamento de dados, um serviço de aplicação ou um serviço de banco de dados. Os projetistas também devem planejar para escalabilidade e, assim, fornecer alguns meios para que os servidores sejam replicados à medida que mais clientes são adicionados ao sistema.

Uma abordagem mais geral de projeto de sistemas distribuídos é projetar o sistema como um conjunto de serviços, sem tentar alocar esses serviços nas camadas do sistema. Cada serviço, ou grupo de serviços relacionados, é implementado usando um componente separado. Em uma arquitetura de componentes distribuídos (Figura 18.10), o sistema é organizado como um conjunto de componentes ou objetos interativos. Esses componentes fornecem uma interface para um conjunto de serviços que eles fornecem. Outros componentes chamam esses serviços através do *middleware*, usando chamadas de procedimento remoto ou chamadas de métodos.

Os sistemas de componentes distribuídos são dependentes do *middleware*, o qual gerencia as interações de componentes, reconcilia as diferenças entre os tipos de parâmetros passados entre componentes e fornece um conjunto de serviços comuns que os componentes de aplicação podem usar. CORBA (ORFALI et al., 1997) foi um dos primeiros exemplos de tal *middleware*, mas hoje em dia ele não é usado amplamente, pois tem sido suplantado por softwares proprietários como o Enterprise Java Beans (EJB) ou .NET.

Tabela 18.1 Uso de padrões de arquitetura cliente-servidor

Arquitetura	Aplicações
Arquitetura cliente-servidor de duas camadas com clientes-magros	Aplicações de sistema legado usadas quando a separação de gerenciamento de dados e de processamento de aplicação são impraticáveis. Os clientes podem acessá-las como serviços, conforme discutido na Seção 18.4. Aplicações computacionalmente intensivas como compiladores com pouco ou nenhum gerenciamento de dados. Aplicações intensivas de dados (navegação e consulta) com processamento de aplicações não intensivo. Navegar na Web é o exemplo mais comum de uma situação em que essa arquitetura é usada.
Arquitetura cliente-servidor de duas camadas com clientes-gordos	Aplicações em que o processamento de aplicação é fornecido por softwares de prateleira (por exemplo, o Microsoft Excel) no cliente. Aplicações em que é requerido processamento computacionalmente intensivo de dados (por exemplo, visualização de dados). Aplicações móveis em que a conectividade com a Internet não pode ser garantida. Algum processamento local usando informações armazenadas em banco de dados, portanto, é possível.
Arquitetura cliente-servidor multicamadas	Aplicações de grande porte com centenas ou milhares de clientes. Aplicações nas quais os dados e a aplicação são voláteis e integrados a dados de várias fontes.

Os benefícios de se usar um modelo de componentes distribuídos para implementação de sistemas distribuídos são os seguintes:

1. A permissão ao projetista de sistema de atrasar decisões sobre onde e como os serviços deverão ser prestados. Os componentes fornecedor de serviços podem executar em qualquer nó da rede. Não é necessário decidir previamente se um serviço é parte de uma camada de gerenciamento de dados, uma camada de aplicação etc.
2. É uma arquitetura de sistemas muito aberta, a qual permite a adição de novos recursos conforme necessário. Novos serviços de sistema podem ser adicionados facilmente sem grandes perturbações ao sistema existente.
3. O sistema é flexível e escalável. Novos componentes ou componentes replicados podem ser adicionados quando a carga sobre o sistema aumenta, sem interromper as outras partes do sistema.
4. É possível reconfigurar o sistema dinamicamente com componentes migrando através da rede conforme necessário. Isso pode ser importante onde estão flutuando padrões de demanda de serviços. Um componente fornecedor de serviços pode migrar para o mesmo processador como objetos requisitor de serviços, aumentando assim o desempenho do sistema.

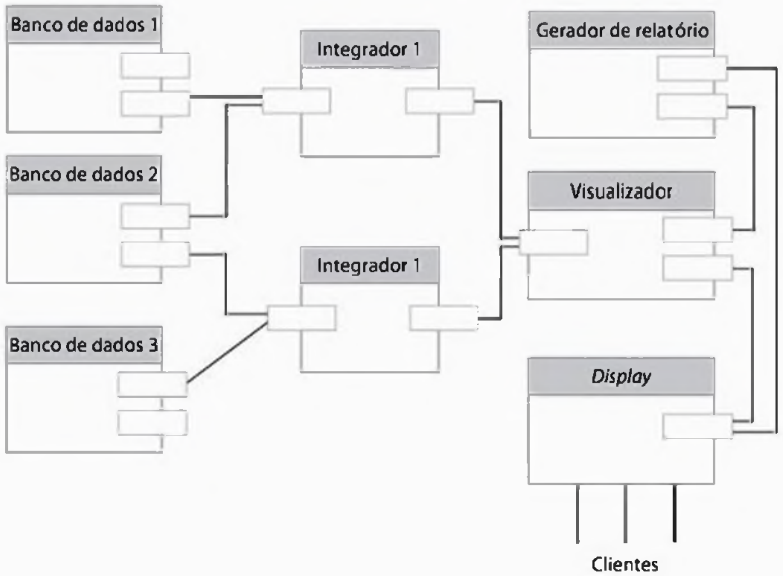
Uma arquitetura de componentes distribuídos pode ser usada como um modelo lógico que permite estruturar e organizar o sistema. Nesse caso, poderá fornecer a funcionalidade de aplicação, unicamente, em termos de serviços e combinações de serviços. Depois, você define como fornecer esses serviços usando um conjunto de componentes distribuídos. Por exemplo, em uma aplicação de varejo pode haver componentes de aplicação interessados no controle de estoque, comunicações com clientes, pedidos de mercadorias, e assim por diante.

Os sistemas de mineração de dados são um bom exemplo de um tipo de sistema no qual uma arquitetura de componentes distribuídos é o melhor padrão de arquitetura para se usar. Um sistema de mineração de dados procura relacionamentos entre os dados que são armazenados em uma série de bancos de dados (Figura 18.11). Os sistemas de mineração de dados geralmente extraem informações de vários bancos de dados separados e realizam o processamento computacional intensivo e exibem seus resultados em gráficos.

Um exemplo de uma aplicação de mineração de dados pode ser um sistema para uma empresa de varejo que vende livros e mercadorias. O departamento de marketing quer encontrar relacionamentos entre as compras de mercadorias e livros de um cliente. Por exemplo, uma proporção relativamente elevada de pessoas que compram pizzas também podem comprar romances policiais. Com esse conhecimento, o negócio pode voltar-se para os clientes que fazem compras de mercadorias específicas com informações sobre novos romances, quando estes são publicados.

Nesse exemplo, cada banco de dados de vendas pode ser sintetizado como um componente distribuído com uma interface que fornece acesso somente para a leitura de seus dados. Os componentes integradores estão interessados em tipos específicos de relacionamentos que coletam informações de todos os bancos de dados para tentar deduzir os relacionamentos. Pode haver um componente integrador que se preocupa com variações sazonais dos bens vendidos e outro que se preocupa com os relacionamentos entre os diferentes tipos de mercadorias.

Figura 18.11 Uma arquitetura de componentes distribuídos para um sistema de mineração de dados



Os componentes visualizadores interagem com os componentes integradores para produzir uma visualização ou um relatório sobre os relacionamentos que forem descobertos. Por causa dos grandes volumes de dados que são manipulados, os componentes visualizadores normalmente apresentam seus resultados graficamente. Finalmente, um componente *display* pode ser responsável por entregar os modelos gráficos para clientes para a apresentação final.

Uma arquitetura de componentes distribuídos é apropriada para esse tipo de aplicação, ao invés de uma arquitetura em camadas, pois novos bancos de dados podem ser adicionados ao sistema sem grandes perturbações. Cada novo banco de dados é acessado adicionando-se outro componente distribuído. Os componentes de acesso ao banco de dados fornecem uma interface simplificada que controla o acesso aos dados. Os bancos de dados acessados podem residir em máquinas diferentes. A arquitetura também torna mais fácil minerar novos tipos de relacionamentos, adicionando novos componentes integradores.

As arquiteturas de componentes distribuídos sofrem de duas grandes desvantagens:

1. Elas são mais complexas para projetar que sistemas cliente-servidor. Os sistemas cliente-servidor multicamadas parecem ser uma forma bastante intuitiva de se pensar sobre os sistemas. Eles refletem muitas transações humanas em que as pessoas solicitam e recebem serviços de outras pessoas que se especializaram em fornecer tais serviços. Por outro lado, as arquiteturas de componentes distribuídos são mais difíceis para as pessoas visualizarem e compreenderem.
2. O *middleware* padronizado para sistemas de componentes distribuídos nunca foi aceito pela comunidade. Diferentes fornecedores, como a Microsoft e a Sun, desenvolveram *middlewares* diferentes e incompatíveis. Esses *middlewares* são complexos e a confiança neles aumenta a complexidade geral dos sistemas distribuídos.

Como resultado desses problemas, as arquiteturas orientadas a serviços (discutidas no Capítulo 19) estão substituindo as arquiteturas de componentes distribuídos em muitas situações. No entanto, os sistemas de componenes distribuídos têm benefícios de desempenho sobre os sistemas orientados a serviços. Geralmente, as comunicações RPC são mais rápidas do que a interação baseada em mensagens usada em sistemas orientados a serviços. As arquiteturas baseadas em componentes, portanto, são mais adequadas para sistemas com alta taxa de transferência em que um grande número de transações precisa ser processado rapidamente.

18.3.5 Arquiteturas ponto-a-ponto

O modelo de computação cliente-servidor discutido nas seções anteriores do capítulo faz uma distinção clara entre os servidores, que são provedores de serviços, e clientes, que são receptores de serviços. Geralmente, esse modelo leva a uma distribuição desigual de carga no sistema, em que os servidores trabalham mais que clientes. Isso pode levar as organizações a investirem muito na capacidade de servidor, enquanto existe capacidade de processamento não usada em centenas ou milhares de PCs usados para acessar os servidores de sistema.

Os sistemas ponto-a-ponto (p2p, do inglês *peer-to-peer*) são sistemas descentralizados em que os processamentos podem ser realizados por qualquer nó na rede. Em princípio, pelo menos, não existem distinções entre clientes e servidores. Em aplicações ponto-a-ponto, todo o sistema é projetado para aproveitar o poder computacional e o armazenamento disponível por meio de uma rede potencialmente enorme de computadores. As normas e protocolos que permitem a comunicação entre os nós são embutidas na própria aplicação, e cada nó deve executar uma cópia dessa aplicação.

As tecnologias ponto-a-ponto têm sido usadas, principalmente, para sistemas pessoais, e não de negócio (ORAM, 2001). Por exemplo, os sistemas de compartilhamento de arquivos com base em protocolos Gnutella e BitTorrent são usados para trocar arquivos de PCs. Os sistemas de mensagens instantâneas, como o ICQ e Jabber, fornecem comunicação direta entre os usuários sem um servidor intermediário. SETI@home é um projeto de longa duração para processar dados de radiotelescópios em PCs domésticos para procurar indícios de vida extraterrestre. Freenet é um banco de dados descentralizado que foi projetado para tornar mais fácil publicar informações anonimamente e para tornar mais difícil para as autoridades suprimirem essa informação. Serviços de voz sobre IP (VOIP), como o Skype, dependem da comunicação ponto-a-ponto entre as partes envolvidas na chamada telefônica ou conferência.

No entanto, os sistemas ponto-a-ponto também estão sendo usados pelas empresas para aproveitarem o poder em suas redes de PC (McDOUGALL, 2000). A Intel e a Boeing têm dois sistemas p2p implementados para aplicações computacionalmente intensivas. Elas aproveitam a capacidade de processamento não usada em computadores locais. Em vez de comprar hardwares caros, de alto desempenho, os processamentos de engenharia

podem ser executados durante a noite, quando os computadores *desktop* não são usados. As empresas também fazem uso extensivo de sistemas p2p comerciais, como sistemas de mensagens e VOIP.

É adequado usar um modelo de arquitetura ponto-a-ponto para um sistema em duas circunstâncias:

1. Quando o sistema é computacionalmente intensivo e é possível separar o processamento necessário para um grande número de computações independentes. Por exemplo, um sistema ponto-a-ponto que ofereça suporte computacional para a descoberta de novas drogas distribui computações que procuram possíveis tratamentos de câncer, analisando um elevado número de moléculas para ver se elas têm as características necessárias para suprimir o crescimento de cânceres. Cada molécula pode ser considerada isoladamente, então não existe nenhuma necessidade de comunicação entre os pontos no mesmo nível de sistema.
2. Sempre que o sistema envolver a troca de informações entre computadores individuais em uma rede e não for necessário que essas informações sejam armazenadas ou gerenciadas centralmente. Exemplos de tais aplicações incluem sistemas de compartilhamento de arquivos que permitem que os pontos troquem de arquivos localmente, como música e arquivos de vídeo e sistemas de telefone que oferecem suporte a comunicações de voz e vídeo entre computadores.

Em princípio, cada nó em uma rede p2p poderia estar ciente de todos os outros nós. Os nós poderiam conectar-se e trocar dados diretamente com qualquer outro nó da rede. Na prática, isso é certamente impossível, por isso os nós são organizados em 'localidades' com alguns nós atuando como pontes para outras localidades de nós. A Figura 18.12 mostra essa arquitetura p2p descentralizada.

Em uma arquitetura descentralizada, os nós da rede não são simplesmente elementos funcionais, mas também comutadores de comunicações que podem rotear dados e controlar os sinais de um nó para outro. Por exemplo, suponha que a Figura 18.12 represente um sistema de gerenciamento de documentos descentralizado. Esse sistema é usado por um consórcio de pesquisadores para compartilhar documentos e cada membro do consórcio mantém seu próprio repositório de documentos. No entanto, quando um documento é recuperado, o nó que o recupera também o disponibiliza para outros nós.

Se alguém precisa de um documento que é armazenado em algum lugar na rede, essa pessoa emite um comando de busca, que é enviado para os nós em sua 'localidade'. Esses nós verificam se eles têm o documento e, em caso afirmativo, devolvem o documento para o solicitante. Se não tiverem, eles roteam a pesquisa para outros nós. Portanto, se n1 emitir uma pesquisa para um documento que está armazenado no n10, essa pesquisa será roteada através dos nós n3, n6 e n9 a n10. Quando o documento é descoberto, o nó que possui o documento envia-o diretamente para o nó solicitante, fazendo uma conexão ponto-a-ponto.

Essa arquitetura descentralizada tem vantagens, pois é altamente redundante e, portanto, tolerante a defeitos e à desconexão de nós da rede. No entanto, as desvantagens são que muitos nós diferentes podem processar a mesma pesquisa e ocorrer um *overhead* significativo em comunicações de pontos replicadas.

Um modelo de arquitetura p2p alternativo, que parte de uma arquitetura p2p pura, é uma arquitetura semicentralizada em que, no âmbito da rede, um ou mais nós atuam como servidores para facilitar as comunicações entre os nós. Isso reduz o tráfego entre eles. A Figura 18.13 ilustra esse modelo.

Em uma arquitetura semicentralizada, a função do servidor (às vezes chamado superponto) é ajudar no estabelecimento de contato entre os pontos da rede ou na coordenação dos resultados de um processamento. Por exemplo, se a Figura 18.13 representa um sistema de mensagens instantâneas, então os nós de rede se comuni-

Figura 18.12 Uma arquitetura p2p descentralizada

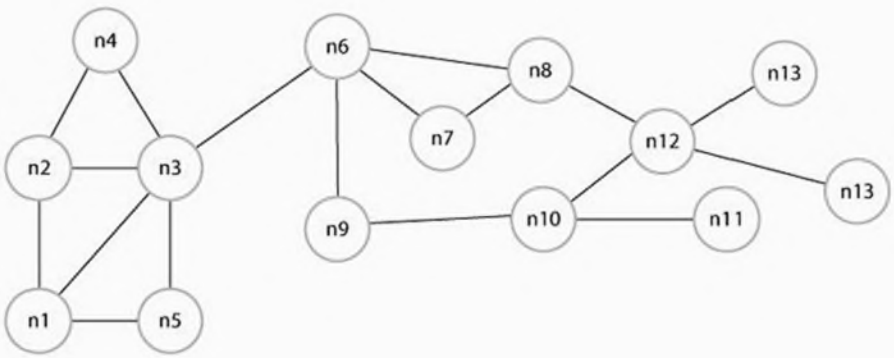
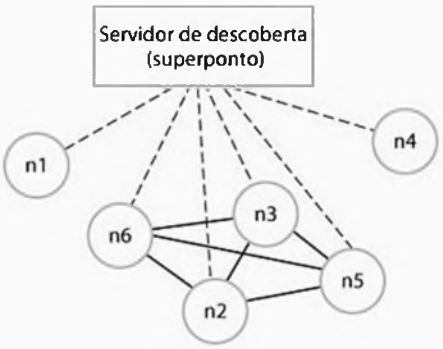


Figura 18.13 Uma arquitetura p2p semicentralizada



cam com o servidor (indicado por linhas tracejadas) para saber quais outros nós estão disponíveis. Uma vez que esses nós são descobertos, pode-se estabelecer a comunicação direta e a conexão com o servidor será desnecessária. Portanto, os nós n2, n3, n5 e n6 estão em comunicação direta.

Em um sistema computacional p2p em que uma computação de processador intensivo é distribuída por meio de um grande número de nós, é normal que alguns nós sejam superpontos. Seu papel é distribuir o trabalho para outros nós, conferir e verificar os resultados da computação.

As arquiteturas ponto-a-ponto permitem o uso eficiente da capacidade por meio de uma rede. No entanto, os principais problemas que têm inibido seu uso são as questões de proteção e confiança. A comunicação ponto-a-ponto envolve abrir seu computador para direcionar as interações com outros pontos, e isso significa que esses sistemas poderiam, potencialmente, acessar qualquer um de seus recursos. A fim de combater isso, você precisa organizar seu sistema para que esses recursos sejam protegidos. Caso esse processo seja feito incorretamente, o sistema poderá ficar inseguro.

Também podem ocorrer problemas quando os pontos em uma rede se comportam deliberadamente de forma maliciosa. Por exemplo, já houve casos em que empresas de música, acreditando que seus direitos autorais estavam sendo violados, 'envenenaram os pontos' disponíveis, deliberadamente. Quando o outro ponto baixa o que eles acham que é uma peça de música, o arquivo real entregue é um *malware*, que pode ser uma versão deliberadamente corrompida de música ou um aviso para o usuário da violação de direitos autorais.

18.4 Softwares como um serviço

Nas seções anteriores, discuti modelos de cliente-servidor e como a funcionalidade pode ser distribuída entre os clientes e os servidores. Para implementar um sistema cliente-servidor, talvez você precise instalar no computador do cliente um programa que se comunique com o servidor, implemente a funcionalidade do cliente e gerencie a interface de usuário. Por exemplo, um cliente de e-mail, como Outlook ou Mac Mail, fornece recursos de gerenciamento de correio em seu próprio computador. Isso evita o problema de alguns sistemas cliente-magro, nos quais todo o processamento é realizado no servidor.

No entanto, os problemas de *overhead* de servidor podem ser significativamente reduzidos, usando-se um *browser* moderno como o software de cliente. As tecnologias Web, como AJAX (HOLDENER, 2008), suportam o gerenciamento eficiente de apresentação de página Web e a computação local por meio de *scripts*. Isso significa que um *browser* pode ser configurado e usado como um cliente, com processamento local significativo. O software de aplicação pode ser pensado como um serviço remoto, que pode ser acessado de qualquer dispositivo que possa executar um *browser*-padrão. Exemplos bem conhecidos disso são sistemas de correio baseados na Web, como Yahoo!® e Gmail®, além de aplicações de escritório, como o Google® Docs.

Essa noção de SaaS envolve a hospedagem remota do software e fornece acesso a ele através da Internet. Os elementos-chave do SaaS são os seguintes:

1. O software é implantado em um servidor (ou, mais comumente, vários servidores) e é acessado por meio de um *browser* de Web. Ele não é implantado em um PC local.
2. O software é de propriedade e gerido por um fornecedor de software, e não pelas organizações que usam o software.

3. Os usuários podem pagar para o software de acordo com a quantidade de uso que fazem dele ou por meio de uma assinatura anual ou mensal. Às vezes, o software tem o uso liberado, mas os usuários devem concordar em aceitar anúncios que financiem o serviço de software.

Para usuários de software, o benefício do SaaS é que os custos de gerenciamento de software são transferidos para o provedor. O provedor é responsável pela correção de *bugs* e instalação das atualizações de software, pelas alterações na plataforma de sistema operacional e pela garantia de que a capacidade do hardware possa atender à demanda. Os custos de gerenciamento de licenças de software são zero. Se alguém tiver vários computadores, não existe necessidade de licença de software para todos eles. Se uma aplicação de software é usada apenas ocasionalmente, o modelo 'pague pelo uso' (*pay-per-use*) pode ser mais barato do que comprar uma aplicação. O software pode ser acessado de dispositivos móveis, como *smart phones*, de qualquer lugar do mundo.

Naturalmente, esse modelo de fornecimento de software tem algumas desvantagens. O principal problema talvez seja os custos de transferência de dados para o serviço remoto. A transferência de dados ocorre em velocidade de rede e, então, transferir uma grande quantidade de dados leva muito tempo. Você também pode ter de pagar o provedor de serviços de acordo com o montante transferido. Outros problemas são a falta de controle sobre a evolução de software (o provedor pode alterar o software quando desejar), além de problemas com leis e regulamentos. Muitos países têm leis que regem o armazenamento, o gerenciamento, a preservação e a acessibilidade de dados, e mover os dados para um serviço remoto pode violar tais leis.

A noção de SaaS e as arquiteturas orientadas a serviços (SOAs), discutidas no Capítulo 19, relacionam-se, obviamente, mas não são as mesmas:

1. O SaaS é uma forma de fornecer funcionalidade em um servidor remoto com acesso de clientes por meio de um *browser* de Web. O servidor mantém os dados e o estado do usuário durante uma sessão de interação. Geralmente, as transações são longas (por exemplo, edição de um documento).
2. A SOA é uma abordagem para a estruturação de um sistema de software como um conjunto de serviços separados, sem estado. Estes podem ser fornecidos por vários provedores e podem ser distribuídos. Normalmente, tratam-se de transações curtas, em que um serviço é chamado, faz alguma coisa e, em seguida, retorna um resultado.

O SaaS é uma maneira de entregar a funcionalidade de aplicação para os usuários, enquanto a SOA é uma tecnologia de implementação para sistemas de aplicações. A funcionalidade implementada pelo uso da SOA precisa aparecer para usuários como serviços. Da mesma forma, os serviços de usuário não precisam ser implementados pelo uso da SOA. No entanto, se o SaaS é implemento usando a SOA, torna-se possível para aplicações usarem APIs de serviço para acessar a funcionalidade de outras aplicações. Em seguida, estas podem ser integradas em sistemas mais complexos; são chamados *mashups* e representam outra abordagem para reúso de software e desenvolvimento rápido de software.

De uma perspectiva de desenvolvimento de software, o processo de desenvolvimento de serviços tem muito em comum com outros tipos de desenvolvimento de software. No entanto, a construção de serviços normalmente não é conduzida pelos requisitos do usuário, mas por suposições do provedor de serviços sobre o que os usuários precisam. Portanto, o software precisa ser capaz de evoluir rapidamente depois que o provedor obtenha *feedback* dos usuários sobre seus requisitos. Portanto, o desenvolvimento ágil com entrega incremental é uma abordagem comumente usada para os softwares que devem ser implantados como serviços.

Ao implementar o SaaS, você precisa considerar que pode haver usuários do software de várias organizações diferentes. São três os fatores que precisam ser considerados:

1. *Configurabilidade*. Como você configura o software para as necessidades específicas de cada organização?
2. *Multilocação*. Como você apresenta para cada usuário do software a impressão de que eles estão trabalhando com sua própria cópia do sistema enquanto, e ao mesmo tempo, fazem uso eficiente dos recursos de sistema?
3. *Escalabilidade*. Como você projeta o sistema para que ele possa ser dimensionado a fim de acomodar um número imprevisível de usuários?

A noção de arquiteturas de linha de produtos, discutida no Capítulo 16, é uma forma de configurar o software para usuários que possuem sobreposição, mas requisitos não idênticos. Você começa com um sistema genérico e o adapta de acordo com os requisitos específicos de cada usuário.

No entanto, isso não funciona para SaaS, pois significaria implantar uma cópia diferente do serviço para cada organização que usa o software. Em vez disso, você precisa projetar a configurabilidade no sistema e fornecer uma interface de configuração que permita aos usuários especificarem suas preferências. Em seguida, você usa esses dados para ajustar o comportamento do software, dinamicamente, enquanto ele é usado. Os recursos de configuração podem permitir:

1. *Gerenciamento de marcas*, em que aos usuários de cada organização são apresentados com uma interface que reflete sua própria organização.
2. *Regras de negócios e fluxos de trabalho*, em que cada organização define suas próprias regras para regerem o uso do serviço e seus dados.
3. *Extensões de banco de dados*, em que cada organização define como o modelo de dados do serviço genérico é ampliado para atender a suas necessidades específicas.
4. *Controle de acesso*, em que os clientes do serviço criam contas individuais para sua equipe e definem os recursos e funções acessíveis para cada um de seus usuários.

A Figura 18.14 ilustra essa situação. Esse diagrama mostra cinco usuários do serviço de aplicação, os quais trabalham para três clientes diferentes do provedor de serviços. Os usuários interagem com o serviço por meio de um perfil de clientes que define a configuração de serviço para seu empregador.

A multilocação é uma situação em que muitos diferentes usuários acessam o mesmo sistema e a arquitetura do sistema é definida para permitir o compartilhamento eficiente dos recursos de sistema. No entanto, para cada usuário deve parecer que ele tem o uso exclusivo do sistema. A multilocação envolve projetar o sistema para que haja uma separação absoluta entre sua funcionalidade e seus dados. Portanto, você deve projetar o sistema para que todas as operações sejam sem estado. Os dados devem ser fornecidos pelo cliente ou devem estar disponíveis em um sistema de armazenamento ou banco de dados que possa ser acessado a partir de qualquer instância do sistema. Os bancos de dados relacionais não são ideais para fornecimento de multilocação e grandes provedores de serviços, como Google", implementaram um banco de dados simples para os dados de usuários.

Um problema específico em sistemas de multilocação é o gerenciamento de dados. A maneira mais simples de fornecer o gerenciamento de dados é garantir que cada cliente tenha seu próprio banco de dados, para que eles possam usá-lo e configurá-lo como quiserem. No entanto, isso requer que o provedor de serviços mantenha muitas instâncias de banco de dados diferentes (um por cliente) para disponibilizá-las sob demanda. Em termos da capacidade de servidor, esse procedimento é ineficiente, além de aumentar o custo global do serviço.

Como alternativa, o provedor de serviços pode usar um único banco de dados com diferentes usuários virtualmente isolados dentro desse banco de dados. Isso é ilustrado na Figura 18.15, na qual se pode ver que entradas de banco de dados também têm um 'identificador de locatário', que liga essas entradas a usuários específicos. Usando visões de banco de dados, você pode extrair as entradas para cada cliente de serviço e, assim, apresentar os usuários desse cliente com um banco de dados virtual e pessoal, o que pode ser estendido para atender às necessidades específicas do cliente, usando os recursos de configuração discutidos anteriormente.

Figura 18.14 Configuração de um sistema de software oferecido como um serviço

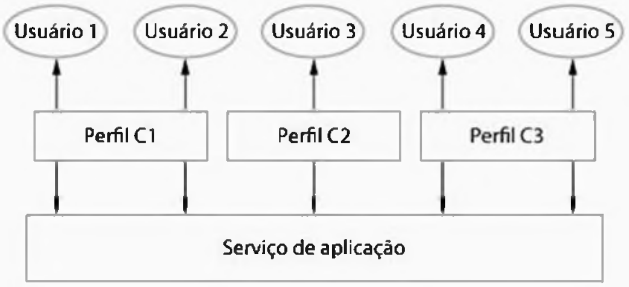


Figura 18.15 Um banco de dados de multilocações

Locatário	Chave	Nome	Endereço
234	C100	XYZ Corp	43, Anystreet, Sometown
234	C110	BigCorp	2, Main St, Motown
435	X234	J. Bowie	56, Mill St, Starville
592	PP37	R. Burns	Alloway, Ayrshire

A escalabilidade é a capacidade do sistema de lidar com o aumento do número de usuários sem reduzir o QoS global que é entregue a qualquer usuário. Geralmente, ao considerar a escalabilidade no contexto do SaaS, você está considerando 'escalamento para fora' ao invés de 'escalamento para cima'. Lembre-se de que 'escalamento para fora' significa adicionar servidores adicionais e, assim, também, aumentar o número de transações que podem ser processadas em paralelo. A escalabilidade é um tópico complexo, o qual não discuto em detalhes mas algumas diretrizes gerais para a implementação de softwares escaláveis são:

1. Desenvolva aplicações em que cada componente é implementado como um serviço simples, sem estado, o qual pode ser executado em qualquer servidor. Portanto, no decurso de uma única transação, um usuário pode interagir com instâncias do mesmo serviço em execução em diversos servidores.
2. Projete o sistema usando a interação assíncrona para que a aplicação não tenha de esperar o resultado de uma interação (por exemplo, uma solicitação de leitura). Isso permite que a aplicação continue a realizar um trabalho útil enquanto está aguardando a interação terminar.
3. Gerencie recursos, como conexões de rede e banco de dados, como um *pool*, para que nenhum servidor específico corra o risco de ficar sem recursos.
4. Projete seu banco de dados para permitir o bloqueio de baixa granularidade. Ou seja, não bloqueie registros inteiros no banco de dados quando apenas parte de um registro está em uso.

A noção de SaaS é uma grande mudança de paradigma para a computação distribuída. Ao invés de uma organização que hospeda várias aplicações em seus servidores, SaaS permite que essas aplicações sejam fornecidas externamente, por diferentes fornecedores. Estamos no meio da transição de um modelo para outro, e é provável que, no futuro, esse processo tenha um efeito significativo sobre a engenharia de sistemas de software corporativos.

PONTOS IMPORTANTES

- Os benefícios de sistemas distribuídos são que eles podem ser dimensionados para lidar com o aumento da demanda, podem continuar a fornecer serviços de usuário (mesmo que algumas partes do sistema falhem) e habilitam o compartilhamento de recursos.
- Algumas questões importantes no projeto de sistemas distribuídos incluem transparência, abertura, escalabilidade, proteção, qualidade de serviço e gerenciamento de falhas.
- Os sistemas cliente-servidor são sistemas distribuídos em que o sistema está estruturado em camadas, com a camada de apresentação implementada em um computador cliente. Os servidores fornecem serviços de gerenciamento de dados, de aplicações e de banco de dados.
- Os sistemas cliente-servidor podem ter várias camadas, com diferentes camadas do sistema distribuídas em computadores diferentes.
- Os padrões de arquitetura para sistemas distribuídos incluem arquiteturas mestre-escravo, arquiteturas cliente-servidor de duas camadas e de múltiplas camadas, arquiteturas de componentes distribuídos e arquiteturas ponto-a-ponto.
- Os componentes de sistemas distribuídos requerem *middleware* para lidar com as comunicações de componentes e para permitir que componentes sejam adicionados e removidos do sistema.
- As arquiteturas ponto-a-ponto são arquiteturas descentralizadas em que não existe distinção entre clientes e servidores. As computações podem ser distribuídas ao longo de muitos sistemas, em organizações diferentes.
- O software como um serviço é uma maneira de implantar aplicações como sistemas cliente-magro-servidor, em que o cliente é um *browser* de Web.

LEITURA COMPLEMENTAR

'Middleware: A model for distributed systems services'. Embora um pouco ultrapassado em partes, esse é um excelente artigo, que oferece uma visão geral e resume o papel do *middleware* em sistemas distribuídos, além de discutir a gama de serviços de *middleware* que podem ser fornecidos. (BERNSTEIN, P. A. *Comm. ACM*, v. 39, n. 2, fev. 1996.) Disponível em: <http://dx.doi.org/10.1145/230798.230809>.

Peer-to-Peer: Harnessing the Power of Disruptive Technologies. Embora esse livro não tenha muita informação sobre arquiteturas p2p, ele é uma excelente introdução à computação p2p e discute a organização e a aborda-

gem usadas em vários sistemas p2p. (ORAM, R. (Org.). *Peer-to-Peer: Harnessing the Power of Disruptive Technologies*. O'Reilly and Associates Inc., 2001.)

'Turning software into a service'. Um artigo com uma boa visão geral que discute os princípios da computação orientada a serviços. Ao contrário de muitos artigos sobre o tema, esse não esconde esses princípios por trás de uma discussão sobre os padrões envolvidos. (TURNER, M.; BUDGEN, D.; BRERETON, P. *IEEE Computer*, v. 36, n. 10, out. 2003.) Disponível em: <http://dx.doi.org/10.1109/MC.2003.1236470>.

Distributed Systems: Principles and Paradigms, 2nd edition. Um livro-texto abrangente que aborda todos os aspectos do projeto e implementação de sistemas distribuídos. No entanto, ele não inclui muitas discussões sobre o paradigma orientado a serviços. (TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems: Principles and Paradigms*. 2. ed. Addison-Wesley, 2007.)

'Software as a Service; The Spark that will Change Software Engineering'. Um pequeno artigo que argumenta que o advento do SaaS vai impulsionar todo o desenvolvimento de software para um modelo iterativo. (GOTH, G. *Distributed Systems Online*, v. 9, n. 7, jul. 2008.) Disponível em: <http://dx.doi.org/10.1109/MDSO.2008.21>.

EXERCÍCIOS

18.1 O que você entende por 'escalabilidade'? Discuta as diferenças entre 'escalabilidade para cima' e 'escalabilidade para fora' e explique quando essas diferentes abordagens para a escalabilidade podem ser usadas.

18.2 Explique por que sistemas de software distribuídos são mais complexos do que os sistemas de software centralizados, em que toda a funcionalidade de sistema é implementada em um único computador.

18.3 Usando um exemplo de uma chamada de procedimento remoto, explique como o *middleware* coordena a interação entre os computadores em um sistema distribuído.

18.4 Qual é a diferença fundamental entre a abordagem cliente-gordo e a abordagem cliente-magro para as arquiteturas de sistemas cliente-servidor?

18.5 Foi solicitada a criação de um sistema protegido que requer autorização e autenticação forte. O sistema deve ser projetado para que as comunicações entre as partes do sistema não possam ser interceptadas e lidas por um invasor. Sugira a arquitetura cliente-servidor mais adequada para esse sistema e, justificando sua resposta, proponha como a funcionalidade deve ser distribuída entre os sistemas cliente e servidor.

18.6 Seu cliente quer desenvolver um sistema de informações de estoque em que os revendedores possam acessar informações sobre empresas e avaliar diferentes cenários de investimento por meio de um sistema de simulação. Cada revendedor(a) usa essa simulação de forma diferente, de acordo com sua experiência e o tipo de estoque em questão. Sugira uma arquitetura cliente-servidor para esse sistema que mostre onde se encontra a funcionalidade. Justifique o modelo do sistema cliente-servidor que você selecionou.

18.7 Usando uma abordagem de componentes distribuídos, proponha uma arquitetura para um sistema nacional de reserva para teatro. Os usuários podem verificar a disponibilidade e reservar os assentos em um grupo de teatros. O sistema deve aceitar a devolução de bilhetes, assim, as pessoas podem devolver seus bilhetes para vendas de última hora para outros clientes.

18.8 Apresente duas vantagens e duas desvantagens de arquiteturas ponto-a-ponto descentralizadas e semicentralizadas.

18.9 Explique por que implantar software como um serviço pode reduzir os custos de suporte de TI para uma empresa. Quais custos adicionais podem surgir caso esse modelo de implantação seja usado?

18.10 Sua empresa pretende parar de usar aplicações de *desktop* para acessar a mesma funcionalidade remotamente, como serviços. Identifique três riscos que podem surgir e dê sugestões de como diminuir esses riscos.

REFERÊNCIAS

BERNSTEIN, P. A. Middleware: A Model for Distributed System Services. *Comm. ACM*, v. 39, n. 2, 1996, p. 86-97.

COULOURIS, G.; DOLLIMORE, J.; KINDBERG, T. *Distributed Systems*: Concepts and Design, 4.ed. Harlow, Reino Unido: Addison-Wesley, 2005.

HOLDENER, A. T. *Ajax*: The Definitive Guide. Sebastopol, Calif.: O'Reilly and Associates, 2008.

McDOUGALL, P. The Power of Peer-To-Peer. *Information Week*, 28 ago. 2000.

NEUMAN, B. C. Scale in Distributed Systems. In: CASAVANT, T.; SINGAL, M. (Orgs.). *Readings in Distributed Computing Systems*. Los Alamitos, Calif.: IEEE Computer Society Press, 1994.

ORAM, A. Peer-to-Peer: Harnessing the Benefits of a Disruptive Technology, 2001.

ORFALI, R.; HARKEY, D. *Client/server Programming with Java and CORBA*. Nova York: John Wiley & Sons, 1998.

ORFALI, R.; HARKEY, D.; EDWARDS, J. *Instant CORBA*. Chichester, Reino Unido: John Wiley & Sons, 1997.

POPE, A. *The CORBA Reference Guide*: Understanding the Common Request Broker Architecture. Boston: Addison-Wesley, 1997.

TANENBAUM, A. S.; VAN STEEN, M. *Distributed Systems*: Principles and Paradigms, 2.ed. Upper Saddle River, NJ: Prentice Hall, 2007.