

SISTEMAS DISTRIBUÍDOS

princípios e paradigmas

2^a edição

Andrew S. Tanenbaum
Maarten Van Steen

Tradução

Arlete Simille Marques
Engenheira Química — UFPR

Revisão Técnica

Wagner Luiz Zucchi

Professor doutor do Departamento de Sistemas Eletrônicos da
Escola Politécnica da Universidade de São Paulo



São Paulo

Brasil Argentina Colômbia Costa Rica Chile Espanha Guatemala México Peru Porto Rico Venezuela



Sumário

Prefácio IX

1. Introdução 1

1.1 Definição de um sistema distribuído	1
1.2 Metas.....	2
1.3 Tipos de sistemas distribuídos	10
1.4 Resumo	18

2. Arquiteturas 20

2.1 Estilos arquitetônicos	20
2.2 Arquiteturas de sistemas	22
2.3 Arquiteturas <i>versus</i> middleware.....	32
2.4 Autogerenciamento em sistemas distribuídos.....	35

3. Processos 42

3.1 Threads.....	42
3.2 Virtualização	48
3.3 Clientes	50
3.4 Servidores	53
3.5 Migração de código	62
3.6 Resumo	67

4. Comunicação 69

4.1 Fundamentos.....	69
4.2 Chamada de procedimento remoto.....	75
4.3 Comunicação orientada a mensagem	84
4.4 Comunicação orientada a fluxo	95
4.6 Resumo	105

5. Nomeação 108

5.1 Nomes, identificadores e endereços	108
5.2 Nomeação simples	110
5.3 Nomeação estruturada	118
5.4 Nomeação baseada em atributo	131
5.5 Resumo	137

6. Sincronização 140

6.1 Sincronização de relógios	140
6.2 Relógios lógicos.....	147
6.3 Exclusão mútua.....	152
6.4 Posicionamento global de nós	157
6.5 Algoritmos de eleição	159
6.6 Resumo	163

7. Consistência e replicação 165

7.1 Introdução	165
7.2 Modelos de consistência centrados em dados	167
7.3 Modelos de consistência centrados no cliente.....	174
7.5 Protocolos de consistência.....	178
7.6 Resumo	185

8. Tolerância a falha 194

8.1 Introdução à tolerância a falha	194
8.2 Resiliência de processo.....	198

VIII Sistemas distribuídos

8.3 Comunicação confiável cliente–servidor	203
8.4 Comunicação confiável de grupo	207
8.5 Comprometimento distribuído.....	215
8.6 Recuperação.....	220
8.7 Resumo	226

9. Segurança 228

9.1 Introdução à segurança	228
9.2 Canais seguros	240
9.3 Controle de acesso	250
9.4 Gerenciamento da segurança	259
9.5 Resumo	266

10. Sistemas distribuídos baseados em objetos 268

10.1 Arquitetura	268
10.2 Processos.....	272
10.3 Comunicação	275
10.4 Nomeação	281
10.5 Sincronização.....	284
10.6 Consistência e replicação.....	285
10.7 Tolerância a falha.....	288
10.8 Segurança.....	291
10.9 Resumo	294

11. Sistemas de arquivos distribuídos 296

11.1 Arquitetura.....	296
11.2 Processos	302
11.3 Comunicação	303
11.4 Nomeação	306
11.5 Sincronização	310
11.6 Consistência e replicação	314
11.7 Tolerância a falha	320
11.8 Segurança	322
11.9 Resumo	328

12. Sistemas distribuídos baseados na Web 330

12.1 Arquitetura	330
12.2 Processos.....	335
12.3 Comunicação	339
12.4 Nomeação	344
12.5 Sincronização.....	345
12.6 Consistência e replicação	346
12.7 Tolerância a falha	353
12.8 Segurança	354
12.9 Resumo	355

13. Sistemas distribuídos baseados em coordenação 357

13.1 Introdução a modelos de coordenação	357
13.2 Arquiteturas	358
13.3 Processos	364
13.4 Comunicação	364
13.5 Nomeação	365
13.6 Sincronização	367
13.7 Consistência e replicação	367
13.8 Tolerância a falha	371
13.9 Segurança	374
13.10 Resumo	375

14. Sugestões de leitura adicional e bibliografia 377

14.1 Sugestões para leitura adicional	377
14.2 Bibliografia em ordem alfabética.....	382

Prefácio

Sistemas distribuídos são uma área da ciência da computação que está mudando rapidamente. Nos últimos anos vêm surgindo novos tópicos muito interessantes, como computação peer-to-peer e redes de sensores, enquanto outros amadureceram muito, como serviços Web e aplicações Web em geral. Mudanças como essas exigiram uma revisão de nosso texto original para que ficasse atualizado.

Esta segunda edição reflete uma grande revisão em comparação com a anterior. Adicionamos um capítulo específico sobre arquiteturas, que reflete o progresso alcançado na organização de sistemas distribuídos. Uma outra diferença importante é que, agora, há muito mais material sobre sistemas descentralizados, em particular computação peer-to-peer. Não nos limitamos a discutir apenas as técnicas básicas, mas também demos atenção às suas aplicações, como compartilhamento de arquivo, divulgação de informações, redes de entrega de conteúdo e sistemas publicar/subscrever.

Assim como esses dois assuntos importantes, ainda discutimos novos assuntos em todo o livro. Por exemplo, adicionamos material sobre redes de sensores, virtualização, clusters de servidores e computação em grade. Demos atenção especial ao autogerenciamento de sistemas distribuídos, um tópico cada vez mais importante dado o contínuo crescimento da escala desses sistemas.

Certamente, modernizamos o material onde foi adequado. Por exemplo, quando discutimos consistência e replicação, era agora focalizamos modelos de consistência mais apropriados para sistemas distribuídos modernos mais do que os modelos originais, que foram talhados para computação distribuída de alto desempenho. Da mesma maneira, adicionamos material sobre modernos algoritmos distribuídos, entre eles algoritmos de sincronização de relógio e de localização baseados em GPS.

Embora isso não seja comum, conseguimos *reduzir* o número total de páginas. Essa redução é causada, em parte, pela exclusão de assuntos como coleta distribuída de lixo e protocolos de pagamento eletrônico, e também pela reorganização dos quatro últimos capítulos.

Como na edição anterior, o livro é dividido em duas partes. Princípios de sistemas distribuídos são discutidos do Capítulo 2 ao Capítulo 9, enquanto as abordagens globais dos modos de desenvolvimento de aplicações distribuídas (os paradigmas) são discutidas do Capítulo 10 ao Capítulo 13. Entretanto, diferente da edição anterior, decidimos não discutir estudos de casos completos nos capítulos dedicados a paradigmas. Em vez disso, agora cada princípio é explicado por meio de um caso representativo. Por exemplo, invocações de objeto são discutidas como um princípio de comunicação no Capítulo 10 sobre sistemas distribuídos baseados em objetos. Essa abordagem nos permitiu condensar o material, mas também tornou a leitura e o estudo mais agradáveis.

Claro que continuamos a recorrer extensivamente à prática para explicar o que são, afinal, sistemas distribuídos. Vários aspectos de sistemas utilizados na vida real, como WebSphere MQ, DNS, GPS, Apache, Corba, Ice, NFS, Akamai, TIB/Rendezvous, Jini e muitos outros são discutidos em todo o livro. Esses exemplos ilustram a tênue divisão entre teoria e prática, que torna a área de sistemas distribuídos tão interessante.

Muitas pessoas contribuíram para este livro de diversas maneiras. Gostaríamos de agradecer em especial a D. Robert Adams, Arno Bakker, Coskun Bayrak, Jacques Chassin de Kercommeaux, Randy Chow, Michel Chaudron, Puneet Singh Chawla, Fabio Costa, Cong Du, Dick Epema, Kevin Fenwick, Chandana Gamage, Ali Ghodsi, Giorgio Ingargiola, Mark Jelasity, Ahmed Kamel, Gregory Kapfhammer, Jeroen Ketema, Onno Kubbe, Patricia Lago, Steve MacDonald, Michael J. McCarthy, M. Tamer Ozsu, Guillaume Pierre, Avi Shahar, Swaminathan Sivasubramanian, Chintan Shah, Ruud Stegers, Paul Tymann, Craig E. Wills, Reuven Yagel e Dakai Zhu pela leitura de partes do manuscrito e por terem ajudado a identificar erros cometidos na edição anterior e oferecido comentários úteis.

Por fim, gostaríamos de agradecer a nossas famílias. Até agora, Suzanne já passou por esse processo dezessete vezes. É muito para mim, mas também para ela. Ela nunca disse: “Agora, chega!”, embora eu tenha certeza de que teve vontade de dizer. Obrigado. Agora, Barbara e Marvin têm uma idéia muito melhor do que professores fazem para viver e sabem quais são as diferenças entre um bom e um mau livro didático. Agora eles são uma inspiração para eu tentar produzir mais bons do que maus livros (AST).

Como tirei uma licença para atualizar o livro, o trabalho de escrever também ficou muito mais agradável para Mariëlle. Ela está apenas começando a se acostumar com ele, mas continua a me dar suporte e a me alertar quando é

tempo de voltar minha atenção a assuntos mais importantes. Eu lhe devo muito. A essa altura, Max e Elke já têm uma idéia muito melhor do que significa escrever um livro, porém, em comparação com o que eles mesmos estão lendo, acham difícil entender o que há de tão interessante nessas coisas estranhas que chamamos sistemas distribuídos. E eu não posso culpá-los (MvS).

Companion Website



O Companion Website desta edição (www.prenhall.com/tanenbaum_br) oferece um conjunto completo de apresentações em PowerPoint para auxiliar os professores a preparar suas aulas, assim como manual de soluções em inglês. Esse material pode ser acessado por meio de uma senha, e, para obtê-la, os professores devem entrar em contato com um representante Pearson ou enviar um e-mail para universitarios@pearsoned.com.

Os editores da edição brasileira agradecem a preciosa colaboração do professor Fabio Kon.

Introdução

 Os sistemas de computação estão passando por uma revolução. Desde 1945, quando começou a era moderna dos computadores, até aproximadamente 1985, os computadores eram grandes e caros. Mesmo os minicomputadores custavam no mínimo dezenas de milhares de dólares cada. O resultado é que a maioria das organizações tinha apenas alguns poucos computadores e, na falta de um modo de conectá-los, eles funcionavam independentemente uns dos outros.

 Entretanto, mais ou menos a partir de meados da década de 1980, dois avanços tecnológicos começaram a mudar essa situação. O primeiro foi o desenvolvimento de microprocessadores de grande capacidade. De início, eram máquinas de 8 bits, mas logo se tornaram comuns CPUs de 16, 32 e 64 bits. Muitas dessas CPUs tinham a capacidade de computação de um mainframe — isto é, um grande computador central —, mas por uma fração do preço dele.

 A quantidade de melhorias que ocorreu na tecnologia de computadores nos últimos 50 anos é verdadeiramente assombrosa e totalmente sem precedentes em outros setores. De uma máquina que custava dez milhões de dólares e executava uma instrução por segundo, chegamos a máquinas que custam mil dólares e podem executar um bilhão de instruções por segundo, um ganho preço/desempenho de 10^{13} . Se os carros tivessem melhorado nessa proporção no mesmo período de tempo, um Rolls Royce custaria agora um dólar e faria um bilhão de milhas por galão. (Infelizmente, é provável que também tivesse um manual de 200 páginas para ensinar como abrir a porta.)

 O segundo desenvolvimento foi a invenção de redes de computadores de alta velocidade. **Redes locais**, ou **LANs (local-area networks)**, permitem que centenas de máquinas localizadas dentro de um edifício sejam conectadas de modo tal que pequenas quantidades de informação possam ser transferidas entre máquinas em alguns microssegundos, ou algo parecido. Maiores quantidades de dados podem ser movimentadas entre máquinas a taxas de 100 milhões a 10 bilhões de bits/s. **Redes de longa distância**, ou **WANs (wide-area networks)**, permitem que milhões de máquinas no mundo inteiro se conectem a velocidades que variam de 64 Kbits/s a gigabits por segundo.

 O resultado dessas tecnologias é que, atualmente, não sómente é viável, mas também fácil, montar sistemas de computação compostos por grandes quantidades de computadores conectados por uma rede de alta velocidade. Esses sistemas costumam ser denominados **redes de computadores** ou **sistemas distribuídos**, em comparação com os **sistemas centralizados** (ou **sistemas monoprocessadores**) anteriores, que consistem em um único computador, seus periféricos e, talvez, alguns terminais remotos.

1.1 Definição de um Sistema Distribuído

Várias definições de sistemas distribuídos já foram dadas na literatura, nenhuma delas satisfatória e de acordo com nenhuma das outras. Para nossa finalidade, é suficiente dar uma caracterização sem ser muito específica:

Um sistema distribuído é um conjunto de computadores independentes que se apresenta a seus usuários como um sistema único e coerente.

 Essa definição tem vários aspectos importantes. O primeiro é que um sistema distribuído consiste em componentes (isto é, computadores) autônomos. Um segundo aspecto é que os usuários, sejam pessoas ou programas, acham que estão tratando com um único sistema. Isso significa que, de um modo ou de outro, os componentes autônomos precisam colaborar. Como estabelecer essa colaboração é o cerne do desenvolvimento de sistemas distribuídos. Observe que nenhuma premissa é adotada em relação ao tipo de computadores. Em princípio, até mesmo dentro de um único sistema, eles poderiam variar desde computadores centrais (mainframes) de alto desempenho até pequenos nós em redes de sensores. Da mesma maneira, nenhuma premissa é adotada quanto ao modo como os computadores são interconectados. Voltaremos a esses aspectos mais adiante neste capítulo.

Em vez de continuar com definições, talvez seja mais útil que nos concentremos em características importantes de sistemas distribuídos. Uma característica importante é que as diferenças entre os vários computadores e o modo como eles se comunicam estão, em grande parte,

2 Sistemas distribuídos

ocultas aos usuários. O mesmo vale para a organização interna do sistema distribuído. Uma outra característica importante é que usuários e aplicações podem interagir com um sistema distribuído de maneira consistente e uniforme, independentemente de onde a interação ocorra.

Em princípio, também deveria ser relativamente fácil expandir ou aumentar a escala de sistemas distribuídos. Essa característica é uma consequência direta de ter computadores independentes, porém, ao mesmo tempo, de ocultar como esses computadores realmente fazem parte do sistema como um todo. Em geral, um sistema distribuído estará continuamente disponível, embora algumas partes possam estar temporariamente avariadas. Usuários e aplicações não devem perceber quais são as partes que estão sendo substituídas ou consertadas, ou quais são as novas partes adicionadas para atender a mais usuários ou aplicações.

Para suportar computadores e redes heterogêneos e, simultaneamente, oferecer uma visão de sistema único, os sistemas distribuídos costumam ser organizados por meio de uma camada de software — que é situada logicamente entre uma camada de nível mais alto, composta de usuários e aplicações, e uma camada subjacente, que consiste em sistemas operacionais e facilidades básicas de comunicação, como mostra a Figura 1.1. Por isso, tal sistema distribuído às vezes é denominado **middleware**.

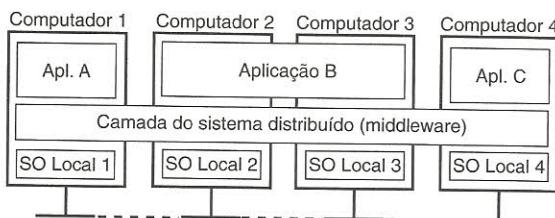


Figura 1.1 Sistema distribuído organizado como middleware. A camada de middleware se estende por várias máquinas e oferece a mesma interface a cada aplicação.

A Figura 1.1 mostra quatro computadores em rede e três aplicações, das quais a aplicação *B* é distribuída para os computadores 2 e 3. A mesma interface é oferecida a cada aplicação. O sistema distribuído proporciona os meios para que os componentes de uma única aplicação distribuída se comuniquem uns com os outros, mas também permite que diferentes aplicações se comuniquem. Ao mesmo tempo, ele oculta, do melhor e mais razoável modo possível, as diferenças em hardware e sistemas operacionais para cada aplicação.

1.2 Metas

O fato de ser possível montar sistemas distribuídos não quer dizer necessariamente que essa seja uma boa idéia. Afinal, dada a tecnologia corrente, também é possí-



blocar quatro drives de disco flexível em um computador pessoal. A questão é que não teria sentido fazer isso. Nesta seção, discutiremos quatro metas importantes que devem ser cumpridas na construção de um sistema distribuído para que valha a pena o esforço. Um sistema distribuído deve oferecer fácil acesso a seus recursos; deve ocultar razoavelmente bem o fato de que os recursos são distribuídos por uma rede; deve ser aberto e deve poder ser expandido.

1.2.1 Acesso a recursos

A principal meta de um sistema distribuído é facilitar aos usuários, e às aplicações, o acesso a recursos remotos e seu compartilhamento de maneira controlada e eficiente. Os recursos podem ser muito abrangentes, mas entre os exemplos típicos estão impressoras, computadores, facilidades de armazenamento, dados, páginas Web e redes, só para citar alguns. Há muitas razões para querer compartilhar recursos, e uma razão óbvia é a economia. Por exemplo, é mais barato permitir que uma impressora seja compartilhada por diversos usuários em um pequeno escritório do que ter de comprar e manter uma impressora direcionada a cada usuário. Do mesmo modo, em termos econômicos, faz sentido compartilhar recursos de alto custo como supercomputadores, sistemas de armazenamento de alto desempenho, imagesetters e outros periféricos caros.

Conectar usuários e recursos também facilita a colaboração e a troca de informações, o que é claramente ilustrado pelo sucesso da Internet com seus protocolos simples para trocar arquivos, correio, documentos, áudio e vídeo. Agora, a conectividade da Internet está levando a várias organizações virtuais nas quais grupos de pessoas muito dispersas geograficamente trabalham juntas por meio de **groupware**, isto é, software para edição colaborativa, teleconferência e assim por diante. Da mesma maneira, a conectividade da Internet possibilitou o comércio eletrônico, que nos permite comprar e vender todos os tipos de mercadoria sem ter de realmente ir a uma loja ou até mesmo sair de casa.

Contudo, à medida que a conectividade e o compartilhamento crescem, a segurança se torna cada vez mais importante. Na prática corrente, os sistemas oferecem pouca proteção contra bisbilhotice ou intrusão na comunicação. Senhas e outras informações sensíveis muitas vezes são enviadas como texto comum — isto é, não criptografado — pela rede, ou armazenadas em servidores que podemos apenas esperar que sejam confiáveis. Nesse sentido, há muito espaço para melhorias. Por exemplo, hoje é possível fazer pedidos de mercadorias informando apenas um número de cartão de crédito. Raramente é solicitada uma prova de que o cliente seja o proprietário do cartão. No futuro, fazer pedidos desse modo só será possível se você realmente provar que possui o cartão em mãos, inserindo-o em uma leitora de cartões.

 Um outro problema de segurança é o rastreamento de comunicações para montar um perfil de preferências de um usuário específico (Wang et al., 1998). Esse rastreamento é uma violação explícita da privacidade, em especial se for feito sem avisar o usuário. Um problema relacionado é que maior conectividade também pode resultar em comunicação indesejável, como envio de mala direta sem permissão, muitas vezes denominada *spam*. Nesses casos, talvez precisemos nos proteger usando filtros especiais de informações que selecionam mensagens de entrada com base em seu conteúdo.

1.2.2 Transparência da distribuição

Uma meta importante de um sistema distribuído é ocultar o fato de que seus processos e recursos estão fisicamente distribuídos por vários computadores. Um sistema distribuído que é capaz de se apresentar a usuários e aplicações como se fosse apenas um único sistema de computador é denominado **transparente**. Em primeiro lugar, vamos examinar quais são os tipos de transparência que existem em sistemas distribuídos. Em seguida, abordaremos a questão mais geral sobre a decisão de se a transparência é sempre requerida.

Tipos de transparência

O conceito de transparência pode ser aplicado a diversos aspectos de um sistema distribuído — os mais importantes são mostrados na Tabela 1.1.

Transparência	Descrição
Acesso	Oculta diferenças na representação de dados e no modo de acesso a um recurso
Localização	Oculta o lugar em que um recurso está localizado
Migração	Oculta que um recurso pode ser movido para outra localização
Relocação	Oculta que um recurso pode ser movido para uma outra localização enquanto em uso
Replicação	Oculta que um recurso é replicado
Concorrência	Oculta que um recurso pode ser compartilhado por diversos usuários concorrentes
Falha	Oculta a falha e a recuperação de um recurso

Tabela 1.1 Diferentes formas de transparência em um sistema distribuído (ISO, 1995).

Transparência de acesso trata de ocultar diferenças em representação de dados e o modo como os recursos podem ser acessados por usuários. Em um nível básico, desejamos ocultar diferenças entre arquiteturas de máquinas, porém o mais importante é chegar a um acordo sobre como os dados devem ser representados por máquinas e sistemas operacionais diferentes. Por exemplo, um sistema distribuído pode ter sistemas de computação que executam sistemas operacionais diferentes, cada um com suas próprias convenções para nomeação de arquivos.

Diferenças entre convenções de nomeação e também o modo como os arquivos devem ser manipulados devem ficar ocultos dos usuários e das aplicações.

Um importante grupo de tipos de transparência tem a ver com a localização de um recurso. **Transparência de localização** refere-se ao fato de que os usuários não podem dizer qual é a localização física de um recurso no sistema. A nomeação desempenha um papel importante para conseguir transparência de localização.

Em particular, pode-se conseguir transparência de localização ao se atribuir somente nomes lógicos aos recursos, isto é, nomes nos quais a localização de um recurso não está secretamente codificada. Um exemplo desse tipo de nome é o URL <http://www.prenhall.com/index.html>, que não dá nenhuma pista sobre a localização do principal servidor Web da Prentice Hall. O URL também não dá nenhuma pista se *index.html* sempre esteve em sua localização corrente ou se foi transferido para lá recentemente. Diz-se que sistemas distribuídos nos quais recursos podem ser movimentados sem afetar o modo como podem ser acessados proporcionam **transparência de migração**. Ainda mais vantajosa é a situação na qual recursos podem ser relocados *enquanto* estão sendo acessados sem que o usuário, diz-se que o sistema suporta **transparência de relocação**. Um exemplo de transparência de relocação é o uso móvel de laptops sem fio, cujos usuários podem continuar a usá-lo quando vão de um lugar a outro sem sequer se desconectar temporariamente.

Como veremos, a replicação desempenha um papel muito importante em sistemas distribuídos. Por exemplo, recursos podem ser replicados para aumentar a disponibilidade ou melhorar o desempenho colocando uma cópia perto do lugar em que ele é acessado. **Transparência de replicação** está relacionada a ocultar o fato de que existem várias cópias de um recurso. Para ocultar a replicação dos usuários, é necessário que todas as réplicas tenham o mesmo nome. Por consequência, um sistema que suporta transparência de replicação em geral também deve suportar transparência de localização porque, caso contrário, seria impossível referir-se a réplicas em diferentes localizações.

Já mencionamos que uma importante meta de sistemas distribuídos é permitir compartilhamento de recursos. Em muitos casos, esse compartilhamento é cooperativo, como no caso da comunicação. Todavia, também há muitos exemplos de compartilhamento competitivo de recursos. Um deles pode ser o caso de dois usuários independentes, em que cada um pode ter armazenado seus arquivos no mesmo servidor de arquivos ou pode acessar as mesmas tabelas em um banco de dados compartilhado. Nesses casos, é importante que cada usuário não perceba que o outro está utilizando o mesmo recurso. Esse fenômeno é denominado **transparência de concorrência**. Uma questão importante é que o acesso concorrente a um recurso compartilhado deixe esse recurso em estado con-

sistente. Pode-se conseguir consistência por meio de travas de acesso, o que dá a cada usuário, um por vez, acesso exclusivo ao recurso desejado. Um mecanismo mais refinado é utilizar transações; porém, como veremos em capítulos posteriores, é bastante difícil implementar transações em sistemas distribuídos.

Uma definição alternativa e popular de um sistema distribuído, devida a Leslie Lamport, é “Você sabe que tem um quando a falha de um computador do qual nunca ouviu falar impede que você faça qualquer trabalho”. Essa descrição pontua uma outra questão importante no projeto de sistemas distribuídos: como tratar falhas. Fazer com que um sistema distribuído seja **transparente à falha** significa que um usuário não percebe que um recurso (do qual possivelmente nunca ouviu falar) deixou de funcionar bem e que, subsequenteamente, o sistema se recuperou da falha. Mascarar falhas é uma das questões mais difíceis em sistemas distribuídos e até mesmo impossíveis de se realizar quando são adotadas certas premissas aparentemente realistas, como discutiremos no Capítulo 8. A principal dificuldade para mascarar falhas está na incapacidade de distinguir entre um recurso morto e um recurso insuportavelmente lento. Por exemplo, quando contatamos um servidor Web ocupado, a certa altura o tempo do browser se esgotará e ele avisará que a página Web não está disponível. Nesse ponto, o usuário não pode concluir se, na verdade, o servidor está avariado.



Grau de transparência

Embora a transparência de distribuição seja geralmente considerada preferível para qualquer sistema distribuído, há situações em que tentar ocultar completamente dos usuários todos os aspectos da distribuição não é uma boa idéia. Um exemplo é requisitar que seu jornal eletrônico apareça em sua caixa postal antes das 7 horas da manhã, hora local, como sempre, enquanto naquele momento você está no outro lado do mundo, onde o fuso horário é diferente. Seu jornal matutino não será aquele ao qual você está acostumado.

Da mesma maneira, não se pode esperar que um sistema distribuído de longa distância que conecta um processo em San Francisco a um processo em Amsterdã oculte o fato de que a Mãe Natureza não permitirá que ele envie uma mensagem de um processo para o outro em menos do que aproximadamente 35 milissegundos. Na prática, quando se está usando uma rede de computadores, isso levará várias centenas de milissegundos. A transmissão de sinal não somente está limitada pela velocidade da luz, mas também pelas capacidades de processamento dos comutadores intermediários.

Também há um compromisso entre um alto grau de transparência e o desempenho de um sistema. Por exemplo, muitas aplicações de Internet tentam contatar um servidor repetidas vezes antes de finalmente desistir. Por consequência, tentar mascarar uma falha transitória de

servidor antes de tentar um outro pode reduzir a velocidade do sistema como um todo. Nesse caso, talvez seja melhor desistir mais cedo ou, ao menos, permitir que o usuário cancele as tentativas para fazer contato.

Um outro exemplo é o de precisar garantir que várias réplicas, localizadas em continentes diferentes, devam ser consistentes o tempo todo. Em outras palavras, se uma cópia for alterada, essa alteração deve ser propagada para todas as cópias antes de permitir qualquer outra operação. É claro que, agora, uma única operação de atualização pode demorar até alguns segundos para ser concluída, algo que não pode ser ocultado dos usuários.

Por fim, há situações em que não é nem um pouco óbvio que ocultar distribuição seja uma boa idéia. À medida que sistemas distribuídos estão se expandindo para dispositivos que as pessoas carregam consigo, e a própria noção de localização e percepção de contexto está se tornando cada vez mais importante, pode ser melhor *expor* a distribuição em vez de tentar ocultá-la. Essa exposição de distribuição ficará mais evidente quando discutirmos sistemas distribuídos embutidos e ubíquos neste capítulo. Como um exemplo simples, considere uma funcionária que quer imprimir um arquivo que está em seu notebook. É melhor enviar o trabalho a ser impresso a uma impressora ocupada que está próxima do que a uma desocupada localizada na sede corporativa em um país diferente.

Também há outros argumentos contra a transparência de distribuição. Reconhecendo que a total transparência de distribuição é simplesmente impossível, devemos nos perguntar se é até mesmo sensato *dissimular* que podemos alcançá-la. Talvez seja muito melhor tornar a distribuição explícita, de modo que o usuário e o desenvolvedor de aplicação nunca sejam levados a acreditar que existe uma coisa denominada transparência. O resultado será que os usuários entenderão de maneira mais clara o comportamento, às vezes inesperado, de um sistema distribuído e, por isso, estarão mais bem preparados para lidar com esse comportamento.

A *comunicação* é que visar à transparência de distribuição pode ser uma bela meta no projeto e na implementação de sistemas distribuídos, mas deve ser considerada em conjunto com outras questões, como desempenho e facilidade de compreensão. Nem sempre vale a pena tentar conseguir total transparência, pois o preço que se paga por isso pode ser surpreendentemente alto.

1.2.3 Abertura

Uma outra meta importante de sistemas distribuídos é a abertura. Um **sistema distribuído aberto** é um sistema que oferece serviços de acordo com regras padronizadas que descrevem a sintaxe e a semântica desses serviços. Por exemplo, em redes de computadores há regras padronizadas que governam o formato, o conteúdo e o significado de mensagens enviadas e recebidas. Tais regras são formalizadas em protocolos. No caso de siste-

mas distribuídos, em geral os serviços são especificados por meio de **interfaces**, que costumam ser descritas em uma **linguagem de definição de interface (Interface Definition Language — IDL)**.

Definições de interfaces escritas em IDL quase sempre capturam apenas a sintaxe de serviços. Em outras palavras, elas especificam com precisão os nomes das funções que estão disponíveis, junto com os tipos dos parâmetros, os valores de retorno, as possíveis exceções que podem surgir e assim por diante. A parte difícil é especificar com precisão o que esses serviços fazem, isto é, a semântica das interfaces. Na prática, tais especificações são sempre dadas de modo informal por meio de linguagem natural.

Se adequadamente especificada, uma definição de interface permite que um processo arbitrário que necessita de certa interface se comunique com um outro processo que fornece aquela interface. Permite também que duas partes independentes construam implementações completamente diferentes dessas interfaces, o que resulta em dois sistemas distribuídos separados que funcionam exatamente do mesmo modo.

Especificações adequadas são completas e neutras. Completa significa que tudo que é necessário para uma implementação foi, de fato, especificado. Contudo, muitas definições de interface não são absolutamente completas, portanto é necessário que um desenvolvedor adicione detalhes específicos da implementação. De igual importância é o fato de que especificações não prescrevem como deve ser a aparência da implementação; elas devem ser neutras. Completude e neutralidade são importantes para interoperabilidade e portabilidade (Blair e Stefani, 1998).

Interoperabilidade caracteriza até que ponto duas implementações de sistemas ou componentes de fornecedores diferentes devem coexistir e trabalhar em conjunto, com base na mera confiança mútua nos serviços de cada um, especificados por um padrão comum.

Portabilidade caracteriza até que ponto uma aplicação desenvolvida para um sistema distribuído *A* pode ser executada, sem modificação, em um sistema distribuído diferente *B* que implementa as mesmas interfaces que *A*.

Uma outra meta importante para um sistema distribuído aberto é que deve ser fácil configurá-lo com base em componentes diferentes (possivelmente de desenvolvedores diferentes). Além disso, deve ser fácil adicionar novos componentes ou substituir os existentes sem afetar os que continuam no mesmo lugar. Em outras palavras, um sistema distribuído aberto deve ser também **extensível**. Por exemplo, em um sistema extensível, deve ser relativamente fácil adicionar partes que são executadas em um sistema operacional diferente, ou até mesmo substituir todo um sistema de arquivo. Como muitos de nós sabemos por experiência própria, é mais fácil falar do que conseguir tal flexibilidade.

Separação entre política e mecanismo

Para conseguir flexibilidade em sistemas distribuídos abertos, é crucial que o sistema seja organizado como um conjunto de componentes relativamente pequenos e de fácil substituição ou adaptação. Isso implica que devemos fornecer definições não somente para as interfaces de nível mais alto, isto é, as que são vistas por usuários e aplicações, mas também definições para interfaces com partes internas do sistema, além de descrever como essas partes interagem.

Essa abordagem é relativamente nova. Muitos sistemas mais antigos, ou mesmo alguns contemporâneos, são construídos segundo uma abordagem monolítica na qual a separação dos componentes é apenas lógica, embora eles sejam implementados como um único e imenso programa. Essa abordagem dificulta a substituição ou adaptação de um componente sem afetar todo o sistema. Portanto, sistemas monolíticos tendem a ser fechados em vez de abertos.

A necessidade de alterar um sistema distribuído é quase sempre causada por um componente que não fornece a política ideal para uma aplicação ou usuário específico. Por exemplo, considere a cache na World Wide Web. Em geral, os browsers permitem que os usuários adaptem sua política de cache especificando o tamanho da cache e se a consistência de um documento em cache deve ser verificada sempre ou se apenas uma vez por sessão. Contudo, o usuário não pode influenciar outros parâmetros de cache, como o período que um documento pode permanecer na cache, ou qual documento deve ser retirado quando a cache estiver cheia. Além disso, é impossível tomar decisões de cache com base no conteúdo de um documento. Um usuário pode querer manter em cache uma tabela de horários de trens porque sabe que esses horários dificilmente mudam, mas nunca as informações sobre as condições de tráfego correntes nas rodovias.

Precisamos de uma separação entre política e mecanismo. No caso de cache na Web, por exemplo, o ideal seria que um browser proporcionasse facilidades apenas para armazenar documentos e, ao mesmo tempo, permitisse aos usuários decidir quais documentos teriam de ser armazenados e por quanto tempo. Na prática, isso pode ser implementado pela oferta de um rico conjunto de parâmetros que o usuário pode estabelecer dinamicamente.

Ainda melhor é que um usuário possa implementar sua própria política sob a forma de um componente que possa ser conectado diretamente ao browser. Claro que esse componente deve ter uma interface que o browser possa entender, de modo que ele possa chamar procedimentos dessa interface.

1.2.4 Escalabilidade

A conectividade mundial pela Internet está rapidamente se tornando tão comum quanto poder enviar um cartão-postal para qualquer pessoa em qualquer lugar do

6 Sistemas distribuídos

mundo. Com isso em mente, a escalabilidade é uma das mais importantes metas de projeto para desenvolvedores de sistemas distribuídos.

A escalabilidade de um sistema pode ser medida segundo, no mínimo, três dimensões diferentes (Neuman, 1994). Em primeiro lugar, um sistema pode ser escalável em relação a seu tamanho, o que significa que é fácil adicionar mais usuários e recursos ao sistema. Em segundo lugar, um sistema escalável em termos geográficos é um sistema no qual usuários e recursos podem estar longe uns dos outros. Em terceiro lugar, um sistema pode ser escalável em termos administrativos, o que significa que ele ainda pode ser fácil de gerenciar, mesmo que abranja muitas organizações administrativas diferentes. Infelizmente, um sistema escalável em uma ou mais dessas dimensões muitas vezes apresenta perda na capacidade de desempenho à medida que é ampliado.

Problemas de escalabilidade

Quando é necessário ampliar um sistema, é preciso resolver problemas de tipos muito diferentes. Em primeiro lugar, vamos considerar a escalabilidade em relação ao tamanho. Se for preciso suportar mais usuários ou recursos, freqüentemente deparamos com as limitações de serviços centralizados, dados e algoritmos (ver Tabela 1.2). Por exemplo, muitos serviços são centralizados no sentido de que são implementados por meio de apenas um único servidor que executa em uma máquina específica no sistema distribuído. O problema com esse esquema é óbvio: o servidor pode se transformar em um gargalo à medida que o número de usuários e aplicações cresce. Ainda que tenhamos capacidades de processamento e armazenagem praticamente ilimitadas, a comunicação com aquele servidor acabará por impedir crescimento ulterior.

Infelizmente, às vezes é inevitável usar apenas um único servidor. Imagine que temos um serviço para gerenciar informações altamente confidenciais como históricos médicos, contas de bancos e assim por diante. Nesses casos, talvez seja melhor implementar o serviço por meio de um único servidor localizado em uma sala separada, de alta segurança, e protegido em relação às outras partes do sistema distribuído por meio de componentes de rede especiais. Copiar o servidor para diversas localizações para melhorar o desempenho pode estar fora de questão porque isso tornaria o serviço menos seguro.

Conceito	Exemplo
Serviços centralizados	Um único servidor para todos os usuários
Dados centralizados	Uma única lista telefônica on-line
Algoritmos centralizados	Fazer roteamento com base em informações completas

Tabela 1.2 Exemplos de limitações de escalabilidade.

Tão ruins quanto serviços centralizados são dados centralizados. Como não perder de vista os números de telefones e endereços de 50 milhões de pessoas? Suponha que cada registro de dados pudesse caber em 50 caracteres. Uma única partição de disco de 2,5 gigabytes proporcionaria armazenamento suficiente. Porém, mais uma vez, nesse caso, ter um único banco de dados sem dúvida saturaria todas as linhas de comunicação que o acessam. Do mesmo modo, imagine como a Internet funcionaria se seu Sistema de Nomes de Domínio (Domain Name System — DNS) ainda estivesse implementado como uma tabela única. O DNS mantém informações de milhões de computadores no mundo inteiro e forma um serviço essencial para localizar servidores Web. Se cada requisição para resolver um URL tivesse de ser passada para aquele único servidor DNS, é claro que ninguém estaria usando a Web — o que, por falar nisso, resloveria o problema.

Por fim, algoritmos centralizados também são má idéia. Em um sistema distribuído de grande porte, uma quantidade enorme de mensagens tem de ser roteada por muitas linhas. De um ponto de vista teórico, um bom modo de fazer isso é colher informações completas sobre a carga em todas as máquinas e linhas, e então executar um algoritmo para computar todas as rotas ótimas. Em seguida, essa informação pode ser propagada por todo o sistema para melhorar o roteamento.

O problema é que colher e transportar todas as informações de entrada e saída também seria má idéia porque essas mensagens sobrecarregariam parte da rede. Na verdade, qualquer algoritmo que colha informações de todos os sites, envie-as a uma única máquina para processamento e então distribua os resultados para todos os sites deve ser, de maneira geral, evitado. Somente algoritmos descentralizados devem ser utilizados. Em geral, esses algoritmos têm as seguintes características, que os distinguem dos algoritmos centralizados:

1. Nenhuma máquina tem informações completas sobre o estado do sistema.
2. As máquinas tomam decisões tendo como base somente informações locais.
3. A falha de uma máquina não arruina o algoritmo.
4. Não há nenhuma premissa implícita quanto à existência de um relógio global.

As três primeiras decorrem do que dissemos até agora. A última talvez seja menos óbvia, mas também é importante. Qualquer algoritmo que comece com “Precisamente às 12:00:00 todas as máquinas anotarão o tamanho de sua fila de saída” falhará porque é impossível conseguir a exata sincronização de todos os relógios, fato que os algoritmos devem levar em conta. Quanto maior o sistema, maior a incerteza. Talvez seja possível conseguir a sincronização de todos os relógios com tolerância de alguns microssegundos em uma única LAN, com considerável esforço, mas fazer isso em escala nacional ou internacional é complicado.

A escalabilidade geográfica tem seus próprios problemas. Uma das principais razões por que hoje é difícil ampliar sistemas distribuídos existentes que foram originalmente projetados para redes locais é que eles são baseados em **comunicação síncrona**. Nessa forma de comunicação, uma parte que requisita um serviço, em geral denominada cliente, fica bloqueada até que uma mensagem seja enviada de volta. Essa abordagem costuma funcionar bem em LANs nas quais a comunicação entre duas máquinas, na pior das hipóteses, demora, comumente, algumas centenas de microsegundos. Todavia, em um sistema de longa distância, precisamos levar em conta que a comunicação entre processos pode demorar centenas de milissegundos, isto é, ela é três ordens de grandeza mais lenta. Construir aplicações interativas usando comunicação síncrona em sistemas de longa distância requer muito cuidado, além de muita paciência.

Um outro problema que atrapalha a escalabilidade geográfica é que a comunicação em redes de longa distância é inherentemente não confiável e quase sempre ponto a ponto. Por comparação, redes locais em geral proporcionam facilidades de comunicação de alta confiança com base em broadcast, o que facilita muito o desenvolvimento de sistemas distribuídos. Considere o problema de localizar um serviço. Em um sistema de área local, um processo pode simplesmente enviar uma mensagem broadcast a todas as máquinas, perguntando a cada uma se está executando o serviço de que ele necessita. Sómente as máquinas que têm aquele serviço respondem, cada uma delas fornecendo seu endereço de rede na mensagem de resposta. Tal esquema de localização é inconcebível em um sistema de longa distância: imagine só o que aconteceria se tentássemos localizar um serviço desse modo na Internet. Em vez disso, é preciso projetar serviços especiais de localização, que talvez tenham alcance mundial e sejam capazes de atender a bilhões de usuários. Voltaremos a abordar esses serviços no Capítulo 5.

A escalabilidade geográfica está fortemente relacionada com problemas de soluções centralizadas que atrapalam a escalabilidade de tamanho. Se tivermos um sistema com muitos componentes centralizados, é claro que a escalabilidade geográfica estará limitada pelos problemas de desempenho e confiabilidade resultantes da comunicação a longa distância. Além disso, nessa circunstância os componentes centralizados resultarão em desperdício de recursos de rede. Imagine que um único servidor de correio seja usado por um país inteiro. Isso significaria que, quando você enviasse um e-mail a seu vizinho, primeiro o e-mail teria de ir até o servidor central de correio, que poderia estar a quilômetros de distância. Claro que esse não é o melhor caminho.

Por fim, uma questão difícil e, em muitos casos, ainda aberta é como ampliar um sistema distribuído por vários domínios administrativos independentes. Um problema importante que precisa ser resolvido é o de políticas con-

flitantes em relação à utilização — e pagamento — de recursos, gerenciamento e segurança.

Muitas vezes, por exemplo, os usuários de um único domínio podem confiar em componentes de um sistema distribuído que residam dentro desse mesmo domínio. Em tais casos, a administração do sistema deve ter testado e certificado aplicações e tomado providências especiais para garantir que os componentes não sofram nenhuma ação indevida. Em essência, os usuários confiam em seus administradores de sistema. Contudo, essa confiança não ultrapassa naturalmente as fronteiras do domínio.

Se um sistema distribuído se expandir para um outro domínio, é preciso tomar duas medidas de segurança. Antes, o sistema distribuído tem de se proteger contra ataques maliciosos do novo domínio: os usuários do novo domínio podem ter somente acesso de leitura ao sistema de arquivos no novo domínio original. Da mesma forma, facilidades como imagesetters caros ou computadores de alto desempenho podem não estar disponíveis para usuários estranhos. Em segundo lugar, o novo domínio tem de se proteger contra ataques maliciosos do sistema distribuído. Um exemplo típico é o download de programas como applets em browsers Web. Basicamente, o novo domínio não sabe que comportamento esperar de tal código estranho e, portanto, pode decidir impor limites severos aos direitos de acesso para esse código. O problema, como veremos no Capítulo 9, é como impor essas limitações.

Técnicas de escalabilidade

Após discutirmos alguns problemas de escalabilidade, surge a questão de como resolvê-los de maneira geral. Na maioria dos casos, problemas de escalabilidade em sistemas distribuídos aparecem como problemas de desempenho causados por capacidade limitada de servidores e rede. Agora, há basicamente apenas três técnicas para ampliar sistemas: ocultar latências de comunicação, distribuição e replicação [ver também Neuman (1994)].

Ocultar latências de comunicação é importante para conseguir escalabilidade geográfica. A idéia básica é simples: tentar evitar, o quanto possível, esperar por respostas a requisições remotas — e potencialmente distantes — de serviços. Vamos supor que um serviço seja requisitado em uma máquina remota. Uma alternativa a esperar por uma resposta do servidor é executar outro trabalho útil no lado do requisitante. Em essência, isso significa construir a aplicação requisitante de modo tal que ela use só **comunicação assíncrona**. Quando chega uma resposta, a aplicação é interrompida e um manipulador especial é chamado para concluir a requisição emitida anteriormente.

A comunicação assíncrona muitas vezes pode ser usada em sistemas de processamento de lotes e em aplicações paralelas, nos quais tarefas mais ou menos independentes podem ser escalonadas para execução enquanto uma outra tarefa está esperando pela conclusão da comunicação. Como alternativa, pode-se iniciar um novo thread de controle para exe-

cutar a requisição. Embora ele fique bloqueado à espera da resposta, outros threads no processo podem continuar.

Contudo, há muitas aplicações que não podem fazer uso efetivo da comunicação assíncrona. Por exemplo, em aplicações interativas, quando um usuário envia uma requisição, em geral ele não terá nada melhor a fazer do que esperar pela resposta. Nesses casos, uma solução muito melhor é reduzir a comunicação global, passando parte da computação que normalmente é executada no servidor para o processo do cliente que está requisitando o serviço.

Um caso típico em que essa abordagem funciona é o acesso a bancos de dados por meio de formulários. O preenchimento de formulários pode ser feito com o envio de uma mensagem separada para cada campo e a espera por um reconhecimento do servidor, como mostra a Figura 1.2(a). O servidor pode verificar se há erros de sintaxe antes de aceitar uma entrada.

Uma solução muito melhor é despachar para o cliente o código para preencher o formulário e possivelmente verificar as entradas, além de fazer com que o cliente

devolva um formulário completo, como mostra a Figura 1.2(b). Essa abordagem de despacho de código atualmente é amplamente suportada pela Web sob a forma de applets Java e Javascript.

Uma outra técnica importante de ampliação é a **distribuição**. A distribuição envolve tomar um componente, subdividi-lo em partes menores e, na seqüência, espalhar essas partes pelo sistema. Um excelente exemplo de distribuição é o Sistema de Nomes de Domínio da Internet. O espaço de nomes do DNS é organizado por hierarquia em uma árvore de **domínios**, dividida em **zonas** sem sobreposição, como mostra a Figura 1.3. Os nomes em cada zona são manipulados por um único servidor de nomes. Sem entrar em muitos detalhes, podemos imaginar cada nome de caminho como o nome de um hospedeiro na Internet e, por isso, associado a um endereço de rede daquele hospedeiro.

Basicamente, resolver um nome significa retornar o endereço de rede do hospedeiro associado. Considere, por exemplo, o nome *nl.vu.cs.flits*. Para resolver esse nome, primeiro ele é passado ao servidor de zona *Z1* (ver Figura 1.3),

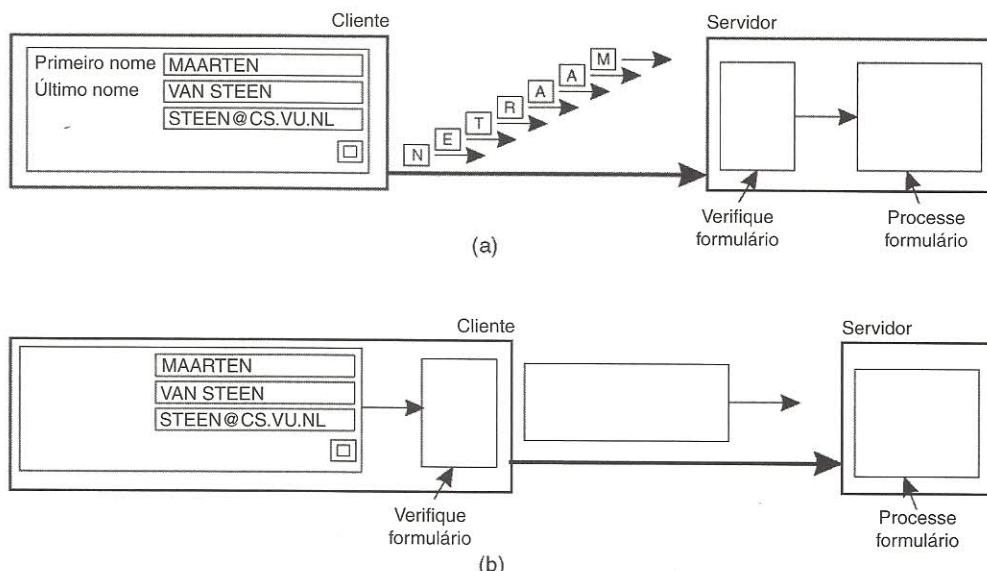


Figura 1.2 A diferença entre deixar (a) um servidor ou (b) um cliente verificar formulários à medida que são preenchidos.

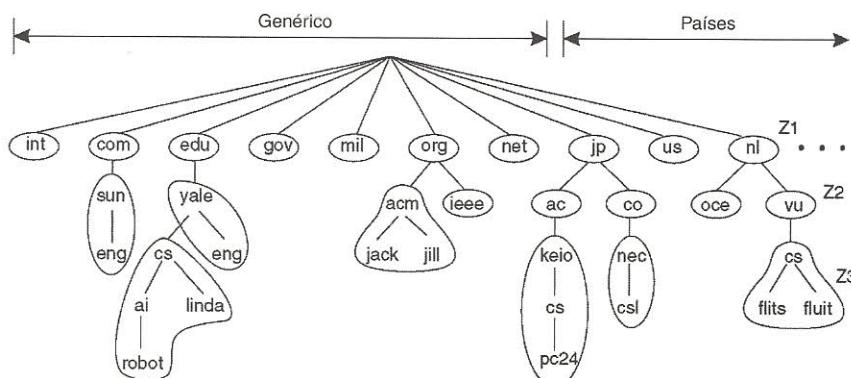


Figura 1.3 Exemplo de divisão do espaço de nomes do DNS em zonas.

que retorna o endereço do servidor para a zona Z2, para a qual o resto do nome, *vu.cs.flits*, pode ser entregue. O servidor para Z2 retornará o endereço do servidor para a zona Z3, que é capaz de manipular a última parte do nome e retornará o endereço do hospedeiro associado.

Esse exemplo ilustra como o *serviço de nomeação* fornecido pelo DNS é distribuído por várias máquinas, evitando, desse modo, que um único servidor tenha de lidar com todas as requisições de resolução de nomes.

Como outro exemplo, considere a World Wide Web. Para a maioria dos usuários, a Web parece ser um enorme sistema de informações baseado em documentos, no qual cada documento tem seu próprio nome exclusivo sob a forma de um URL. Em termos de conceito, pode até parecer que há apenas um único servidor. Entretanto, a Web é fisicamente distribuída por um grande número de servidores, e cada um deles manuseia certa quantidade de documentos da Web. O nome do servidor que manuseia um documento está codificado no URL do documento. Somente graças a essa distribuição de documentos é que foi possível aumentar a Web até seu tamanho atual.

Considerando que problemas de escalabilidade frequentemente aparecem sob a forma de degradação do desempenho, em geral é uma boa idéia **replicar** componentes por um sistema distribuído. A replicação não somente aumenta a disponibilidade, mas também ajuda a equilibrar a carga entre componentes, o que resulta em melhor desempenho. Além disso, em sistemas de ampla dispersão geográfica, ter uma cópia por perto pode ocultar grande parte dos problemas de latência de comunicação já mencionados.

Cache é uma forma especial de replicação, embora muitas vezes a distinção entre as duas seja difícil de compreender ou até mesmo artificial. Como no caso da replicação, a cache resulta em fazer uma cópia de um recurso, em geral na proximidade do acesso do cliente àquele recurso. Entretanto, ao contrário da replicação, a cache é uma decisão tomada pelo cliente de um recurso, e não por seu proprietário. Além disso, a cache acontece sob demanda, ao passo que a replicação costuma ser planejada antecipadamente.

Tanto a cache quanto a replicação têm uma séria desvantagem que pode causar efeitos adversos na escalabilidade. Como nessa circunstância temos várias cópias de um recurso, se uma delas for modificada, ficará diferente das outras. Por consequência, cache e replicação resultam em problemas de **consistência**.

Até que ponto as inconsistências podem ser toleradas depende em grande parte da utilização de um recurso. Muitos usuários da Web acham aceitável que seus browsers retornem um documento em cache cuja validade não tenha sido verificada nos últimos minutos passados. Contudo, também há muitos casos em que é preciso cumprir fortes garantias de consistência, tal como no caso de bolsas de valores e leilões eletrônicos. O problema com a forte consistência é que uma atualização deve ser imediatamente propagada para

todas as outras cópias. Além do mais, se duas atualizações ocorrerem ao mesmo tempo, freqüentemente também é exigida a atualização de cada cópia na mesma ordem.

Situações como essa em geral requerem algum mecanismo de sincronização global. Infelizmente, é extremamente difícil ou até impossível implementar esses mecanismos de modo escalável porque as leis da física insistem em que os fótons e os sinais elétricos obedeçam a um limite de velocidade de 3×10^8 m/s (a velocidade da luz). Por consequência, ampliar um sistema por replicação pode introduzir outras soluções inherentemente não escaláveis. Voltaremos à replicação e à consistência no Capítulo 7.

Ao considerarmos essas técnicas de ampliação de escala de um sistema, poderíamos argumentar que a escalabilidade de tamanho é a menos problemática do ponto de vista técnico. Em muitos casos, o simples aumento da capacidade de uma máquina resolverá a questão, ao menos temporariamente, e talvez a custos significativos. A escalabilidade geográfica é um problema muito mais difícil, porque é a Mãe Natureza que está atrapalhando. Ainda assim, a prática mostra que combinar técnicas de distribuição, replicação e cache com diferentes formas de consistência costuma ser suficiente em muitos casos.

Por fim, escalabilidade administrativa parece ser a mais difícil, em parte porque também precisamos resolver problemas que não são técnicos (como políticas de organizações e colaboração humana). Não obstante, houve progresso nessa área simplesmente *ignorando* domínios administrativos. A introdução, e agora a utilização, disseminada de tecnologia peer-to-peer demonstra o que pode ser conseguido se os usuários finais simplesmente tomarem o controle (Aberer e Hauswirth, 2005; Lua et al., 2005; Oram, 2001). Contudo, vamos deixar claro que, na melhor das hipóteses, a tecnologia peer-to-peer pode ser apenas uma solução parcial para a escalabilidade administrativa. Mas, em algum momento, teremos de tratar dela.

1.2.5 Cidades

Agora já deve estar claro que desenvolver sistemas distribuídos pode ser uma tarefa dificílima. Como veremos muitas vezes em todo este livro, há tantas questões a considerar ao mesmo tempo que parece poder ser apenas a complexidade o único resultado. Mesmo assim, seguindo alguns princípios de projeto, podem-se desenvolver sistemas distribuídos que cumpram à risca as metas que estabelecemos neste capítulo. Muitos princípios seguem as regras básicas da engenharia decente de software e não serão repetidos aqui.

Contudo, sistemas distribuídos são diferentes do software tradicional porque os componentes estão dispersos por uma rede. Não levar essa dispersão em conta durante o projeto é o que torna tantos sistemas desnecessariamente complexos e resulta em erros que precisam ser consertados mais tarde. Peter Deutsch, que, na época, trabalhava na Sun Microsystems, formulou esses erros como

as seguintes premissas falsas que todos adotam ao desenvolver uma aplicação distribuída pela primeira vez:

1. A rede é confiável.
2. A rede é segura.
3. A rede é homogênea.
4. A topologia não muda.
5. A latência é zero.
6. A largura de banda é infinita.
7. O custo de transporte é zero.
8. Há só um administrador.

Observe como essas premissas se referem a propriedades exclusivas de sistemas distribuídos: confiabilidade, segurança, heterogeneidade e topologia da rede; latência e largura de banda; custos de transporte e, por fim, domínios administrativos. No desenvolvimento de aplicações não distribuídas, é provável que a maioria dessas questões nem apareça.

A maior parte dos princípios que discutimos neste livro está imediatamente relacionada a essas premissas. Em todos os casos, discutiremos soluções para problemas causados pelo fato de uma ou mais premissas serem falsas. Por exemplo, redes confiáveis simplesmente não existem, o que leva à impossibilidade de conseguir transparência à falha. Dedicaremos um capítulo inteiro para tratar do fato de que a comunicação por rede é inherentemente insegura. Já discutimos que sistemas distribuídos precisam levar em conta a heterogeneidade. Na mesma toada, quando discutimos replicação para resolver problemas de escalabilidade, estamos, em essência, atacando problemas de latência e largura de banda. Também abordaremos questões de gerenciamento em vários pontos desta obra, tratando das falsas premissas do transporte a custo zero e de um único domínio administrativo.

1.3 Tipos de Sistemas Distribuídos

Antes de começarmos a discutir os princípios de sistemas distribuídos, vamos examinar com maior atenção os vários tipos de sistemas distribuídos. A seguir faremos

a distinção entre sistemas de computação distribuídos, sistemas de informação distribuídos e sistemas embutidos distribuídos.

1.3.1 Sistemas de computação distribuídos

Uma classe importante de sistemas distribuídos é a utilizada para tarefas de computação de alto desempenho. Em termos estritos, podemos fazer uma distinção entre dois subgrupos. Na **computação de cluster**, o hardware subjacente consiste em um conjunto de estações de trabalho ou PCs semelhantes, conectados por meio de uma rede local de alta velocidade. Além disso, cada nó executa o mesmo sistema operacional.

A situação fica bem diferente no caso da **computação em grade**. Esse subgrupo consiste em sistemas distribuídos que costumam ser montados como federação de computadores, na qual cada sistema pode cair sob um domínio administrativo diferente, e pode ser muito diferente no que tange a hardware, software e tecnologia de rede empregada.

Sistemas de computação de cluster

Sistemas de computação de cluster tornaram-se populares quando a razão preço/desempenho de computadores pessoais e estações de trabalho melhorou. A certa altura ficou atraente, em termos financeiros e técnicos, construir um supercomputador que usasse tecnologia de prateleira simplesmente conectando uma série de computadores relativamente simples a uma rede de alta velocidade. Em quase todos os casos, a computação de cluster é usada para programação paralela na qual um único programa, intensivo em computação, é executado em paralelo em várias máquinas.

Um exemplo bem conhecido de um computador de cluster é formado pelos clusters Beowulf baseados em Linux, cuja configuração geral é mostrada na Figura 1.4. Cada cluster consiste em um conjunto de nós de computação controlados e acessados por meio de um único nó mestre. As tarefas típicas do mestre são manipular a alocação de nós a um determinado programa paralelo, manter uma fila de jobs apresentados e proporcionar uma

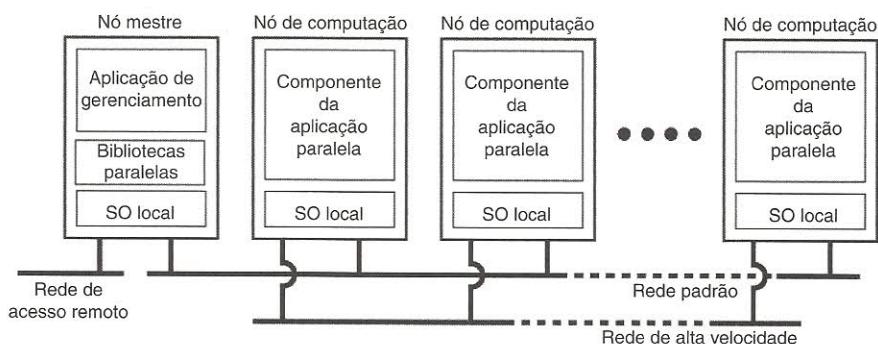


Figura 1.4 Exemplo de um sistema de computação de cluster.

interface para os usuários do sistema. Assim, na verdade, o mestre executa o middleware necessário para a execução de programas e o gerenciamento do cluster, ao passo que, para os nós de computação, muitas vezes basta um sistema operacional padrão e nada mais.

Uma parte importante desse middleware é formada pelas bibliotecas para execução de programas paralelos. Como discutiremos no Capítulo 4, na realidade, muitas dessas bibliotecas fornecem somente facilidades avançadas de comunicação por mensagem, mas não são capazes de manter segurança, processos com falhas e assim por diante.

Como alternativa para essa organização hierárquica, o sistema Mosix adota uma abordagem simétrica (Amar et al., 2004). Esse sistema tenta prover uma **imagem de sistema único** de um cluster, o que significa que, para um processo, um computador de cluster oferece a transparência de distribuição definitiva porque parece ser um único computador. Como já mencionamos, é impossível proporcionar tal imagem sob todas as circunstâncias. No caso do Mosix, o alto grau de transparência é conseguido com a permissão de uma forma dinâmica e preventiva de migração de processos entre os nós que compõem o cluster.

A migração de processos permite que um usuário inicie uma aplicação em qualquer nó (denominado nó nativo), após o que ele pode se mover transparentemente para outros nós a fim de, por exemplo, fazer uso eficiente de recursos. Voltaremos à migração de processos no Capítulo 3.

Sistemas de computação em grade

Um aspecto característico da computação de cluster é sua homogeneidade. Na maioria dos casos, os computadores que compõem um cluster são, em grande parte, os mesmos, todos têm o mesmo sistema operacional e todos estão conectados à mesma rede. Por comparação, sistemas de computação em grade têm alto grau de heterogeneidade: nenhuma premissa é adotada em relação a hardware, sistemas operacionais, redes, domínios administrativos, políticas de segurança e assim por diante.

Uma questão fundamental em um sistema de computação em grade é que recursos de diferentes organizações são reunidos para permitir a colaboração de um grupo de pessoas ou instituições. Tal colaboração é realizada sob a forma de uma **organização virtual**. As pessoas que per-

tencem à mesma organização virtual têm direitos de acesso aos recursos fornecidos por aquela organização. Entre os recursos típicos estão servidores de computação (entre eles supercomputadores, possivelmente implementados como computadores de cluster), facilidades de armazenamento e bancos de dados. Além disso, também podem ser oferecidos equipamentos especiais em rede, como telescópios, sensores e outros.

Dada a sua natureza, grande parte do software para realizar computação em grade é desenvolvida com a finalidade de prover acesso a recursos de diferentes domínios administrativos, e somente para usuários e aplicações que pertençam a uma organização virtual específica. Por essa razão, o foco costuma ser dirigido às questões de arquitetura. Uma arquitetura proposta por Foster et al. (2001) é mostrada na Figura 1.5.

A arquitetura consiste em quatro camadas. A mais baixa, denominada *camada-base*, provê interfaces para recursos locais em um site específico. Observe que essas interfaces são projetadas para permitir compartilhamento de recursos dentro de uma organização virtual. A tarefa típica dessas interfaces é prover funções para consultar o estado e as capacidades de um recurso, em conjunto com funções para o gerenciamento de recursos propriamente dito. Por exemplo: travar recursos.

A *camada de conectividade* consiste em protocolos de comunicação para suportar transações da grade que abrangam a utilização de múltiplos recursos. Por exemplo, são necessários protocolos para transferir dados entre recursos, ou para o simples acesso de um recurso desde uma localização remota. Além disso, a camada de conectividade conterá protocolos de segurança para autenticar usuários e recursos. Observe que, em muitos casos, usuários humanos não são autenticados; em vez disso, são autenticados programas que agem em nome dos usuários. Nesse sentido, delegar direitos de um usuário a programas é uma função importante que precisa ser suportada na camada de conectividade. Daremos mais detalhes sobre a delegação quando discutirmos segurança em sistemas distribuídos.

A *camada de recursos* é responsável pelo gerenciamento de um único recurso. Ela utiliza as funções fornecidas pela camada de conectividade e chama diretamente as interfaces disponibilizadas pela camada-base. Por exemplo, essa cama-

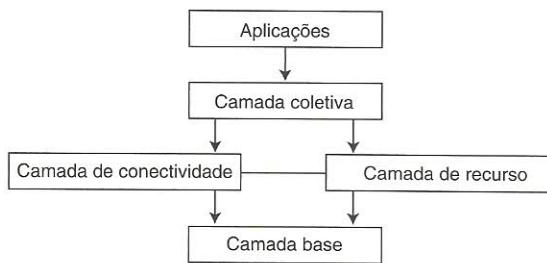


Figura 1.5 Arquitetura em camadas para sistemas de computação em grade.

da oferecerá funções para obter informações de configuração sobre um recurso específico ou, em geral, para realizar operações específicas como criar um processo ou ler dados. Portanto, a camada de recursos é considerada responsável pelo controle de acesso e, por isso, dependerá da autenticação realizada como parte da camada de conectividade.

A camada seguinte na hierarquia é a *camada coletiva*. Ela trata de manipular o acesso a múltiplos recursos e normalmente consiste em serviços para descoberta de recursos, alocação e escalonamento de tarefas para múltiplos recursos, replicação de dados e assim por diante. Diferentemente das camadas de conectividade e de recursos, que consistem em um conjunto padronizado e relativamente pequeno de protocolos, a camada coletiva pode consistir em muitos protocolos diferentes para muitas finalidades diferentes, que refletem o amplo espectro de serviços que ela pode oferecer a uma organização virtual.

Por fim, a *camada de aplicação* consiste em aplicações que funcionam dentro de uma organização virtual e que fazem uso do ambiente de computação em grade.

Normalmente, as camadas coletiva, de conectividade e de recursos formam o cerne daquilo que poderia ser denominado camada de middleware em grade. Em conjunto, essas camadas dão acesso e gerenciam recursos que estão potencialmente dispersos por vários sites. Uma observação importante da perspectiva do middleware é que, com a computação em grade, a noção de um site (ou unidade administrativa) é comum. Essa prevalência é enfatizada pela tendência gradual à migração para uma **arquitetura orientada a serviços** na qual sites ofereçam acesso às várias camadas por meio de um conjunto de serviços Web (Joseph et al., 2004).

A essa altura, isso nos levou à definição de uma arquitetura alternativa conhecida como **arquitetura de serviços de grade aberta** (*Open Grid Services Architecture — OGSA*). Essa arquitetura consiste em várias camadas e muitos componentes, o que a torna bastante complexa. A complexidade parece ser o destino de qualquer processo de padronização. Detalhes sobre a OGSA podem ser encontrados em Foster et al. (2005).

1.3.2 Sistemas de informação distribuídos

Uma outra classe importante de sistemas distribuídos é encontrada em organizações que se defrontaram com uma profusão de aplicações em rede para as quais a interoperabilidade se mostrou uma experiência dolorosa. Muitas das soluções de middleware existentes são resultado do trabalho com uma infra-estrutura na qual era mais fácil integrar aplicações a um sistema de informações de âmbito empresarial (Bernstein, 1996; Alonso et al., 2004).

Podemos distinguir vários níveis nos quais ocorreu a integração. Em muitos casos, uma aplicação em rede consistia simplesmente em um servidor que executava aquela aplicação, freqüentemente incluindo um banco de dados, e a disponibilizava para programas remotos, denominados

clientes. Esses clientes podiam enviar uma requisição ao servidor para executar uma operação específica e depois receber uma resposta que era devolvida. Integração no nível mais baixo permitiria que clientes empacotassem várias requisições, possivelmente para diferentes servidores, em uma única requisição maior, e as enviassem para execução como uma **transação distribuída**. A idéia fundamental era que todas as — ou nenhuma das — requisições seriam executadas.

À medida que as aplicações se tornavam mais sofisticadas e eram gradualmente separadas em componentes independentes, notavelmente distinguindo componentes de banco de dados de componentes de processamento, ficou claro que a integração também deveria ocorrer de modo que permitisse às aplicações se comunicar diretamente umas com as outras. Isso resultou, atualmente, em uma enorme indústria dedicada à **integração de aplicações empresariais** (*Enterprise Application Integration — EAI*). A seguir, abordaremos essas duas formas de sistemas distribuídos.

Sistemas de processamento de transações

Para deixar nossa discussão mais clara, vamos nos concentrar em aplicações de banco de dados. Na prática, operações em um banco de dados costumam ser realizadas sob a forma de **transações**. Programar a utilização de transações requer primitivas especiais que devem ser fornecidas pelo sistema distribuído subjacente ou pelo sistema de linguagem em tempo de execução. Exemplos típicos de primitivas de transação são mostrados na Tabela 1.3.

A lista exata de primitivas depende dos tipos de objetos que estão sendo usados na transação (Gray e Reuter, 1993). Em um sistema de correio, poderia haver primitivas para enviar, receber e repassar correio. Em um sistema de contabilidade, elas poderiam ser bastante diferentes. Entretanto, **READ** e **WRITE** são exemplos típicos. Declarações ordinárias, chamadas de procedimento e assim por diante, também são permitidas dentro de uma transação. Em particular, mencionamos que chamadas a procedimentos remotos (*Remote Procedure Calls — RPCs*), isto é, chamadas de procedimento em servidores remotos, também costumam ser encapsuladas em uma transação, o que resulta no que é conhecido como **RPC transacional**. Discutiremos RPCs minuciosamente no Capítulo 4.

Primitiva	Descrição
BEGIN_TRANSACTION	Marque o início de uma transação
END_TRANSACTION	Termine a transação e tente comprometê-la
ABORT_TRANSACTION	Elimine a transação e restaure os valores antigos
READ	Leia dados de um arquivo, tabela ou de outra forma
WRITE	Escreva dados para um arquivo, tabela ou de outra forma

Tabela 1.3 Exemplos de primitivas para transações.

`BEGIN_TRANSACTION` e `END_TRANSACTION` são usadas para delimitar o escopo de uma transação. As operações entre elas formam o corpo da transação. O aspecto característico de uma transação é que todas essas operações são executadas ou nenhuma é executada. Elas podem ser procedimentos de biblioteca, chamadas de sistema ou declarações entre parênteses em uma linguagem, dependendo da implementação.

Essa propriedade tudo-ou-nada das transações é uma das quatro propriedades características que elas têm. Mais especificamente, transações são:

1. Atômicas: para o mundo exterior, a transação acontece como se fosse indivisível.
2. Consistentes: a transação não viola invariantes de sistema.
3. Isoladas: transações concorrentes não interferem umas com as outras.
4. Duráveis: uma vez comprometida uma transação, as alterações são permanentes.

Essas propriedades costumam ser citadas por suas letras iniciais: **ACID**.

A primeira propriedade fundamental exibida por todas as transações é que elas são **atômicas**. Essa propriedade garante que cada transação aconteça completamente, ou não aconteça; e, se acontecer, será como uma única ação indivisível e instantânea. Enquanto uma transação está em progresso, outros processos, estejam ou não envolvidos em transações, não podem ver nenhum dos estados intermediários.

A segunda propriedade afirma que elas são **consistentes**. Isso quer dizer que, se o sistema tiver certos invariantes que devem valer sempre, se eles forem válidos antes da transação, também o serão após a transação. Por exemplo, em um sistema bancário, um variante fundamental é a lei da conservação do dinheiro. Após toda transferência interna, a quantidade de dinheiro no banco tem de ser a mesma que era antes da transferência; contudo, por um breve instante durante a transação, esse variante pode ser violado. Todavia, a violação não é visível fora da transação.

A terceira propriedade diz que as transações são **isoladas** ou **serializáveis**. Isso significa que, se duas ou mais transações são executadas ao mesmo tempo, o resultado final para cada uma delas e para outros processos se apresentará como se todas as transações fossem executadas em sequência em certa ordem, dependente do sistema.

A quarta propriedade diz que as transações são **duráveis**. Refere-se ao fato de que, não importa o que aconteça, uma vez comprometida uma transação, ela continua, e os resultados tornam-se permanentes. Nenhuma falha após o comprometimento pode desfazer os resultados ou provocar sua perda.

A durabilidade será discutida minuciosamente no Capítulo 8.

Até aqui, transações foram definidas em um único banco de dados. Uma **transação aninhada** é construída com base em uma quantidade de subtransações, como mostra a Figura 1.6. A transação do nível mais alto pode se ramificar e gerar ‘filhos’ que executam em paralelo uns aos outros em máquinas diferentes para obter ganho de desempenho ou simplificar a programação. Cada um desses filhos também pode executar uma ou mais subtransações ou se ramificar e gerar seus próprios filhos.

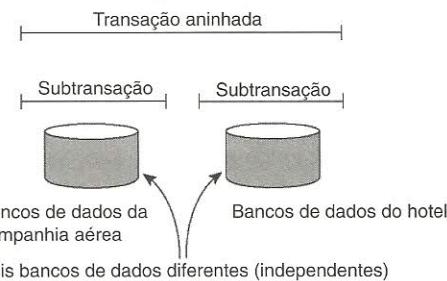


Figura 1.6 Transação aninhada.

Subtransações dão origem a um problema sutil, porém importante. Imagine que uma transação inicia várias subtransações em paralelo e uma delas se compromete, tornando seus resultados visíveis à transação-pai. Após mais computação, a transação-pai é abortada, restaurando todo o sistema ao estado em que estava antes que a transação do nível mais alto começasse. Por consequência, os resultados da subtransação que se comprometeu devem ser anulados. Portanto, a permanência que citamos anteriormente se aplica somente às transações do nível mais alto.

Visto que transações podem ser aninhadas até uma profundidade arbitrária, é preciso considerável administração para conseguir que tudo esteja correto. No entanto, a semântica é clara. Quando qualquer transação ou subtransação começa, ela recebe, em termos conceituais, uma cópia privada de todos os dados presentes no sistema inteiro, a qual pode manipular como desejar. Se ela abortar, seu universo privado desaparece como se nunca tivesse existido. Se ela se comprometer, seu universo privado substitui o universo do pai. Assim, se uma subtransação estiver comprometida e mais tarde for iniciada uma nova subtransação, a segunda vê os resultados produzidos pela primeira. Da mesma forma, se uma transação do nível mais alto abortar, todas as suas subtransações subjacentes também têm de ser abortadas.

Transações aninhadas são importantes em sistemas distribuídos porque proporcionam um modo natural de distribuir uma transação por várias máquinas. Elas seguem uma divisão *lógica* do trabalho da transação original. Por exemplo, uma transação para o planejamento de uma viagem pela qual é preciso reservar três vôos pode ser subdividida logicamente em até três subtransações. Cada uma dessas subtransações pode ser gerenciada em separado e independentemente das outras duas.

Quando os sistemas de middleware empresarial começaram, o componente que manipulava transações distribuídas, ou aninhadas, formava o núcleo para a integração de aplicações no nível do servidor ou do banco de dados. Esse componente era denominado **monitor de processamento de transação**, ou, de forma abreviada, **monitor TP**. Sua principal tarefa era permitir que uma aplicação acessasse vários servidores/bancos de dados oferecendo a ela um modelo de programação transacional, como mostra a Figura 1.7.

Integração de aplicações empresariais

Como já mencionamos, quanto mais as aplicações se desvinculavam dos bancos de dados sobre os quais eram construídas, mais evidente ficava que eram necessárias facilidades para integrar aplicações independentemente de seus bancos de dados. Em particular, componentes de aplicação deveriam poder se comunicar diretamente uns com os outros, e não apenas por meio do comportamento de requisição/resposta que era suportado por sistemas de processamento de transações.

Essa necessidade de comunicação entre aplicações resultou em muitos modelos diferentes de comunicação que serão discutidos minuciosamente neste livro — por essa razão, por enquanto faremos aqui apenas uma breve descrição. A principal idéia era que aplicações existentes pudessem trocar informações diretamente, como mostra a Figura 1.8.

Existem vários tipos de middleware de comunicação. Com **chamadas de procedimento remoto (Remote Procedure Calls — RPC)**, um componente de aplicação pode efetivamente enviar uma requisição a um outro componente de aplicação executando uma chamada de procedimento local, que resulta no empacotamento da requisição como uma mensagem e em seu envio ao chamador. Da mesma forma, o resultado será enviado de volta e devolvido à aplicação como o resultado da chamada de procedimento.

À medida que a popularidade da tecnologia de objeto aumentava, foram desenvolvidas técnicas que permitiam chamadas a objetos remotos, o que resultou naquilo que denominamos **invocações de método remoto (Remote Method Invocations — RMI)**. Uma RMI é, em essência, o mesmo que uma RPC, exceto que funciona com objetos em vez de com aplicações.

A desvantagem da RPC e da RMI é que ambos, o chamador e o chamado, precisam estar ligados e em funcionamento no momento da comunicação. Além disso, eles precisam saber exatamente como se referir um ao outro. Esse forte acoplamento muitas vezes é percebido como uma séria desvantagem e resultou no que conhecemos como **middleware orientado a mensagem** ou, simplesmente, **MOM (Message-oriented Middleware)**. Nesse caso, as aplicações apenas enviam mensagens a pontos lógicos de contato, que freqüentemente são descritos por meio de um sujeito. Da mesma forma, as aplica-

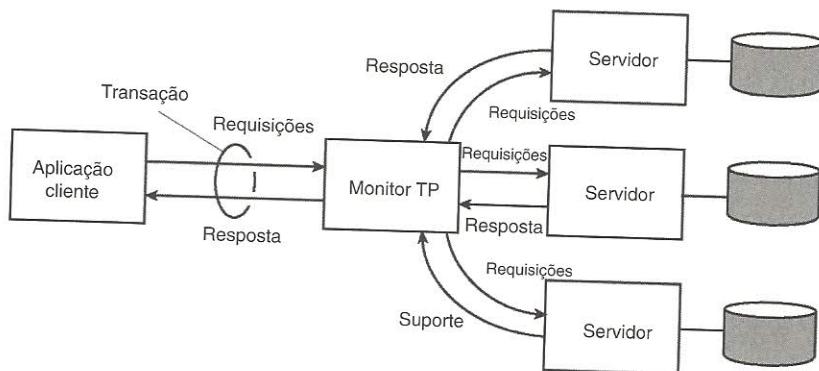


Figura 1.7 O papel do monitor TP em sistemas distribuídos.

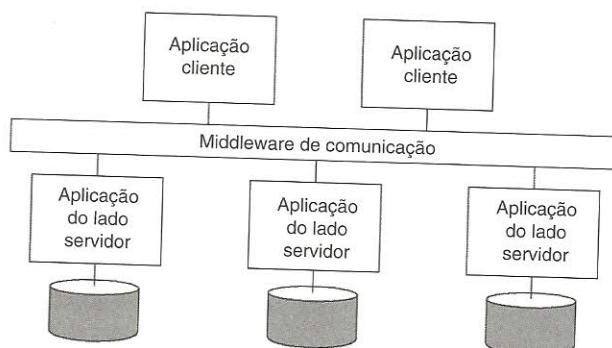


Figura 1.8 Middleware como facilitador de comunicação em integração de aplicações empresariais.

ções podem indicar seu interesse por um tipo específico de mensagem, após o que o middleware de comunicação cuidará para que todas as mensagens sejam entregues a essas aplicações. Esses sistemas, denominados **publicar/subscrever**, formam uma classe importante e em expansão de sistemas distribuídos. Nós os discutiremos minuciosamente no Capítulo 13.

1.3.3 Sistemas distribuídos pervasivos

Os sistemas distribuídos que discutimos até aqui são, em grande parte, caracterizados por sua estabilidade: os nós são fixos e têm uma conexão mais ou menos permanente e de alta qualidade com uma rede. Até certo ponto, essa estabilidade tem sido conseguida por meio de várias técnicas que são discutidas neste livro e que visam a obter transparência de distribuição. Um exemplo: a profusão de técnicas para mascarar falhas e recuperação dará a impressão de que as coisas podem dar errado apenas raramente. Da mesma maneira, conseguimos ocultar aspectos relacionados com a real localização de um nó na rede, o que permite, efetivamente, que usuários e aplicações acreditam que os nós continuam onde estão.

Contudo, a questão ficou muito diferente com a introdução de dispositivos de computação móveis e embutidos. Atualmente encontramos sistemas distribuídos nos quais a instabilidade é o comportamento esperado. Nesses sistemas, que denominamos **sistemas distribuídos pervasivos**, os equipamentos costumam ser caracterizados por seu pequeno tamanho, pela alimentação por bateria, por sua mobilidade e por terem somente uma conexão sem fio, se bem que nem todas essas características se aplicam a todos os dispositivos. Além do mais, tais características não precisam ser necessariamente interpretadas como restritivas, como é ilustrado pelas possibilidades dos modernos smart phones (Roussos et al., 2005).

Como seu nome sugere, um sistema distribuído perversivo é parte de nosso entorno; por isso, é, em geral, inerentemente distribuído. Um aspecto importante é a ausência geral de controle administrativo humano. Na melhor das hipóteses, os dispositivos podem ser configurados por seus proprietários; porém, quanto ao mais, eles precisam descobrir automaticamente seu ambiente e ‘se encaixar’ o melhor que puderem. Grimm et al. (2004) tornaram esse ‘encaixar’ mais exato pela formulação dos três requisitos para aplicações perversivas apresentados a seguir:

1. Adotar mudanças contextuais.
2. Incentivar composição ad hoc.
3. Reconhecer compartilhamento como padrão.

Adotar mudanças contextuais significa que um dispositivo deve estar continuamente ciente do fato de que seu ambiente pode mudar o tempo todo. Uma das mudanças mais simples é descobrir que uma rede não está mais disponível porque um usuário está se movimentando entre estações-bases. Nesse caso, a aplicação deve reagir,

possivelmente conectando-se a uma outra rede, ou tomando outras providências adequadas.

Incentivar composição ad hoc refere-se ao fato de que muitos dispositivos em sistemas pervasivos serão utilizados de modos muito diferentes por usuários diferentes. O resultado é que a configuração do conjunto de aplicações que executa em um dispositivo, seja pelo usuário, seja por interposição automatizada, porém controlada, tem de ser fácil.

Um aspecto muito importante de sistemas pervasivos é que, em geral, os dispositivos se juntam ao sistema para acessar — e possivelmente fornecer — informações. Isso requer meios para ler, armazenar, gerenciar e compartilhar informação com facilidade. À luz da conectividade intermitente e em constante mutação dos dispositivos, é muito provável que o espaço no qual residem informações acessíveis mudará o tempo todo.

Mascolo et al. (2004) bem como Niemela e Latvakoski (2004) chegaram a conclusões semelhantes: na presença de mobilidade, dispositivos devem suportar a adaptação fácil e dependente de aplicação a seu ambiente local. Também devem ser capazes de descobrir serviços com eficiência e reagir de acordo. Considerando esses requisitos, a esta altura já ficou claro que, na realidade, não existe transparência de distribuição em sistemas pervasivos. De fato, a distribuição de dados, processos e controle é *inerente* a esses sistemas, razão por que talvez seja melhor tão-somente expor a distribuição, em vez de ocultá-la. Vamos estudar, agora, alguns exemplos concretos de sistemas pervasivos.

Sistemas domésticos

Um tipo cada vez mais popular de sistema perversivo, mas que talvez seja o menos restrito, são sistemas montados ao redor de redes domésticas. Em geral, esses sistemas são compostos de um ou mais computadores pessoais. Porém, o mais importante é que integram eletrônicos de consumo típicos como aparelhos de TV, equipamentos de áudio e vídeo, dispositivos para jogos, smart phones, PDAs e outros equipamentos de uso pessoal em um único sistema. Além disso, podemos esperar que todos os tipos de dispositivos, como eletrodomésticos de cozinha, câmeras de vigilância, relógios, controladores de iluminação e assim por diante, serão conectados a um único sistema distribuído.

Da perspectiva de sistema, há vários desafios que precisam ser enfrentados antes que os sistemas pervasivos domésticos se tornem realidade. Um desafio importante é que tal sistema deve ser completamente autoconfigurável e autogerenciável. Não se pode esperar que usuários finais estejam dispostos ou sejam capazes de manter um sistema distribuído doméstico ligado e em funcionamento se seus componentes forem propensos a erros, como acontece com muitos dos dispositivos existentes hoje.

Muito já foi conseguido por meio dos padrões **Universal Plug and Play (UPnP)**, pelos quais dispositivos obtêm automaticamente endereços IP, podem descobrir uns

aos outros e assim por diante (UPnP Forum, 2003). Contudo, é preciso mais. Por exemplo, não está claro como o software e o firmware presentes em dispositivos podem ser atualizados com facilidade sem intervenção manual, ou quando ocorrem as atualizações, de modo que a compatibilidade com outros dispositivos não seja violada.

Uma outra questão premente é o gerenciamento daquilo que é conhecido como um *espaço pessoal*. Reconhecendo que um sistema doméstico consiste em muitos dispositivos compartilhados, bem como pessoais, e que os dados em um sistema doméstico também estão sujeitos a restrições de compartilhamento, muita atenção é dedicada à percepção desses espaços pessoais. Por exemplo, parte do espaço pessoal de Alice pode consistir em sua agenda, fotos da família, um diário, músicas e vídeos que ela comprou etc. Esses ativos pessoais devem ser armazenados de maneira que Alice tenha acesso a eles sempre que desejar. Além disso, partes desse espaço pessoal devem estar — temporariamente — acessíveis a outros, como no caso de ela precisar participar de uma reunião de negócios.

Felizmente, as coisas podem ficar mais simples. Há muito tempo considera-se que espaços pessoais relacionados com sistemas domésticos são inherentemente distribuídos por vários dispositivos. É óbvio que tal dispersão poderá resultar facilmente em problemas de sincronização. Todavia, esses problemas podem ser amenizados devido ao rápido crescimento da capacidade de discos rígidos, aliado à redução de seu tamanho. Configurar uma unidade de armazenamento de vários terabytes para um computador pessoal não é, realmente, um problema.

Da mesma maneira, discos rígidos portáteis com capacidade de centenas de gigabytes estão sendo colocados dentro de reprodutores de mídia portáteis relativamente pequenos. Com o crescimento contínuo dessas capacidades, é possível que logo vejamos sistemas pervasivos domésticos adotarem uma arquitetura na qual uma única máquina funcionará como mestra (e ficará escondida em algum lugar do porão, perto do aquecimento central) e todos os outros dispositivos fixos simplesmente oferecerão uma interface conveniente para os seres humanos. Então,

os dispositivos pessoais ficarão repletos de informações diárias necessárias, mas sua capacidade de armazenamento nunca se esgotará.

Contudo, ter armazenamento suficiente não resolve o problema do gerenciamento de espaços pessoais. Ser capaz de armazenar enormes quantidades de dados muda o problema para o armazenamento de dados *relevantes* e para a capacidade de achá-los mais tarde. Cada vez mais veremos sistemas pervasivos, como redes domésticas, equipados com o que denominamos **recomendadores**, programas que consultam o que os outros usuários armazenaram, de modo a identificar gostos semelhantes e, na sequência, deduzir qual conteúdo colocar no espaço pessoal de alguém. Uma observação interessante é que a quantidade de informações de que os programas recomendadores necessitam para fazer seu trabalho costuma ser pequena o suficiente para permitir que sejam executados em PDAs (Miller et al., 2004).

Sistemas eletrônicos para tratamento de saúde

Uma outra classe de sistemas pervasivos importante e que está começando a fazer sucesso é a relacionada ao tratamento eletrônico (pessoal) de saúde. Com o aumento do custo do tratamento médico, estão sendo desenvolvidos novos dispositivos para monitorar o bem-estar de indivíduos e entrar automaticamente em contato com médicos quando necessário. Em muitos desses sistemas, uma meta importante é evitar que as pessoas sejam hospitalizadas.

Sistemas para tratamentos de saúde costumam ser equipados com vários sensores organizados em uma rede de área corporal (*Body-area Network* — BAN), de preferência sem fio. Uma questão importante é que, na pior das hipóteses, tal rede deve incomodar uma pessoa o mínimo possível. Com essa finalidade em vista, a rede deve ser capaz de funcionar quando a pessoa estiver em movimento, sem que esta precise estar presa por fios elétricos a dispositivos imóveis.

Esse requisito resulta em duas organizações óbvias, como mostra a Figura 1.9. Na primeira, um hub central é parte da BAN e colhe dados conforme necessário. Esses

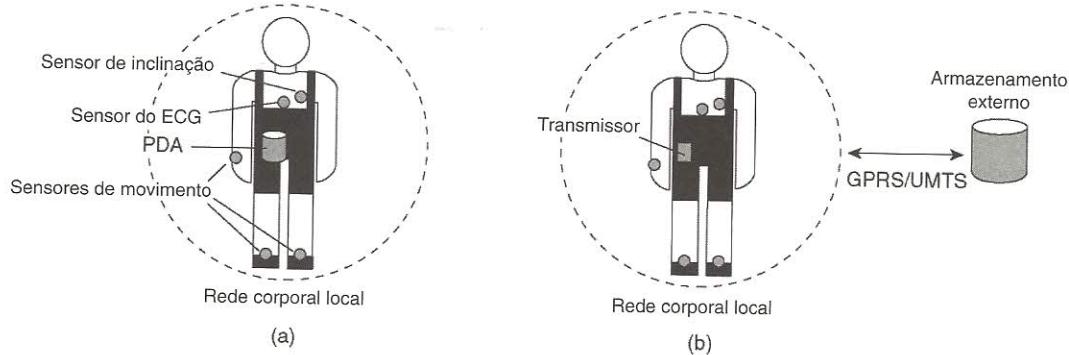


Figura 1.9 Monitoração de uma pessoa em um sistema eletrônico pervasivo de tratamento de saúde utilizando (a) um hub local ou (b) uma conexão contínua sem fio.

dados são descarregados periodicamente em um dispositivo de armazenamento de maior capacidade. A vantagem desse esquema é que o hub também pode gerenciar a BAN. Na segunda, a BAN está ligada continuamente a uma rede externa, mas uma vez por uma conexão sem fio, à qual envia dados monitorados. Será preciso disponibilizar técnicas separadas para gerenciar a BAN. Claro que também poderão existir mais conexões com um médico ou com outras pessoas.

Da perspectiva do sistema distribuído, deparamos imediatamente com questões como:

1. Onde e como os dados monitorados deverão ser armazenados?
2. Como podemos evitar a perda de dados cruciais?
3. Qual é a infra-estrutura necessária para gerar e transmitir sinais de alerta?
4. Como os médicos podem dar retorno on-line?
5. Como pode ser alcançada a extrema robustez do sistema de monitoração?
6. Quais são as questões de segurança e como as políticas adequadas podem ser impostas?

Diferentemente dos sistemas domésticos, não podemos esperar que a arquitetura de sistemas pervasivos de tratamento de saúde tenda a passar para sistemas de um único servidor e que seus dispositivos de monitoração operem com funcionalidade mínima. Ao contrário: por razões de eficiência, os dispositivos e redes de áreas corporais terão de suportar **processamento de dados na rede**, o que significa que os dados de monitoração terão de ser agregados antes de ser armazenados permanentemente ou enviados a um médico. Diferentemente do caso de sistemas de informação distribuídos, ainda não há uma resposta clara para essas questões.

Redes de sensores

Nosso último exemplo de sistemas pervasivos são as redes de sensores. Em muitos casos, essas redes são parte da tecnologia que habilita a pervasividade, e veremos que muitas soluções para redes de sensores são aproveitadas por aplicações pervasivas. O que torna as redes de sensores interessantes da perspectiva de sistema distribuído é que em praticamente todos os casos elas são usadas para processar informações. Nesse sentido, elas fazem mais do que apenas fornecer serviços de comunicação, que é o objetivo principal das redes de computadores tradicionais.

Akyildiz et al. (2002) dão uma visão geral de acordo com a perspectiva de rede. Zhao e Guibas (2004) dão uma introdução às redes de sensores orientadas a sistemas. Estreitamente relacionadas com as redes de sensores são as **redes em malha**, que, em essência, formam um conjunto de nós (fixos) que se comunicam por meio de ligações sem fio. Essas redes podem formar a base para muitos sistemas distribuídos de médio porte. Akyildiz et al. (2005) dão uma visão geral dessas redes.

Normalmente, uma rede de sensores consiste em dezenas a centenas de milhares de nós relativamente pequenos, cada um equipado com um dispositivo de sensoriamento. A maioria das redes de sensores usa comunicação sem fio, e os nós com freqüência são alimentados por bateria. Seus recursos limitados, sua capacidade restrita de comunicação e demanda reprimida de consumo de energia exigem que a eficiência ocupe um dos primeiros lugares da lista de critérios de projeto.

A relação com sistemas distribuídos pode ser esclarecida considerando redes de sensores como bancos de dados distribuídos. Essa visão é bastante comum e fácil de entender quando se percebe que muitas redes de sensores são montadas para aplicações de medição e vigilância (Bonnet et al., 2002). Nesses casos, um operador gostaria de extrair informações de (uma parte de) uma rede simplesmente emitindo consultas como “Qual é a carga de tráfego na direção norte na Rodovia 1?”. Essas consultas são parecidas com as consultas tradicionais em bancos de dados. Nesse caso, é provável que a resposta tenha de ser dada por meio da colaboração de muitos sensores localizados ao longo da Rodovia 1, deixando, ao mesmo tempo, os outros sensores intactos.

Para organizar uma rede de sensores como um banco de dados distribuído há, em essência, dois extremos, como mostra a Figura 1.10. No primeiro, os sensores não cooperam; simplesmente enviam seus dados a um banco de dados centralizado, localizado no site do operador. No outro extremo, as consultas são repassadas a sensores relevantes e permite-se que cada um processe uma resposta, o que requer que o operador agregue, de modo sensato, as respostas devolvidas.

Nenhuma dessas soluções é muito atraente. A primeira requer que os sensores enviem pela rede todos os seus dados medidos, o que pode desperdiçar recursos de rede e energia. A segunda solução também pode ser pernudária, porque despreza as capacidades de agregação dos sensores que permitiriam o retorno de uma quantidade muito menor de dados ao operador. Portanto, é preciso facilitar o **processamento de dados na rede**, como encontramos também em sistemas pervasivos de tratamento de saúde.

O processamento de dados na rede pode ser feito de várias maneiras. Uma óbvia é repassar uma consulta a todos os nós sensores ao longo de uma árvore que abrange todos os nós e, na seqüência, agrregar os resultados à medida que são propagados de volta à raiz em que está localizado o iniciador. A agregação ocorrerá onde dois ou mais ramos da árvore se encontrarem. Pode até parecer que esse sistema é simples, mas ele introduz questões difíceis:

1. Como montar (dinamicamente) uma árvore eficiente em uma rede de sensores?
2. Como ocorre a agregação de resultados? Ela pode ser controlada?
3. O que acontece quando enlaces de rede falham?

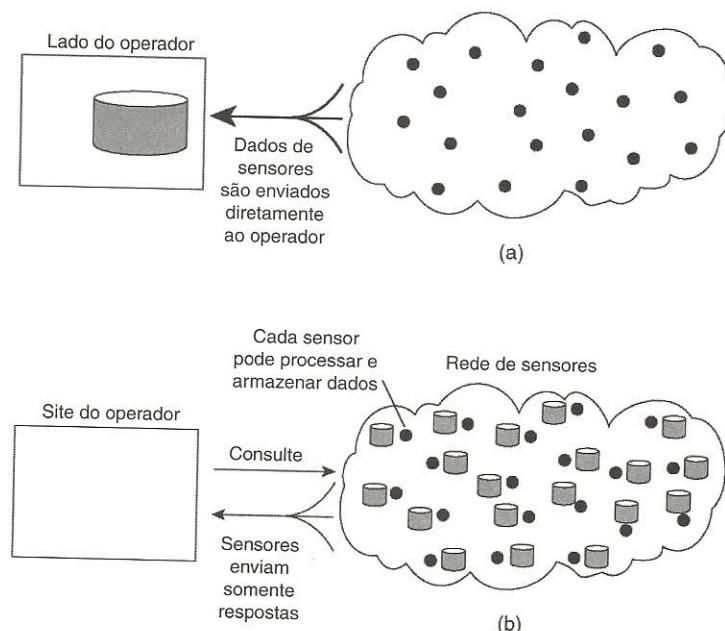


Figura 1.10 Organizando um banco de dados de rede de sensores e, ao mesmo tempo, armazenando e processando dados (a) somente no site do operador ou (b) somente nos sensores.

Essas questões são parcialmente resolvidas pelo TinyDB, que implementa uma interface declarativa (banco de dados) com redes de sensores sem fio. Em essência, o TinyDB pode usar qualquer algoritmo de roteamento baseado em árvore. Um nó intermediário colherá e agritará os resultados de seus filhos, junto com suas próprias constatações, e os enviará em direção à raiz. Para dar eficiência ao sistema, as consultas abrangem um período que leva em conta o cuidadoso escalonamento das operações, de modo que o consumo de recursos da rede e de energia seja ótimo. Se quiser mais detalhes, consulte Madden et al. (2005).

Contudo, quando consultas podem ser iniciadas em diferentes pontos da rede, usar árvores de uma única raiz, como no TinyDB, pode não ser suficientemente eficiente. Como alternativa, redes de sensores podem ser equipadas com nós especiais para os quais são repassados resultados, bem como as consultas relacionadas a esses resultados. Para dar um exemplo simples, consultas e resultados relacionados a leituras de temperatura são colhidos em um lugar diferente dos relacionados às medições de umidade. Essa abordagem corresponde diretamente à noção de sistemas publicar/subscrever, que discutiremos minuciosamente no Capítulo 13.

1.4 Resumo

Sistemas distribuídos consistem em computadores autônomos que trabalham juntos para dar a aparência de um único sistema coerente. Uma importante vantagem é

que eles facilitam a integração em um único sistema de diferentes aplicações que executam em computadores diferentes. Uma outra vantagem importante é que, quando adequadamente projetados, sistemas distribuídos podem ser ampliados com facilidade em relação ao tamanho da rede subjacente. Muitas vezes essas vantagens vêm à custa de software mais complexo, degradação do desempenho e, também, freqüentemente, de menor segurança. Não obstante, há considerável interesse mundial na construção e instalação de sistemas distribuídos.

Sistemas distribuídos costumam ter como meta ocultar grande parte das complexidades relacionadas à distribuição de processos, dados e controle. Contudo, essa transparência de distribuição não é apenas conseguida à custa do desempenho, mas, em situações práticas, ela nunca pode ser totalmente alcançada. O fato de ser necessário estabelecer compromissos de modo a obter várias formas de transparência de distribuição é inerente ao projeto de sistemas distribuídos, e é fácil que elas complicuem a sua compreensão.

As coisas ficam ainda mais complicadas porque muitos desenvolvedores adotam premissas iniciais sobre a rede subjacente que estão fundamentalmente erradas. Mais tarde, quando essas premissas são abandonadas, pode ficar difícil mascarar comportamentos indesejáveis. Um exemplo típico é adotar como premissa que a latência da rede não é significativa. Mais tarde, quando chega a hora de transferir um sistema existente para uma rede de longa distância, as latências ocultas podem afetar profundamente o projeto original do sistema. Outras ciladas incluem admitir que a rede é confiável, estática, segura e homogênea.

Existem tipos diferentes de sistemas distribuídos que podem ser classificados como orientados a suporte de computação, processamento de informações e pervasividade. Em geral, sistemas de computação distribuídos são utilizados para aplicações de alto desempenho que muitas vezes se originaram do campo da computação paralela. Uma enorme classe de sistemas distribuídos pode ser encontrada em ambientes tradicionais de escritório, nos quais os bancos de dados desempenham importante papel.

Normalmente, sistemas de processamento de transações são utilizados nesses ambientes. Por fim, há uma classe emergente de sistemas distribuídos na qual os componentes são pequenos e o sistema é composto ad hoc; porém, acima de tudo, eles não são mais gerenciados por meio de um administrador de sistemas. Essa última classe tem como representantes típicos os ambientes de computação ubíquos.

Problemas

6. Por que nem sempre é uma boa idéia visar à implementação do mais alto grau de transparência possível?
7. O que é um sistema distribuído aberto e quais são os benefícios que a abertura proporciona?
8. Descreva, com exatidão, o que quer dizer *sistema escalável*.
9. Pode-se conseguir escalabilidade pela aplicação de diferentes técnicas. Quais são essas técnicas?
10. Explique o que significa *organização virtual* e dê uma sugestão para uma possível implementação dessas organizações.
11. Dissemos que, quando uma transação é abortada, o mundo é restaurado a seu estado anterior, como se a transação nunca tivesse acontecido. Mentimos. Dê um exemplo no qual restaurar o mundo é impossível.
12. Executar transações aninhadas requer certo tipo de coordenação. Explique o que um coordenador deveria realmente fazer.
13. Argumentamos que a transparência de distribuição pode não estar presente em sistemas perversos. Essa declaração não vale para todos os tipos de transparências. Dê um exemplo.
14. Já demos alguns exemplos de sistemas distribuídos perversos: sistemas domésticos, sistemas eletrônicos para tratamento de saúde e redes de sensores. Amplie essa lista com mais exemplos.
15. **(Tarefa de laboratório)** Esboce um projeto para um sistema doméstico composto de um servidor de mídia em separado, que leva em conta a ligação com um cliente sem fio. Esse último está conectado a um equipamento (análgico) de áudio e vídeo e transforma as seqüências de mídia digital em saída analógica. O servidor executa em uma máquina separada, possivelmente conectada à Internet, mas não há nenhum teclado nem monitor conectado a ela.