Tanenbaum, A.S and van Steen, M. "Distributed Systems", 3rd Ed. (v. 3.03), https://www.distributed-systems.net, 2020.

Capítulo 1

Introdução

O ritmo em que os sistemas de computador mudam foi, é e continua sendo esmagador. De 1945, quando começou a era dos computadores modernos, até cerca de 1985, os computadores eram grandes e caros. Além disso, por falta de uma maneira de conectá-los, esses computadores operavam independentemente um do outro.

A partir de meados da década de 1980, porém, dois avanços tecnológicos começaram a mudar essa situação. A primeira foi o desenvolvimento de microprocessadores poderosos . Inicialmente, eram máquinas de 8 bits, mas logo CPUs de 16, 32 e 64 bits se tornaram comuns. Com CPUs multicore, agora estamos enfrentando o desafio de adaptar e desenvolver programas para explorar o paralelismo. De qualquer forma, a geração atual de máquinas tem o poder computacional dos mainframes implantados há 30 ou 40 anos, mas por 1/1000 do preço ou menos.

O segundo desenvolvimento foi a invenção de redes de computadores de alta velocidade . **Redes locais** ou **LANs** permitem que milhares de máquinas dentro de um prédio sejam conectadas de tal forma que pequenas quantidades de informação possam ser transferidas em alguns microssegundos ou mais. Grandes quantidades de dados podem ser movidas entre máquinas a taxas de bilhões de bits por segundo **(bps). Redes de** longa distância ou **WANs** permitem que centenas de milhões de máquinas em todo o mundo sejam conectadas em velocidades que variam de dezenas de milhares a centenas de milhões de bps e mais.

Paralelamente ao desenvolvimento de máquinas cada vez mais poderosas e conectadas em rede , também pudemos testemunhar a miniaturização de sistemas de computador com talvez o smartphone como o resultado mais impressionante. Embalados com sensores, muita memória e uma CPU poderosa, esses dispositivos são nada menos que computadores completos. Claro, eles também têm recursos de rede. Na mesma linha, os chamados [plug computer]plug computadores são

Uma versão deste capítulo foi publicada como “Uma Breve Introdução aos Sistemas Distribuídos”, Computação, v. 98(10):967-1009, 2016.

encontrar o caminho para o mercado. Esses pequenos computadores, geralmente do tamanho de um adaptador de energia, podem ser conectados diretamente a uma tomada e oferecem desempenho próximo ao de um desktop.

O resultado dessas tecnologias é que agora é não apenas viável, mas fácil montar um sistema de computação composto por um grande número de computadores em rede, sejam eles grandes ou pequenos. Esses computadores são geralmente dispersos geograficamente, razão pela qual geralmente se diz que formam um **sistema distribuído.** O tamanho de um sistema distribuído pode variar de um punhado de dispositivos a milhões de computadores. A rede de interconexão pode ser com fio, sem fio ou uma combinação de ambos. Além disso, os sistemas distribuídos geralmente são altamente dinâmicos, no sentido de que os computadores podem entrar e sair, com a topologia e o desempenho da rede subjacente mudando quase continuamente.

Neste capítulo, fornecemos uma exploração inicial de sistemas distribuídos e seus objetivos de projeto e, em seguida, discutimos alguns tipos bem conhecidos de sistemas.

1.1 O que é um sistema distribuído?

Várias definições de sistemas distribuídos foram dadas na literatura, nenhuma delas satisfatória, e nenhuma delas de acordo com qualquer uma das outras. Para nossos propósitos, é suficiente dar uma caracterização vaga:

> Um sistema distribuído é uma coleção de elementos de computação autônomos que aparecem para seus usuários como um único sistema coerente.

Esta definição refere-se a duas características dos sistemas distribuídos. A primeira é que um sistema distribuído é uma coleção de elementos de computação, cada um capaz de se comportar independentemente um do outro. Um elemento de computação, que geralmente chamaremos de **nó,** pode ser um dispositivo de hardware ou um processo de software. Uma segunda característica é que os usuários (sejam pessoas ou aplicativos) acreditam estar lidando com um único sistema. Isso significa que, de uma forma ou de outra, os nós autônomos precisam colaborar. Como estabelecer essa colaboração está no centro do desenvolvimento de sistemas distribuídos. Observe que não estamos fazendo nenhuma suposição sobre o tipo de nós. Em princípio, mesmo dentro de um único sistema, eles podem variar de computadores mainframe de alto desempenho a pequenos dispositivos em redes de sensores. Da mesma forma, não fazemos suposições sobre a maneira como os nós estão interconectados.

Característica 1: Coleção de elementos de computação autônomos

Os sistemas distribuídos modernos podem, e geralmente consistirão, em todos os tipos de nós, desde computadores muito grandes de alto desempenho até pequenos computadores plugados.

ou até mesmo dispositivos menores. Um princípio fundamental é que os nós podem agir independentemente uns dos outros, embora deva ser óbvio que, se eles se ignoram , não adianta colocá-los no mesmo sistema distribuído. Na prática, os nós são programados para atingir objetivos comuns, que são realizados através da troca de mensagens entre si. Um nó reage às mensagens que chegam, que são então processadas e, por sua vez, levam a uma comunicação adicional por meio da passagem de mensagens.

Uma observação importante é que, como consequência de se lidar com nós independentes, cada um terá sua própria noção de tempo. Em outras palavras, nem sempre podemos supor que existe algo como um **relógio global.** Essa falta de uma referência comum de tempo leva a questões fundamentais sobre sincronização e coordenação dentro de um sistema distribuído, que discutiremos extensivamente no Capítulo 6. O fato de estarmos lidando com uma coleção de nós implica que também podemos precisar para gerenciar a associação e organização dessa coleção. Em outras palavras, podemos precisar registrar quais nós podem ou não pertencer ao sistema e também fornecer a cada membro uma lista de nós com os quais ele pode se comunicar diretamente.

Gerenciar **a associação ao grupo** pode ser extremamente difícil, mesmo que apenas por razões de controle de admissão. Para explicar, fazemos uma distinção entre grupos abertos e fechados. Em um **grupo aberto,** qualquer nó pode ingressar no sistema distribuído, o que significa efetivamente que ele pode enviar mensagens para qualquer outro nó do sistema. Em contraste, com um **grupo fechado,** apenas os membros desse grupo podem se comunicar entre si e é necessário um mecanismo separado para permitir que um nó entre ou saia do grupo.

Não é difícil ver que o controle de admissão pode ser difícil. Primeiro, é necessário um mecanismo para autenticar um nó e, como veremos no Capítulo 9, se não for projetado adequadamente, o gerenciamento da autenticação pode facilmente criar um gargalo de escalabilidade. Em segundo lugar, cada nó deve, em princípio, verificar se está de fato se comunicando com outro membro do grupo e não, por exemplo, com um intruso visando causar estragos. Finalmente, considerando que um membro pode se comunicar facilmente com não membros, se a confidencialidade for um problema na comunicação dentro do sistema distribuído, podemos estar enfrentando problemas de confiança.

No que diz respeito à organização do acervo, a prática mostra que um sistema distribuído é muitas vezes organizado como uma **rede de sobreposição** [Tarkoma, 2010]. Nesse caso, um nó é normalmente um processo de software equipado com uma lista de outros processos para os quais ele pode enviar mensagens diretamente. Também pode ser o caso de um vizinho precisar ser consultado primeiro. A passagem de mensagens é então feita por meio de canais TCP/IP ou UDP, mas, como veremos no Capítulo 4, recursos de nível superior também podem estar disponíveis. Existem aproximadamente dois tipos de redes de sobreposição:

Sobreposição estruturada: neste caso, cada nó possui um conjunto bem definido de vizinhos com os quais pode se comunicar. Por exemplo, os nós são organizados em uma árvore ou anel lógico.

Sobreposição não estruturada: Nessas sobreposições, cada nó tem um número de referência para outros nós selecionados aleatoriamente.

Em qualquer caso, uma rede overlay deve, em princípio, estar sempre **conectada,** o que significa que entre quaisquer dois nós há sempre um caminho de comunicação que permite que esses nós encaminhem mensagens de um para o outro. Uma classe bem conhecida de sobreposições é formada por **redes peer-to-peer (P2P).** Exemplos de sobreposições serão discutidos em detalhes no Capítulo 2 e capítulos posteriores. É importante perceber que a organização de nós requer um esforço especial e que às vezes é uma das partes mais complexas do gerenciamento de sistemas distribuídos.

Característica 2: Sistema único coerente

Como mencionado, um sistema distribuído deve aparecer como um único sistema coerente. Em alguns casos, os pesquisadores chegaram a dizer que deveria haver uma visão de sistema único, o que significa que os usuários finais nem deveriam perceber que estão lidando com o fato de que processos, dados e controle estão dispersos em um computador. rede. Alcançar uma visão de sistema único muitas vezes é pedir demais, razão pela qual, em nossa definição de sistema distribuído, optamos por algo mais fraco, ou seja, que pareça coerente. Grosso modo, um sistema distribuído é coerente se se comporta de acordo com as expectativas de seus usuários. Mais especificamente, em um único sistema coerente, a coleção de nós como um todo opera da mesma forma, não importa onde, quando e como ocorre a interação entre um usuário e o sistema.

Oferecer uma visão única e coerente costuma ser bastante desafiador. Por exemplo, requer que um usuário final não seja capaz de dizer exatamente em qual computador um processo está sendo executado no momento, ou mesmo que parte de uma tarefa tenha sido gerada para outro processo em execução em outro lugar. Da mesma forma, onde os dados são armazenados não deve ser uma preocupação, e também não deve importar que o sistema possa estar replicando dados para melhorar o desempenho. Essa chamada **transparência de distribuição,** que discutiremos mais detalhadamente na Seção 1.2, é um objetivo importante do projeto de sistemas distribuídos. De certa forma, é semelhante à abordagem adotada em muitos sistemas operacionais do tipo Unix, nos quais os recursos são acessados por meio de uma interface unificadora de sistema de arquivos, ocultando efetivamente as diferenças entre arquivos, dispositivos de armazenamento e memória principal, mas também redes.

No entanto, lutar por um único sistema coerente introduz uma importante compensação. Como não podemos ignorar o fato de que um sistema distribuído consiste em vários nós em rede, é inevitável que, a qualquer momento, apenas uma parte do sistema falhe. Isso significa que o comportamento inesperado em que, por exemplo, alguns aplicativos podem continuar sendo executados com sucesso enquanto outros param, é uma realidade que precisa ser tratada. Embora **falhas parciais** sejam inerentes a qualquer sistema complexo, em sistemas distribuídos elas são

particularmente difícil de esconder. Isso levou a vencedora do Prêmio Turing, Leslie Lamport, a descrever um sistema distribuído como "[. . .] um em que a falha de um computador que você nem sabia que existia pode tornar seu próprio computador inutilizável."

Middleware e sistemas distribuídos

Para auxiliar o desenvolvimento de aplicativos distribuídos, os sistemas distribuídos são frequentemente organizados para ter uma camada separada de software que é colocada logicamente sobre os respectivos sistemas operacionais dos computadores que fazem parte do sistema. Essa organização é mostrada na Figura 1.1, levando ao que é conhecido como **middleware** [Bernstein, 1996].

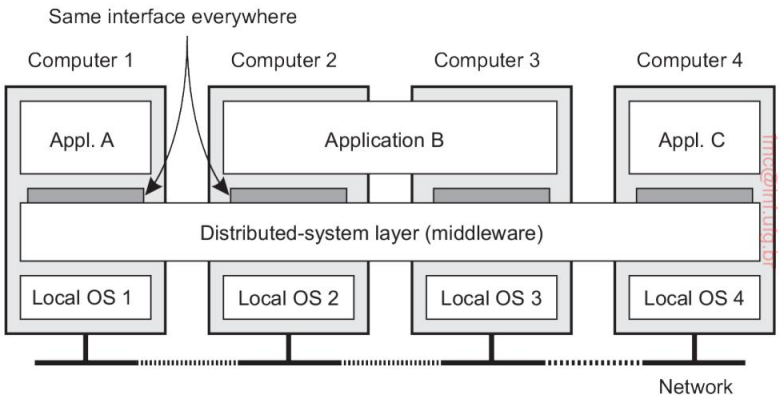

Figura 1.1: Um sistema distribuído organizado em uma camada de middleware, que se estende por várias máquinas, oferecendo a cada aplicação a mesma interface.

A Figura 1.1 mostra quatro computadores em rede e três aplicativos, dos quais o aplicativo B é distribuído pelos computadores 2 e 3. Cada aplicativo recebe a mesma interface. O sistema distribuído fornece os meios para que os componentes de um único aplicativo distribuído se comuniquem entre si, mas também permitem que diferentes aplicativos se comuniquem. Ao mesmo tempo, oculta, da melhor forma e razoavelmente possível, as diferenças de hardware e sistemas operacionais de cada aplicativo.

Em certo sentido, o middleware é o mesmo para um sistema distribuído que um sistema operacional é para um computador: um gerenciador de recursos que oferece seus aplicativos para compartilhar e distribuir eficientemente esses recursos em uma rede. Além do gerenciamento de recursos, oferece serviços que também podem ser encontrados na maioria dos sistemas operacionais, incluindo:

• Facilidades para comunicação entre aplicações. • Serviços de segurança. • Serviços de contabilidade. • Mascaramento e recuperação de falhas.

A principal diferença com seus equivalentes de sistema operacional é que os serviços de middleware são oferecidos em um ambiente de rede. Observe também que a maioria dos serviços é útil para muitos aplicativos. Nesse sentido, o middleware também pode ser visto como um contêiner de componentes e funções comumente usados que agora não precisam mais ser implementados por aplicativos separadamente. Para ilustrar melhor esses pontos, vamos considerar brevemente alguns exemplos de serviços de middleware típicos.

Comunicação: Um serviço de comunicação comum é o chamado **Remote Chamada de Procedimento (RPC).** Um serviço RPC, ao qual retornaremos no Capítulo 4, permite que um aplicativo invoque uma função que é implementada e executada em um computador remoto como se estivesse disponível localmente. Para isso , o desenvolvedor precisa apenas especificar o cabeçalho da função expresso em uma linguagem de programação especial, a partir da qual o subsistema RPC pode gerar o código necessário que estabelece invocações remotas.

Transações: Muitos aplicativos fazem uso de vários serviços que são distribuídos entre vários computadores. O middleware geralmente oferece suporte especial para executar esses serviços de forma tudo ou nada, comumente chamada de **transação atômica.** Nesse caso, o desenvolvedor do aplicativo precisa apenas especificar os serviços remotos envolvidos e, seguindo um protocolo padronizado, o middleware garante que todos os serviços sejam invocados, ou nenhum.

Composição do serviço: Está se tornando cada vez mais comum desenvolver novos aplicativos pegando programas existentes e colando-os. Este é notavelmente o caso de muitos aplicativos baseados na Web, em particular aqueles conhecidos como **serviços da Web** [Alonso et al., 2004]. O middleware baseado na Web pode ajudar padronizando a maneira como os serviços da Web são acessados e fornecendo os meios para gerar suas funções em uma ordem específica. Um exemplo simples de como a composição de serviço é implementada é formada por **mashups:** páginas da Web que combinam e agregam dados de diferentes fontes. Mashups bem conhecidos são aqueles baseados em mapas do Google em que os mapas são aprimorados com informações extras, como planejadores de viagem ou previsões do tempo em tempo real .

Confiabilidade: Como último exemplo, tem havido uma grande quantidade de pesquisas sobre o fornecimento de funções aprimoradas para a construção de aplicativos distribuídos confiáveis. O kit de ferramentas Horus [van Renesse et al., 1994] permite que um desenvolvedor construa uma aplicação como um grupo de processos de forma que qualquer mensagem enviada por um processo seja garantida para ser recebida por todos ou nenhum outro processo. Como se vê, essas garantias podem simplificar bastante o desenvolvimento de aplicativos distribuídos e geralmente são implementadas como parte do middleware.

Nota 1.1 (Nota histórica: O termo middleware)
Embora o termo middleware tenha se tornado popular em meados da década de 1990, provavelmente foi mencionado pela primeira vez em um relatório em uma conferência de engenharia de software da OTAN, editada por Peter Naur e Brian Randell em outubro de 1968 [Naur e Randell, 1968]. De fato, o middleware foi colocado precisamente entre aplicativos e rotinas de serviço (o equivalente a sistemas operacionais).

1.2 Objetivos do projeto

Só porque é possível construir sistemas distribuídos não significa necessariamente que seja uma boa ideia. Nesta seção, discutimos quatro objetivos importantes que devem ser atendidos para que a construção de um sistema distribuído valha a pena. Um sistema distribuído deve tornar os recursos facilmente acessíveis; deve ocultar o fato de que os recursos são distribuídos em uma rede; deve ser aberto; e deve ser escalável.

Suporte ao compartilhamento de recursos

Um objetivo importante de um sistema distribuído é tornar mais fácil para os usuários (e aplicativos) acessar e compartilhar recursos remotos. Os recursos podem ser praticamente qualquer coisa, mas exemplos típicos incluem periféricos, instalações de armazenamento, dados, arquivos, serviços e redes, para citar apenas alguns. Há muitas razões para querer compartilhar recursos. Uma razão óbvia é a da economia. Por exemplo, é mais barato ter uma única instalação de armazenamento confiável de ponta compartilhada do que ter que comprar e manter armazenamento para cada usuário separadamente.

Conectar usuários e recursos também facilita a colaboração e a troca de informações, conforme ilustrado pelo sucesso da Internet com seus protocolos simples para troca de arquivos, correio, documentos, áudio e vídeo.
A conectividade da Internet permitiu que grupos de pessoas geograficamente dispersos trabalhassem juntos por meio de todo tipo de **groupware,** ou seja, software para edição colaborativa, teleconferência etc., como ilustram empresas multinacionais de desenvolvimento de software que terceirizaram grande parte de sua produção de código para a Ásia.

No entanto, o compartilhamento de recursos em sistemas distribuídos talvez seja mais bem ilustrado pelo sucesso de redes peer-to-peer de compartilhamento de arquivos como o **BitTorrent.** Esses sistemas distribuídos tornam extremamente simples para os usuários compartilhar arquivos pela Internet. As redes ponto a ponto são frequentemente associadas à distribuição de arquivos de mídia, como áudio e vídeo. Em outros casos, a tecnologia é usada para distribuir grandes quantidades de dados, como no caso de atualizações de software, serviços de backup e sincronização de dados em vários servidores.

Nota 1.2 (Mais informações: Compartilhando pastas em todo o mundo)
Para ilustrar onde estamos quando se trata de integração perfeita de recursos de compartilhamento de recursos em um ambiente de rede, agora são implantados serviços baseados na Web que permitem que um grupo de usuários coloque arquivos em uma pasta compartilhada especial que é mantidos por terceiros em algum lugar na Internet. Usando software especial, a pasta compartilhada mal se distingue de outras pastas no computador de um usuário.
Com efeito, esses serviços substituem o uso de um diretório compartilhado em um local distribuído sistema de arquivos, disponibilizando dados para usuários independentes da organização que eles pertençam e independentemente de onde estejam. O serviço é oferecido para diferentes sistemas operacionais. Onde exatamente os dados são armazenados está completamente oculto do usuário final.

Tornando a distribuição transparente

Um objetivo importante de um sistema distribuído é esconder o fato de que seus processos e os recursos são fisicamente distribuídos em vários computadores possivelmente separados por grandes distâncias. Em outras palavras, ele tenta fazer a distribuição de processos e recursos **transparentes,** ou seja, invisíveis, para usuários finais e formulários.

Tipos de transparência de distribuição

O conceito de transparência pode ser aplicado a vários aspectos de uma distribuição sistema, dos quais os mais importantes estão listados na Figura 1.2. Nós usamos o termo objeto para significar um processo ou um recurso.

Descrição da transparência	
Acesso	Ocultar diferenças na representação de dados e como um objeto é acessado
Localização	Ocultar onde um objeto está localizado
Realocação	Ocultar que um objeto pode ser movido para outro local enquanto em uso
Migração	Ocultar que um objeto pode se mover para outro local
Replicação	Ocultar que um objeto é replicado
Simultaneidade	Ocultar que um objeto pode ser compartilhado por vários usuários
Falha	Ocultar a falha e a recuperação de um objeto

Figura 1.2: Diferentes formas de transparência em um sistema distribuído (ver ISO [1995]). Um objeto pode ser um recurso ou um processo.

A transparência de acesso lida com a ocultação de diferenças na representação de dados e a forma como os objetos podem ser acessados. Em um nível básico, queremos esconder

diferenças nas arquiteturas de máquina, mas o mais importante é que alcançamos acordo sobre como os dados devem ser representados por diferentes máquinas e sistemas operacionais. Por exemplo, um sistema distribuído pode ter sistemas de computador que executam diferentes sistemas operacionais, cada um com suas próprias convenções de nomenclatura de arquivos . Diferenças nas convenções de nomenclatura, diferenças nas operações de arquivo ou diferenças em como a comunicação de baixo nível com outros processos deve ser local, são exemplos de problemas de acesso que devem ser preferencialmente ocultados usuários e aplicativos.

Um grupo importante de tipos de transparência diz respeito à localização de um processo ou recurso. A **transparência de localização** refere-se ao fato de que os usuários não podem dizer onde um objeto está fisicamente localizado no sistema. A nomeação desempenha um papel importante na obtenção de transparência de localização. Em particular, a localização a transparência muitas vezes pode ser alcançada atribuindo apenas nomes lógicos aos recursos, isto é, nomes nos quais a localização de um recurso não é secretamente codificado. Um exemplo de tal nome é o **localizador uniforme de recursos (URL)** http://www.prenhall.com/index.html, que não dá nenhuma pista sobre o real localização do servidor Web principal da Prentice Hall. O URL também não dá nenhuma pista, pois se o arquivo index.html sempre esteve em seu local atual ou foi mudou-se recentemente para lá. Por exemplo, todo o site pode ter sido movido de um data center para outro, mas os usuários não devem notar. Este último é um exemplo de **transparência de realocação,** que está se tornando cada vez mais importante na o contexto da computação em nuvem ao qual retornaremos mais adiante neste capítulo.

Onde a transparência de realocação se refere a ser movido pelo sistema, **a transparência de migração** é oferecida por um sistema distribuído quando suporta a mobilidade de processos e recursos iniciados pelos usuários, sem afetar a comunicação e as operações em andamento. Um exemplo típico é a comunicação entre telefones celulares: independentemente de duas pessoas estão realmente se movendo, os telefones celulares permitirão que eles continuem a conversa. Outros exemplos que vêm à mente incluem rastreamento online e rastreamento de mercadorias enquanto elas estão sendo transportadas de um lugar para outro, e teleconferência (parcialmente) usando dispositivos equipados com Internet.

Como veremos, a replicação desempenha um papel importante em sistemas distribuídos. Por exemplo, os recursos podem ser replicados para aumentar a disponibilidade ou melhorar o desempenho colocando uma cópia próxima ao local de acesso. A **transparência de replicação** trata de ocultar o fato de que várias cópias de um recurso existe, ou que vários processos estão operando em alguma forma de bloqueio modo para que um possa assumir quando outro falhar. Para ocultar a replicação de usuários, é necessário que todas as réplicas tenham o mesmo nome. Consequentemente, um sistema que suporta transparência de replicação geralmente deve suportar transparência de localização também, porque de outra forma seria impossível referem-se a réplicas em locais diferentes.

Já mencionamos que um objetivo importante dos sistemas distribuídos é

para permitir o compartilhamento de recursos. Em muitos casos, o compartilhamento de recursos é feito de forma cooperativa, como no caso dos canais de comunicação. No entanto, também há muitos exemplos de compartilhamento competitivo de recursos. Por exemplo, dois usuários independentes podem ter seus arquivos armazenados no mesmo servidor de arquivos ou podem estar acessando 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 chamado de **transparência de concorrência.** Uma questão importante é que o acesso simultâneo a um recurso compartilhado deixa esse recurso em um estado consistente. A consistência pode ser alcançada por meio de mecanismos de bloqueio, pelos quais os usuários, por sua vez, recebem acesso exclusivo ao recurso desejado. Um mecanismo mais refinado é fazer uso de transações, mas elas podem ser difíceis de implementar em um sistema distribuído, principalmente quando a escalabilidade é um problema.

Por último, mas certamente não menos importante, é importante que um sistema distribuído forneça **transparência de falhas.** Isso significa que um usuário ou aplicativo não percebe que alguma parte do sistema não funciona corretamente e que o sistema subsequentemente (e automaticamente) se recupera dessa falha. Mascarar falhas é uma das questões mais difíceis em sistemas distribuídos e é até mesmo impossível quando certas suposições aparentemente realistas são feitas, como discutiremos no Capítulo 8. processo e uma resposta dolorosamente lenta. Por exemplo, ao entrar em contato com um servidor da Web ocupado , um navegador eventualmente atingirá o tempo limite e informará que a página da Web não está disponível. Nesse ponto, o usuário não pode dizer se o servidor está realmente inativo ou se a rede está muito congestionada.

Grau de transparência de distribuição

Embora a transparência de distribuição seja geralmente considerada preferível para qualquer sistema distribuído, há situações em que tentar ocultar cegamente todos os aspectos de distribuição dos usuários não é uma boa ideia. Um exemplo simples é solicitar que seu jornal eletrônico apareça em sua caixa postal antes das 7h , horário local, como de costume, enquanto você está do outro lado do mundo vivendo em um fuso horário diferente. Seu jornal matinal não será o jornal matinal que você está acostumado.

Da mesma forma, um sistema distribuído de área ampla que conecta um processo em São Francisco a um processo em Amsterdã não pode esconder o fato de que a Mãe Natureza não permitirá que ele envie uma mensagem de um processo para outro em menos de aproximadamente 35 milissegundos. . A prática mostra que, na verdade, são necessárias várias centenas de milissegundos usando uma rede de computadores. A transmissão do sinal não é limitada apenas pela velocidade da luz, mas também por capacidades de processamento limitadas e atrasos nos interruptores intermediários.

Há também um trade-off entre um alto grau de transparência e o desempenho de um sistema. Por exemplo, muitos aplicativos da Internet repetidamente

tente entrar em contato com um servidor antes de finalmente desistir. Conseqüentemente, tentar mascarar uma falha transitória do servidor antes de tentar outra pode tornar o sistema mais lento como um todo. Nesse caso, talvez fosse melhor desistir antes, ou pelo menos deixar o usuário cancelar as tentativas de contato.

Outro exemplo é onde precisamos garantir que várias réplicas, localizadas em diferentes continentes, sejam 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 uma única operação de atualização agora pode levar alguns segundos para ser concluída, algo que não pode ser ocultado usuários.

Finalmente, há situações em que não é tão óbvio que ocultar a distribuição seja uma boa ideia. À medida que os sistemas distribuídos estão se expandindo para dispositivos que as pessoas carregam e onde a própria noção de localização e reconhecimento de contexto está se tornando cada vez mais importante, pode ser melhor expor a distribuição em vez de tentar escondê-la. Um exemplo óbvio é fazer uso de serviços baseados em localização, que muitas vezes podem ser encontrados em telefones celulares, como encontrar o take-away chinês mais próximo ou verificar se algum de seus amigos está por perto.

Há também outros argumentos contra a transparência da distribuição. Reconhecendo que a transparência total da distribuição é simplesmente impossível, devemos nos perguntar se é mesmo sensato fingir que podemos alcançá-la. Pode ser muito melhor tornar a distribuição explícita para que o usuário e o desenvolvedor do aplicativo nunca sejam levados a acreditar que existe algo como transparência. O resultado será que os usuários entenderão muito melhor o comportamento (às vezes inesperado) de um sistema distribuído e, portanto, estarão muito mais preparados para lidar com esse comportamento.

Nota 1.3 (Discussão: Contra a transparência da distribuição)
Vários pesquisadores argumentaram que ocultar a distribuição levará a complicar ainda mais o desenvolvimento de sistemas distribuídos, exatamente pelo motivo de que a transparência total da distribuição nunca pode ser alcançada. Uma técnica popular para obter transparência de acesso é estender chamadas de procedimento para servidores remotos. No entanto, Waldo et al. [1997] já apontavam que tentar esconder a distribuição por meio de tais chamadas de procedimento remoto pode levar a uma semântica mal compreendida, pela simples razão de que uma chamada de procedimento muda quando executada em um link de comunicação defeituoso.

Como alternativa, vários pesquisadores e profissionais estão agora defendendo menos transparência, por exemplo, usando mais explicitamente a comunicação no estilo de mensagem , ou postando solicitações mais explicitamente e obtendo resultados de máquinas remotas, como é feito na Web ao buscar Páginas. Tais soluções serão discutidas em detalhes no próximo capítulo.

Um ponto de vista um tanto radical é adotado por Wams [2011] ao afirmar que falhas parciais impedem confiar na execução bem-sucedida de um serviço remoto. Se tal confiabilidade não puder ser garantida, é melhor sempre realizar apenas

execuções, levando ao princípio **de copiar antes de usar .** De acordo com este princípio, os dados só podem ser acessados após terem sido transferidos para a máquina do processo que deseja esses dados. Além disso, a modificação de um item de dados não deve ser feita. Em vez disso, ele só pode ser atualizado para uma nova versão. Não é difícil imaginar que muitos outros problemas virão à tona. No entanto, Wams mostra que muitos aplicativos existentes podem ser adaptados para essa abordagem alternativa sem sacrificar a funcionalidade.

A conclusão é que buscar transparência na distribuição pode ser um bom objetivo ao projetar e implementar sistemas distribuídos, mas que deve ser considerado em conjunto com outras questões como desempenho e compreensibilidade. O preço para alcançar a transparência total pode ser surpreendentemente alto.

Estar aberto

Outro objetivo importante dos sistemas distribuídos é a abertura. Um **sistema distribuído aberto** é essencialmente um sistema que oferece componentes que podem ser facilmente usados ou integrados a outros sistemas. Ao mesmo tempo, um sistema distribuído aberto geralmente consiste em componentes que se originam de outros lugares.

Interoperabilidade, composição e extensibilidade

Ser aberto significa que os componentes devem aderir a regras padrão que descrevem a sintaxe e a semântica do que esses componentes têm a oferecer (ou seja, qual serviço eles fornecem). Uma abordagem geral é definir serviços por meio de **interfaces** usando uma **Interface Definition Language (IDL).** As definições de interface escritas em um IDL quase sempre capturam apenas a sintaxe dos serviços. Em outras palavras, eles especificam precisamente os nomes das funções que estão disponíveis junto com os tipos dos parâmetros, valores de retorno, possíveis exceções que podem ser levantadas e assim por diante. A parte difícil é especificar precisamente o que esses serviços fazem, ou seja, a semântica das interfaces. Na prática, tais especificações são dadas de forma informal por meio de linguagem natural.

Se especificada corretamente, uma definição de interface permite que um processo arbitrário que precise de uma determinada interface converse com outro processo que forneça essa interface. Ele também permite que duas partes independentes construam implementações completamente diferentes dessas interfaces, levando a dois componentes separados que operam exatamente da mesma maneira.

As especificações adequadas são completas e neutras. Completo significa que tudo o que é necessário para fazer uma implementação foi realmente especificado. No entanto, muitas definições de interface não estão completas, de modo que é necessário que um desenvolvedor adicione detalhes específicos da implementação.

Igualmente importante é o fato de que as especificações não prescrevem como deve ser a aparência de uma implementação; eles devem ser neutros.

Conforme apontado em Blair e Stefani [1998], a integridade e a neutralidade são importantes para a interoperabilidade e a portabilidade. A **interoperabilidade** caracteriza a extensão pela qual duas implementações de sistemas ou componentes de diferentes fabricantes podem coexistir e trabalhar em conjunto simplesmente confiando nos serviços um do outro conforme especificado por um padrão comum. A **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.

Outro objetivo importante para um sistema distribuído aberto é que deve ser fácil configurar o sistema a partir de diferentes componentes (possivelmente de diferentes desenvolvedores). Além disso, deve ser fácil adicionar novos componentes ou substituir os existentes sem afetar os componentes que permanecem no lugar.
Em outras palavras, um sistema distribuído aberto também deve ser **extensível.** Por exemplo, em um sistema extensível, deve ser relativamente fácil adicionar partes executadas em um sistema operacional diferente ou até mesmo substituir um sistema de arquivos inteiro.

Nota 1.4 (Discussão: Sistemas abertos na prática)
Naturalmente, o que acabamos de descrever é uma situação ideal. A prática mostra que muitos sistemas distribuídos não são tão abertos quanto gostaríamos e que ainda é necessário muito esforço para juntar várias partes e peças para fazer um sistema distribuído. Uma saída para a falta de abertura é simplesmente revelar todos os detalhes sangrentos de um componente e fornecer aos desenvolvedores o código-fonte real. Essa abordagem está se tornando cada vez mais popular, levando aos chamados projetos de código aberto, onde grandes grupos de pessoas contribuem para melhorar e depurar sistemas. É certo que isso é o mais aberto que um sistema pode ser, mas se é a melhor maneira é questionável.

Separando a política do mecanismo

Para obter flexibilidade em sistemas distribuídos abertos, é crucial que o sistema seja organizado como uma coleção de componentes relativamente pequenos e facilmente substituíveis ou adaptáveis. Isso implica que devemos fornecer definições não apenas das interfaces de nível mais alto, ou seja, aquelas vistas por usuários e aplicativos, mas também definições de interfaces para partes internas do sistema e descrever como essas partes interagem. Essa abordagem é relativamente nova. Muitos sistemas mais antigos e até contemporâneos são construídos usando uma abordagem monolítica na qual os componentes são separados apenas logicamente, mas implementados como um grande programa. Essa abordagem dificulta a substituição ou adaptação de um componente sem afetar todo o sistema. Assim, os sistemas monolíticos tendem a ser fechados em vez de abertos.

A necessidade de alterar um sistema distribuído geralmente é causada por um componente que não fornece a política ideal para um usuário ou aplicativo específico.

Como exemplo, considere o armazenamento em cache em navegadores da Web. Existem muitos parâmetros diferentes que precisam ser considerados:

Armazenamento: onde os dados devem ser armazenados em cache? Normalmente, haverá um cache na memória próximo ao armazenamento em disco. Neste último caso, a posição exata no sistema de arquivos local precisa ser considerada.

Isenção: Quando o cache fica cheio, quais dados devem ser removidos para que páginas recém-buscadas podem ser armazenadas?

Compartilhamento: Cada navegador faz uso de um cache privado ou um cache deve ser compartilhado entre navegadores de usuários diferentes?

Atualizando: quando um navegador verifica se os dados armazenados em cache ainda estão atualizados?

Os caches são mais eficazes quando um navegador pode retornar páginas sem precisar entrar em contato com o site original. No entanto, isso tem o risco de retornar dados obsoletos. Observe também que as taxas de atualização são altamente dependentes de quais dados são realmente armazenados em cache: enquanto os horários dos trens dificilmente mudam, esse não é o caso das páginas da Web que mostram as condições atuais do tráfego nas rodovias ou, pior ainda, os preços das ações.

O que precisamos é de uma separação entre política e mecanismo. No caso do cache da Web, por exemplo, um navegador deve, idealmente, fornecer recursos para armazenar apenas documentos e, ao mesmo tempo, permitir que os usuários decidam quais documentos serão armazenados e por quanto tempo. Na prática, isso pode ser implementado oferecendo um rico conjunto de parâmetros que o usuário pode definir (dinamicamente). Ao levar isso um passo adiante, um navegador pode até oferecer recursos para conectar políticas que um usuário implementou como um componente separado.

Nota 1.5 (Discussão: Uma separação estrita é realmente o que precisamos?)
Em teoria, separar estritamente as políticas dos mecanismos parece ser o caminho a seguir. No entanto, há um trade-off importante a ser considerado: quanto mais rigorosa a separação, mais precisamos ter certeza de que oferecemos a coleção apropriada de mecanismos. Na prática, isso significa que um rico conjunto de recursos é oferecido, levando a muitos parâmetros de configuração. Como exemplo, o popular navegador Firefox vem com algumas centenas de parâmetros de configuração. Imagine como o espaço de configuração explode ao considerar grandes sistemas distribuídos que consistem em muitos componentes. Em outras palavras, a separação estrita de políticas e mecanismos pode levar a problemas de configuração altamente complexos.

Uma opção para aliviar esses problemas é fornecer padrões razoáveis, e isso é o que muitas vezes acontece na prática. Uma abordagem alternativa é aquela em que o sistema observa seu próprio uso e altera dinamicamente as configurações dos parâmetros. Isso leva ao que é conhecido como **sistemas autoconfiguráveis.** No entanto, o fato de que muitos mecanismos precisam ser oferecidos para suportar uma ampla gama de políticas muitas vezes torna a codificação de sistemas distribuídos muito complicada. As políticas de codificação em um sistema distribuído podem reduzir consideravelmente a complexidade, mas ao preço de menos flexibilidade.

Encontrar o equilíbrio certo na separação de políticas de mecanismos é uma das razões pelas quais projetar um sistema distribuído é muitas vezes mais uma arte do que uma ciência.

Ser escalável

Para muitos de nós, a conectividade mundial através da Internet é tão comum quanto enviar um cartão postal para qualquer pessoa em qualquer lugar do mundo. Além disso, onde até recentemente estávamos acostumados a ter computadores desktop relativamente poderosos para aplicativos de escritório e armazenamento, agora estamos testemunhando que esses aplicativos e serviços estão sendo colocados no que foi chamado de “nuvem”, levando a um aumento de muito dispositivos menores em rede, como computadores tablet. Com isso em mente, a escalabilidade tornou-se um dos objetivos de design mais importantes para desenvolvedores de sistemas distribuídos.

Dimensões de escalabilidade

A escalabilidade de um sistema pode ser medida ao longo de pelo menos três dimensões diferentes (ver [Neuman, 1994]):

Escalabilidade de tamanho: Um sistema pode ser escalável em relação ao seu tamanho, o que significa que podemos adicionar facilmente mais usuários e recursos ao sistema sem qualquer perda perceptível de desempenho.

Escalabilidade geográfica: Um sistema geograficamente escalável é aquele em que os usuários e os recursos podem estar distantes, mas o fato de que os atrasos de comunicação podem ser significativos dificilmente é notado.

Escalabilidade administrativa: Um sistema escalável administrativamente é aquele que ainda pode ser facilmente gerenciado, mesmo que abranja muitas organizações administrativas independentes.

Vamos dar uma olhada em cada uma dessas três dimensões de escalabilidade.

Escalabilidade de tamanho. Quando um sistema precisa ser dimensionado, tipos muito diferentes de problemas precisam ser resolvidos. Vamos primeiro considerar a escala em relação ao tamanho.
Se mais usuários ou recursos precisam ser suportados, muitas vezes somos confrontados com as limitações dos serviços centralizados, embora muitas vezes por motivos muito diferentes. Por exemplo, muitos serviços são centralizados no sentido de que são implementados por meio de um único **servidor** rodando em uma máquina específica no sistema distribuído. Em um cenário mais moderno, podemos ter um grupo de servidores colaborativos colocados em um cluster de máquinas fortemente acopladas fisicamente colocadas no mesmo local. O problema com esse esquema é óbvio: o servidor, ou grupo de servidores, pode simplesmente se tornar um gargalo quando precisar processar um número crescente de solicitações. Para ilustrar como isso

pode acontecer, vamos supor que um serviço seja implementado em uma única máquina. Nesse caso, existem essencialmente três causas principais para se tornar um gargalo:

- A capacidade computacional, limitada pelas CPUs
- A capacidade de armazenamento, incluindo a taxa de transferência de E/S
- A rede entre o usuário e o serviço centralizado

Vamos primeiro considerar a capacidade computacional. Imagine um serviço para calcular rotas ideais levando em consideração as informações de tráfego em tempo real. Não é difícil imaginar que isso pode ser principalmente um serviço vinculado à computação que requer vários (dezenas) segundos para concluir uma solicitação. Se houver apenas uma única máquina disponível, até mesmo um sistema moderno de ponta acabará tendo problemas se o número de solicitações aumentar além de um determinado ponto.

Da mesma forma, mas por motivos diferentes, teremos problemas ao ter um serviço que é principalmente vinculado a E/S. Um exemplo típico é um mecanismo de pesquisa centralizado mal projetado. O problema com as consultas de pesquisa baseadas em conteúdo é que, essencialmente, precisamos corresponder uma consulta a um conjunto de dados inteiro. Mesmo com técnicas avançadas de indexação, ainda podemos enfrentar o problema de ter que processar uma enorme quantidade de dados que excede a capacidade da memória principal da máquina que executa o serviço. Como consequência, grande parte do tempo de processamento será determinado pelos acessos relativamente lentos ao disco e pela transferência de dados entre o disco e a memória principal. A simples adição de mais discos ou de maior velocidade provará não ser uma solução sustentável, pois o número de solicitações continua a aumentar.

Finalmente, a rede entre o usuário e o serviço também pode ser a causa da baixa escalabilidade. Imagine um serviço de vídeo sob demanda que precisa transmitir vídeo de alta qualidade para vários usuários. Um fluxo de vídeo pode facilmente exigir uma largura de banda de 8 a 10 Mbps, o que significa que, se um serviço estabelecer conexões ponto a ponto com seus clientes, em breve poderá atingir os limites da capacidade da rede de suas próprias linhas de transmissão de saída.

Existem várias soluções para atacar a escalabilidade do tamanho que discutimos abaixo depois de analisar a escalabilidade geográfica e administrativa.

Nota 1.6 (Avançado: Analisando a capacidade de atendimento)
Problemas de escalabilidade de tamanho para serviços centralizados podem ser formalmente analisados usando a teoria de filas e fazendo algumas suposições simplificadoras. Em um nível conceitual, um serviço centralizado pode ser modelado como o sistema de enfileiramento simples mostrado na Figura 1.3: os pedidos são enviados ao serviço onde são enfileirados até novo aviso. Assim que o processo puder lidar com uma próxima solicitação, ele a buscará na fila, fará seu trabalho e produzirá uma resposta. Seguimos amplamente Menasce e Almeida [2002] na explicação do desempenho de um serviço centralizado.

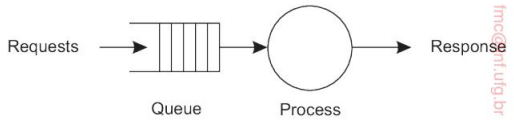

Figura 1.3: Um modelo simples de um serviço como sistema de filas.

Em muitos casos, podemos supor que a fila tem capacidade infinita, o que significa que não há restrição quanto ao número de solicitações que podem ser aceitas para processamento posterior. A rigor, isso significa que a taxa de chegada de solicitações não é influenciada pelo que está atualmente na fila ou sendo processado. Assumindo que a taxa de chegada de requisições é de $\ddot{y}$ requisições por segundo, e que a capacidade de processamento do serviço é de μ requisições por segundo, pode-se calcular que a fração de tempo pk que há k requisições no sistema é igual a:

$$\ddot{y}\ \text{pk} = 1 \ddot{y}\ \mu^{-} \quad \frac{eu}{m}^{k}$$

Se definirmos a **utilização** U de um serviço como a fração de tempo em que ele está ocupado, então claramente,

$$U = \ddot{y}_{k>0}\ \text{pk} = 1 \ddot{y}\ \text{p0} = \ddot{y}\ \overline{\text{pk}} = (1 \ddot{y}\ U)U\ \mu^{k}$$

Podemos então calcular o número médio N de solicitações no sistema como

$$\overline{N} = \ddot{y}_{k\ddot{y}0}\ k \cdot \text{pk} = \ddot{y}\ \underset{k\ddot{y}0}{k} \cdot (1 \ddot{y}\ U)U^{k} = (1 \ddot{y}\ U)\ \ddot{y}_{k\ddot{y}0}\ k\, u\, \_\_ = \frac{(1 \ddot{y}\ U)U}{(1 - U)^{2}} = \frac{\text{DENTRO}}{1 - U}.$$

O que realmente nos interessa é o tempo de resposta R: quanto tempo leva para o serviço processar uma solicitação, incluindo o tempo gasto na fila.

Para isso, precisamos do throughput médio X. Considerando que o serviço está "ocupado" quando pelo menos uma requisição está sendo processada, e que isso acontece com um throughput de μ solicitações por segundo, e durante uma fração U do total tempo, temos:

$$X = \underset{\text{servidor no trabalho}}{U \cdot \mu_{(\text{Com}}} + \underset{\text{servidor ocioso}}{(1 \ddot{y}\ U) \cdot 0_{(\text{Com}}} = \ddot{y}_{\mu} \cdot \mu = \ddot{y}$$

Usando a fórmula de Little [Trivedi, 2002], podemos então derivar o tempo de resposta como

$$R = \frac{\overline{N}}{X} = \frac{S}{1 - U}\ \ddot{y}\ \frac{R}{S} = \frac{1}{1 - U}$$

1 onde $S = \mu^{-}$, o tempo real de serviço. Observe que, se U for muito pequeno, a razão de tempo de resposta para serviço será próxima de 1, o que significa que uma solicitação é processada praticamente instantaneamente e na velocidade máxima possível. No entanto, assim que a utilização se aproxima de 1, vemos que a razão de tempo de resposta ao servidor aumenta rapidamente para valores muito altos, significando efetivamente que o sistema está chegando perto de uma moagem

parar. É aqui que vemos surgir problemas de escalabilidade. A partir deste modelo simples, podemos ver que a única solução é diminuir o tempo de serviço S. Deixamos como exercício para o leitor explorar como S pode ser diminuído.

Escalabilidade geográfica. A escalabilidade geográfica tem seus próprios problemas. Uma das principais razões pelas quais ainda é difícil dimensionar os sistemas distribuídos existentes que foram projetados para redes locais é que muitos deles são baseados em **comunicação síncrona.** Nesta forma de comunicação, uma parte solicitante do serviço, geralmente chamada de **cliente,** bloqueia até que uma resposta seja enviada de volta do **servidor que** implementa o serviço. Mais especificamente, muitas vezes vemos um padrão de comunicação que consiste em muitas interações cliente-servidor, como pode ser o caso de transações de banco de dados. Essa abordagem geralmente funciona bem em LANs onde a comunicação entre duas máquinas geralmente é de algumas centenas de microssegundos. No entanto, em um sistema de área ampla, precisamos levar em conta que a comunicação entre processos pode ser centenas de milissegundos, três ordens de magnitude mais lenta. Construir aplicativos usando comunicação síncrona em sistemas de longa distância requer muito cuidado (e não apenas um pouco de paciência), notadamente com um rico padrão de interação entre cliente e servidor.

Outro problema que dificulta a escalabilidade geográfica é que a comunicação em redes de longa distância é inerentemente muito menos confiável do que em redes locais. Além disso, também precisamos lidar com largura de banda limitada. O efeito é que as soluções desenvolvidas para redes de área local nem sempre podem ser facilmente portadas para um sistema de área ampla. Um exemplo típico é o streaming de vídeo. Em uma rede doméstica, mesmo tendo apenas links sem fio, é bastante simples garantir um fluxo estável e rápido de quadros de vídeo de alta qualidade de um servidor de mídia para um monitor . Simplesmente colocar o mesmo servidor longe e usar uma conexão TCP padrão para o monitor certamente falhará: limitações de largura de banda surgirão instantaneamente , mas também manter o mesmo nível de confiabilidade pode facilmente causar dores de cabeça.

Ainda outro problema que surge quando os componentes estão distantes é o fato de que os sistemas de área ampla geralmente têm apenas recursos muito limitados para comunicação multiponto. Em contraste, as redes locais geralmente suportam mecanismos eficientes de transmissão. Tais mecanismos têm se mostrado extremamente úteis para a descoberta de componentes e serviços, o que é essencial do ponto de vista gerencial. Em sistemas de longa distância, precisamos desenvolver serviços separados, como serviços de nomes e diretórios para os quais as consultas podem ser enviadas. Esses serviços de suporte, por sua vez, também precisam ser escaláveis e, em muitos casos, não existem soluções óbvias, como veremos em capítulos posteriores.

Escalabilidade administrativa. Finalmente, uma questão difícil, e em muitos casos aberta, é como dimensionar um sistema distribuído em vários administradores independentes.

domínios trativos. Um grande problema que precisa ser resolvido é o das políticas conflitantes com relação ao uso de recursos (e pagamento), gerenciamento e segurança.

Para ilustrar, há muitos anos os cientistas procuram soluções para compartilhar seus equipamentos (muitas vezes caros) no que é conhecido como **grade computacional.** Nessas grades, um sistema distribuído global é construído como uma federação de sistemas distribuídos locais, permitindo que um programa executado em um computador da organização A acesse diretamente os recursos da organização B.

Por exemplo, muitos componentes de um sistema distribuído que residem em um único domínio geralmente podem ser confiáveis por usuários que operam nesse mesmo domínio. Nesses casos, a administração do sistema pode ter testado e certificado aplicativos e pode ter tomado medidas especiais para garantir que tais componentes não possam ser adulterados. Em essência, os usuários confiam em seus administradores de sistema. No entanto, essa confiança não se expande naturalmente entre os limites do domínio.

Nota 1.7 (Exemplo: Um radiotelescópio moderno)
Como exemplo, considere o desenvolvimento de um radiotelescópio moderno, como o Observatório Pierre Auger [Abraham et al., 2004]. O sistema final pode ser considerado como um sistema distribuído federado:

- O próprio radiotelescópio pode ser um sistema distribuído sem fio desenvolvido como uma grade de alguns milhares de nós sensores, cada um coletando sinais de rádio e colaborando com nós vizinhos para filtrar eventos relevantes. Os nós mantêm dinamicamente uma árvore coletora pela qual os eventos selecionados são roteados para um ponto central para análise posterior.
- O ponto central precisa ser um sistema razoavelmente poderoso, capaz de armazenar e processar os eventos enviados a ele pelos nós sensores. Este sistema é necessariamente colocado na proximidade dos nós sensores, mas de outra forma deve ser considerado para operar de forma independente. Dependendo de sua funcionalidade, pode operar como um pequeno sistema distribuído local. Em particular, ele armazena todos os eventos registrados e oferece acesso a sistemas remotos pertencentes a parceiros do consórcio.
- A maioria dos parceiros tem sistemas distribuídos locais (geralmente na forma de um cluster de computadores) que eles usam para processar os dados coletados pelo telescópio. Neste caso, os sistemas locais acessam diretamente o ponto central do telescópio usando um protocolo de comunicação padrão. Naturalmente, muitos resultados produzidos dentro do consórcio são disponibilizados para cada parceiro.

Vê-se, assim, que o sistema completo cruzará as fronteiras de vários domínios administrativos, e que medidas especiais são necessárias para garantir que os dados que deveriam ser acessíveis apenas a parceiros (específicos) do consórcio não possam ser divulgados a terceiros não autorizados. Como alcançar a escalabilidade administrativa não é óbvio.

Se um sistema distribuído se expandir para outro domínio, dois tipos de segurança

medidas precisam ser tomadas. Primeiro, o sistema distribuído precisa se proteger contra ataques maliciosos do novo domínio. Por exemplo, os usuários do novo domínio podem ter apenas acesso de leitura ao sistema de arquivos em seu domínio original. Da mesma forma, recursos como setters de imagem caros ou computadores de alto desempenho podem não estar disponíveis para usuários não autorizados.

Segundo, o novo domínio precisa se proteger contra ataques maliciosos do sistema distribuído. Um exemplo típico é o download de programas como applets em navegadores da Web. Basicamente, o novo domínio não sabe o que esperar desse código estrangeiro. O problema, como veremos no Capítulo 9, é como impor essas limitações.

Como um contra-exemplo de sistemas distribuídos abrangendo vários domínios administrativos que aparentemente não sofrem de problemas de escalabilidade administrativa, considere as modernas redes peer-to-peer de compartilhamento de arquivos. Nesses casos, os usuários finais simplesmente instalam um programa que implementa as funções de pesquisa e download distribuídas e, em poucos minutos, podem começar a baixar os arquivos. Outros exemplos incluem aplicativos peer-to-peer para telefonia pela Internet, como Skype [Baset e Schulzrinne, 2006], e aplicativos de streaming de áudio assistidos por pares, como Spotify [Kreitz e Niemelä, 2010]. O que esses sistemas distribuídos têm em comum é que os usuários finais, e não as entidades administrativas, colaboram para manter o sistema funcionando. Na melhor das hipóteses, organizações administrativas subjacentes , como **Provedores de Serviços de Internet (ISPs) ,** podem policiar o tráfego de rede causado por esses sistemas ponto a ponto, mas até agora esses esforços não foram muito eficazes.

Técnicas de dimensionamento

Discutir alguns dos problemas de escalabilidade nos leva à questão de como esses problemas geralmente podem ser resolvidos. Na maioria dos casos, problemas de escalabilidade em sistemas distribuídos aparecem como problemas de desempenho causados pela capacidade limitada de servidores e rede. O simples aumento de sua capacidade (por exemplo, aumentando a memória, atualizando CPUs ou substituindo módulos de rede) geralmente é uma solução, conhecida como **expansão.** Quando se trata de **dimensionar,** ou seja, expandir o sistema distribuído essencialmente implantando mais máquinas, existem basicamente apenas três técnicas que podemos aplicar: ocultar latências de comunicação, distribuição de trabalho e replicação (veja também Neuman [1994]).

Ocultar latências de comunicação. Ocultar latências de comunicação é aplicável no caso de escalabilidade geográfica. A ideia básica é simples: tente evitar ao máximo esperar por respostas a solicitações de serviços remotos.

Por exemplo, quando um serviço é solicitado em uma máquina remota, uma alternativa à espera de uma resposta do servidor é fazer outro trabalho útil ao lado do solicitante. Essencialmente, isso significa construir o aplicativo solicitante de forma que use apenas **comunicação assíncrona.** Quando uma resposta chega, o aplicativo é interrompido e um manipulador especial é chamado

para completar o pedido anteriormente emitido. A comunicação assíncrona geralmente pode ser usada em sistemas de processamento em lote e aplicativos paralelos nos quais tarefas independentes podem ser agendadas para execução enquanto outra tarefa aguarda a conclusão da comunicação. Alternativamente, um novo thread de controle pode ser iniciado para realizar a solicitação. Embora bloqueie a espera da resposta, outros threads no processo podem continuar.

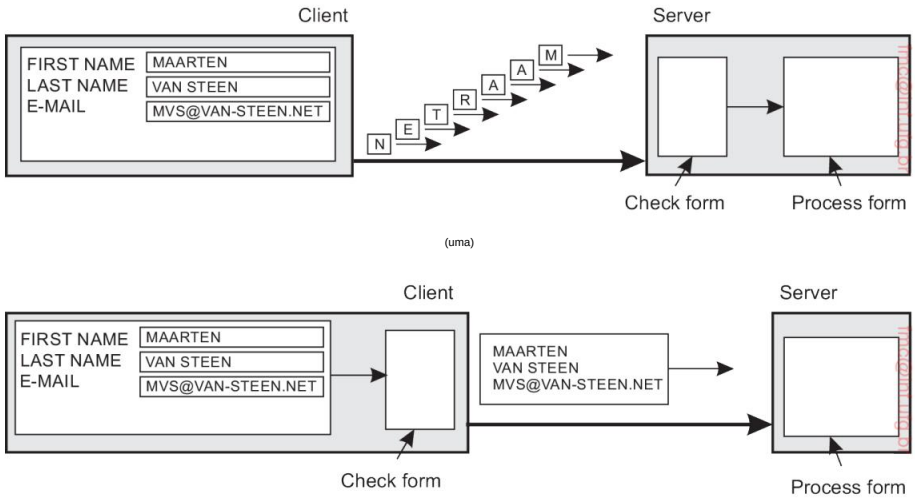

Figura 1.4: A diferença entre deixar (a) um servidor ou (b) um cliente verificar os formulários enquanto eles estão sendo preenchidos.

No entanto, existem muitos aplicativos que não podem fazer uso efetivo da comunicação assíncrona. Por exemplo, em aplicativos interativos, quando um usuário envia uma solicitação, ele geralmente não tem nada melhor para fazer do que esperar pela resposta. Nesses casos, uma solução muito melhor é reduzir a comunicação geral, por exemplo, movendo parte da computação que normalmente é feita no servidor para o processo cliente que solicita o serviço. Um caso típico em que essa abordagem funciona é acessar bancos de dados usando formulários. O preenchimento dos formulários pode ser feito enviando uma mensagem separada para cada campo e aguardando uma confirmação do servidor, conforme mostrado na Figura 1.4(a). Por exemplo, o servidor pode verificar erros sintáticos antes de aceitar uma entrada.

Uma solução muito melhor é enviar o código para preencher o formulário e possivelmente verificar as entradas para o cliente e fazer com que o cliente retorne um formulário preenchido, conforme mostrado na Figura 1.4(b). Essa abordagem de código de remessa é amplamente suportada pela Web por meio de applets Java e Javascript.

Particionamento e distribuição. Outra técnica de dimensionamento importante é **o particionamento e distribuição,** que envolve pegar um componente, dividi-lo

em partes menores e, posteriormente, espalhando essas partes pelo sistema. Um bom exemplo de particionamento e distribuição é o Internet Domain Name System (DNS). O espaço de nomes DNS é organizado hierarquicamente em uma árvore de **domínios,** que são divididos em **zonas não sobrepostas,** conforme mostrado para o DNS original na Figura 1.5. Os nomes em cada zona são tratados por um único servidor de nomes. Sem entrar em muitos detalhes agora (voltaremos ao DNS extensivamente no Capítulo 5), pode-se pensar que cada nome de caminho é o nome de um host na Internet e, portanto, está associado a um endereço de rede desse host. Basicamente, resolver um nome significa retornar o endereço de rede do host associado. Considere, por exemplo, o nome flits.cs.vu.nl. Para resolver este nome, primeiro ele é passado para o servidor da zona Z1 (veja a Figura 1.5) que retorna o endereço do servidor para a zona Z2, para o qual o restante do nome, flits.cs.vu, pode ser entregue. O servidor para Z2 retornará o endereço do servidor para a zona Z3, que é capaz de lidar com a última parte do nome e retornará o endereço do host associado.

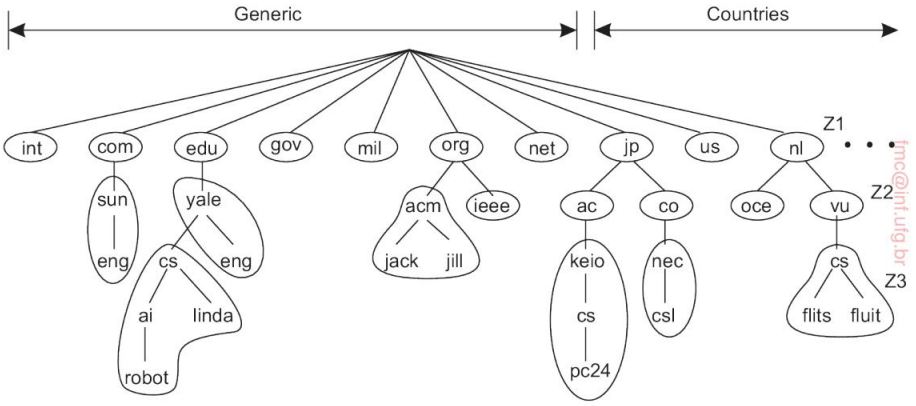

Figura 1.5: Um exemplo de divisão do espaço de nomes DNS (original) em zonas.

Este exemplo ilustra como o **serviço de nomeação** fornecido pelo DNS é distribuído em várias máquinas, evitando assim que um único servidor tenha que lidar com todas as solicitações 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 na forma de uma URL. Conceitualmente, pode até parecer que existe apenas um único servidor. No entanto, a Web é particionada fisicamente e distribuída em algumas centenas de milhões de servidores, cada um lidando com vários documentos da Web. O nome do servidor que manipula um documento é codificado na URL desse documento. É somente por causa dessa distribuição de documentos que a Web foi capaz de escalar para seu tamanho atual.

Replicação. Considerando que os problemas de escalabilidade geralmente aparecem na forma de degradação de desempenho, geralmente é uma boa ideia **replicar** componentes em um sistema distribuído. A replicação não apenas aumenta a disponibilidade, mas também ajuda a equilibrar a carga entre os componentes, levando a um melhor desempenho. Além disso, em sistemas geograficamente dispersos, ter uma cópia por perto pode ocultar muitos dos problemas de latência de comunicação mencionados anteriormente.

O **cache** é uma forma especial de replicação, embora a distinção entre os dois seja muitas vezes difícil de fazer ou mesmo artificial. Como no caso da replicação, o armazenamento em cache resulta na cópia de um recurso, geralmente na proximidade do cliente que está acessando esse recurso. No entanto, ao contrário da replicação, o cache é uma decisão tomada pelo cliente de um recurso e não pelo proprietário de um recurso.

Há uma séria desvantagem no armazenamento em cache e na replicação que pode afetar negativamente a escalabilidade. Como agora temos várias cópias de um recurso, modificar uma cópia torna essa cópia diferente das outras. Consequentemente, o armazenamento em cache e a replicação levam a problemas de **consistência .**

Até que ponto as inconsistências podem ser toleradas depende muito do uso de um recurso. Por exemplo, muitos usuários da Web consideram aceitável que seu navegador retorne um documento em cache cuja validade não foi verificada nos últimos minutos. No entanto, também existem muitos casos em que é necessário cumprir fortes garantias de consistência, como é o caso das bolsas de valores eletrónicas e dos leilões. O problema com a consistência forte é que uma atualização deve ser propagada imediatamente para todas as outras cópias. Além disso, se duas atualizações ocorrerem simultaneamente, muitas vezes também será necessário que as atualizações sejam processadas na mesma ordem em todos os lugares, introduzindo um problema de ordenação global adicional. Para agravar ainda mais os problemas, combinar consistência com outras propriedades desejáveis, como disponibilidade, pode ser simplesmente impossível, como discutimos no Capítulo 8.

A replicação, portanto, geralmente requer algum mecanismo de sincronização global . Infelizmente, tais mecanismos são extremamente difíceis ou mesmo impossíveis de implementar de forma escalável, mesmo que sozinhos, porque as latências de rede têm um limite inferior natural. Consequentemente, o dimensionamento por replicação pode introduzir outras soluções inerentemente não escaláveis. Retornamos extensivamente à replicação e consistência no Capítulo 7.

Discussão. Ao considerar essas técnicas de dimensionamento, pode-se argumentar que a escalabilidade de tamanho é a menos problemática do ponto de vista técnico. Em muitos casos, aumentar a capacidade de uma máquina salvará o dia, embora talvez haja um alto custo monetário a pagar. A escalabilidade geográfica é um problema muito mais difícil, pois as latências da rede são naturalmente limitadas por baixo.
Como consequência, podemos ser forçados a copiar dados para locais próximos de onde os clientes estão, levando a problemas de manutenção de cópias consistentes. Prática

mostra que a combinação de técnicas de distribuição, replicação e armazenamento em cache com diferentes formas de consistência geralmente leva a soluções aceitáveis. Finalmente, a escalabilidade administrativa parece ser o problema mais difícil de resolver, em parte porque precisamos lidar com questões não técnicas, como políticas de organizações e colaboração humana. A introdução e o uso generalizado da tecnologia peer-to-peer demonstrou com sucesso o que pode ser alcançado se os usuários finais forem colocados no controle [Lua et al., 2005; Oram, 2001]. No entanto, as redes peer-to-peer obviamente não são a solução universal para todos os problemas de escalabilidade administrativa.

Armadilhas

Já deve estar claro que desenvolver um sistema distribuído é uma tarefa formidável. Como veremos muitas vezes ao longo deste livro, há tantas questões a serem consideradas ao mesmo tempo que parece que apenas a complexidade pode ser o resultado. No entanto, seguindo uma série de princípios de design, sistemas distribuídos podem ser desenvolvidos que aderem fortemente aos objetivos que estabelecemos neste capítulo.

Os sistemas distribuídos diferem do software tradicional porque os componentes estão dispersos em uma rede. Não levar essa dispersão em consideração durante o tempo de projeto é o que torna tantos sistemas desnecessariamente complexos e resulta em falhas que precisam ser corrigidas posteriormente. Peter Deutsch, na época trabalhando na Sun Microsystems, formulou essas falhas como as seguintes suposições falsas que muitos fazem ao desenvolver um aplicativo distribuído pela primeira vez:

- A rede é confiável • A rede é segura
- A rede é homogênea • A topologia não muda • A latência é zero • A largura de banda é infinita • O custo de transporte é zero • Existe um administrador

Observe como essas suposições se relacionam com propriedades exclusivas de sistemas distribuídos: confiabilidade, segurança, heterogeneidade e topologia da rede; latência e largura de banda; custos de transporte; e finalmente domínios administrativos. Ao desenvolver aplicativos não distribuídos, a maioria desses problemas provavelmente não aparecerá.

A maioria dos princípios que discutimos neste livro relaciona-se imediatamente a essas suposições. Em todos os casos, discutiremos soluções para problemas causados pelo fato de uma ou mais suposições serem falsas. Por exemplo, redes confiáveis simplesmente não existem e levam à impossibilidade de obter transparência de falhas. Dedicamos um capítulo inteiro para tratar do fato de que

a comunicação em rede é inerentemente insegura. Já argumentamos que os sistemas distribuídos precisam ser abertos e levar em conta a heterogeneidade. Da mesma forma, ao discutir a replicação para resolver problemas de escalabilidade, estamos essencialmente abordando problemas de latência e largura de banda. Também abordaremos questões de gerenciamento em vários pontos ao longo deste livro.

1.3 Tipos de sistemas distribuídos

Antes de começar a discutir os princípios dos sistemas distribuídos, vamos primeiro dar uma olhada nos vários tipos de sistemas distribuídos. Fazemos uma distinção entre sistemas de computação distribuídos, sistemas de informação distribuídos e sistemas pervasivos (que são naturalmente distribuídos).

Computação distribuída de alto desempenho

Uma classe importante de sistemas distribuídos é aquela usada para tarefas de computação de alto desempenho. Grosso modo, pode-se fazer uma distinção entre dois subgrupos. Na **computação em cluster,** o hardware subjacente consiste em uma coleção de estações de trabalho ou PCs semelhantes, intimamente conectados por meio de uma rede local de alta velocidade. Além disso, cada nó executa o mesmo sistema operacional.

A situação torna-se muito diferente no caso da **computação em grade.** Esse subgrupo consiste em sistemas distribuídos que geralmente são construídos como uma federação de sistemas de computador, onde cada sistema pode estar sob um domínio administrativo diferente e pode ser muito diferente quando se trata de hardware, software e tecnologia de rede implantada.

Da perspectiva da computação em grade, o próximo passo lógico é simplesmente terceirizar toda a infraestrutura necessária para aplicativos de computação intensiva. Em essência, é disso que se trata **a computação em nuvem :** fornecer as facilidades para construir dinamicamente uma infraestrutura e compor o que é necessário a partir dos serviços disponíveis. Ao contrário da computação em grade, que está fortemente associada à computação de alto desempenho, a computação em nuvem é muito mais do que apenas fornecer muitos recursos. Discutiremos isso brevemente aqui, mas retornaremos a vários aspectos ao longo do livro.

Nota 1.8 (Mais informações: Processamento paralelo)
A computação de alto desempenho começou mais ou menos com a introdução de **máquinas multiprocessadores.** Nesse caso, várias CPUs são organizadas de forma que todas tenham acesso à mesma memória física, conforme mostrado na Figura 1.6(a). Em contraste, em um **sistema multicomputador** vários computadores estão conectados através de uma rede e não há compartilhamento de memória principal, como mostra a Figura 1.6(b). O modelo de memória compartilhada mostrou-se altamente conveniente para melhorar o desempenho dos programas e foi relativamente fácil de programar.

Sua essência é que vários threads de controle estão sendo executados ao mesmo tempo, enquanto todos os threads têm acesso a dados compartilhados. O acesso a esses dados é controlado por meio de mecanismos de sincronização bem compreendidos, como semáforos (veja Ben Ari [2006] ou Herlihy e Shavit [2008] para obter mais informações sobre o desenvolvimento de programas paralelos). Infelizmente, o modelo não escala facilmente: até agora, foram desenvolvidas máquinas nas quais apenas algumas dezenas (e às vezes centenas) de CPUs têm acesso eficiente à memória compartilhada. Até certo ponto, estamos vendo as mesmas limitações para processadores multicore.

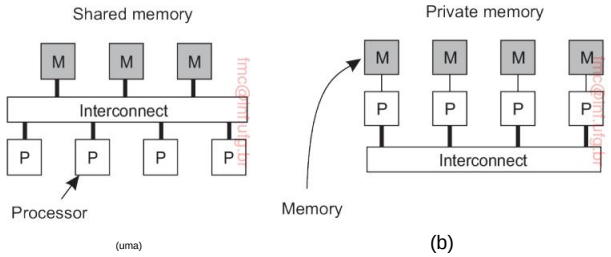

Figura 1.6: Uma comparação entre as arquiteturas (a) multiprocessador e (b) multicomputador .

Para superar as limitações dos sistemas de memória compartilhada, a computação de alto desempenho mudou para sistemas de memória distribuída. Essa mudança também significou que muitos programas tiveram que usar a passagem de mensagens em vez de modificar os dados compartilhados como meio de comunicação e sincronização entre threads. Infelizmente, os modelos de passagem de mensagens provaram ser muito mais difíceis e propensos a erros em comparação com os modelos de programação de memória compartilhada. Por esta razão, tem havido pesquisas significativas na tentativa de construir os chamados **multicomputadores de memória compartilhada distribuída,** ou simplesmente **sistema DSM** [Amza et al., 1996].

Em essência, um sistema DSM permite que um processador enderece um local de memória em outro computador como se fosse uma memória local. Isso pode ser feito usando técnicas existentes disponíveis para o sistema operacional, por exemplo, mapeando todas as páginas de memória principal dos vários processadores em um único espaço de endereço virtual. Sempre que um processador A endereça uma página localizada em outro processador B, ocorre uma falha de página em A , permitindo que o sistema operacional em A busque o conteúdo da página referenciada em B da mesma forma que normalmente faria localmente a partir do disco. Ao mesmo tempo, o processador B seria informado de que a página não está acessível no momento.

Essa ideia elegante de imitar sistemas de memória compartilhada usando multicomputadores eventualmente teve que ser abandonada pela simples razão de que o desempenho nunca poderia atender às expectativas dos programadores, que prefeririam recorrer a modelos de programação de passagem de mensagens muito mais intrincados, mas com melhor desempenho (previsivelmente) .

Um efeito colateral importante de explorar os limites hardware-software do processamento paralelo é uma compreensão completa dos modelos de consistência, aos quais retornaremos extensivamente no Capítulo 7.

Computação em cluster

Os sistemas de computação em cluster tornaram-se populares quando a relação preço/desempenho de computadores pessoais e estações de trabalho melhorou. A certa altura, tornou-se financeira e tecnicamente atraente construir um supercomputador usando tecnologia de prateleira simplesmente conectando uma coleção de computadores relativamente simples em uma rede de alta velocidade. Em praticamente todos os casos, a computação em cluster é usada para programação paralela na qual um único programa (com muita computação) é executado em paralelo em várias máquinas.

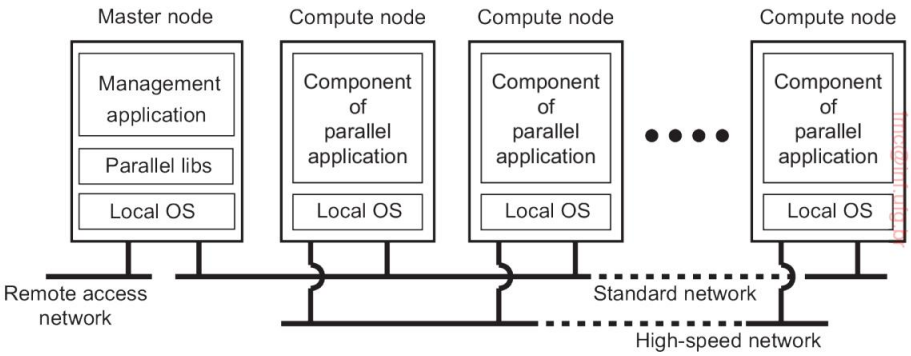

Figura 1.7: Um exemplo de um sistema de computação em cluster.

Um exemplo amplamente aplicado de um computador de cluster é formado por **clusters Beowulf baseados em Linux,** cuja configuração geral é mostrada na Figura 1.7. Cada cluster consiste em uma coleção de nós de computação que são controlados e acessados por meio de um único nó mestre. O mestre normalmente lida com a alocação de nós para um programa paralelo específico, mantém uma fila em lote de trabalhos enviados e fornece uma interface para os usuários do sistema. Como tal, o mestre realmente executa o middleware necessário para a execução de programas e gerenciamento do cluster, enquanto os nós de computação são equipados com um sistema operacional padrão estendido com funções típicas de middleware para comunicação, armazenamento, tolerância a falha Além do nó mestre, os nós de computação são vistos como altamente idênticos.

Uma abordagem ainda mais simétrica é seguida no **sistema MOSIX** [Amar et al., 2004]. O MOSIX tenta fornecer uma **imagem de sistema único** de um cluster, o que significa que, para um processo, um computador de cluster oferece a transparência de distribuição final, parecendo ser um único computador. Como mencionamos, fornecer tal imagem em todas as circunstâncias é impossível. No caso do MOSIX, o alto grau de transparência é proporcionado ao permitir que os processos migrem de forma dinâmica e preventiva entre os nós que compõem o cluster. A migração de processo permite que um usuário inicie um aplicativo em qualquer nó (referido como nó inicial), após o qual ele pode mover-se de forma transparente para outros nós, por exemplo, para fazer uso eficiente dos recursos. Nós voltaremos a

processo de migração no Capítulo 3. Abordagens semelhantes na tentativa de fornecer uma imagem de sistema único são comparadas por [Lottiaux et al., 2005].

No entanto, vários computadores de cluster modernos estão se afastando dessas arquiteturas simétricas para soluções mais híbridas nas quais o middleware é particionado funcionalmente em diferentes nós, conforme explicado por En gelmann et al. [2007]. A vantagem de tal separação é óbvia: ter nós de computação com sistemas operacionais dedicados e leves provavelmente fornecerá desempenho ideal para aplicativos de computação intensiva. Da mesma forma, a funcionalidade de armazenamento provavelmente pode ser tratada de maneira otimizada por outros nós especialmente configurados, como servidores de arquivos e diretórios. O mesmo vale para outros serviços de middleware dedicados, incluindo gerenciamento de tarefas, serviços de banco de dados e talvez acesso geral à Internet para serviços externos.

Computação em grade

Uma característica da computação em cluster tradicional é sua homogeneidade. Na maioria dos casos, os computadores em um cluster são basicamente os mesmos, têm o mesmo sistema operacional e estão todos conectados por meio da mesma rede.

No entanto, como acabamos de discutir, tem havido uma tendência para arquiteturas mais híbridas nas quais os nós são configurados especificamente para determinadas tarefas. Essa diversidade é ainda mais prevalente em **sistemas de computação em grade:** não são feitas suposições sobre semelhança de hardware, sistemas operacionais, redes, domínios administrativos, políticas de segurança etc.

Uma questão chave em um sistema de computação em grade é que recursos de diferentes organizações são reunidos para permitir a colaboração de um grupo de pessoas de diferentes instituições, formando de fato uma federação de sistemas.

Tal colaboração é realizada na forma de uma **organização virtual.** Os processos pertencentes à mesma organização virtual têm direitos de acesso aos recursos que são fornecidos a essa organização. Normalmente, os recursos consistem em servidores de computação (incluindo supercomputadores, possivelmente implementados como computadores em cluster), instalações de armazenamento e bancos de dados. Além disso, dispositivos especiais em rede, como telescópios, sensores, etc., também podem ser fornecidos.

Dada a sua natureza, grande parte do software para realizar a computação em grade envolve o fornecimento de acesso a recursos de diferentes domínios administrativos e apenas aos usuários e aplicativos que pertencem a uma organização virtual específica. Por esta razão, o foco é muitas vezes em questões de arquitetura. Uma arquitetura proposta inicialmente por Foster et al. [2001] é mostrado na Figura 1.8, que ainda constitui a base para muitos sistemas de computação em grade.

A arquitetura consiste em quatro camadas. A camada de malha mais baixa fornece interfaces para recursos locais em um site específico. Observe que essas interfaces são personalizadas para permitir o compartilhamento de recursos em uma organização virtual. Normalmente, eles fornecem funções para consultar o estado e os recursos de um recurso, juntamente com funções para gerenciamento real de recursos (por exemplo, recursos de bloqueio).

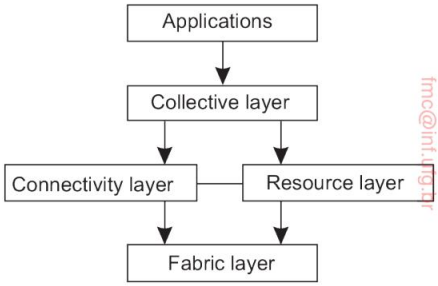

Figura 1.8: Uma arquitetura em camadas para sistemas de computação em grade.

A camada de conectividade consiste em protocolos de comunicação para suportar transações de grade que abrangem o uso de vários recursos. Por exemplo, protocolos são necessários para transferir dados entre recursos ou simplesmente para acessar um recurso de um local remoto. Além disso, a camada de conectividade conterá protocolos de segurança para autenticar usuários e recursos. Observe que, em muitos casos, os usuários humanos não são autenticados; em vez disso, os programas que atuam em nome dos usuários são autenticados. Nesse sentido, delegar direitos de um usuário a programas é uma função importante que precisa ser suportada na camada de conectividade. Voltamos à delegação ao discutir segurança em sistemas distribuídos no Capítulo 9.

A camada de recursos é responsável por gerenciar um único recurso. Ele usa as funções fornecidas pela camada de conectividade e chama diretamente as interfaces disponibilizadas pela camada de malha. Por exemplo, essa camada oferecerá funções para obter informações de configuração de um recurso específico ou, em geral, para realizar operações específicas, como criar um processo ou ler dados. A camada de recursos é, portanto, responsável pelo controle de acesso e, portanto, contará com a autenticação realizada como parte da camada de conectividade.

A próxima camada na hierarquia é a camada coletiva. Ele lida com o acesso a vários recursos e normalmente consiste em serviços para descoberta de recursos, alocação e agendamento de tarefas em vários recursos, replicação de dados e assim por diante. Ao contrário da conectividade e da camada de recursos, cada uma consistindo em uma coleção padrão relativamente pequena de protocolos, a camada coletiva pode consistir em muitos protocolos diferentes refletindo o amplo espectro de serviços que ela pode oferecer a uma organização virtual.

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

Normalmente, o coletivo, a conectividade e a camada de recursos formam o coração do que poderia ser chamado de camada de middleware de grade. Essas camadas fornecem acesso e gerenciamento de recursos que estão potencialmente dispersos em vários sites.

Uma observação importante de uma perspectiva de middleware é que em grid

calcular a noção de um site (ou unidade administrativa) é comum. Essa prevalência é enfatizada pela mudança gradual em direção a uma **arquitetura orientada a serviços** na qual os sites oferecem acesso às várias camadas por meio de uma coleção de serviços da Web [Joseph et al., 2004]. Isso, até agora, levou à definição de uma arquitetura alternativa conhecida como **Open Grid Services Architecture (OGSA)** [Foster et al., 2006]. OGSA baseia-se nas ideias originais formuladas por Foster et al. [2001], mas ter passado por um processo de padronização torna-o complexo, para dizer o mínimo. As implementações de OGSA geralmente seguem os padrões de serviço da Web.

Computação em nuvem

Enquanto os pesquisadores pensavam em como organizar grades computacionais que fossem facilmente acessíveis, as organizações encarregadas de administrar os data centers enfrentavam o problema de abrir seus recursos para os clientes. Eventualmente , isso levou ao conceito de **computação utilitária** pelo qual um cliente poderia carregar tarefas para um data center e ser cobrado por recurso. A computação utilitária formou a base para o que hoje é chamado de **computação em nuvem.**

Seguindo Vaquero et al. [2008], a computação em nuvem é caracterizada por um pool de recursos virtualizados facilmente utilizáveis e acessíveis. Quais e como os recursos são usados podem ser configurados dinamicamente, fornecendo a base para a escalabilidade: se mais trabalho precisar ser feito, um cliente pode simplesmente adquirir mais recursos. O vínculo com a computação utilitária é formado pelo fato de que a computação em nuvem geralmente é baseada em um modelo pay-per-use em que as garantias são oferecidas por meio de **acordos de nível de serviço (SLAs) customizados.**

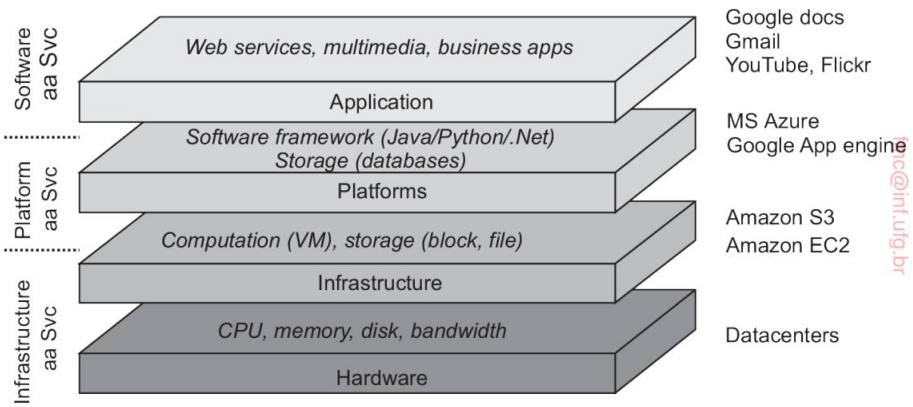

Figura 1.9: A organização das nuvens (adaptado de Zhang et al. [2010]).

Na prática, as nuvens são organizadas em quatro camadas, conforme mostrado na Figura 1.9 (veja também Zhang et al. [2010]):

Hardware: A camada mais baixa é formada pelos meios para gerenciar o hardware necessário: processadores, roteadores, mas também sistemas de energia e refrigeração. Geralmente é implementado em data centers e contém os recursos que os clientes normalmente nunca conseguem ver diretamente.

Infraestrutura: Esta é uma camada importante que forma a espinha dorsal da maioria das plataformas de computação em nuvem. Ele implementa técnicas de virtualização (discutidas na Seção 3.2) para fornecer aos clientes uma infraestrutura que consiste em armazenamento virtual e recursos de computação. Na verdade, nada é o que parece: a computação em nuvem evolui em torno da alocação e gerenciamento de dispositivos de armazenamento virtual e servidores virtuais.

Plataforma: Pode-se argumentar que a camada de plataforma fornece a um cliente de computação em nuvem o que um sistema operacional fornece aos desenvolvedores de aplicativos, ou seja, os meios para desenvolver e implantar facilmente aplicativos que precisam ser executados em uma nuvem. Na prática, um desenvolvedor de aplicativos recebe uma API específica do fornecedor, que inclui chamadas para carregar e executar um programa na nuvem desse fornecedor. De certa forma, isso é comparável à família exec de chamadas de sistema do Unix , que recebe um arquivo executável como parâmetro e o passa para o sistema operacional a ser executado.

Também como sistemas operacionais, a camada de plataforma fornece abstrações de nível superior para armazenamento e tal. Por exemplo, como discutiremos com mais detalhes posteriormente, o **sistema de armazenamento Amazon S3** [Murty, 2008] é oferecido ao desenvolvedor do aplicativo na forma de uma API que permite que arquivos (criados localmente) sejam organizados e armazenados em **buckets.** Um bucket é um pouco comparável a um diretório. Ao armazenar um arquivo em um bucket, esse arquivo é carregado automaticamente na nuvem da Amazon.

Aplicativo: Os aplicativos reais são executados nessa camada e são oferecidos aos usuários para personalização adicional. Exemplos bem conhecidos incluem aqueles encontrados em suítes de escritório (processadores de texto, aplicativos de planilhas, aplicativos de apresentação e assim por diante). É importante perceber que esses aplicativos são executados novamente na nuvem do fornecedor. Como antes, eles podem ser comparados ao conjunto tradicional de aplicativos que são enviados durante a instalação de um sistema operacional.

Os provedores de computação em nuvem oferecem essas camadas a seus clientes por meio de várias interfaces (incluindo ferramentas de linha de comando, interfaces de programação e interfaces da Web), levando a três tipos diferentes de serviços:

- **Infraestrutura como serviço (IaaS)** cobrindo o hardware e a infraestrutura camada.
- **Platform-as-a-Service (PaaS)** cobrindo a camada de plataforma. • **Software-as-a-Service (SaaS)** no qual seus aplicativos são cobertos.

A partir de agora, fazer uso de nuvens é relativamente fácil, e discutiremos em capítulos posteriores exemplos mais concretos de interfaces para provedores de nuvem. Como consequência, a computação em nuvem como meio de terceirização de infraestruturas de computação local tornou-se uma opção séria para muitas empresas. No entanto, ainda há uma série de obstáculos sérios, incluindo bloqueio de provedor, questões de segurança e privacidade e dependência da disponibilidade de serviços, para citar alguns (ver também Armbrust et al. [2010]). Além disso, como os detalhes sobre como os cálculos específicos de nuvem são realmente executados geralmente são ocultos e até talvez desconhecidos ou imprevisíveis, pode ser impossível atender às demandas de desempenho com antecedência. Além disso, Li et al. [2010] mostraram que diferentes provedores podem facilmente apresentar perfis de desempenho muito diferentes.

A computação em nuvem não é mais um hype e certamente uma alternativa séria para manter grandes infraestruturas locais, mas ainda há muito espaço para melhorias.

Nota 1.9 (Avançado: a computação em nuvem é mais barata?)

Uma das razões importantes para migrar para um ambiente de nuvem é que pode ser muito mais barato em comparação com a manutenção de uma infraestrutura de computação local. Existem muitas maneiras de calcular a economia, mas, como se vê, apenas para casos simples e óbvios os cálculos diretos fornecerão uma perspectiva realista. Hajjat et ai. [2010] propõem uma abordagem mais completa, levando em consideração que parte de uma suíte de aplicativos é migrada para uma nuvem, e a outra parte continua sendo operada em uma infraestrutura local. O cerne de seu método é fornecer o modelo certo de um conjunto de aplicativos corporativos.

O núcleo de sua abordagem é formado por um conjunto potencialmente grande de componentes de software. Supõe-se que cada aplicativo corporativo consiste em componentes. Além disso, cada componente C_i é considerado executado em servidores N_i. Um exemplo simples é um componente de banco de dados a ser executado por um único servidor. Um exemplo mais elaborado é um aplicativo da Web para calcular rotas de bicicleta, consistindo em um front-end de servidor Web para renderizar páginas HTML e aceitar entrada do usuário, um componente para calcular caminhos mais curtos (talvez sob diferentes restrições) e um componente de banco de dados contendo vários mapas.

Cada aplicação é modelada como um grafo direcionado, no qual um vértice representa um componente e um arco h ÿÿi, ji o fato de que os dados fluem do componente C_i para o componente C_j. Cada arco tem dois pesos associados: $T_{i,j}$ representa o número de transações por unidade de tempo levando os dados a fluir de C_i para C_j e $S_{i,j}$ o tamanho médio dessas transações (ou seja, a quantidade média de dados por transação). Eles assumem que $T_{i,j}$ e $S_{i,j}$ são conhecidos, tipicamente obtidos através de Medidas.

A migração de um conjunto de aplicativos de uma infraestrutura local para a nuvem se resume a encontrar um plano de migração ideal M: descobrir para cada um quanto n_i de seus servidores N_i devem ser movidos para a nuvem, tal como para o componente C_i, que os benefícios monetários resultantes de M, reduzidos pelos custos adicionais de comunicação pela Internet, são máximos. Um plano M também deve atender às seguintes restrições:

1. As restrições da política são atendidas. Por exemplo, pode haver dados cuja localização seja legalmente exigida na infraestrutura local de uma organização.
2. Como a comunicação agora é feita parcialmente através de links de Internet de longa distância, pode ser que certas transações entre componentes se tornem muito mais lentas. Um plano M é aceitável apenas se quaisquer latências adicionais não violarem restrições de atraso específicas.
3. As equações de equilíbrio de fluxo devem ser respeitadas: as transações continuam operando corretamente e as solicitações ou dados não são perdidos durante uma transação.

Vejamos agora os benefícios e os custos da Internet de um plano de migração.

Benefícios Para cada plano de migração M, pode-se esperar uma economia monetária expressa como Benefits(M), porque menos máquinas ou conexões de rede precisam ser mantidas. Em muitas organizações, esses custos são conhecidos de modo que pode ser relativamente simples calcular a economia. Por outro lado, há custos a serem feitos para usar a nuvem. Hajjat et ai. [2010] fazem uma distinção simplificada entre o benefício Bc de migrar um componente de computação intensiva e o benefício Bs de migrar um componente de armazenamento intensivo. Se houver componentes com uso intensivo de computação e Ms com uso intensivo de armazenamento, teremos Benefits(M) = Bc · Mc + Bs · Ms. Obviamente, modelos muito mais sofisticados também podem ser implantados

Custos da Internet Para calcular o aumento dos custos de comunicação porque os componentes estão espalhados pela nuvem, bem como pela infraestrutura local, precisamos levar em consideração as solicitações iniciadas pelo usuário. Para simplificar as coisas, não fazemos distinção entre usuários internos (ou seja, membros da empresa) e usuários externos (como veríamos no caso de aplicativos da Web). O tráfego de usuários antes da migração pode ser expresso como:

$$T_{\text{local,inet}} = \ddot{y}_{\text{Lá}} (T_{\text{user},i} \text{User},i + \text{Você},_{\text{user}} S_{i,\text{user}})$$

onde Tuser,i denota o número de transações por unidade de tempo levando os dados a fluir dos usuários para Ci . Temos interpretações análogas para Ti,user, Suseri e Si,user.

Para cada componente Ci , seja Ci,local os servidores que continuam operando na infraestrutura local, e Ci,cloud seus servidores que são colocados na nuvem. Observe que |Ci,nuvem| = n . Para simplificar, suponha que um servidor da Ci,local distribua o tráfego nas mesmas proporções que um servidor da Ci,cloud. Estamos interessados na taxa de transações entre servidores locais, servidores em nuvem e entre servidores locais e em nuvem, após a migração. Seja sk o servidor do componente Ck e denote por fk a fração nk/Nk . Temos então para a taxa de transações T após a migração i,j : ˙

$$T_{i,j}^{\ddot{y}} = \begin{cases} \ddot{y} (1 \ddot{y} f_i) \cdot (1 \ddot{y} f_j) \cdot T_{i,j} & \text{quando } s_i \ddot{y} C_{i,\text{local}} \text{ e } s_j \ddot{y} C_{j,\text{local}} \\ (1 \ddot{y} f_i) \cdot f_j \cdot T_{i,j} & \text{quando } s_i \ddot{y} C_{i,\text{local}} \text{ e } s_j \ddot{y} C_{j,\text{nuvem}} \\ f_i \cdot (1 \ddot{y} f_j) \cdot T_{i,j} & \text{quando } s_i \ddot{y} C_{i,\text{nuvem}} \text{ e } s_j \ddot{y} C_{j,\text{local}} \\ f_i \cdot f_j \cdot T_{i,j} & \text{quando } s_i \ddot{y} C_{i,\text{nuvem}} \text{ e } s_j \ddot{y} C_{j,\text{nuvem}} \end{cases}$$

S ý é a quantidade de dados associados a T i,j ý eu, j . Observe que fk denota a fração de servidores do componente Ck que são movidos para a nuvem. Em outras palavras, (1 ÿ fk) é a fração que fica na infraestrutura local. Deixamos para o leitor dar uma expressão para T i,user.

ý

Finalmente, deixe costlocal,inet e costcloud,inet denotar os custos de Internet por unidade para tráfego de e para a infraestrutura local e nuvem, respectivamente. Ignorando algumas sutilezas explicadas em [Hajjat et al., 2010], podemos calcular o tráfego local da Internet após a migração como:

$$Tr\ddot{y}_{local,inet} = \ddot{y}_{Ci,local\ ,Cj,local} (T^{\ddot{y}}_{Si\ ,ji,jj,ij,i} S^{\ddot{y}}_{+T} {}^{\ddot{y}} {}^{\ddot{y}}) + \ddot{y}_{Cj,\ local} (T^{\ddot{y}}_{usuário,j} S^{\ddot{y}}_{usuário,j} + T \ {}^{\ddot{y}}_{j,usuários} {}^{\ddot{y}}_{j,usuário})$$

e, da mesma forma, para o tráfego de Internet na nuvem após a migração:

$$Tr\ddot{y}_{nuvem,inet} = \ddot{y}_{Ci,nuvem,Cj,nuvem} (T^{\ddot{y}}_{i,ji,jj,ij,i} S^{\ddot{y}}_{+T} {}^{\ddot{y}} S^{\ddot{y}}) + \ddot{y}_{Cj,\ nuvem} (T^{\ddot{y}}_{usuário,j} S^{\ddot{y}}_{usuário,j} + T \ {}^{\ddot{y}}_{j,usuárioSj} {}^{\ddot{y}}_{,usuário})$$

Juntos, isso leva a um modelo para o aumento dos custos de comunicação na Internet:

costlocal,inet(Trÿ local,inet ÿ Trlocal,inet) + costcloud,inetTrÿ cloud,inet

Claramente, responder à pergunta se migrar para a nuvem é mais barato requer muitas informações detalhadas e um planejamento cuidadoso sobre exatamente o que migrar. Hajjat et ai. [2010] fornecem um primeiro passo para tomar uma decisão informada. O modelo deles é mais detalhado do que estamos dispostos a explicar aqui. Um aspecto importante que não abordamos é que a migração de componentes também significa que uma atenção especial deverá ser dada à migração de componentes de segurança. O leitor interessado deve consultar seu artigo.

Sistemas de informação distribuídos

Outra classe importante de sistemas distribuídos é encontrada em organizações que foram confrontadas com uma variedade de aplicativos em rede, mas para as quais a interoperabilidade acabou sendo uma experiência dolorosa. Muitas das soluções de middleware existentes são o resultado de trabalhar com uma infraestrutura na qual era mais fácil integrar aplicativos em um **sistema de informação corporativo** [Alonso et al., 2004; Bernstein, 1996; Hohpe e Woolf, 2004].

Podemos distinguir vários níveis em que a integração pode ocorrer. Em muitos casos, um aplicativo em rede consiste simplesmente em um servidor que executa esse aplicativo (geralmente incluindo um banco de dados) e o disponibiliza para programas remotos, chamados **clientes.** Esses clientes enviam uma solicitação ao servidor para executar uma operação específica, após a qual uma resposta é enviada de volta. A integração no nível mais baixo permite que os clientes envolvam várias solicitações, possivelmente para servidores diferentes, em uma única solicitação maior e a executem como uma **transação distribuída.** A ideia chave é que todas ou nenhuma das requisições sejam executadas.

À medida que os aplicativos se tornaram mais sofisticados e foram gradualmente separados

em componentes independentes (principalmente distinguindo componentes de banco de dados de componentes de processamento), ficou claro que a integração também deveria ocorrer permitindo que os aplicativos se comunicassem diretamente uns com os outros. Isso agora levou a uma enorme indústria que se concentra na **integração de aplicativos corporativos (EAI).**

Processamento de transações distribuídas

Para esclarecer nossa discussão, nos concentramos em aplicativos de banco de dados. Na prática, as operações em um banco de dados são realizadas na forma de **transações.** A programação usando transações requer primitivas especiais que devem ser fornecidas pelo sistema distribuído subjacente ou pelo sistema de tempo de execução da linguagem. Exemplos típicos de primitivas de transação são mostrados na Figura 1.10. A lista exata de primitivas depende de quais tipos de objetos estão sendo usados na transação [Gray e Reuter, 1993; Bernstein e Newcomer, 2009]. Em um sistema de correio, pode haver primitivas para enviar, receber e encaminhar correio. Em um sistema de contabilidade, eles podem ser bem diferentes. READ e WRITE são exemplos típicos, no entanto. Declarações comuns, chamadas de procedimento e assim por diante também são permitidas dentro de uma transação. Em particular, **chamadas de procedimento remoto (RPC),** ou seja, chamadas de procedimento para servidores remotos, muitas vezes também são encapsuladas em uma transação, levando ao que é conhecido como **RPC transacional.** Discutimos RPCs extensivamente na Seção 4.2.

Primitivo	Descrição
BEGIN_TRANSACTION	Marcar o início de uma transação
END_TRANSACTION	Encerre a transação e tente confirmar
ABORT_TRANSACTION	Mate a transação e restaure os valores antigos
LER	Ler dados de um arquivo, uma tabela ou de outra forma
ESCREVA	Gravar dados em um arquivo, uma tabela ou de outra forma

Figura 1.10: Exemplo de primitivas para transações.

BEGIN_TRANSACTION e END_TRANSACTION são usados para delimitar o escopo de uma transação. As operações entre eles formam o corpo da transação. A característica de uma transação é que todas essas operações são executadas ou nenhuma é executada. Podem ser chamadas de sistema, procedimentos de biblioteca ou instruções de colchetes em uma linguagem, dependendo da implementação.

Essa propriedade de tudo ou nada das transações é uma das quatro propriedades características que as transações possuem. Mais especificamente, as transações aderem às chamadas propriedades **ACID :**

- **Atômica:** Para o mundo exterior, a transação acontece de forma indivisível •
- **Consistente:** A transação não viola invariantes do sistema

• **Isolado:** as transações simultâneas não interferem umas nas outras • **Durável:** uma vez que uma transação é confirmada, as alterações são permanentes

Em sistemas distribuídos, as transações são frequentemente construídas como um número de subtransações, formando conjuntamente uma **transação aninhada,** conforme mostrado na Figura 1.11. A transação de nível superior pode separar os filhos que são executados em paralelo uns com os outros, em máquinas diferentes, para ganhar desempenho ou simplificar a programação. Cada um desses filhos também pode executar uma ou mais subtransações ou bifurcar seus próprios filhos.

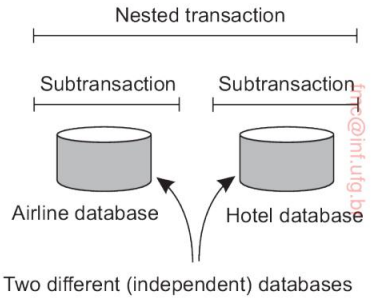

Figura 1.11: Uma transação aninhada.

As subtransações dão origem a um problema sutil, mas importante. Imagine que uma transação inicia várias subtransações em paralelo, e uma dessas confirmações, tornando seus resultados visíveis para a transação pai. Após mais computação, o pai aborta, restaurando todo o sistema ao estado que tinha antes do início da transação de nível superior. Consequentemente, os resultados da subtransação que cometeu devem, no entanto, ser desfeitos. Assim, a permanência acima referida aplica-se apenas às transações de nível superior.

Como as transações podem ser aninhadas arbitrariamente em profundidade, é necessária uma administração considerável para fazer tudo certo. A semântica é clara, no entanto. Quando qualquer transação ou subtransação é iniciada, ela recebe conceitualmente uma cópia privada de todos os dados em todo o sistema para manipular como desejar. Se abortar, seu universo privado simplesmente desaparece, como se nunca tivesse existido. Se ele confirmar, seu universo privado substituirá o universo do pai. Assim, se uma subtransação for confirmada e, posteriormente, uma nova subtransação for iniciada, a segunda verá os resultados produzidos pela primeira. Da mesma forma, se uma transação envolvente (nível superior) for abortada, todas as suas subtransações subjacentes também deverão ser abortadas. E se várias transações forem iniciadas simultaneamente, o resultado será como se elas fossem executadas sequencialmente em alguma ordem não especificada.

As transações aninhadas são importantes em sistemas distribuídos, pois fornecem uma maneira natural de distribuir uma transação em várias máquinas. Eles seguem uma divisão lógica do trabalho da transação original. Por exemplo, uma transação para planejar uma viagem em que três voos diferentes precisam ser

reservado pode ser logicamente dividido em três subtransações. Cada uma dessas subtransações pode ser gerenciada separadamente e independentemente das outras duas.

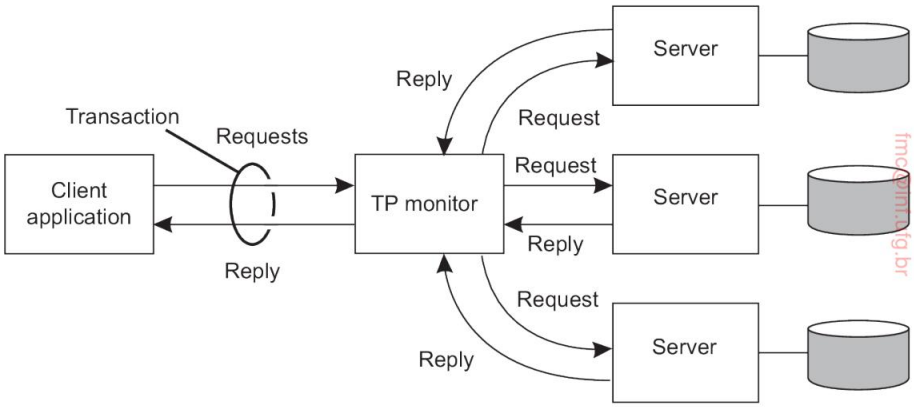

Figura 1.12: O papel de um monitor TP em sistemas distribuídos.

Nos primeiros dias dos sistemas de middleware corporativos, o componente que manipulava transações distribuídas (ou aninhadas) formava o núcleo para integração de aplicativos no nível do servidor ou do banco de dados. Esse componente foi chamado **de monitor de processamento de transações** ou monitor **TP** para abreviar. Sua principal tarefa era permitir que uma aplicação acessasse vários servidores/bancos de dados, oferecendo a ela um modelo de programação transacional, conforme mostrado na Figura 1.12. Essencialmente, o monitor de TP coordenou o comprometimento de subtransações seguindo um protocolo padrão conhecido como **commit distribuído,** que discutimos na Seção 8.5.

Uma observação importante é que os aplicativos que desejam coordenar várias subtransações em uma única transação não precisam implementar essa coordenação por conta própria. Com o simples uso de um monitor TP, essa coordenação foi feita para eles. É exatamente aí que o middleware entra em ação: ele implementa serviços que são úteis para muitos aplicativos, evitando que esses serviços tenham que ser reimplementados repetidamente pelos desenvolvedores de aplicativos.

Integração de aplicativos corporativos

Conforme mencionado, quanto mais aplicativos se desacoplam dos bancos de dados sobre os quais foram construídos, mais evidente fica a necessidade de recursos para integrar aplicativos independentemente de seus bancos de dados. Em particular, os componentes do aplicativo devem ser capazes de se comunicar diretamente entre si e não apenas por meio do comportamento de solicitação/resposta que era suportado pelos sistemas de processamento de transações.

Essa necessidade de comunicação entre aplicativos levou a muitos modelos de comunicação diferentes. A ideia principal era que os aplicativos existentes pudessem trocar informações, conforme mostrado na Figura 1.13.

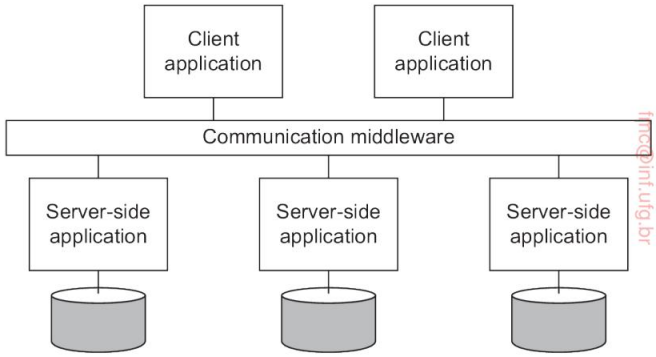

Figura 1.13: Middleware como facilitador de comunicação na integração de aplicativos corporativos.

Existem vários tipos de middleware de comunicação. Com **procedimento remoto chamadas (RPC),** um componente de aplicação pode efetivamente enviar uma solicitação para outro componente do aplicativo fazendo uma chamada de procedimento local, o que resulta na solicitação sendo empacotada como uma mensagem e enviada ao receptor. Da mesma forma, o resultado será enviado de volta e devolvido ao aplicativo como resultado da chamada de procedimento.

À medida que a popularidade da tecnologia de objetos aumentou, técnicas foram desenvolvidas para permitir chamadas para objetos remotos, levando ao que é conhecido como **controle remoto . invocações de método (RMI).** Um RMI é essencialmente o mesmo que um RPC, exceto que opera em objetos em vez de funções.

RPC e RMI têm a desvantagem de que o chamador e o receptor precisam estar funcionando no momento da comunicação. Além disso, eles precisam saber exatamente como se referir uns aos outros. Esse acoplamento estreito é muitas vezes visto como uma séria desvantagem e levou ao que é conhecido como **middleware orientado a mensagens,** ou simplesmente **MOM.** Nesse caso, os aplicativos enviam mensagens para pontos de contato lógicos, muitas vezes descritos por meio de um assunto. Da mesma maneira, aplicativos podem indicar seu interesse por um tipo específico de mensagem, após qual o middleware de comunicação cuidará para que essas mensagens sejam entregues a essas aplicações. Esses chamados sistemas **de publicação-assinatura** formam uma classe importante e em expansão de sistemas distribuídos.

> **Nota 1.10** (Mais informações: Sobre a integração de aplicativos)
> Apoiar a integração de aplicativos corporativos é uma meta importante para muitas empresas de médio porte.

produtos de dleware. Em geral, existem quatro maneiras de integrar aplicativos [Hohpe e Woolf, 2004]:

Transferência de arquivos: A essência da integração por meio da transferência de arquivos é que um aplicativo produz um arquivo contendo dados compartilhados que são posteriormente lidos por outros aplicativos. A abordagem é tecnicamente muito simples, tornando-a atraente. A desvantagem, no entanto, é que há muitas coisas que precisam ser acordadas:

- Formato e layout do arquivo: texto, binário, sua estrutura e assim por diante. Atualmente, o XML tornou-se popular, pois seus arquivos são, em princípio, autodescritivos.
- Gerenciamento de arquivos: onde são armazenados, como são nomeados, quem é responsável pela exclusão dos arquivos?
- Propagação de atualização: Quando um aplicativo produz um arquivo, pode haver vários aplicativos que precisam ler esse arquivo para fornecer a visão de um único sistema coerente, como argumentamos na Seção 1.1. Como consequência, às vezes, programas separados precisam ser implementados para notificar aplicativos sobre atualizações de arquivos.

Banco de **dados compartilhado:** Muitos dos problemas associados à integração por meio de arquivos são aliviados ao usar um banco de dados compartilhado. Todos os aplicativos terão acesso aos mesmos dados e, muitas vezes, por meio de uma linguagem de alto nível, como SQL. Além disso, é fácil notificar os aplicativos quando ocorrem alterações, pois os gatilhos geralmente fazem parte dos bancos de dados modernos. Existem, no entanto, dois grandes inconvenientes. Primeiro, ainda há a necessidade de projetar um esquema de dados comum, o que pode estar longe de ser trivial se o conjunto de aplicativos que precisam ser integrados não for totalmente conhecido antecipadamente. Segundo, quando há muitas leituras e atualizações, um banco de dados compartilhado pode facilmente se tornar um gargalo de desempenho.

Chamada de procedimento remoto: A integração por meio de arquivos ou um banco de dados pressupõe implicitamente que as alterações de um aplicativo podem facilmente acionar outros aplicativos para agir. No entanto, a prática mostra que, às vezes, pequenas alterações devem realmente fazer com que muitos aplicativos executem ações. Nesses casos, não é realmente a mudança de dados que importa, mas a execução de uma série de ações.

Séries de ações são melhor capturadas através da execução de um procedimento (que pode, por sua vez, levar a todos os tipos de mudanças nos dados compartilhados). Para evitar que cada aplicativo precise conhecer todas as partes internas dessas ações (como implementadas por outro aplicativo), as técnicas de encapsulamento padrão devem ser usadas, como implantadas com chamadas de procedimento tradicionais ou invocações de objeto. Para tais situações, um aplicativo pode oferecer melhor um procedimento para outros aplicativos na forma de uma chamada de procedimento remoto ou RPC. Em essência, um RPC permite que um aplicativo A faça uso das informações disponíveis apenas para o aplicativo B, sem dar a A acesso direto a essas informações. Há muitas vantagens e desvantagens nas chamadas de procedimento remoto, que são discutidas em profundidade no Capítulo 4.

Mensagens: A principal desvantagem dos RPCs é que o chamador e o receptor precisam estar funcionando ao mesmo tempo para que a chamada seja bem-sucedida. No entanto, em

Em muitos cenários, esta atividade simultânea é muitas vezes difícil ou impossível de garantir. Nesses casos, é necessário oferecer um sistema de mensagens carregando solicitações do aplicativo A para realizar uma ação no aplicativo B. O sistema de mensagens garante que, eventualmente, a solicitação seja entregue e, se necessário, que uma resposta seja retornada também. Obviamente, o envio de mensagens não é a panaceia para a integração de aplicativos: também apresenta problemas de formatação e layout de dados, exige que um aplicativo saiba para onde enviar uma mensagem, precisa haver cenários para lidar com mensagens perdidas e assim por diante. Assim como os RPCs, discutiremos essas questões extensivamente no Capítulo 4.

O que essas quatro abordagens nos dizem é que a integração de aplicativos geralmente não será simples. O middleware (na forma de um sistema distribuído), no entanto, pode ajudar significativamente na integração, fornecendo os recursos certos, como suporte para RPCs ou mensagens. Como dito, a integração de aplicativos corporativos é um campo de destino importante para muitos produtos de middleware.

Sistemas pervasivos

Os sistemas distribuídos discutidos até agora são amplamente caracterizados por sua estabilidade: os nós são fixos e têm uma conexão mais ou menos permanente e de alta qualidade a uma rede. Até certo ponto, essa estabilidade é alcançada por meio de várias técnicas para alcançar a transparência da distribuição. Por exemplo, há muitas maneiras de criar a ilusão de que apenas ocasionalmente os componentes podem falhar. Da mesma forma, existem todos os tipos de meios para ocultar a localização real da rede de um nó, permitindo efetivamente que usuários e aplicativos acreditem que os nós permanecem no lugar.

No entanto, as coisas mudaram desde a introdução de dispositivos de computação móvel e incorporados, levando ao que geralmente são chamados de **sistemas pervasivos.** Como o próprio nome sugere, os sistemas pervasivos destinam-se a se misturar naturalmente ao nosso ambiente. Eles também são naturalmente sistemas distribuídos e certamente atendem à caracterização que demos na Seção 1.1.

O que os torna únicos em comparação com os sistemas de computação e informação descritos até agora é que a separação entre usuários e componentes do sistema é muito mais indistinta. Muitas vezes não há uma única interface dedicada, como uma combinação de tela/teclado. Em vez disso, um sistema abrangente é frequentemente equipado com muitos **sensores** que captam vários aspectos do comportamento de um usuário. Da mesma forma, pode ter uma infinidade de **atuadores** para fornecer informações e feedback, muitas vezes até com o objetivo de orientar o comportamento.

Muitos dispositivos em sistemas pervasivos são caracterizados por serem pequenos, movidos a bateria, móveis e possuir apenas uma conexão sem fio, embora nem todas essas características se apliquem a todos os dispositivos. Essas características não são necessariamente restritivas, como ilustram os smartphones [Roussos et al., 2005] e seu papel no que hoje é denominado **Internet das Coisas** [Mattern and

Floerkemeier, 2010; Stankovic, 2014]. No entanto, notadamente o fato de que muitas vezes precisamos lidar com os meandros da comunicação sem fio e móvel, exigirá soluções especiais para tornar um sistema abrangente o mais transparente ou discreto possível.

A seguir, fazemos uma distinção entre três tipos diferentes de sistemas pervasivos, embora haja uma sobreposição considerável entre os três tipos: sistemas de computação ubíquos, sistemas móveis e redes de sensores. Essa distinção nos permite focar em diferentes aspectos dos sistemas pervasivos.

Sistemas de computação ubíquos

Até agora, falamos sobre sistemas pervasivos para enfatizar que seus elementos se espalharam por muitas partes do nosso ambiente. Em um sistema de computação onipresente, vamos um passo além: o sistema é penetrante e continuamente presente. O último significa que um usuário estará continuamente interagindo com o sistema, muitas vezes nem mesmo sabendo que a interação está ocorrendo. Poslad [2009] descreve os principais requisitos para um **sistema de computação ubíquo** aproximadamente da seguinte forma:

1. **(Distribuição) Os** dispositivos são conectados em rede, distribuídos e acessíveis de maneira transparente 2. **(Interação)** A interação entre usuários e dispositivos é altamente discreta
   seguro
3. **(Consciência do contexto)** O sistema está ciente do contexto de um usuário para otimizar a interação 4. **(Autonomia)** Os dispositivos operam de forma autônoma sem intervenção humana
   e são, portanto, altamente autogerenciados
5. **(Inteligência)** O sistema como um todo pode lidar com uma ampla gama de ações e interações dinâmicas

Vamos considerar brevemente esses requisitos de uma perspectiva de sistemas distribuídos .

De Anúncios. 1: Distribuição. Como mencionado, um sistema de computação ubíquo é um exemplo de sistema distribuído: os dispositivos e outros computadores que formam os nós de um sistema são simplesmente conectados em rede e trabalham juntos para formar a ilusão de um único sistema coerente. A distribuição também vem naturalmente: haverá dispositivos próximos aos usuários (como sensores e atuadores), conectados a computadores ocultos e talvez até operando remotamente em uma nuvem. A maioria, se não todos, os requisitos relativos à transparência de distribuição mencionados na Seção 1.2 devem, portanto, ser válidos.

De Anúncios. 2: Interação. Quando se trata de interação com os usuários, os sistemas de computação ubíqua diferem muito em comparação com os sistemas que temos

discutindo até agora. Os usuários finais desempenham um papel proeminente no projeto de sistemas ubíquos, o que significa que atenção especial precisa ser dada à forma como ocorre a interação entre os usuários e o sistema central. Para sistemas de computação ubíquos , grande parte da interação por humanos será implícita, com uma **ação implícita** sendo definida como “que não visa principalmente interagir com um sistema computadorizado, mas que tal sistema entende como entrada” [Schmidt, 2000] . Em outras palavras, um usuário pode não estar ciente do fato de que a entrada está sendo fornecida a um sistema de computador. De uma certa perspectiva, pode-se dizer que a computação ubíqua aparentemente esconde interfaces.

Um exemplo simples é quando as configurações do banco do motorista de um carro, volante e espelhos são totalmente personalizadas. Se Bob se sentar, o sistema reconhecerá que está lidando com Bob e, posteriormente, fará os ajustes apropriados. O mesmo acontece quando Alice usa o carro, enquanto um usuário desconhecido será orientado a fazer seus próprios ajustes (a serem lembrados para mais tarde). Este exemplo já ilustra um importante papel dos sensores na computação ubíqua, nomeadamente como dispositivos de entrada que são usados para identificar uma situação (uma pessoa específica aparentemente querendo dirigir), cuja análise de entrada leva a ações (fazer ajustes). Por sua vez, as ações podem levar a reações naturais, por exemplo, Bob altera ligeiramente as configurações do assento. O sistema terá que levar em consideração todas as ações (implícitas e explícitas) do usuário e reagir de acordo.

De Anúncios. 3: Consciência do contexto. Reagir à entrada sensorial, mas também à entrada explícita dos usuários, é mais fácil de dizer do que de fazer. O que um sistema de computação ubíquo precisa fazer é levar em conta o contexto no qual as interações ocorrem. A consciência de contexto também diferencia os sistemas de computação ubíquos dos sistemas mais tradicionais que discutimos anteriormente, e é descrita por Dey e Abowd [2000] como “qualquer informação que possa ser usada para caracterizar a situação de entidades (ou seja, se uma pessoa, lugar ou ou objeto) que são considerados relevantes para a interação entre um usuário e um aplicativo, incluindo o próprio usuário e o aplicativo”. Na prática, o contexto é muitas vezes caracterizado por localização, identidade, tempo e atividade: onde, quem, quando e o quê. Um sistema precisará ter a entrada (sensorial) necessária para determinar um ou vários desses tipos de contexto.

O que é importante do ponto de vista de sistemas distribuídos é que os dados brutos coletados por vários sensores sejam elevados a um nível de abstração que possa ser usado por aplicativos. Um exemplo concreto é detectar onde uma pessoa está, por exemplo, em termos de coordenadas de GPS e, posteriormente, mapear essa informação para um local real, como a esquina de uma rua ou uma loja específica ou outra instalação conhecida. A questão é onde ocorre esse processamento de entrada sensorial : todos os dados coletados em um servidor central conectado a um banco de dados com informações detalhadas sobre uma cidade, ou é o smartphone do usuário onde o mapeamento é feito? Claramente, há trocas a serem consideradas.

Dey [2010] discute abordagens mais gerais para a construção de aplicativos sensíveis ao contexto. Quando se trata de combinar flexibilidade e distribuição de potencial, os chamados **espaços de dados compartilhados** nos quais os processos são desacoplados no tempo e no espaço são atraentes, mas, como veremos em capítulos posteriores, sofrem de problemas de escalabilidade. Uma pesquisa sobre consciência de contexto e sua relação com middleware e sistemas distribuídos é fornecida por Baldauf et al

De Anúncios. 4: Autonomia. Um aspecto importante da maioria dos sistemas de computação onipresentes é que o gerenciamento de sistemas explícitos foi reduzido ao mínimo. Em um ambiente de computação onipresente, simplesmente não há espaço para um administrador de sistemas manter tudo funcionando. Como consequência, o sistema como um todo deve ser capaz de agir de forma autônoma e reagir automaticamente às mudanças. Isso requer uma miríade de técnicas, das quais várias serão discutidas ao longo deste livro. Para dar alguns exemplos simples, pense no seguinte:

Alocação de endereço: Para que os dispositivos em rede se comuniquem, eles precisam de um endereço IP. Os endereços podem ser alocados automaticamente usando protocolos como o Dynamic Host Configuration Protocol (DHCP) [Droms, 1997] (que requer um servidor) ou Zeroconf [Guttman, 2001].

Adicionando dispositivos: Deve ser fácil adicionar dispositivos a um sistema existente. Um passo em direção à configuração automática é realizado pelo **protocolo Universal Plug and Play (UPnP)** [UPnP Forum, 2008]. Usando UPnP, os dispositivos podem descobrir uns aos outros e certificar-se de que podem configurar canais de comunicação entre eles.

Atualizações automáticas: muitos dispositivos em um sistema de computação onipresente devem poder verificar regularmente pela Internet se seu software deve ser atualizado. Nesse caso, eles podem baixar novas versões de seus componentes e, idealmente, continuar de onde pararam.

É certo que estes são exemplos muito simples, mas a imagem deve ser clara de que a intervenção manual deve ser reduzida ao mínimo. Discutiremos muitas técnicas relacionadas ao autogerenciamento em detalhes ao longo do livro.

De Anúncios. 5: Inteligência. Por fim, Poslad [2009] menciona que os sistemas de computação ubíqua costumam usar métodos e técnicas do campo da inteligência artificial. O que isso significa é que, em muitos casos, uma ampla variedade de algoritmos e modelos avançados precisa ser implantado para lidar com entradas incompletas, reagir rapidamente a um ambiente em mudança, lidar com eventos inesperados e assim por diante. A extensão em que isso pode ou deve ser feito de forma distribuída é crucial do ponto de vista dos sistemas distribuídos. Infelizmente, soluções distribuídas para muitos problemas no campo da inteligência artificial ainda não foram encontradas, o que significa que pode haver uma tensão na

o primeiro requisito de dispositivos em rede e distribuídos e processamento avançado de informações distribuídas.

Sistemas de computação móvel

Como mencionado, a mobilidade geralmente forma um componente importante de sistemas pervasivos, e muitos, senão todos os aspectos que acabamos de discutir também se aplicam à **computação móvel.** Existem várias questões que colocam a computação móvel de lado para sistemas pervasivos em geral (ver também Adelstein et al. [2005] e Tarkoma e Kangasharju [2009]).

Primeiro, os dispositivos que fazem parte de um sistema móvel (distribuído) podem variar muito. Normalmente, a computação móvel agora é feita com dispositivos como smartphones e tablets. No entanto, tipos completamente diferentes de dispositivos estão agora usando o Internet Protocol (IP) para se comunicar, colocando a computação móvel em uma perspectiva diferente. Esses dispositivos incluem controles remotos , pagers, crachás ativos, equipamentos automotivos, vários dispositivos habilitados para GPS e assim por diante. Uma característica de todos esses dispositivos é que eles usam comunicação sem fio. Móvel implica sem fio assim parece (embora haja exceções às regras).

Em segundo lugar, na computação móvel, supõe-se que a localização de um dispositivo mude ao longo do tempo. A mudança de localização tem seus efeitos em muitas questões. Por exemplo, se a localização de um dispositivo muda regularmente, talvez o mesmo aconteça com os serviços que estão disponíveis localmente. Como consequência, podemos precisar prestar atenção especial à descoberta dinâmica de serviços, mas também permitir que os serviços anunciem sua presença. Na mesma linha, muitas vezes também queremos saber onde um dispositivo realmente está. Isso pode significar que precisamos conhecer as coordenadas geográficas reais de um dispositivo, como em aplicativos de rastreamento e rastreamento, mas também pode exigir que possamos simplesmente detectar sua posição na rede (como em IP móvel [Perkins, 2010; Perkins et al., 2011].

Mudar de local também tem um efeito profundo na comunicação. Para ilustrar, considere uma rede **ad hoc móvel (sem fio),** geralmente abreviada como **MANET.** Suponha que dois dispositivos em uma MANET tenham se descoberto no sentido de que conhecem o endereço de rede um do outro. Como roteamos mensagens entre os dois? As rotas estáticas geralmente não são sustentáveis , pois os nós ao longo do caminho de roteamento podem facilmente sair do alcance do vizinho, invalidando o caminho. Para MANETs grandes, usar caminhos de configuração a priori não é uma opção viável. Estamos lidando aqui com as chamadas **redes tolerantes à interrupção:** redes nas quais a conectividade entre dois nós simplesmente não pode ser garantida. Obter uma mensagem de um nó para outro pode ser problemático, para dizer o mínimo.

O truque nesses casos não é tentar configurar um caminho de comunicação da origem ao destino, mas confiar em dois princípios. Primeiro, como discutiremos na Seção 4.4, o uso de técnicas especiais baseadas em inundação permitirá que uma mensagem se espalhe gradualmente por uma parte da rede, para eventualmente

chegar ao destino. Obviamente, qualquer tipo de inundação impõe comunicação redundante, mas esse pode ser o preço que temos que pagar. Em segundo lugar, em uma rede tolerante a interrupções, permitimos que um nó intermediário armazene uma mensagem recebida até encontrar outro nó para o qual possa transmiti-la. Em outras palavras, um nó se torna um portador temporário de uma mensagem, conforme esboçado na Figura 1.14. Eventualmente, a mensagem deve chegar ao seu destino.

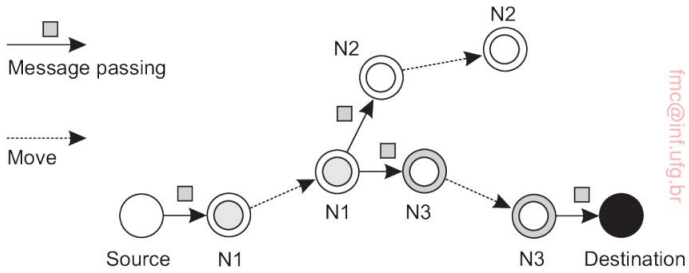

Figura 1.14: Passando mensagens em uma rede (móvel) tolerante a interrupções.

Não é difícil imaginar que a passagem seletiva de mensagens para os nós encontrados pode ajudar a garantir uma entrega eficiente. Por exemplo, se sabe-se que os nós pertencem a uma determinada classe e a origem e o destino pertencem à mesma classe, podemos decidir passar mensagens apenas entre os nós dessa classe. Da mesma forma, pode ser eficiente passar mensagens apenas para nós bem conectados, ou seja, nós que estiveram ao alcance de muitos outros nós no passado recente. Uma visão geral é fornecida por Spyropoulos et al. [2010].

Nota 1.11 (Avançado: Redes sociais e padrões de mobilidade)
Não surpreendentemente, a computação móvel está fortemente ligada ao paradeiro dos seres humanos. Com o crescente interesse por redes sociais complexas [Vega Redondo, 2007; Jackson, 2008] e a explosão do uso de smartphones, diversos grupos estão buscando combinar análise de comportamento social e disseminação de informações nas chamadas **redes pocket-switched** [Hui et al., 2005]. Estas últimas são redes em que os nós são formados por pessoas (ou, na verdade, seus dispositivos móveis), e os links são formados quando duas pessoas se encontram, permitindo que seus dispositivos troquem dados.

A ideia básica é permitir que a informação se espalhe usando as comunicações ad hoc entre as pessoas. Ao fazê-lo, torna-se importante compreender a estrutura de um grupo social. Um dos primeiros a examinar como a consciência social pode ser explorada em redes móveis foi Miklas et al. [2007]. Em sua abordagem, baseada em vestígios de encontros entre pessoas, duas pessoas são caracterizadas como amigas ou estranhas. Os amigos interagem com frequência, onde o número de encontros recorrentes entre estranhos é baixo. O objetivo é garantir que uma mensagem de Alice para Bob seja finalmente entregue.

Como se vê, quando Alice adota uma estratégia pela qual ela entrega a mensagem a cada um de seus amigos, e que cada um desses amigos passa a mensagem para Bob assim que é encontrado, pode garantir que a mensagem chegue a Bob com um atraso superior a aproximadamente 10% do atraso melhor atingível. Qualquer outra estratégia, como encaminhar a mensagem para apenas 1 ou 2 amigos, tem um desempenho muito pior.

Passar uma mensagem para um estranho não tem efeito significativo. Em outras palavras, faz uma enorme diferença se os nós levarem em conta as relações de amigos, mas mesmo assim ainda é necessário adotar criteriosamente uma estratégia de encaminhamento.

Para grandes grupos de pessoas, são necessárias abordagens mais sofisticadas. Em primeiro lugar, pode acontecer que as mensagens precisem ser enviadas entre pessoas em comunidades diferentes. O que entendemos por comunidade? Se considerarmos uma rede social (onde um vértice representa uma pessoa, e um link o fato de duas pessoas terem uma relação social), então uma comunidade é, grosso modo, um grupo de vértices no qual existem muitos links entre seus membros e apenas poucos. ligações com vértices em outros grupos [Newman, 2010]. Infelizmente, muitos algoritmos de detecção de comunidades requerem informações completas sobre a estrutura social, tornando -os praticamente inviáveis para otimizar a comunicação em redes móveis.

Hui et ai. [2007] propõem uma série de algoritmos de detecção de comunidades descentralizadas . Em essência, esses algoritmos dependem de permitir que um nó i (1) detecte o conjunto de nós que ele encontra regularmente, chamado de conjunto familiar Fi , e (2) expanda incrementalmente sua comunidade local Ci , começando com Ci = Fi e, à medida que Ci cresce, ficom ele, Ci .

No caso mais simples, um nó j é adicionado a uma comunidade Ci da seguinte forma:

$$\text{O nó i adiciona j a Ci quando } \frac{|Fj \ddot{y} Ci|}{|Fj|} > \ddot{y} \text{ para algum } \ddot{y} > 0$$

Em outras palavras, quando a fração do conjunto familiar de j se sobrepõe substancialmente à comunidade de i, então o nó i deve adicionar j à sua comunidade. Além disso, temos o seguinte para mesclar comunidades:

$$\text{Mesclar duas comunidades quando } |Ci \ddot{y} Cj| > \ddot{y}|Ci \ddot{y} Cj| \text{ para algum } \ddot{y} > 0$$

o que significa que duas comunidades devem ser mescladas quando tiverem um número significativo de membros em comum. (Em seus experimentos, Hui et al. descobriram que definir $\ddot{y} = \ddot{y} = 0{,}6$ leva a bons resultados.)

Conhecer comunidades, em combinação com a conectividade de um nó em uma comunidade ou globalmente, pode ser usado posteriormente para encaminhar mensagens de forma eficiente em uma rede tolerante a interrupções, conforme explicado por Hui et al. [2011].

Obviamente, muito do desempenho de um sistema de computação móvel depende de como os nós se movem. Em particular, para pré-avaliar a eficácia de novos protocolos ou algoritmos, é importante ter uma ideia de quais padrões de mobilidade são realmente realistas. Por muito tempo, não havia muitos dados sobre esses padrões, mas experimentos recentes mudaram isso.

Vários grupos começaram a coletar estatísticas sobre a mobilidade humana, cujos traços são usados para conduzir simulações. Além disso, traços foram usados para derivar modelos de mobilidade mais realistas (ver, por exemplo, Kim et al. [2006b]). No entanto,

compreender os padrões de mobilidade humana em geral continua a ser um problema difícil. González et al. [2008] relatam esforços de modelagem baseados em dados coletados de 100.000 usuários de telefones celulares durante um período de seis meses. Eles observaram que o comportamento do deslocamento poderia ser representado pela seguinte distribuição relativamente simples :

$$P[\ddot{y}r] = (\ddot{y}r + \ddot{y}r0)\ \ddot{y}\ddot{y} \cdot e^{\ddot{y}\ddot{y}r/mr}$$

em que ÿr é o deslocamento real e ÿr0 = 1,5km um deslocamento inicial constante. Com $\ddot{y}$ = 1,75 e $\ddot{y}$ = 400, isso leva à distribuição mostrada na Figura 1.15.

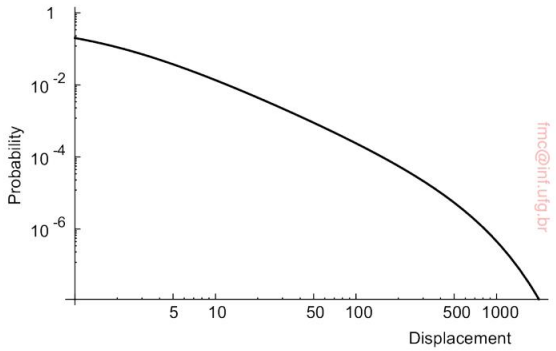

Figura 1.15: A distribuição do deslocamento de usuários de telefones celulares (móveis).

Podemos concluir que as pessoas tendem a ficar paradas. De fato, análises posteriores revelaram que as pessoas tendem a retornar ao mesmo lugar após 24, 48 ou 72 horas, mostrando claramente que as pessoas tendem a ir aos mesmos lugares. Em um estudo de acompanhamento, Song et al. [2010] poderia de fato mostrar que a mobilidade humana é realmente bem previsível.

Redes de sensores

Nosso último exemplo de sistemas pervasivos são **as redes de sensores.** Essas redes, em muitos casos, fazem parte da tecnologia de habilitação para difusão e vemos que muitas soluções para redes de sensores retornam em aplicações pervasivas. O que torna as redes de sensores interessantes do ponto de vista de um sistema distribuído é que elas são mais do que apenas uma coleção de dispositivos de entrada. Em vez disso, como veremos, os nós sensores geralmente colaboram para processar eficientemente os dados detectados de uma maneira específica da aplicação, tornando-os muito diferentes, por exemplo, das redes de computadores tradicionais. Akyildiz et ai. [2002] e Akyildiz et al. [2005] fornecem uma visão geral de uma perspectiva de rede. Uma introdução mais orientada a sistemas para redes de sensores é dada por Zhao e Guibas [2004], mas também Karl e Willig [2005] se mostrarão úteis.

Uma rede de sensores geralmente consiste de dezenas a centenas ou milhares de nós relativamente pequenos, cada um equipado com um ou mais dispositivos sensores. Dentro

Além disso, os nós podem muitas vezes atuar como atuadores [Akyildiz e Kasimoglu, 2004], um exemplo típico é a ativação automática de sprinklers quando um incêndio é detectado. Muitas redes de sensores usam comunicação sem fio e os nós geralmente são alimentados por bateria. Seus recursos limitados, recursos de comunicação restritos e consumo de energia restrito exigem que a eficiência esteja no topo da lista de critérios de projeto.

Ao fazer zoom em um nó individual, vemos que, conceitualmente, eles não diferem muito dos computadores “normais”: acima do hardware há uma camada de software semelhante ao que os sistemas operacionais tradicionais oferecem, incluindo acesso à rede de baixo nível, acesso a sensores e atuadores, gerenciamento de memória e assim por diante. Normalmente, o suporte para serviços específicos está incluído, como localização, armazenamento local (pense em dispositivos flash adicionais) e facilidades de comunicação convenientes, como mensagens e roteamento. No entanto, semelhante a outros sistemas de computador em rede, é necessário suporte adicional para implantar efetivamente aplicativos de rede de sensores. Em sistemas distribuídos, isso assume a forma de middleware. Para redes de sensores, em vez de olhar para middleware, é melhor ver que tipo de suporte de programação é fornecido, o que foi amplamente pesquisado por Mottola e Picco [2011].

Um aspecto típico no suporte à programação é o escopo fornecido pelas primitivas de comunicação. Esse escopo pode variar entre endereçar a vizinhança física de um nó e fornecer primitivas para comunicação em todo o sistema. Além disso, também pode ser possível endereçar um grupo específico de nós. Da mesma forma, os cálculos podem ser restritos a um nó individual, um grupo de nós ou afetar todos os nós. Para ilustrar, Welsh e Mainland [2004] usam as chamadas **regiões abstratas** que permitem a um nó identificar uma vizinhança de onde pode, por exemplo, coletar informações:

```
1  região = k_neest_region.create(8); lendo =
2  get_sensor_reading(); region.putvar(chave_leitura,
3  leitura); max_id = region.reduce(OP_MAXID,
4  leitura_key);
```

Na linha 1, um nó primeiro cria uma região de seus oito vizinhos mais próximos, após o que busca um valor de seu(s) sensor(es). Essa leitura é posteriormente gravada na região previamente definida a ser definida usando a chave read_key. Na linha 4, o nó verifica cuja leitura do sensor na região definida foi a maior, que é retornada na variável max_id.

Como outro exemplo relacionado, considere uma rede de sensores como a implementação de um banco de dados distribuído, que é, segundo Mottola e Picco [2011], uma das quatro formas possíveis de acessar dados. Essa visão de banco de dados é bastante comum e fácil de entender ao perceber que muitas redes de sensores são implantadas para aplicações de medição e vigilância [Bonnet et al., 2002]. Nesses casos, um operador gostaria de extrair informações de (uma parte) da rede simplesmente emitindo consultas como “Qual é a carga de tráfego no sentido norte na rodovia 1 como Santa Cruz?” Tais consultas assemelham-se às dos tradicionais

bancos de dados. Nesse caso, a resposta provavelmente precisará ser fornecida por meio da colaboração de muitos sensores ao longo da rodovia 1, deixando outros sensores intocados.

Para organizar uma rede de sensores como um banco de dados distribuído, existem basicamente dois extremos, como mostra a Figura 1.16. Primeiro, os sensores não cooperam, mas simplesmente enviam seus dados para um banco de dados centralizado localizado no local do operador. O outro extremo é encaminhar consultas para sensores relevantes e permitir que cada um calcule uma resposta, exigindo que o operador agregue as respostas.

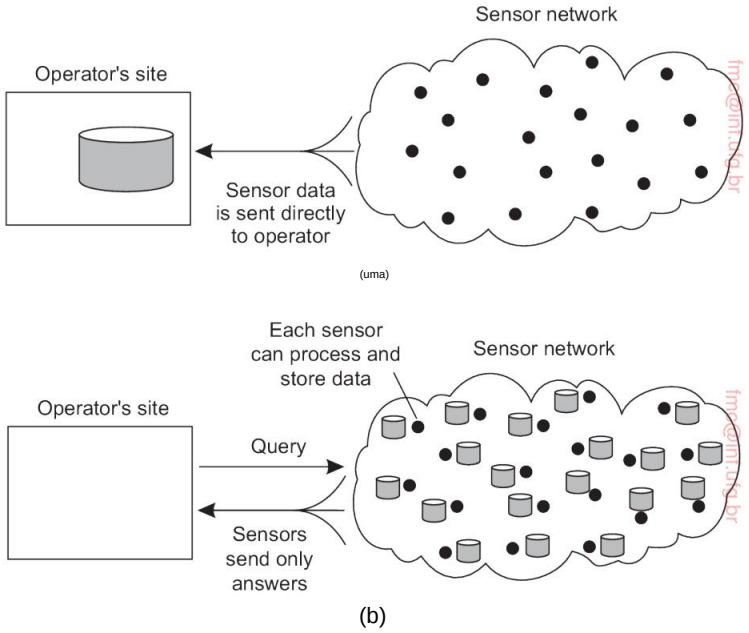

Figura 1.16: Organização de um banco de dados de rede de sensores, enquanto armazena e processa dados (a) apenas no local do operador ou (b) apenas nos sensores.

Nenhuma dessas soluções é muito atraente. A primeira exige que os sensores enviem todos os dados medidos pela rede, o que pode desperdiçar recursos e energia da rede. A segunda solução também pode ser um desperdício, pois descarta os recursos de agregação dos sensores que permitiriam que muito menos dados fossem devolvidos ao operador. O que é necessário são facilidades para **processamento de dados em rede,** semelhantes ao exemplo anterior de regiões abstratas.

O processamento na rede pode ser feito de várias maneiras. Uma óbvia é encaminhar uma consulta para todos os nós sensores ao longo de uma árvore que abrange todos os nós e, posteriormente, agregar os resultados à medida que são propagados de volta à raiz, onde o iniciador está localizado. A agregação ocorrerá quando dois ou mais ramos da árvore se unirem. Por mais simples que esse esquema possa

som, introduz questões difíceis:

• Como configuramos (dinamicamente) uma árvore eficiente em uma rede de sensores? • Como ocorre a agregação de resultados? Pode ser controlado? • O que acontece quando os links de rede falham?

Essas questões foram parcialmente abordadas no TinyDB, que implementa uma interface declarativa (banco de dados) para redes de sensores sem fio [Madden et al., 2005]. Em essência, o TinyDB pode usar qualquer algoritmo de roteamento baseado em árvore. Um nó intermediário coletará e agregará os resultados de seus filhos, juntamente com suas próprias descobertas, e os enviará para a raiz. Para tornar as coisas mais eficientes, as consultas abrangem um período de tempo, permitindo um planejamento cuidadoso das operações para que os recursos e a energia da rede sejam consumidos de maneira otimizada.

No entanto, quando as consultas podem ser iniciadas de diferentes pontos na rede , o uso de árvores de raiz única, como no TinyDB, pode não ser eficiente o suficiente. Como alternativa, as redes de sensores podem ser equipadas com nós especiais para onde são encaminhados os resultados, bem como as consultas relacionadas a esses resultados. Para dar um exemplo simples, consultas e resultados relacionados a leituras de temperatura podem ser coletados em um local diferente daqueles relacionados a medições de umidade . Essa abordagem corresponde diretamente à noção de sistemas de publicação/assinatura.

Nota 1.12 (Avançado: Quando a energia começa a se tornar crítica)
Como mencionado, muitas redes de sensores precisam operar com um orçamento de energia proveniente do uso de baterias ou outras fontes de alimentação limitadas. Uma abordagem para reduzir o consumo de energia é deixar os nós ativos apenas parte do tempo. Mais especificamente, suponha que um nó esteja repetidamente ativo durante unidades de tempo Tativas e, entre esses períodos ativos, ele seja suspenso para unidades Tsuspended . A fração de tempo que um nó está ativo é conhecido como seu **ciclo de trabalho** ÿ, ou seja,

$$t = \frac{\text{Tativo}}{\text{Tativo} + \text{Suspenso}}$$

Os valores para *ÿ* são tipicamente da ordem de 10 a 30%, mas quando uma rede precisa permanecer operacional por períodos superiores a muitos meses, ou mesmo anos, atingir valores tão baixos quanto 1% são críticos.

Um problema com redes com ciclo de trabalho é que, em princípio, os nós precisam estar ativos ao mesmo tempo, caso contrário a comunicação simplesmente não seria possível. Considerando que enquanto um nó está suspenso, apenas seu relógio local continua tiquetaqueando, e que esses relógios estão sujeitos a desvios, acordar no mesmo horário pode ser problemático. Isto é particularmente verdadeiro para redes com ciclos de trabalho muito baixos.

Quando um grupo de nós está ativo ao mesmo tempo, diz-se que os nós formam um **grupo sincronizado.** Existem essencialmente dois problemas que precisam ser resolvidos. Primeiro, precisamos garantir que os nós em um grupo sincronizado permaneçam ativos ao mesmo tempo. Na prática, isso acaba sendo relativamente simples

se cada nó comunicar informações sobre sua hora local atual. Então, ajustes simples do relógio local farão o truque. O segundo problema é mais difícil, ou seja, como dois grupos sincronizados diferentes podem ser mesclados em um em que todos os nós são sincronizados. Vejamos mais de perto o que estamos enfrentando. A maior parte da discussão a seguir é baseada no material de Voulgaris et al. [2016].

Para que dois grupos sejam mesclados, precisamos primeiro garantir que um grupo detecte o outro. De fato, se seus respectivos períodos ativos são completamente disjuntos, não há esperança de que qualquer nó de um grupo possa receber uma mensagem de um nó do outro grupo. Em um método de detecção ativo, um nó enviará uma mensagem de junção durante seu período de suspensão. Em outras palavras, enquanto ele está suspenso, ele acorda temporariamente para elicitar nós em outros grupos para se juntarem. Qual é a chance de outro nó pegar essa mensagem? Perceba que precisamos considerar apenas o caso quando $\ddot{y} < 0,5$, caso contrário, dois períodos ativos sempre se sobrepõem, o que significa que dois grupos podem detectar facilmente a presença um do outro. A probabilidade Pda de que uma mensagem de junção possa ser captada durante o período ativo de outro nó é igual a

$$\text{Pda} = \frac{\text{Tativo}}{\text{Suspenso}} = \frac{t}{1 - p}$$

Isso significa que para valores baixos de ÿ, Pda também é muito pequeno.

Em um método de detecção passiva, um nó pula o estado suspenso com (uma probabilidade muito baixa) Pdp, ou seja, ele simplesmente permanece ativo durante as unidades de tempo Tsuspended após seu período ativo. Durante esse tempo, ele poderá receber todas as mensagens enviadas por seus vizinhos, que são, por definição, membros de um grupo sincronizado diferente. Experimentos mostram que a detecção passiva é inferior à detecção ativa .

Simplesmente declarar que dois grupos sincronizados precisam se fundir não é suficiente: se A e B descobriram um ao outro, qual grupo adaptará as configurações de ciclo de trabalho do outro? Uma solução simples é usar uma noção de IDs de cluster. Cada nó começa com um ID escolhido aleatoriamente e efetivamente também um grupo sincronizado tendo apenas a si mesmo como membro. Depois de detectar outro grupo B, todos os nós do grupo A se juntam a B se e somente se o ID do cluster de B for maior que o de A.

A sincronização pode ser melhorada consideravelmente usando as chamadas mensagens de junção direcionadas. Sempre que um nó N recebe uma mensagem de junção de um grupo A com um ID de cluster menor, ele obviamente não deve ingressar em A. No entanto, como N agora sabe quando é o período ativo de A , ele pode enviar uma mensagem de junção exatamente durante esse período. Obviamente, a chance de um nó de A receber essa mensagem é muito alta, permitindo que os nós de A se juntem ao grupo de N. Além disso, quando um nó decide ingressar em outro grupo, ele pode enviar uma mensagem especial aos membros do grupo, dando a oportunidade de ingressar rapidamente também.

A Figura 1.17 mostra a rapidez com que grupos sincronizados podem se fundir usando duas estratégias diferentes. Os experimentos são baseados em uma rede móvel de 4.000 nós usando padrões de mobilidade realistas. Os nós têm um ciclo de trabalho inferior a 1%. Esses experimentos mostram que trazer mesmo uma grande rede móvel com ciclo de trabalho para um estado em que todos os nós estejam ativos ao mesmo tempo é bastante viável. Para mais informações, ver Voulgaris et al. [2016].

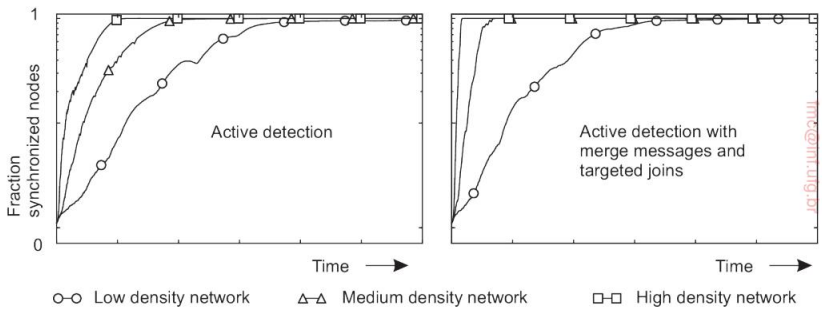

Figura 1.17: A velocidade com que diferentes grupos sincronizados podem se fundir.

1.4 Resumo

Os sistemas distribuídos consistem em computadores autônomos que trabalham juntos para dar a aparência de um único sistema coerente. Essa combinação de comportamento coletivo independente, porém coerente, é alcançada pela coleta de protocolos independentes de aplicativos no que é conhecido como middleware: uma camada de software logicamente colocada entre sistemas operacionais e aplicativos distribuídos. Os protocolos incluem aqueles para comunicação, transações, composição de serviços e, talvez o mais importante, confiabilidade.

As metas de design para sistemas distribuídos incluem o compartilhamento de recursos e a garantia de abertura. Além disso, os designers visam esconder muitos dos meandros relacionados à distribuição de processos, dados e controle. No entanto, esta transparência de distribuição não tem apenas um preço de desempenho, em situações práticas nunca pode ser totalmente alcançada. O fato de que compensações precisam ser feitas entre a obtenção de várias formas de transparência de distribuição é inerente ao projeto de sistemas distribuídos e pode facilmente complicar seu entendimento. Um objetivo específico de design difícil que nem sempre combina bem com a obtenção de transparência de distribuição é a escalabilidade. Isso é particularmente verdadeiro para a escalabilidade geográfica, caso em que ocultar latências e restrições de largura de banda pode ser difícil. Da mesma forma, a escalabilidade administrativa pela qual um sistema é projetado para abranger vários domínios administrativos pode facilmente entrar em conflito com os objetivos para alcançar a transparência da distribuição.

As coisas são ainda mais complicadas pelo fato de que muitos desenvolvedores inicialmente fazem suposições sobre a rede subjacente que estão fundamentalmente erradas. Mais tarde, quando as suposições são descartadas, pode ser difícil mascarar o comportamento indesejado. Um exemplo típico é assumir que a latência da rede não é significativa. Outras armadilhas incluem assumir que a rede é confiável, estática, segura e homogênea.

Existem diferentes tipos de sistemas distribuídos que podem ser classificados como orientados para suportar computações, processamento de informações e difusão. Os sistemas de computação distribuída são normalmente implantados para aplicativos de alto desempenho, muitas vezes originários do campo da computação paralela. Um campo que emergiu do processamento paralelo foi inicialmente a computação em grade com um forte foco no compartilhamento mundial de recursos, levando ao que hoje é conhecido como computação em nuvem. A computação em nuvem vai além da computação de alto desempenho e também oferece suporte a sistemas distribuídos encontrados em ambientes de escritório tradicionais, onde vemos os bancos de dados desempenhando um papel importante. Normalmente, os sistemas de processamento de transações são implementados nesses ambientes. Finalmente, uma classe emergente de sistemas distribuídos é onde os componentes são pequenos, o sistema é composto de forma ad hoc, mas acima de tudo não é mais gerenciado por um administrador de sistema. Esta última classe é tipicamente representada por ambientes de computação pervasivos, incluindo sistemas de computação móvel, bem como ambientes ricos em sens