

6

Sincronização

Nos capítulos anteriores, estudamos processos e comunicação entre processos. Embora a comunicação seja importante, não é tudo. Uma questão intimamente ligada a ela é o modo como processos cooperam e sincronizam uns com os outros. A cooperação é atingida, em parte, por meio de nomeação, que permite aos processos ao menos compartilhar recursos ou entidades em geral.

Neste capítulo, vamos nos concentrar principalmente no modo como os processos podem sincronizar. Por exemplo, é importante que vários processos não acessem simultaneamente um recurso compartilhado como uma impressora mas, ao contrário, cooperem para garantir um ao outro acesso temporário exclusivo. Um outro exemplo é que vários processos às vezes podem concordar com a ordenação de eventos, por exemplo, se a mensagem *m1* do processo *P* foi enviada antes ou depois da mensagem *m2* do processo *Q*.

Ocorre que a sincronização em sistemas distribuídos costuma ser muito mais difícil em comparação com a sincronização em sistemas monoprocessadores ou multiprocessadores. Os problemas e soluções discutidos neste capítulo são, por sua natureza, bastante gerais e ocorrem em variadas situações em sistemas distribuídos.

Começaremos com uma discussão sobre a questão da sincronização baseada em tempo real, seguida pela sincronização na qual o que importa são apenas questões de ordenação relativa, e não de ordenação em tempo absoluto.

Em muitos casos é importante que um grupo de processos possa designar um processo como coordenador, o que pode ser feito por meio de algoritmos de eleição. Discutiremos vários algoritmos de eleição em uma seção específica.

Há muitos tipos e espécies de algoritmos distribuídos que foram desenvolvidos para tipos variados de sistemas distribuídos. Muitos exemplos (e mais referências) podem ser encontrados em Andrews (2000) e Guerraoui e Rodrigues (2006). Abordagens mais formais para uma profusão de algoritmos podem ser encontradas em Attiya e Welch (2004), Lynch (1996) e Tel (2000).

6.1 Sincronização de Relógios

Em um sistema centralizado, o tempo não é ambíguo. Quando um processo quer saber a hora, faz uma chamada de sistema, e o núcleo responde. Se o processo *A* perguntar a hora e, um pouco mais tarde, o processo *B* também perguntar a hora, o valor que *B* obtém será mais alto (ou possivelmente igual ao valor que *A* obteve. Porém, por certo, não será mais baixo. Em um sistema distribuído, conseguir acordo nos horários não é trivial.

Imagine, por um instante, as implicações da falta de um horário global no programa *make* do Unix, só para dar um exemplo. Em Unix, programas grandes normalmente são divididos em vários arquivos-fonte, de modo que uma alteração em um arquivo-fonte requer que apenas um arquivo seja recompilado, e não todos os arquivos. Se um programa consistir em cem arquivos, não ter de recompilar tudo porque um arquivo foi alterado aumenta bastante a velocidade à qual os programadores podem trabalhar.

O modo de funcionamento normal do *make* é simples. Quando o programador terminou de alterar todos os arquivos-fonte, ele executa *make*, que examina os horários em que todos os arquivos-fonte e arquivos-objeto foram modificados da última vez. Se o horário do arquivo-fonte *input.c* for 2151 e o horário do arquivo-objeto *input.o* for 2150, *make* sabe que *input.c* foi alterado desde o momento em que *input.o* foi criado e, assim, que *input.c* deve ser recompilado. Por outro lado, se o horário de *output.c* for 2144 e o horário de *output.o* for 2145, nenhuma compilação será necessária. Por isso, *make* percorre todos os arquivos-fonte para descobrir quais deles precisam ser recompilados e chama o compilador para fazer isso.

Agora, imagine o que poderia acontecer em um sistema distribuído no qual não houvesse nenhum acordo global sobre horários. Suponha que o horário de *output.o* seja 2144 como citado antes e que, logo depois, *output.c* tenha sido modificado, mas recebeu o horário 2143 porque o relógio de sua máquina estava um pouco atrasado, como mostra a Figura 6.1. *Make* não chamará o compilador. Sendo assim, o programa binário executável resultante conterá uma mistura de arquivos-objeto dos fonte抗igos e dos fonte novos.

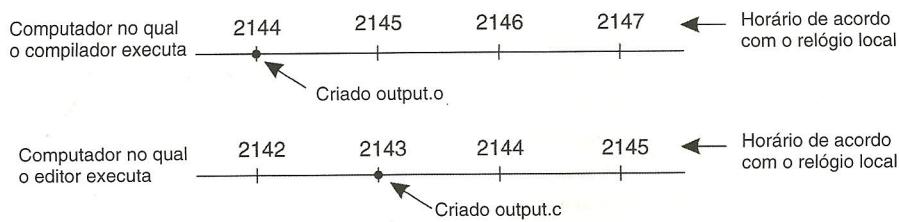


Figura 6.1 Quando cada máquina tem seu próprio relógio, um evento que ocorreu após outro evento pode, ainda assim, receber um horário anterior.

Ele provavelmente falhará, e o programador ficará confuso tentando entender o que está errado no código.

Há muitos outros exemplos em que é necessário um controle exatos de horários. O exemplo que demos pode ser facilmente reformulado para arquivar marcas de tempo em geral. Além disso, pense em domínios de aplicação como corretagem de ativos financeiros, auditoria de segurança e sensoriamento colaborativo, e ficará claro que a temporização exata é importante. Uma vez que a marcação do tempo é tão básica para o modo de pensar das pessoas, e o efeito de não ter todos os relógios sincronizados pode ser tão drástico, nada mais adequado do que começar nosso estudo da sincronização com a simples pergunta: é possível sincronizar todos os relógios em um sistema distribuído? A resposta é surpreendentemente complicada.

6.1 Relógios físicos

Quase todos os computadores têm um circuito para monitorar a passagem do tempo. Apesar do uso disseminado do termo ‘relógio’ para se referir a esses dispositivos, na verdade eles não são relógios no sentido usual da palavra. **Temporizador** talvez seja uma palavra melhor. Um temporizador de computador usualmente é um cristal de quartzo lapidado e usinado com precisão. Quando mantidos sob tensão, cristais de quartzo oscilam a uma freqüência bem definida que depende do tipo de cristal, de como ele foi lapidado e da magnitude da tensão. Associados com cada cristal há dois registradores, um **contador** e um **registrator de retenção**. Cada oscilação do cristal reduz uma unidade do contador. Quando o contador chega a zero é gerada uma interrupção e o contador é recarregado pelo registrator de retenção. Desse modo, é possível programar um temporizador para gerar uma interrupção 60 vezes por segundo ou a qualquer outra freqüência desejada. Cada interrupção é denominada **ciclo de relógio**.

Quando o sistema é inicializado, ele usualmente solicita ao usuário que digite a data e a hora, que então são convertidas para o número de ciclos de relógio após alguma data inicial conhecida e armazenada na memória. A maioria dos computadores tem uma RAM CMOS especial suportada por bateria, de modo que a data e a hora não precisam ser digitadas em ativações subsequentes.

A cada ciclo de relógio, o procedimento do serviço de interrupção soma uma unidade à hora armazena da na memória. Desse modo, o relógio (de software) é mantido atualizado.

Com um único computador e um único relógio, não há problema se esse relógio estiver um pouco defasado. Uma vez que todos os processos na máquina usam o mesmo relógio, eles ainda serão internamente consistentes. Por exemplo, se o horário do arquivo *input.c* for 2151 e o horário do arquivo *input.o* for 2150, *make* recompilará o arquivo-fonte ainda que o relógio esteja defasado por dois ciclos e os horários verdadeiros sejam 2153 e 2152, respectivamente. O que realmente importa são os horários relativos.

Logo que forem introduzidas CPUs múltiplas, cada uma com seu próprio relógio, a situação sofre uma mudança radical. Embora a freqüência à qual um oscilador de cristal funciona seja em geral razoavelmente estável, é impossível garantir que todos os cristais em diferentes computadores funcionem exatamente à mesma freqüência. Na prática, quando um sistema tem n computadores, todos os n cristais funcionarão a taxas ligeiramente diferentes, o que faz com que os relógios (de software) gradativamente saiam de sincronia e informem valores diferentes quando lidos. Essa diferença nos valores dos horários é denominada **defasagem de relógio**. Em consequência dessa defasagem entre relógios, programas que esperam que o horário associado com um arquivo, objeto, processo ou uma mensagem esteja correto e seja independente da máquina na qual foi gerado (isto é, de qual relógio é usado) podem falhar, como já vimos no exemplo do *make*.

Em alguns sistemas (por exemplo, sistemas de tempo real), a hora real marcada pelo relógio é importante. Sob essas circunstâncias, são necessários relógios físicos externos. Por razões de eficiência e redundância, em geral considera-se desejável ter vários relógios físicos, o que resulta em dois problemas: 1) como sincronizá-los com relógios do mundo real, e 2) como sincronizar os relógios um com o outro?

Antes de responder a essas perguntas, vamos fazer uma ligeira digressão e ver como o tempo é realmente medido. Na verdade, medir o tempo não é, nem de longe, tão fácil como poderíamos imaginar, em especial quando se requer alta precisão. Desde a invenção dos relógios

mecânicos no século XVII, o tempo tem sido medido por meios astronômicos. Todo dia o sol nasce no horizonte leste, depois sobe até uma altura máxima no céu e, por fim, mergulha no oeste. O evento da passagem do sol pelo seu ponto aparente mais alto no céu é denominado **trânsito solar**. Esse evento ocorre aproximadamente ao meio-dia, todos os dias. O intervalo entre dois trânsitos consecutivos do sol é denominado **dia solar**. Visto que há 24 horas em um dia, cada hora com 3.600 segundos, o **segundo solar** é definido exatamente como 1/86.400 de um dia solar. A geometria do cálculo do dia solar médio é mostrada na Figura 6.2.

Na década de 1940, foi estabelecido que o período de rotação da Terra não é constante. A Terra está desacelerando devido ao atrito das marés e ao arraste atmosférico. Com base em estudos sobre os padrões de crescimento em corais antigos, agora os geólogos acreditam que há 300 milhões de anos havia aproximadamente 400 dias por ano. Não foi o comprimento do ano (o tempo de uma viagem ao redor do sol) que mudou; simplesmente, o dia é que ficou mais longo. Além dessa tendência de longo prazo, também ocorrem variações de curto prazo no comprimento do dia, provavelmente causadas por turbulências nas profundidades do núcleo da Terra, que é de ferro fundido. Essas revelações levaram os astrônomos a calcular o comprimento do dia medindo uma grande quantidade de dias e tomando a média antes de dividir por 86.400. A quantidade resultante foi denominada **segundo solar médio**.

Com a invenção do relógio atômico em 1948, tornou-se possível medir o tempo com muito mais exatidão, e independentemente dos movimentos erráticos da Terra, contando transições do átomo de césio 133. Os profissionais de física tomaram dos astrônomos a tarefa de contar o tempo e definiram o segundo como o tempo que o átomo de césio 133 leva para fazer exatamente 9.192.631.770 transições. Esse número foi escolhido de modo que o segundo atômico fosse igual ao segundo solar médio no

ano em que foi lançado. Hoje, vários laboratórios ao redor do mundo têm relógios de césio 133 e cada um deles informa periodicamente ao Bureau International de l'Heure (BIH), em Paris, quantas vezes seu relógio pulsou. O BIH calcula a média desses valores e produz a **hora atômica internacional** (International Atomic Time) ou **TAI**. Assim, a TAI é apenas o número médio de ciclos dos relógios de césio 133 desde a meia-noite de 1º de janeiro de 1958 (o início da contagem do tempo) dividido por 9.192.631.770.

Embora seja muito estável e esteja disponível para quem quiser se dar ao trabalho de comprar um relógio de césio, a TAI apresenta um sério problema: hoje, 86.400 segundos TAI equivalem a aproximadamente 3 ms a menos do que um dia solar médio (porque o dia solar médio está ficando mais longo a cada dia). Usar a TAI para medir o tempo significaria que, no decorrer dos anos, o meio-dia seria cada vez mais cedo, até que, a certa altura, ocorreria de madrugada. Por certo todos perceberiam isso e poderíamos ter o mesmo tipo de situação que ocorreu em 1582 quando um decreto do papa Gregório XIII eliminou dez dias do calendário. Esse evento causou revoltas nas ruas porque os donos de terras exigiam um mês inteiro de aluguel e os banqueiros, um mês inteiro de juros, enquanto os empregadores se recusavam a pagar aos trabalhadores os dez dias em que não trabalharam, só para mencionar alguns dos conflitos. Por questão de princípio, os países protestantes se recusaram a obedecer a qualquer decreto papal e não aceitaram o calendário gregoriano por 170 anos.

O BIH resolve o problema ao introduzir **segundos extras** sempre que a discrepância entre a hora TAI e a hora solar alcança 800 ms. A utilização de segundos extras é ilustrada na Figura 6.3. Essa correção dá origem ao sistema de medição do tempo baseado em segundos TAI constantes, mas que fica em fase com o movimento aparente do sol. Ele é denominado **hora coordenada universal**.

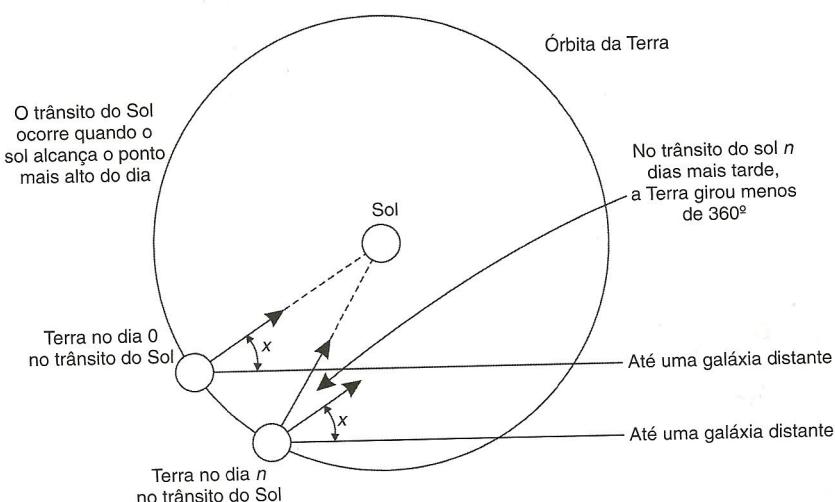


Figura 6.2 Cálculo do dia solar médio.



Figura 6.3 Segundos TAI têm comprimento constante, diferentes dos segundos solares.
Os segundos extras são introduzidos quando necessário para se manterem em fase com o sol.

versal (Universal Coordinated Time), ou UTC. O sistema UTC é a base de toda a moderna medição civil do tempo. Em essência, ele substituiu o antigo padrão, o Greenwich Mean Time (tempo médio de Greenwich) que é a hora astronômica.

A maioria das empresas geradoras de energia elétrica sincroniza a temporização de seus relógios de 60 Hz ou 50 Hz com o UTC; portanto, quando o BIH anuncia um segundo extra, essas empresas elevam sua freqüência para 61 Hz ou 51 Hz durante 60 ou 50 segundos, a fim de adiantar todos os relógios em sua área de distribuição. Visto que 1 segundo é um intervalo perceptível para um computador, um sistema operacional que precisa manter horários exatos durante certo período de anos deve ter um software especial para lidar com segundos extras quando eles são anunciados (a menos que usem a linha de fornecimento de energia para medir tempo, o que, de modo geral, é um método muito grosseiro). O número total de segundos introduzidos no UTC até agora é aproximadamente 30.

Para fornecer UTC a quem precisa da hora exata, o National Institute of Standard Time (Nist) opera uma estação de rádio de ondas curtas cujo prefixo é WWV, em Fort Collins, Colorado. A WWV transmite um pulso curto no início de cada segundo UTC. A precisão da própria WWV é de ± 1 ms, porém, devido a flutuações atmosféricas aleatórias que podem afetar o comprimento do caminho do sinal, na prática a precisão não é melhor do que ± 10 ms. Na Inglaterra, a estação MSF, que transmite de Rugby, Warwickshire, oferece um serviço semelhante, assim como estações situadas em vários outros países.

Há vários satélites em órbita terrestre que também oferecem serviço UTC. O satélite operacional ambiental geoestacionário (Geostationary Environment Operational Satellite — Geos) pode oferecer UTC com exatidão de até 0,5 ms, e alguns outros satélites se saem ainda melhor.

Usar serviços de rádio de ondas curtas ou de satélites requer conhecer a exata posição relativa entre remetente e receptor, de modo a compensar o atraso de propagação do sinal. Radiorreceptores para WWV, Geos e outras fontes UTC estão disponíveis no comércio.

6.1.2 Sistema de posicionamento global

Como passo em direção aos problemas da sincronização de relógios propriamente dita, consideraremos, em primeiro lugar, um problema relacionado, ou seja, a determinação da posição geográfica de alguém em qualquer lugar da Terra. Esse problema de posicionamento é, em si, resolvido por meio de um sistema distribuído dedicado altamente específico denominado GPS, que é um acrônimo para *global positioning system (sistema de posicionamento global)*. O GPS é um sistema distribuído baseado em satélite lançado em 1978. Embora tenha sido utilizado principalmente para aplicações militares, nos últimos anos ele foi adotado em muitas aplicações civis, em particular na área da navegação comercial. Contudo, existem muitos outros domínios de aplicações. Por exemplo, agora, telefones GPS permitem aos interlocutores monitorar suas respectivas posições, característica que pode mostrar ser extremamente útil quando você se perder ou estiver em dificuldades. Esse princípio também pode ser aplicado com facilidade ao monitoramento de outras coisas, entre elas animais de estimação, crianças, carros, barcos e assim por diante. Uma excelente visão geral do GPS é dada por Zogg (2002).

O GPS usa 29 satélites, cada um circulando em uma órbita a uma altura aproximada de 20.000 km. Cada satélite tem até quatro relógios atômicos que são calibrados periodicamente por estações especiais na Terra. Um satélite transmite continuamente sua posição em broadcast e anexa marcas de tempo a cada mensagem, informando sua hora local. Essa transmissão broadcast permite que todo receptor na Terra calcule com precisão sua própria posição usando, em princípio, somente três satélites. Para explicar, a princípio vamos considerar que todos os relógios, entre eles o do receptor, estejam sincronizados.

Para calcular uma posição considere, em primeiro lugar, o caso bidimensional, como mostra a Figura 6.4, na qual estão desenhados dois satélites, junto com os círculos, que representam pontos que estão a mesma distância de cada satélite respectivo. O eixo *y* representa a altura, enquanto o eixo *x* representa uma linha reta ao longo da superfície da Terra, no nível do mar. Ignorando o ponto mais

elevado, vemos que a interseção dos dois círculos é um ponto único, nesse caso, talvez no alto de alguma montanha.

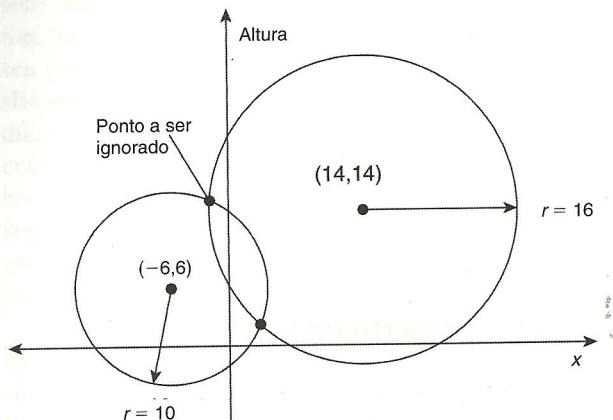


Figura 6.4 Cálculo de uma posição em um espaço bidimensional.

Esse princípio de interseção de círculos pode ser expandido para três dimensões, o que significa que precisamos de três satélites para determinar longitude, latitude e altitude de um receptor na Terra. Todo esse posicionamento é razoavelmente direto, mas as coisas tornam-se complicadas quando não podemos mais supor que todos os relógios estão perfeitamente sincronizados.

Há também dois fatos importantes do mundo real que precisamos levar em conta:

1. Leva um certo tempo para que os dados sobre a posição de um satélite cheguem ao receptor.
2. De modo geral, o relógio do receptor não está em sincronia com o de um satélite.

Suponha que a marca de tempo de um satélite seja totalmente exata. Seja Δ_r o desvio do relógio do receptor em relação à hora real. Quando o receptor recebe uma mensagem enviada pelo satélite i com a marca de tempo T_i , o atraso medido do receptor, Δ_i , consiste em dois componentes: o atraso propriamente dito, mais seu próprio desvio:

$$\Delta_i = (T_{agora} - T_i) + \Delta_r$$

Como os sinais viajam à velocidade da luz, c , a distância medida do satélite é, claramente, $c\Delta_i$. Sendo

$$d_i = c(T_{agora} - T_i)$$

a distância real entre o receptor e o satélite, a distância medida pode ser expressa por $d_i + c\Delta_r$. A distância real é calculada apenas por:

$$d_i = \sqrt{(x_i - x_r)^2 + (y_i - y_r)^2 + (z_i - z_r)^2}$$

onde x_i , y_i e z_i são as coordenadas do satélite i . O que vemos agora é que, se tivermos quatro satélites, teremos quatro equações com quatro incógnitas, o que nos permi-

te resolver as coordenadas x_r , y_r e z_r para o receptor, e também Δ_r . Em outras palavras, uma medição de GPS também dará uma indicação da hora real. Mais adiante neste capítulo, voltaremos à determinação de posições seguindo uma abordagem semelhante.

Até aqui, adotamos como premissa que as medidas são perfeitamente exatas. Claro que não são. Uma razão é que o GPS não leva em conta segundos extras. Em outras palavras, há um desvio sistemático em relação à hora UTC que, em 1º de janeiro de 2006, era de 14 segundos. Esse erro pode ser facilmente compensado por software. Contudo, há muitas outras fontes de erros, que começam pelo fato de que os relógios atômicos nos satélites nem sempre estão em perfeita sincronia, a posição de um satélite não é conhecida com exatidão, a precisão do relógio do receptor é finita, a velocidade de propagação do sinal não é constante (porque os sinais perdem velocidade, por exemplo, ao entrar na ionosfera) e assim por diante. Além do mais, todos sabemos que a Terra não é uma esfera perfeita, o que resulta em mais correções.

De modo geral, calcular uma posição exata está longe de ser uma tarefa trivial e requer noção de muitos detalhes assustadores. Não obstante, até mesmo com receptores GPS relativamente baratos, o posicionamento pode alcançar uma precisão dentro de uma faixa de 1 a 5 metros. Além do mais, receptores profissionais (que podem ser facilmente ligados a uma rede de computadores) têm um erro declarado de menos de 20 a 35 nanosegundos. Mais uma vez, referimo-nos à excelente visão geral de Zogg (2002) como uma primeira etapa para conhecer os detalhes.

6.1.3 Algoritmos de sincronização de relógios

Se uma máquina tiver um receptor WWV, a meta é manter todas as outras máquinas sincronizadas com ela. Se nenhuma máquina tiver receptores WWV, cada uma monitora seu próprio horário, e o objetivo é manter todas as máquinas o mais juntas possível. Muitos algoritmos foram propostos para fazer essa sincronização, e um levantamento deles pode ser encontrado em Ramanathan et al. (1990).

Todos os algoritmos têm o mesmo modelo subjacente do sistema. Cada máquina deve ter um temporizador que provoca uma interrupção H vezes por segundo. Quando o temporizador esgota o tempo fixado, o manipulador de interrupção soma 1 a um relógio de software que monitora o número de ciclos de relógio (interrupções) que ocorreram desde um instante determinado com o qual todos concordaram antes. Vamos denominar C o valor desse relógio. Mais especificamente, quando a hora UTC é t , o valor do relógio na máquina p é $C_p(t)$. Idealmente, teríamos $C_p(t) = t$ para todo p e todo t . Em outras palavras, seria perfeito que $C'_p(t) = dC/dt$ fosse 1. $C'_p(t)$ é denominado **frequênci**a do relógio de p 's no tempo t . A **defasagem** do relógio é definida como $C'_p(t) - 1$ e denota a magnitude da diferença entre a frequênci

relógio perfeito. O **deslocamento** em relação a uma hora específica t é $C_p(t) - t$.

Temporizadores reais não interrompem exatamente H vezes por segundo. Teoricamente, um temporizador com $H = 60$ deve gerar 216.000 ciclos por hora. Na prática, o erro relativo que se obtém com modernos chips temporizadores é de aproximadamente 10^{-5} , o que significa que determinada máquina pode obter um valor na faixa de 215.998 a 216.002 ciclos por hora. Mais exatamente, se existir alguma constante ρ tal que

$$1 - \rho \leq \frac{dC}{dt} \leq 1 + \rho$$

pode-se dizer que o temporizador está funcionando dentro de sua especificação. A constante ρ é especificada pelo fabricante e é conhecida como **taxa máxima de deriva**. Note que a taxa máxima de deriva especifica até que ponto a defasagem de um relógio pode chegar. Relógios adiantados, perfeitos e atrasados são mostrados na Figura 6.5.

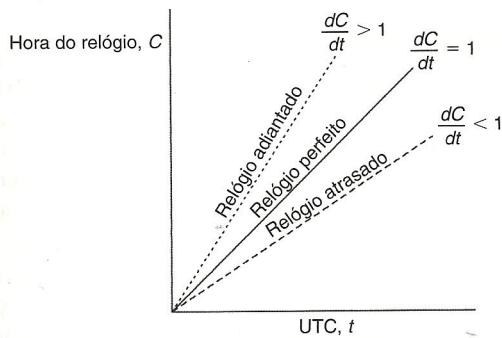


Figura 6.5 Relação entre a hora do relógio e a hora UTC quando as taxas de ciclos de relógios são diferentes.

Se dois relógios derivarem em direções opostas em relação à UTC, passado um tempo Δt após a sincronização entre os dois, eles podem apresentar uma defasagem de até $2\rho \Delta t$. Se os projetistas de sistemas operacionais quiserem garantir que a defasagem entre dois relógios nunca seja maior do que δ , eles devem ser sincronizados novamente (em software) no mínimo a cada $\delta/2\rho$ segundos. A diferença entre os vários algoritmos encontra-se exatamente na maneira como essa sincronização periódica é feita.

Protocolo de tempo de rede

Uma abordagem comum a muitos protocolos e proposta originalmente por Cristian (1989) é deixar que os clientes consultem um servidor de tempo. Este pode fornecer a hora corrente exata, por exemplo, porque está equipado com um receptor WWV ou um relógio de precisão. Claro que, então, o problema é que, quando se contacta o servidor, os atrasos de mensagens farão com que a hora fornecida esteja desatualizada. A solução é achar uma boa estimativa para esses atrasos. Considere a situação delineada na Figura 6.6.

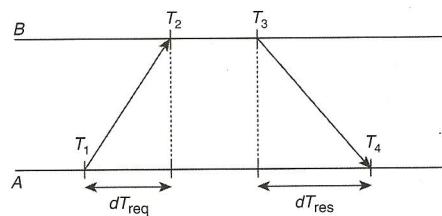


Figura 6.6 Obtenção da hora corrente por meio de um servidor de tempo.

Nesse caso, A enviará uma requisição a B , com uma marca de tempo cujo valor é T_1 . B , por sua vez, registrará a hora em que recebeu T_2 (obtida de seu próprio relógio local) e retornará uma resposta com uma marca de tempo de valor T_3 , enviando também o valor T_2 informado anteriormente. Por fim, A registra a hora da chegada da resposta, T_4 . Vamos supor que os atrasos de propagação de A até B sejam aproximadamente os mesmos que de B até A , o que significa que $T_2 - T_1 \approx T_4 - T_3$. Nesse caso, A pode estimar seu deslocamento em relação a B como

$$\theta = T_3 - \frac{(T_2 - T_1) + (T_4 - T_3)}{2} = \frac{(T_2 - T_1) + (T_3 - T_4)}{2}$$

Logicamente, o tempo não pode correr para trás. Se o relógio de A estiver adiantado, $\theta < 0$, significa que A deveria, em princípio, atrasar seu relógio. Isso não é permitido porque poderia causar sérios problemas, como um arquivo-objeto compilado logo após a alteração do relógio ter um horário anterior ao do arquivo-fonte que foi modificado um pouco antes da alteração do relógio.

Tal alteração deve ser introduzida gradativamente. Um modo de fazer isso é o seguinte: suponha que o temporizador esteja ajustado para gerar cem interrupções por segundo. Normalmente, cada interrupção somaria 10 m à hora. Para atrasar, a rotina de interrupção soma apenas 9 ms por vez, até que a correção tenha sido feita. De modo semelhante, o relógio pode ser adiantado gradativamente com a soma de 11 m a cada interrupção, em vez de adiantar tudo de uma vez só.

No caso do **protocolo de tempo de rede** (network time protocol — NTP), esse protocolo é ajustado entre pares de servidores. Em outras palavras, B também consultará A para saber qual é sua hora corrente. O deslocamento θ é calculado como mostrado antes, junto com a estimativa δ para o atraso:

$$\delta = \frac{(T_2 - T_1) + (T_4 - T_3)}{2}$$

Oito pares de valores (θ, δ) são armazenados em buffer, finalmente com a adoção do valor mínimo encontrado para δ como a melhor estimativa para o atraso entre os dois servidores e, na seqüência, o valor associado θ como a estimativa mais confiável do deslocamento.

A aplicação simétrica de NTP deveria, em princípio, também permitir que B ajustasse seu relógio com o de A . Contudo, se tivermos certeza de que a precisão do relógio de B é melhor, esse ajuste seria tolice. Para resolver esse problema, o NTP divide servidores em estratos. Um servidor que tenha um **relógio de referência** tal como um receptor WWV, ou um relógio atômico, é conhecido como **servidor do estrato 1** (diz-se que o relógio, em si, opera no estrato 0). Quando A contata B , só ajustará seu horário se seu próprio nível de estrato for mais alto do que o de B . Além do mais, após a sincronização, o nível de estrato de A se tornará uma unidade mais alta do que o de B . Em outras palavras, se B for um servidor de estrato k , então A se tornará um servidor de estrato $(k + 1)$ se seu nível de estrato original já era maior do que k . Devido à simetria do NTP, se o nível de estrato de A for *mais baixo* do que o de B , B se ajustará a A .

O NTP tem muitos aspectos importantes e muitos deles estão relacionados a identificar e mascarar erros, mas também a ataques contra a segurança. O NTP é descrito em Mills (1992) e sabe-se que alcança uma precisão, em âmbito mundial, na faixa de 1 a 50 ms. De início, sua mais nova versão (NTPv4) foi documentada somente por meio de sua implementação mas, agora, uma descrição detalhada pode ser encontrada em Mills (2006).

O algoritmo de Berkeley

Em muitos algoritmos como o NTP, o servidor de tempo é passivo. Outras máquinas lhe perguntam a hora periodicamente e ele se limita a responder a essas consultas. No Unix de Berkeley, é adotada a abordagem exatamente oposta (Gusella e Zatti, 1989). Nesse caso, o servidor de tempo (na verdade, um daemon de tempo) é ativo e consulta todas as máquinas de tempos em tempos para perguntar qual é a hora que cada uma está marcando. Com base nas respostas, ele calcula um horário médio e diz a todas as outras máquinas que adiantem seus relógios até o novo horário ou atrasem seus relógios até que tenham obtido alguma redução especificada. Esse método é adequado para um sistema no qual nenhuma máquina tenha receptor WWV. A hora do daemon de tempo tem de

ser ajustada manualmente pelo operador de tempos em tempos. O método é ilustrado na Figura 6.7.

Na Figura 6.7(a), às 3:00, o daemon de tempo informa a hora que ele próprio está marcando e pergunta qual é a hora que cada uma das outras máquinas está marcando. Na Figura 6.7(b), elas respondem informando o quanto estão adiantadas ou atrasadas em relação ao daemon. De posse desses números, o daemon de tempo calcula a hora média e informa a cada máquina como deve ajustar seu relógio [veja a Figura 6.7(c)].

Note que, para muitas finalidades, é suficiente que todas as máquinas concordem com a mesma hora. Não é essencial que essa hora também esteja de acordo com a hora real anunciada por rádio a cada hora. Se, em nosso exemplo da Figura 6.7, a hora marcada pelo relógio do daemon nunca fosse calibrada manualmente, não haveria dano nenhum contanto que nenhum dos outros nós se comunicasse com computadores externos. Todos concordariam alegremente com uma hora corrente, ainda que seu valor não tenha nenhuma relação com a realidade.

Sincronização de relógios em redes sem fio

Uma importante vantagem dos sistemas distribuídos mais tradicionais é que podemos disponibilizar servidores de tempo com facilidade e eficiência. Além disso, a maioria das máquinas pode entrar em contato umas com as outras, o que permite uma disseminação de informações relativamente simples. Essas premissas deixam de ser válidas em redes sem fio, em particular, redes de sensores. Os nós são restritos em relação a recursos, e o roteamento por múltiplos saltos é caro. Além disso, muitas vezes é importante otimizar algoritmos para consumo de energia. Essas e outras observações resultaram no projeto de algoritmos de sincronização de relógios muito diferentes para redes sem fio. A seguir, consideraremos uma solução específica. Sivrikaya e Yener (2004) dão uma breve visão geral de outras soluções. Um levantamento extensivo pode ser encontrado em Sundararaman et al. (2005).

Sincronização em broadcast de referência (reference broadcast synchronization — RBS) é um protocolo de sincronização de relógios muito diferente de outras propos-

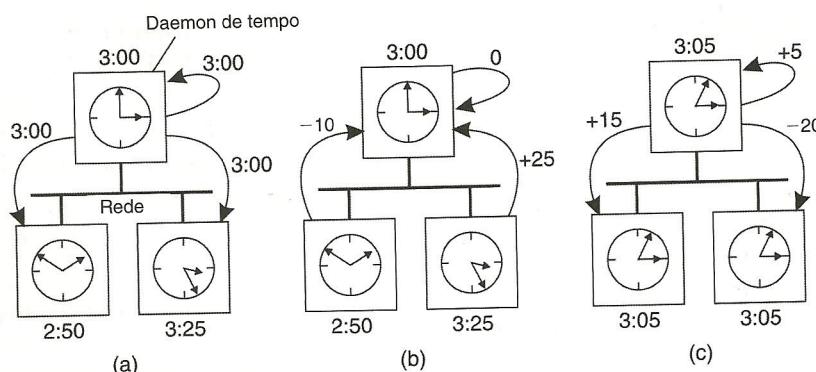


Figura 6.7 (a) O daemon de tempo pergunta a todas as outras máquinas os valores marcados por seus relógios.
(b) As máquinas respondem. (c) O daemon de tempo informa a todas como devem ajustar seus relógios.

tas (Elson et al., 2002). Em primeiro lugar, o protocolo não adota como premissa que há um único nó que tenha disponível um valor exato da hora real. Em vez de visar a dar a todos os nós a hora UTC, ele visa à mera sincronização dos relógios em âmbito interno, exatamente como o algoritmo de Berkeley. Em segundo lugar, as soluções que discutimos até agora são projetadas para colocar o remetente e o receptor em sincronia seguindo, em essência, um protocolo de duas vias. O RBS se desvia desse padrão permitindo que somente os receptores sincronizem, mantendo o remetente fora do laço.

Em RBS, um remetente transmite uma mensagem de referência em broadcast que permitirá a seus receptores ajustar os relógios. Uma observação fundamental é que, em uma rede de sensores, o tempo para propagar um sinal para outros nós é aproximadamente constante, contanto que não seja adotada nenhuma premissa de roteamento por múltiplos saltos. Nesse caso, o tempo de propagação é medido desde o momento em que a mensagem sai da interface de rede do remetente. Em decorrência, duas importantes fontes de variação presentes em transferência de mensagens deixam de desempenhar um papel na estimativa de atrasos: o tempo gasto para construir a mensagem e o tempo gasto para acessar a rede. Esse princípio é mostrado na Figura 6.8.

Note que, em protocolos como o NTP, uma marca de tempo é adicionada à mensagem antes de ela ser passada para a interface de rede. Além do mais, como redes sem fio são baseadas em um protocolo de contenção, de modo geral não há como dizer quanto tempo levará até que uma mensagem possa ser realmente transmitida. Esses fatores não determinísticos são eliminados em RBS. O que permanece é o tempo de entrega no receptor, mas esse tempo varia consideravelmente menos do que o tempo de acesso à rede.

A idéia subjacente ao RBS é simples: quando um nó transmite em broadcast uma mensagem de referência m , cada nó p simplesmente registra a hora $T_{p,m}$ em que recebeu m . Note que $T_{p,m}$ é lida no relógio local de p . Ignorando a defasagem de relógio, dois nós, p e q , podem tro-

car seus respectivos horários de entrega de modo a estimar o deslocamento relativo entre eles:

$$\text{Deslocamento } [p,q] = \frac{\sum_{k=1}^M (T_{p,k} - T_{q,k})}{M}$$

onde M é o número total de mensagens de referência enviadas. Essa informação é importante: o nó p saberá o valor do relógio de q em relação a seu próprio valor. Além do mais, se ele simplesmente armazenar esses deslocamentos, não há necessidade de ajustar seu próprio relógio, o que economiza energia.

Infelizmente, os relógios podem derivar um do outro. O efeito é que o simples cálculo do deslocamento médio como foi feito anteriormente não funcionará: os últimos valores enviados serão menos exatos do que os primeiros. Ademais, à medida que o tempo passa, o deslocamento presumivelmente aumenta. Elson et al. usam um algoritmo muito simples para compensar esse efeito: em vez de calcular a média, eles aplicam regressão linear padrão para calcular o deslocamento como uma função:

$$\text{Deslocamento } [p,q](t) = \alpha t + \beta$$

As constantes α e β são calculadas pelos pares $(T_{p,k}, T_{q,k})$. Essa nova forma permitirá um cálculo muito mais preciso do valor corrente do relógio de q pelo nó p e vice-versa.

6.2 Relógios Lógicos

Até aqui, consideramos que a sincronização de relógios está naturalmente relacionada com a hora real. Todavia, também vimos que pode ser suficiente que cada nó concorde com uma hora corrente, sem que essa hora seja a mesma que a hora real. Podemos avançar mais um passo. Para executar *make*, por exemplo, é adequado que dois nós concordem que *input.o* seja desatualizado por uma nova versão de *input.c*. Nesse caso, monitorar os eventos de cada um (tal como a produção

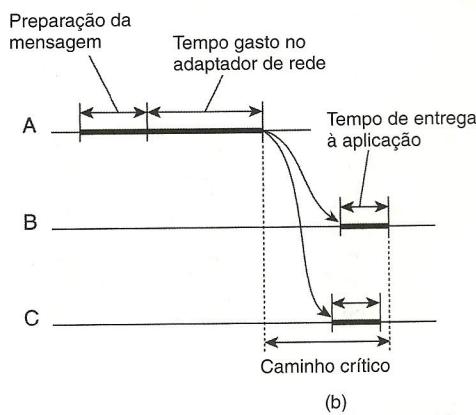
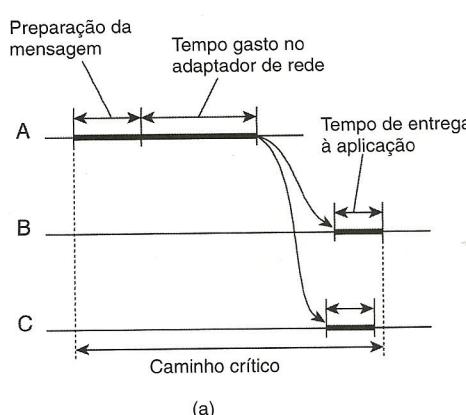


Figura 6.8 (a) Caminho crítico usual na determinação de atrasos de rede. (b) Caminho crítico no caso de RBS.

de uma nova versão de *input.c*) é o que importa. No caso desses algoritmos, o convencional é denominar esses relógios como **relógios lógicos**.

Em um artigo clássico, Lamport (1978) mostrou que, embora a sincronização entre relógios seja possível, ela não precisa ser absoluta. Se dois processos não interagirem, não é necessário que seus relógios sejam sincronizados porque a falta de sincronização não seria observável e, portanto, não poderia causar problemas. Além do mais, ele destacou que, de modo geral, o que importa não é que todos os processos concordem com a hora exata, mas com a ordem em que os eventos ocorrem. No exemplo do *make*, o que conta é se *input.c* é mais velho ou mais novo que *input.o*, e não a hora exata em que foram criados.

Nesta seção, discutiremos o algoritmo de Lamport, que sincroniza relógios lógicos. Ademais, discutiremos uma extensão da abordagem de Lamport, denominada marcas de tempo vetoriais.

6.2.1 Relógios lógicos de Lamport

Para sincronizar relógios lógicos, Lamport definiu uma relação denominada **acontece antes**. A expressão $a \rightarrow b$ é lida como ‘*a* acontece antes de *b*’ e significa que todos os processos concordam que primeiro ocorre um evento *a* e, depois, um evento *b*. A relação ‘acontece antes’ pode ser observada diretamente em duas situações:

1. Se *a* e *b* são eventos do mesmo processo, e *a* ocorre antes de *b*, então $a \rightarrow b$ é verdadeira.
2. Se *a* é o evento de uma mensagem sendo enviada por um processo, e *b* é o evento da mensagem sendo recebida por um outro processo, então $a \rightarrow b$ também é verdadeira. Uma mensagem não pode ser recebida antes de ser enviada, ou até ao mesmo tempo que é enviada, visto que ela leva uma quantidade de tempo finita, diferente de zero, para chegar.

A relação ‘acontece antes’ é transitiva, portanto se $a \rightarrow b$ e $b \rightarrow c$, então $a \rightarrow c$. Se dois eventos, *x* e *y*, acontecem em processos diferentes que não trocam mensagens (nem mesmo indiretamente via terceiros), então $x \rightarrow y$ não é verdadeira, mas $y \rightarrow x$ também não é. Diz-se que esses eventos são **concorrentes**, o que significa, apenas, que nada pode ser dito (ou nem precisa ser dito) sobre quando os eventos aconteceram ou qual evento aconteceu em primeiro lugar.

O que precisamos é um modo de medir uma noção de tempo tal que, para cada evento *a*, possamos designar um valor de tempo $C(a)$ com o qual todos os processos concordam. Esses valores de tempo devem ter a propriedade de se $a \rightarrow b$, então $C(a) < C(b)$. Expressando em outras palavras as condições que declaramos antes, se *a* e *b* são dois eventos dentro de um mesmo processo, e *a* ocorre antes de *b*, então $C(a) < C(b)$. De modo semelhante, se *a* é o envio de uma mensagem por um processo e *b* é o recebimento dessa mensagem por um outro processo, então $C(a)$ e $C(b)$ devem ser atribuídos de tal maneira que todos concordem com os valores de $C(a)$ e $C(b)$ sendo $C(a) < C(b)$. Além disso, o tempo de relógio, C , deve sempre correr para a frente (aumentar), nunca para trás (diminuir). Os tempos podem ser corrigidos pela adição de um valor positivo, nunca por subtração.

Agora, vamos examinar o algoritmo de Lamport proposto para designar tempo a eventos. Considere os três processos representados na Figura 6.9(a). Os processos executam em máquinas diferentes, cada uma com seu próprio relógio, que funciona a sua própria velocidade. Como podemos ver na figura, quando o relógio pulsa 6 vezes no processo P_1 , pulsou 8 vezes no processo P_2 e 10 vezes no processo P_3 . Cada relógio funciona a uma taxa constante, mas as taxas são diferentes devido às diferenças nos cristais.

No tempo 6, o processo P_1 envia a mensagem m_1 ao processo P_2 . O tempo que essa mensagem leva para chegar depende do relógio no qual você se baseia. Seja como for, o relógio no processo P_2 marca 16 quando a mensa-

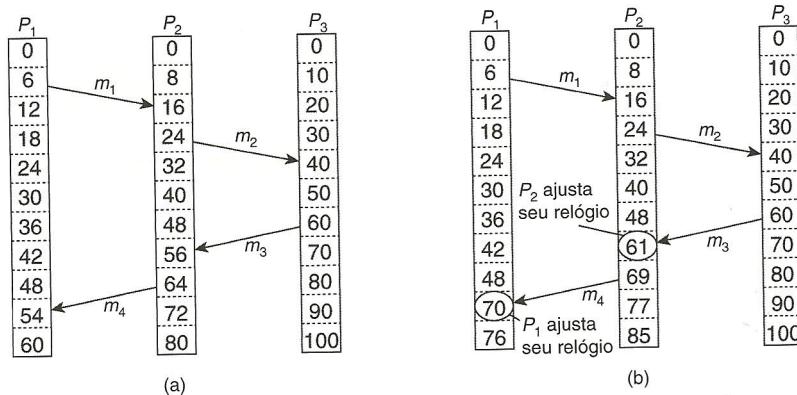


Figura 6.9 (a) Três processos, cada um com seu próprio relógio. Os relógios funcionam a taxas diferentes.
(b) O algoritmo de Lamport corrige os relógios.

gem chega. Se a mensagem transportar com ela o tempo de início, 6, o processo P_2 concluirá que ela levou 10 pulsos para fazer sua jornada. Esse valor certamente é coerente. Segundo esse raciocínio, a mensagem m_2 de P_2 a P_3 leva 16 pulsos, novamente um valor plausível.

Agora considere a mensagem m_3 . Ela sai do processo P_3 em 60 e chega em P_2 em 56. Da mesma maneira, a mensagem m_4 de P_2 a P_1 sai em 64 e chega em 54. Esses valores são claramente implausíveis. É essa situação que deve ser evitada.

A solução de Lamport resulta diretamente da relação ‘acontece antes’. Visto que m_3 saiu em 60, ela deve chegar em 61 ou mais tarde. Portanto, cada mensagem transporta o tempo de envio conforme o relógio do remetente. Quando uma mensagem chega e o relógio do receptor mostra um valor anterior ao tempo em que a mensagem foi enviada, o receptor adianta seu relógio para ficar uma unidade a mais do tempo de envio. Na Figura 6.9(b) vemos que, agora, m_3 chega em 61. De modo semelhante, m_4 chega em 70.

Para preparar nossa discussão sobre relógios vetoriais, vamos formular esse procedimento com mais precisão. Nesse ponto, é importante distinguir três camadas diferentes de software, como já tínhamos encontrado no Capítulo 1: a rede, uma camada de middleware e uma camada de aplicação, como mostra a Figura 6.10. O que apresentamos a seguir é típico de uma camada de middleware.

Para implementar relógios lógicos de Lamport, cada processo P_i mantém um contador local C_i . Esses contadores são atualizados conforme as etapas apresentadas a seguir (Raynal e Singhal, 1996):

1. Antes de executar um evento (isto é, enviar uma mensagem pela rede, entregar uma mensagem a uma aplicação, ou qualquer outro evento interno), P_i executa $C_i \leftarrow C_i + 1$.
2. Quando o processo P_i envia uma mensagem m a P_j , ajusta a marca de tempo de m , $ts(m)$, para igual a C_i após ter executado a etapa anterior.
3. Ao receber uma mensagem m , o processo P_j ajusta seu próprio contador local para $C_j \leftarrow \max\{C_j, ts(m)\}$ e, depois disso, executa a primeira etapa e entrega a mensagem à aplicação.

Em algumas situações, é desejável um requisito adicional: dois eventos nunca, jamais, ocorrem exatamente ao mesmo tempo. Para atingir esse objetivo, podemos anexar o número do processo no qual o evento ocorre à extrema- dade menos significativa do tempo, separado por um ponto decimal. Por exemplo, um evento que ocorreu no tempo 40 no processo P_i receberá a marca de tempo 40.i.

Note que, designando ao evento o tempo $C(a) \leftarrow C_i(a)$ se a acontecer no processo P_i no tempo $C_i(a)$, temos uma implementação distribuída do valor do tempo global que procurávamos desde o início.

Exemplo: Multicast totalmente ordenado

Como uma aplicação de relógios lógicos de Lamport, considere a situação em que um banco de dados foi replicado em vários sites. Por exemplo, para melhorar o desempenho de consulta, um banco pode colocar cópias de um banco de dados de contas correntes em duas cidades diferentes, digamos, Nova York e San Francisco. Uma consulta é sempre repassada para a cópia mais próxima. O preço de uma resposta rápida a uma consulta é pago, em parte, por custos mais altos de atualização, porque cada operação de atualização deve ser executada em cada réplica.

Na verdade, há um requisito mais restritivo no que diz respeito a atualizações. Suponha que um cliente em San Francisco queira depositar \$ 100 em sua conta que, no instante em questão, contém \$ 1.000. Ao mesmo tempo, um funcionário do banco em Nova York inicia uma atualização pela qual a conta do cliente recebe o acréscimo de 1% de juros. Ambas as atualizações devem ser executadas em ambas as cópias do banco de dados. Contudo, devido a atrasos de comunicação na rede subjacente, as atualizações podem chegar na ordem em que mostra a Figura 6.11.

A atualização da operação do cliente é realizada em San Francisco antes da atualização do lançamento de juros. Ao contrário, a cópia da conta na réplica de Nova York é primeiramente atualizada com o 1% de juros e depois com o depósito de \$ 100. Por consequência, o banco de dados de San Francisco registrará uma quantia total de \$ 1.111, ao passo que o banco de dados de Nova York registrará \$ 1.110.

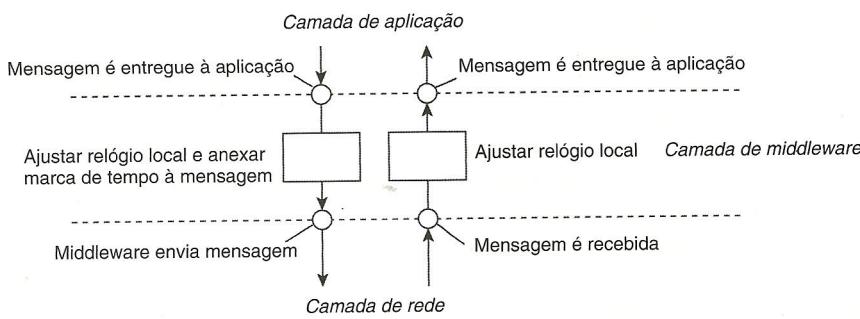


Figura 6.10 Posicionamento de relógios lógicos de Lamport em sistemas distribuídos.

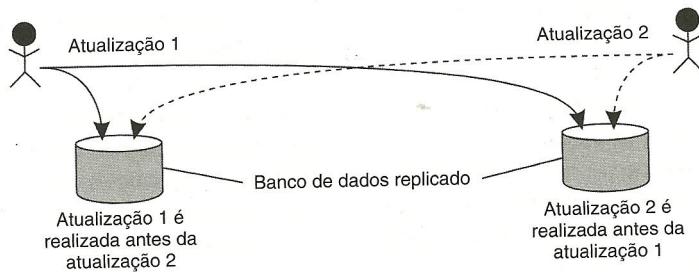


Figura 6.11 Atualização de banco de dados replicado que o deixa em estado inconsistente.

O problema que enfrentamos é que as duas operações de atualização deveriam ter sido executadas na mesma ordem em cada cópia. Embora faça diferença se o depósito é processado antes ou depois da atualização do depósito dos juros ou ao contrário, a ordem seguida não é importante do ponto de vista de consistência. A questão importante é que ambas as cópias devem ser exatamente as mesmas. Em geral, situações como essas requerem um **multicast totalmente ordenado**, isto é, uma operação multicast pela qual todas as mensagens são entregues na mesma ordem a cada receptor. Os relógios lógicos de Lamport podem ser usados para implementar multicast totalmente ordenado de modo completamente distribuído.

Considere um grupo de processos que enviam mensagens multicast uns aos outros. Cada mensagem sempre transportará a marca de tempo correspondente ao tempo (lógico) corrente de seu remetente. Quando uma mensagem é enviada em multicast, ela é conceitualmente também enviada ao remetente. Além disso, consideramos que mensagens do mesmo remetente são recebidas na ordem em que foram enviadas e que nenhuma mensagem foi perdida.

Quando um processo recebe uma mensagem, ela é colocada em uma fila de cache local, ordenada conforme sua marca de tempo. O receptor envia mensagens multicast de reconhecimento aos outros processos. Note que, se seguirmos o algoritmo de Lamport para ajustar relógios locais, a marca de tempo da mensagem recebida é mais baixa do que a marca de tempo da mensagem de reconhecimento. O aspecto interessante dessa abordagem é que, a certa altura, todos os processos terão a mesma cópia da fila local (contanto que nenhuma mensagem seja removida).

Um processo só pode entregar uma mensagem enfileirada à aplicação que ele estiver executando quando essa mensagem estiver no início da fila e tiver sido reconhecida por cada um dos outros processos. Nesse ponto, a mensagem é retirada da fila e entregue à aplicação; os reconhecimentos associados podem ser simplesmente removidos. Como cada processo tem a mesma cópia da fila, todas as mensagens são entregues na mesma ordem em todos os lugares. Em outras palavras, estabelecemos multicast totalmente ordenado.

Como veremos em capítulos posteriores, o multicast totalmente ordenado é um veículo importante para serviços replicados nos quais a consistência entre as

réplicas é mantida permitindo que elas executem as mesmas operações na mesma ordem em todos os lugares. Como as réplicas seguem, em essência, as mesmas transições na mesma máquina de estado finito, essa operação também é conhecida como **replicação de estado de máquina** (Schneider, 1990).

6.2.2 Relógios vetoriais

Relógios lógicos de Lamport resultam em uma situação em que todos os eventos em um sistema distribuído são totalmente ordenados e têm a seguinte propriedade: se o evento a aconteceu antes do evento b , a também será posicionado nessa ordem antes de b , isto é, $C(a) < C(b)$.

Contudo, com relógios de Lamport, nada se pode dizer sobre a relação entre dois eventos, a e b , pela mera comparação entre seus valores de tempo, $C(a)$ e $C(b)$, respectivamente. Em outras palavras, se $C(a) < C(b)$, isso não implica necessariamente que a realmente ocorreu antes de b . É preciso algo mais para fazer isso.

Como explicação, considere as mensagens enviadas pelos três processos mostrados na Figura 6.12. Denote por $T_{snd}(m_i)$ o instante lógico em que a mensagem m_i foi enviada e, da mesma maneira, por $T_{rcv}(m_i)$ o instante em que ela foi recebida. Por dedução, sabemos que, para cada mensagem, $T_{snd}(m_i) < T_{rcv}(m_i)$. Mas, de modo geral, o que podemos concluir de $T_{rcv}(m_i) < T_{snd}(m_j)$?

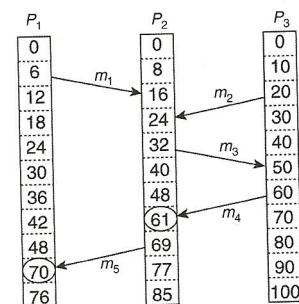


Figura 6.12 Transmissão de mensagens concorrentes com utilização de relógios lógicos.

No caso em que $m_i = m_1$ e $m_j = m_3$, sabemos que esses valores correspondem a eventos que ocorreram no processo P_2 , o que significa que m_3 foi, de fato, enviada após o recebimento da mensagem m_1 . Isso pode indicar que o

envio da mensagem m_3 dependeu do que foi recebido por meio da mensagem m_1 . Contudo, sabemos também que $T_{rcv}(m_1) < T_{snd}(m_2)$; entretanto, o envio de m_2 nada tem a ver com o recebimento de m_1 .

O problema é que os relógios de Lamport não capturam **causalidade**. Causalidade pode ser capturada por meio de **relógios vetoriais**. Um relógio vetorial $VC(a)$ designado a um evento a tem a seguinte propriedade: se $VC(a) < VC(b)$ para algum evento b , sabe-se que o evento a precede por causalidade o evento b . Relógios vetoriais são construídos de modo que permitam a cada processo P_i manter um vetor VC_i com as duas propriedades seguintes:

1. $VC_i[i]$ é o número de eventos que ocorreram em P_i até o instante em questão. Em outras palavras, $VC_i[i]$ é o relógio lógico local no processo P_i .
2. Se $VC_i[j] = k$, então P_i sabe que k eventos ocorreram em P_j . Portanto, P_i conhece o tempo local em P_j .

A primeira propriedade é mantida incrementando $VC_i[i]$ na ocorrência de cada novo evento que ocorrer no processo P_i . A segunda propriedade é mantida por meio das caronas que os vetores pegam com as mensagens que são enviadas. Em particular, ocorrem as seguintes etapas:

1. Antes de executar um evento (isto é, enviar uma mensagem pela rede, entregar uma mensagem a uma aplicação ou qualquer outro evento interno), P_i executa $VC_i[i] \leftarrow VC_i[i] + 1$.
2. Quando o processo P_i envia uma mensagem m a P_j , ele iguala a marca de tempo (vetorial) de m , $ts(m)$, à marca de tempo de VC_i , após ter executado a etapa anterior.
3. Ao receber uma mensagem m , o processo P_j ajusta seu próprio vetor fixando $VC_j[k] \leftarrow \max\{VC_j[k], ts(m)[k]\}$ para cada k ; em seguida, executa a primeira etapa e entrega a mensagem à aplicação.

Note que, se a marca de tempo de um evento a for $ts(a)$, então $ts(a)[i] - 1$ é o número de eventos processados em P_i que precedem a por causalidade. Em consequência, quando P_j recebe uma mensagem de P_i com marca de tempo $ts(m)$, ele sabe o número de eventos que ocorreram em P_i e que precedem por causalidade o envio de m . Porém, o mais importante é que P_j também é informado de quantos eventos ocorreram em outros processos, antes de P_i enviar a mensagem m . Em outras palavras, a marca de tempo $ts(m)$ informa ao receptor quantos eventos ocorreram em outros processos antes do envio de m e dos quais m pode depender por causalidade.

Imposição de comunicação causal

Com o uso de relógios vetoriais, agora é possível garantir que uma mensagem seja entregue somente se todas as mensagens que a precederem por causalidade

também tenham sido recebidas. Para habilitar tal esquema, consideraremos que as mensagens são transmitidas em multicast dentro de um grupo de processos. Note que esse **multicast ordenado por causalidade** é mais fraco do que o multicast totalmente ordenado que discutimos antes. Especificamente, se duas mensagens não estiverem relacionadas uma com a outra de modo nenhum, não nos importaremos com a ordem em que elas são entregues às aplicações. Elas podem até mesmo ser entregues em ordem diferente e em localizações diferentes.

Além do mais, consideraremos que os relógios só são ajustados quando enviam e recebem mensagens. Em particular, ao enviar uma mensagem, o processo P_i só incrementará $VC_i[i]$ de 1. Quando receber uma mensagem m com marca de tempo $ts(m)$, ele só ajustará $VC_i[k]$ para $\max\{VC_i[k], ts(m)[k]\}$ para cada k .

Agora, suponha que P_j recebe de P_i uma mensagem m com marca de tempo (vetorial) $ts(m)$. Sendo assim, a entrega da mensagem à camada de aplicação será atrasada até que as duas condições seguintes sejam cumpridas:

1. $ts(m)[i] = VC_j[i] + 1$
2. $ts(m)[k] \leq VC_j[k]$ para todo $k \neq i$

A primeira condição afirma que m é a próxima mensagem que P_j estava esperando do processo P_i . A segunda condição afirma que P_j viu todas as mensagens que foram vistas por P_i quando este enviou a mensagem m . Note que o processo P_j não precisa atrasar a entrega de suas próprias mensagens.

Como exemplo, considere três processos, P_0 , P_1 e P_2 , como mostra a Figura 6.13. No tempo local $(1,0,0)$, P_0 envia a mensagem m aos outros dois processos. Após o recebimento dessa mensagem por P_1 , este decide enviar m^* , que chega a P_2 mais cedo do que m . Nesse ponto, a entrega de m^* é atrasada por P_2 até que m tenha sido recebida e entregue à camada de aplicação de P_2 .

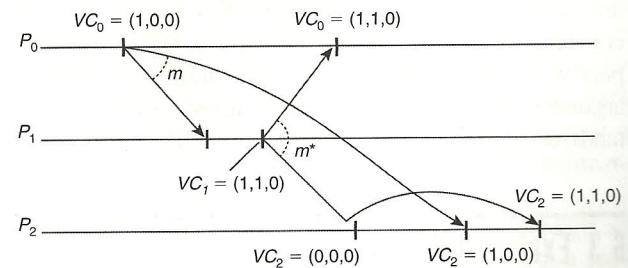


Figura 6.13 Imposição de comunicação causal.

Observação sobre entrega ordenada de mensagens

Alguns sistemas de middleware, em particular o Isis e seu sucessor, Horus (Birman e Van Renesse, 1994), fornecem suporte para multicast totalmente ordenado e multicast (confiável) ordenado por causalidade. Há certa con-

trovésia sobre se tal suporte deveria ser fornecido como parte da camada de comunicação de mensagens ou se as aplicações deveriam se encarregar da ordenação (veja, por exemplo, Cheriton e Skeen, 1993; Birman, 1994). A questão ainda não foi dirimida, porém o mais importante é que os argumentos são válidos ainda hoje.

Há dois problemas principais quando se permite que o middleware se encarregue da ordenação de mensagens. Em primeiro lugar, como o middleware não pode dizer o que uma mensagem realmente contém, só é possível capturar causalidade *potencial*. Por exemplo, duas mensagens completamente independentes enviadas pelo mesmo remetente sempre serão marcadas como relacionadas por causalidade pela camada de middleware. Essa abordagem é excessivamente restritiva e pode resultar em problemas de eficiência.

Um segundo problema é que nem toda causalidade pode ser capturada. Considere um painel eletrônico de mensagens e suponha que Alice apresente um texto. Se, em seguida, ela telefonar para Bob e contar o que acabou de escrever, Bob pode apresentar um outro texto como resposta sem ter visto o que Alice apresentou no painel. Em outras palavras, existe uma causalidade entre as apresentações de Bob e de Alice devida à comunicação *externa*. Essa causalidade não é capturada pelo sistema de painel eletrônico de mensagens.

Em essência, questões de ordenação, assim como muitas outras questões de comunicação específicas de aplicação, podem ser adequadamente resolvidas ao se examinar a aplicação com a qual está ocorrendo a comunicação de mensagens, o que também é conhecido como **argumento fim-a-fim** em projeto de sistemas (Saltzer et al., 1984). Uma desvantagem de ter somente soluções no nível de aplicação é que um desenvolvedor é forçado a se concentrar em questões que não estão relacionadas imediatamente com a funcionalidade central da aplicação. Por exemplo, a ordenação pode não ser o problema mais importante no desenvolvimento de um sistema de troca de mensagens em um painel eletrônico de mensagens. Nesse caso, pode ser conveniente ter uma camada de comunicação subjacente para tratar da ordenação. Vamos encontrar várias vezes o argumento fim-a-fim, em particular quando estivermos tratando de segurança em sistemas distribuídos.

6.3 Exclusão Mútua

Uma questão fundamental em sistemas distribuídos é a concorrência e a colaboração entre vários processos. Em muitos casos, isso também significa que processos vão precisar acessar simultaneamente os mesmos recursos. Para evitar que tais acessos concorrentes corrompam o recurso ou o tornem inconsistente, são necessárias soluções que garantam acesso mutuamente exclusivo pelos processos. Nesta seção, estudaremos alguns dos mais importantes algoritmos distribuídos propostos. Um levan-

tamento de algoritmos distribuídos para exclusão mútua é dado por Saxena e Rai (2003). Mais antigo, porém ainda relevante, é o de Velazquez (1993).

6.3.1 Visão geral

Algoritmos distribuídos de exclusão mútua podem ser classificados em duas categorias diferentes. Em **soluções baseadas em ficha**, consegue-se a exclusão mútua com a passagem de uma mensagem especial entre os processos, conhecida como **ficha**. Há só uma ficha disponível, e quem quer que a tenha pode acessar o recurso compartilhado. Ao terminar, a ficha é passada adiante para o processo seguinte. Se um processo que tenha a ficha não estiver interessado em acessar o recurso, ele apenas a passa adiante.

Soluções baseadas em ficha têm algumas propriedades importantes. Em primeiro lugar, dependendo do modo como os processos são organizados, elas podem garantir, com razoável facilidade, que todo processo terá oportunidade de acessar o recurso. Em outras palavras, a ficha evita a **inanição**. Em segundo lugar, também fica fácil evitar **deadlocks**, isto é, que vários processos fiquem esperando uns pelos outros para prosseguir, o que contribui para a otimização do processo. Infelizmente, a principal desvantagem de soluções baseadas em fichas é bastante séria: quando a ficha se perde (por exemplo, porque o processo que a detém falhou), é preciso iniciar um complicado procedimento distribuído para assegurar a criação de uma nova ficha, porém, acima de tudo, essa deve ser a única ficha.

Como alternativa, muitos algoritmos distribuídos de exclusão mútua seguem uma **abordagem baseada em permissão**. Nesse caso, um processo que quiser acessar o recurso em primeiro lugar solicita a permissão de outros processos. Há diferentes modos de conceder tal permissão; a seguir, vamos considerar alguns deles.

6.3.2 Algoritmo centralizado

O modo mais direto de conseguir exclusão mútua em um sistema distribuído é simular como ela é feita em um sistema monoprocessador. Um processo é eleito como o coordenador. Sempre que um processo quiser acessar um recurso compartilhado, envia uma mensagem de requisição ao coordenador declarando qual recurso quer acessar e solicitando permissão. Se nenhum outro processo estiver acessando aquele recurso naquele momento, o coordenador devolve uma resposta concedendo a permissão, como mostra a Figura 6.14(a). Quando a resposta chega, o processo requisitante pode seguir adiante.

Agora, suponha que um outro processo, 2 na Figura 6.14(b), peça permissão para acessar o recurso. O coordenador sabe que um outro processo diferente já está utilizando o recurso, portanto ele não pode dar a permissão. O método exato usado para negar permissão é dependente do sistema. Na Figura 6.14(b), o coordenador apenas se

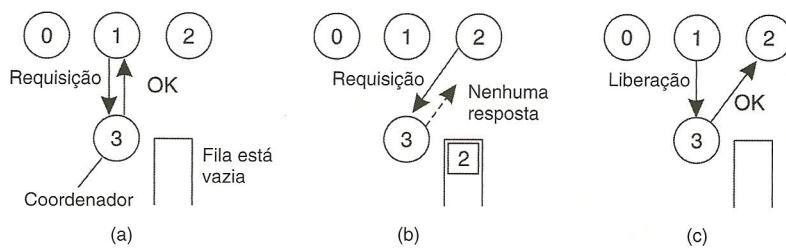


Figura 6.14 (a) O processo 1 solicita ao coordenador permissão para acessar um recurso compartilhado. A permissão é concedida. (b) Depois, o processo 2 solicita permissão para acessar o mesmo recurso. O coordenador não responde. (c) Quando o processo 1 libera o recurso, informa ao coordenador, que então responde a 2.

abstém de responder e, com isso, bloqueia o processo 2 que está esperando por uma resposta. Como alternativa, ele poderia enviar uma resposta dizendo “permissão negada”. Seja como for, por enquanto ele coloca a requisição de 2 na fila e espera mais mensagens.

Quando o processo 1 conclui a utilização do recurso, envia uma mensagem ao coordenador a fim de liberar seu acesso exclusivo, como mostra a Figura 6.14(c). O coordenador retira o primeiro item da fila de requisições adiadas e envia ao processo requisitante uma mensagem de concessão de permissão. Se o processo ainda estiver bloqueado — isto é, se essa for a primeira mensagem para ele —, ele desbloqueia e acessa o recurso. Se uma mensagem explícita que negue a permissão já tiver sido enviada, o processo terá de sondar o tráfego de entrada ou bloquear mais tarde. Seja como for, quando ele vir a concessão de permissão, também pode continuar.

É fácil ver que o algoritmo garante exclusão mútua: o coordenador só permite o acesso de um processo por vez ao recurso. Além disso, ele também é justo, visto que as permissões são concedidas na ordem em que as requisições foram recebidas. Nenhum processo jamais esperará para sempre (não há inanição). O esquema também é fácil de implementar e requer somente três mensagens por utilização de recurso (requisição, concessão, liberação). Sua simplicidade o torna uma solução atraente para muitas situações práticas.

A abordagem centralizada também tem deficiências. O coordenador é um ponto de falha único, portanto, se ele falhar, todo o sistema pode cair. Se os processos normalmente bloquearem após emitir uma requisição, não podem distinguir um coordenador inativo de uma ‘permissão negada’, visto que, em ambos os casos, nenhuma mensagem volta. Além disso, em um sistema de grande porte, um coordenador único pode se tornar um gargalo de desempenho. Ainda assim, em muitos casos, os benefícios proporcionados por sua simplicidade compensam as desvantagens potenciais. Entretanto, soluções distribuídas não são necessariamente as melhores, como ilustra nosso exemplo seguinte.

6.3.3 Algoritmo descentralizado

Ter um único coordenador costuma ser uma abordagem ruim. Vamos estudar uma solução totalmente descentralizada. Lin et al. (2004) propõem usar um algoritmo de votação que pode ser executado usando um sistema baseado em DHT. Em essência, a solução desses algoritmos amplia o coordenador central da maneira que explicaremos a seguir. Adota-se como premissa que cada recurso é replicado n vezes. Toda réplica tem seu próprio coordenador para controlar o acesso por processos concorrentes.

Todavia, sempre que um processo quiser acessar o recurso, ele vai precisar apenas obter um voto majoritário de $m > n/2$ coordenadores. Diferente do esquema centralizado que acabamos de discutir, consideraremos que, quando um coordenador não der permissão para acessar um recurso (o que fará quando tiver concedido permissão a um outro processo), ele informará ao requisitante.

Em essência, esse esquema torna a solução centralizada original menos vulnerável a falhas de um único coordenador. A premissa é que, quando um coordenador falhar, ele se recuperará rapidamente, mas esquecerá qualquer voto que tenha dado antes de falhar. Um outro modo de entender isso é que o próprio coordenador reiniçia a si mesmo em momentos arbitrários. O risco que estamos correndo é que um reinício fará o coordenador esquecer que, antes, já tinha concedido permissão para algum processo acessar o recurso. Em decorrência, após o reinício ele poderá conceder essa mesma permissão incorretamente, mais uma vez, a um outro processo.

Seja p a probabilidade de que um coordenador se reinicie durante um intervalo de tempo Δt . Então, a probabilidade $P[k]$ de que k entre m coordenadores se reiniciem durante o mesmo intervalo é

$$P[k] = \binom{m}{k} p^k (1-p)^{m-k}$$

Dado que no mínimo $2m - n$ coordenadores precisam se reiniciar para violar a correção do mecanismo de votação, a probabilidade de que tal violação ocorra é $\sum_{k=2m-n}^n P[k]$. Para termos uma idéia do que isso poderia significar, suponha que estejamos lidando com um sis-

tema baseado em DHT do qual cada nó participa durante aproximadamente três horas seguidas. Seja Δt 10 segundos, o que é considerado um valor conservador para um único processo querer acessar um recurso compartilhado. (Para alocações muito longas são necessários mecanismos diferentes.) Com $n = 32$ e $m = 0,75n$, a probabilidade de violação da correção é menos do que 10^{-40} . Essa probabilidade com certeza é menor do que a disponibilidade de qualquer recurso.

Para implementar esse esquema, Lin et al. (2004) usam um sistema baseado em DHT no qual um recurso é replicado n vezes. Considere que o recurso é conhecido sob seu nome exclusivo *rname*. Por conseguinte, podemos supor que o nome da i -ésima réplica é *rname-i*, que então é usada para calcular uma única chave por meio de uma função de hash conhecida. Em decorrência, todo processo pode gerar as n chaves dado o nome de um recurso e, na seqüência, consultar cada nó responsável por uma réplica (e controlar o acesso a essa réplica).

Se a permissão para acessar o recurso for negada, isto é, um processo obtiver menos do que m votos, considera-se que ele desistirá durante um período de tempo escolhido aleatoriamente e fará nova tentativa mais tarde. O problema desse esquema é que, se muitos nós quiserem acessar o mesmo recurso, a utilização decresce rapidamente. Em outras palavras, haverá tantos nós competindo para obter acesso que, a certa altura, nenhum deles conseguirá votos suficientes, e o recurso deixará de ser utilizado. Uma solução para resolver esse problema pode ser encontrada em Lin et al. (2004).

6.3.4 Algoritmo distribuído

Para muitos, ter um algoritmo correto segundo as leis da probabilidade não é bom o bastante. Portanto, os pesquisadores procuraram algoritmos distribuídos determinísticos de exclusão mútua. O artigo sobre sincronização de relógios publicado por Lamport em 1978 apresentou o primeiro. Ricart e Agrawala (1981) o tornaram mais eficiente. Nesta seção, descreveremos o método desses autores.

O algoritmo de Ricart e Agrawala requer que haja uma ordenação total de todos os eventos no sistema. Isto é, para qualquer par de eventos, como mensagens, não pode haver ambigüidade sobre qual realmente aconteceu em primeiro lugar. O algoritmo de Lamport apresentado na Subseção 6.2.1 é um modo de conseguir essa ordenação e pode ser usado para fornecer marcas de tempo para exclusão mútua distribuída.

O algoritmo funciona como descreveremos a seguir. Quando um processo quer acessar um recurso compartilhado, monta uma mensagem que contém o nome do recurso, seu número de processo e a hora corrente (lógica). Depois, envia a mensagem a todos os outros processos, fato que, conceitualmente, inclui ele mesmo. Adota-

se como premissa que o envio de mensagens é confiável; ou seja, nenhuma mensagem se perde.

Quando um processo recebe uma mensagem de requisição de um outro processo, a ação que ele executa depende de seu próprio estado em relação ao recurso nomeado na mensagem. Três casos têm de ser claramente distinguidos:

1. Se o receptor não estiver acessando o recurso e não quiser acessá-lo, devolve uma mensagem *OK* ao remetente.
2. Se o receptor já tiver acesso ao recurso, simplesmente não responde. Em vez disso, coloca a requisição em uma fila.
3. Se o receptor também quiser acessar o recurso, mas ainda não o fez, ele compara a marca de tempo da mensagem que chegou com a marca de tempo contida na mensagem que enviou para todos. A mais baixa vence. Se a marca de tempo da mensagem que acabou de chegar for mais baixa, o receptor devolve uma mensagem *OK*. Se a marca de tempo de sua própria mensagem for mais baixa, o receptor enfileira a requisição que está chegando e nada envia.

Após enviar requisições que peçam permissão, um processo se detém e espera até que todos tenham dado permissão. Logo que todas as permissões tenham entrado, ele pode seguir adiante. Quando conclui, envia mensagens *OK* para todos os processos que estão em sua fila e remove todos eles da fila.

Vamos tentar entender por que o algoritmo funciona. Se não houver conflito, é claro que ele funciona. Contudo, suponha que dois processos tentem acessar o recurso ao mesmo tempo, como mostra a Figura 6.15(a).

O processo 0 envia a todos uma requisição com marca de tempo 8, enquanto, simultaneamente, o processo 2 envia a todos uma requisição com a marca de tempo 12. O processo 1 não está interessado no recurso, portanto envia *OK* a ambos os remetentes. Ambos os processos, 0 e 2, vêem o conflito e compararam marcas de tempo. O processo 2 vê que perdeu, portanto dá permissão a 0 enviando *OK*. Agora, o processo 0 enfileira a requisição do processo 2 para mais tarde processá-la e acessa o recurso, como mostra a Figura 6.15(b). Quando conclui, remove de sua fila a requisição de 2 e envia uma mensagem *OK* ao processo 2, permitindo que este siga em frente, como mostra a Figura 6.15(c). O algoritmo funciona porque, em caso de conflito, a marca de tempo mais baixa vence, e todos concordam com a ordenação das marcas de tempo.

Note que a situação na Figura 6.15 teria sido, em essência, diferente se o processo 2 tivesse enviado sua mensagem antes, de modo que 0 a tivesse recebido e concedido permissão antes de emitir sua própria requisição. Nesse caso, 2 teria notado que ele próprio já tinha acesso

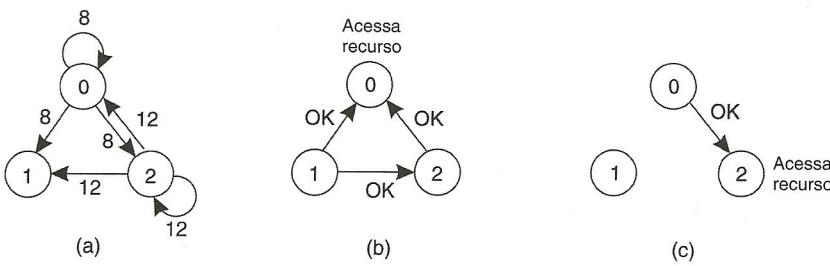


Figura 6.15 (a) Dois processos querem acessar um recurso compartilhado no mesmo momento. (b) O processo 0 tem a marca de tempo mais baixa, portanto vence. (c) Quando o processo 0 conclui, também envia uma mensagem OK, portanto, agora, 2 pode seguir adiante.

ao recurso na hora da requisição e a teria enfileirado em vez de enviar uma resposta.

Como acontece com o algoritmo centralizado que discutimos antes, a exclusão mútua é garantida sem deadlock nem inanição. O número de mensagens requeridas por entrada nessa circunstância é $2(n - 1)$, onde o número total de processos no sistema é n . O melhor de tudo é que não existe nenhum ponto de falha único.

Infelizmente, o ponto de falha único foi substituído por n pontos de falha. Se qualquer processo falhar, não responderá às requisições. Esse silêncio será interpretado (incorrectamente) como recusa de permissão e, por isso, bloqueará todas as tentativas subsequentes feitas por todos os processos para entrar em todas as regiões críticas. Uma vez que a probabilidade de um dos n processos falhar é, no mínimo, n vezes maior do que a probabilidade de um único coordenador falhar, conseguimos substituir um algoritmo ruim por um que é mais do que n vezes pior e que, além disso, requer muito mais tráfego de rede.

O algoritmo pode ser consertado pelo mesmo estratagema que propusemos antes. Quando uma requisição chega, o receptor sempre envia uma resposta, seja concedendo ou recusando permissão. Sempre que uma requisição ou uma resposta se perder, o remetente esgota a temporização de espera e continua tentando até que uma resposta volte ou que o remetente conclua que o destinatário está morto. Após uma requisição ser negada, o remetente deve bloquear à espera de uma mensagem OK subsequente.

Um outro problema desse algoritmo é que ou uma primitiva de comunicação multicast deve ser usada, ou cada processo deve manter, ele mesmo, a lista de associação ao grupo, incluindo processos que entram no grupo, saem do grupo e caem. O método funciona melhor com pequenos grupos de processos que nunca mudam seus grupos de associação.

Por fim, lembre-se de que um dos problemas do algoritmo centralizado é que fazer com que ele manipule todas as requisições pode resultar em um gargalo. No algoritmo distribuído, todos os processos estão envolvidos em todas as decisões referentes ao acesso ao recurso compartilhado. Se um processo for incapaz de manipular

a carga, é improvável que forçar todos a fazer exatamente a mesma coisa em paralelo ajudará muito.

São possíveis várias pequenas melhorias nesse algoritmo. Por exemplo, obter permissão de todos é realmente excessivo. Basta haver um método para impedir que dois processos acessem o recurso ao mesmo tempo. O algoritmo pode ser modificado para dar permissão quando tiver obtido permissão de uma maioria simples dos outros processos, em vez de obtê-la de todos eles. Claro que, nessa variação, após um processo ter concedido permissão a um outro processo, não poderá dar a mesma permissão a um outro até que o primeiro tenha sido concluído.

Ainda assim, esse algoritmo é mais lento, mais complicado, mais caro e menos robusto do que o original centralizado. Por que se dar ao trabalho de estudá-lo nessas condições? Uma razão é que ele mostra que um algoritmo distribuído é, no mínimo, possível, algo que não era óbvio quando começamos. Além disso, destacando suas deficiências, podemos estimular futuros teóricos a tentar produzir algoritmos que sejam realmente úteis. Por fim, assim como comer espinafre e aprender latim na escola, por alguma razão um tanto misteriosa afirma-se que algumas coisas são boas para você. Pode-se levar algum tempo para descobrir exatamente quais.

6.3.5 Algoritmo Token Ring

Uma abordagem completamente diferente para conseguir exclusão mútua por esquemas determinísticos em um sistema distribuído é ilustrada na Figura 6.16. Nesse caso, temos uma rede de barramento, como mostra a Figura 6.16(a) (por exemplo, Ethernet), sem nenhuma ordenação inherente dos processos. Um anel lógico é construído em software e a cada processo é designada uma posição no anel, como mostra a Figura 6.16(b). As posições no anel podem ser alocadas em ordem numérica de endereços de rede ou por alguns outros meios. Não importa qual é a ordenação; o que importa é que cada processo saiba de quem é a vez depois dele mesmo.

Quando o anel é inicializado, o processo 0 recebe uma **ficha**. A ficha circula ao redor do anel. Ela é passada do processo k para o processo $k + 1$ (valor em módulo

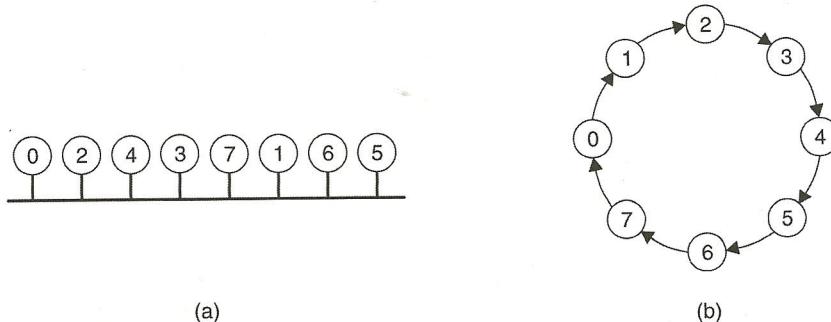


Figura 6.16 (a) Grupo de processos não ordenados em uma rede. (b) Um anel lógico é construído em software.

do tamanho do anel) em mensagens ponto-a-ponto. Quando um processo adquire a ficha de seu vizinho, verifica para confirmar se precisa acessar o recurso compartilhado. Caso necessite, o processo segue adiante, faz todo o trabalho que precisa fazer e libera o recurso. Após concluir, passa a ficha ao longo do anel. Não é permitido acessar o recurso novamente, de imediato, usando a mesma ficha.

Se um processo receber a ficha de seu vizinho e não estiver interessado no recurso, ele apenas passa a ficha adiante. Em consequência, quando nenhum processo precisar do recurso, a ficha apenas circula a grande velocidade pelo anel.

É fácil ver a correção desse algoritmo. Só um processo tem a ficha a qualquer instante, portanto só um processo pode realmente obter o recurso. Visto que a ficha circula entre os processos em uma ordem bem definida, não pode ocorrer inanição. Tão logo um processo decida que quer ter acesso ao recurso, na pior das hipóteses ele terá de esperar que cada um dos outros processos use o recurso.

Como sempre, esse algoritmo também tem problemas. Se a ficha se perder, precisa ser regenerada. Na verdade, detectar que ela se perdeu é difícil, visto que a quantidade de tempo entre aparições sucessivas da ficha na rede é ilimitada. O fato de a ficha não ser localizada durante uma hora não significa que ela se perdeu; talvez alguém ainda a esteja usando.

O algoritmo também encontra dificuldade se um processo cair, mas a recuperação é mais fácil do que nos outros casos. Se exigirmos que um processo que recebe a ficha reconheça o recebimento, um processo morto será detectado quando seu vizinho tentar lhe passar a ficha e não conseguir. Nesse ponto, o processo morto pode ser removido do grupo e o portador da ficha pode pular o processo morto e passar a ficha ao próximo membro a quem pertencer a vez, ou para o próximo depois deste, se necessário. Claro que isso requer que todos mantenham a configuração corrente do anel.

6.3.6 Comparação entre os quatro algoritmos

É instrutivo fazer uma breve comparação entre os quatro algoritmos de exclusão mútua que estudamos. Na Tabela 6.1 apresentamos uma lista dos algoritmos e três propriedades fundamentais: o número de mensagens requeridas para um processo acessar e liberar um recurso compartilhado, o atraso antes que um acesso possa ocorrer (considerando que as mensagens passam em seqüência por uma rede) e alguns problemas associados com cada algoritmo.

O algoritmo centralizado é mais simples e também o mais eficiente. Requer apenas três mensagens para entrar e sair de uma região crítica: uma requisição, uma permissão para entrar e uma liberação para sair. No caso descentralizado, vemos que essas mensagens precisam ser executadas para cada um dos m coordenadores, mas agora é

Algoritmo	Mensagens por entrada/saída	Atraso antes da entrada (em número de tempos de mensagens)	Problemas
Centralizado	3	2	Queda do coordenador
Descentralizado	$3mk, k = 1,2,\dots$	$2m$	Inanição, baixa eficiência
Distribuído	$2(n-1)$	$2(n-1)$	Queda de qualquer processo
Token Ring	$1 \text{ a } \infty$	$0 \text{ a } n-1$	Ficha perdida; processo cai

Tabela 6.1 Comparação entre quatro algoritmos de exclusão mútua.

possível que seja preciso fazer várias tentativas (para as quais introduzimos a variável k). O algoritmo distribuído requer $n - 1$ mensagens de requisição, uma para cada um dos outros processos, e $n - 1$ mensagens adicionais de permissão, para um total de $2(n - 1)$. (Supomos que somente canais de comunicação ponto-a-ponto sejam usados.) Com o algoritmo Token Ring, o número é variável. Se todo processo quiser entrar em uma região crítica constantemente, cada ficha resultará em uma entrada e saída para uma média de uma mensagem por entrada em região crítica. No outro extremo, às vezes a ficha pode circular durante horas sem que ninguém se interesse por ela. Nesse caso, o número de mensagens por entrada em uma região crítica é ilimitado.

O atraso desde o momento em que o processo precisa entrar em uma região crítica até sua real entrada também varia para os quatro algoritmos. Quando o tempo de utilização de um recurso for curto, o fator dominante no atraso é o próprio mecanismo de acesso a um recurso. Quando os recursos são usados durante um longo período, o fator dominante é a espera para que todos tenham a sua vez. Na Tabela 6.1 mostramos o primeiro caso. Leva somente dois tempos de mensagem para entrar em uma região crítica no caso centralizado, mas $3mk$ tempos para o caso descentralizado, onde k é o número de tentativas que precisam ser feitas. Adotando como premissa que as mensagens são enviadas uma após a outra, são necessários $2(n - 1)$ tempos de mensagem no caso distribuído. Para o Token Ring, o tempo varia de 0 (a ficha acabou de chegar) a $n - 1$ (a ficha acabou de sair).

Por fim, todos os algoritmos, exceto o descentralizado, são muito afetados por quedas. Providências especiais e complexidade adicional devem ser introduzidas para evitar que uma queda derrube o sistema inteiro. É irônico que os algoritmos distribuídos sejam ainda mais sensíveis a quedas do que os centralizados. Em um sistema projetado para ser tolerante a falha, nenhum deles seria adequado, mas, se as quedas não forem muito frequentes, até poderiam servir. O algoritmo descentralizado é menos sensível a quedas, mas os processos podem sofrer de inanição e são necessárias providências especiais para garantir eficiência.

6.4 Posicionamento Global de Nós

Quando o número de nós em um sistema distribuído cresce, torna-se cada vez mais difícil para qualquer nó monitorar os outros. Saber onde cada nó está pode ser importante para executar algoritmos distribuídos como os de roteamento, multicast, colocação de dados, busca e assim por diante. Já vimos diferentes exemplos nos quais grandes conjuntos de nós são organizados em topologias específicas que facilitam a execução eficiente de tais algo-

ritmos. Nesta seção, examinamos uma outra organização que está relacionada a questões de temporização.

Em **redes de sobreposição geométrica**, a cada nó é designada uma posição em um espaço geométrico dimensional m , tal que a distância entre dois nós nesse espaço reflete uma métrica de desempenho do mundo real. O exemplo mais simples e mais aplicado é aquele em que a distância corresponde a uma latência entre nós. Em outras palavras, dados dois nós, P e Q , a distância $d(P, Q)$ reflete o tempo que levaria para uma mensagem ir de P a Q e vice-versa.

Há muitas aplicações de redes de sobreposição geométrica. Considere a situação em que um site Web no servidor O foi replicado para vários servidores, S_1, \dots, S_k na Internet. Quando um cliente C requisita uma página de O , este pode decidir redirecionar essa requisição para o servidor mais próximo de C , isto é, aquele que dará o melhor tempo de resposta. Se a localização geométrica de C for conhecida, bem como a de cada servidor de réplica, então O pode simplesmente escolher o servidor S_i para o qual $d(C, S_i)$ é mínima. Note que tal seleção requer somente processamento local em O . Em outras palavras, não há, por exemplo, nenhuma necessidade de amostrar todas as latências entre C e cada um dos servidores de réplica.

Um outro exemplo, cujos detalhes esmiuçaremos no próximo capítulo, é a colocação ótima da réplica. Considere, mais uma vez, um site Web que colheu as posições de seus clientes. Se o site fosse replicar seu conteúdo para K servidores, ele poderia calcular as K melhores posições em que colocar réplicas, de modo que o tempo médio de resposta cliente-réplica fosse mínimo. Executar esses cálculos é praticamente viável se clientes e servidores ocuparem posições geométricas que refletem latências entre nós.

Como um último exemplo, considere o **roteamento baseado em posição** (Araujo e Rodrigues, 2005; Stojmenovic, 2002). Em tais esquemas, uma mensagem é repassada a seu destinatário usando somente informações de posicionamento; por exemplo, um algoritmo ingênuo de roteamento que permita a cada nó repassar uma mensagem ao vizinho mais próximo do destinatário. Embora seja fácil mostrar que esse algoritmo específico pode não convergir, ele ilustra que somente informações locais são usadas para tomar uma decisão. Não há nenhuma necessidade de propagar informações de enlaces ou semelhantes a todos os nós presentes na rede, como acontece com algoritmos de roteamento convencionais.

Em teoria, posicionar um nó em um espaço geométrico m -dimensional requer $m + 1$ medições de distância até nós que estejam em posições conhecidas. É fácil de ver isso considerando o caso $m = 2$, como mostra a Figura 6.17. Supondo que o nó P queira calcular sua própria posição, ele contata três outros nós cujas posições sejam

conhecidas e mede sua distância até cada um deles. Contatar somente um nó informaria a P o círculo no qual ele está localizado; contatar somente dois nós lhe informaria a posição da interseção de dois círculos (que, em geral, consiste em dois pontos); na seqüência, um terceiro nó permitiria que P calculasse sua localização real.

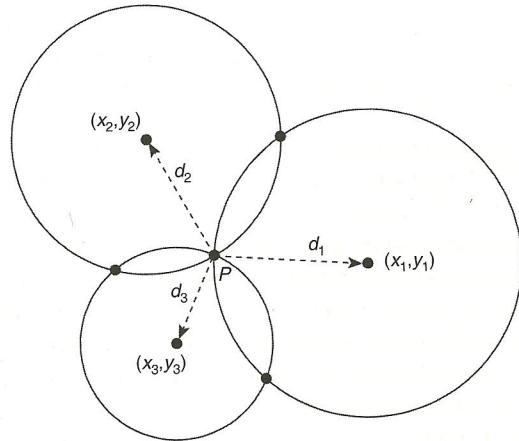


Figura 6.17 Cálculo da posição de um nó em um espaço bidimensional.

Exatamente como em GPS, o nó P pode calcular suas próprias coordenadas (x_P, y_P) resolvendo as três equações com as duas incógnitas x_P e y_P :

$$d_i = \sqrt{(x_i - x_P)^2 + (y_i - y_P)^2} \quad (i = 1, 2, 3)$$

Como dissemos, de modo geral, d_i corresponde a medir a latência entre P e o nó em (x_i, y_i) . Essa latência pode ser estimada como a metade do atraso de viagem de ida e volta, mas é preciso ficar claro que seu valor será diferente ao longo do tempo. O efeito é um posicionamento diferente sempre que P quiser recalcular sua posição. Além do mais, se outros nós usassem a posição corrente de P para calcular suas próprias coordenadas, então deve ficar claro que o erro no posicionamento de P afetará a exatidão do posicionamento de outros nós.

Ademais, também deve ficar claro que, de modo geral, as distâncias medidas por nós diferentes não serão nem mesmo consistentes. Por exemplo, consideremos que estamos calculando distâncias em um espaço unidimensional, como mostra a Figura 6.18. Nesse exemplo, vemos que, embora R meça sua distância até Q como 2,0 e $d(P,Q)$ foi medida como 1,0, quando R medir $d(P,R)$ achará 3,2, que é claramente inconsistente com as outras duas medições.

A Figura 6.18 também sugere como essa situação pode ser melhorada. Em nosso exemplo simples, poderíamos resolver as inconsistências pelo mero cálculo de posições em um espaço bidimensional. Todavia, isso, por si só, não é uma solução geral quando estivermos tratando com muitas medições. Na realidade, considerando que as medições de latência da Internet podem violar a **desigualdade triangular**, em geral é impossível resolver completamente as inconsistências. A desigualdade triangular afirma que, em um espaço geométrico, para quaisquer três nós arbitrários P , Q e R , sempre deve ser verdadeira que $d(P,R) \leq d(P,Q) + d(Q,R)$.

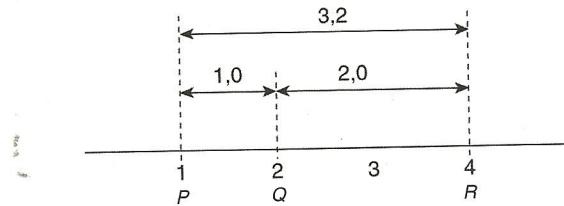


Figura 6.18 Medições inconsistentes de distâncias em um espaço unidimensional.

Há vários modos de abordar essas questões. Um deles, proposto por Ng e Zhang (2002), é usar L nós especiais b_1, \dots, b_L , conhecidos como **marcos**. Marcos medem suas latências aos pares $d(b_i, b_j)$ e, na seqüência, deixam que um nó central calcule as coordenadas para cada marco. Com essa finalidade, o nó central procura minimizar a seguinte função erro agregado:

$$\sum_{i=1}^L \sum_{j=i+1}^L \left[\frac{d(b_i, b_j) - \hat{d}(b_i, b_j)}{d(b_i, b_j)} \right]^2$$

onde $\hat{d}(b_i, b_j)$ corresponde à *distância geométrica*, isto é, à distância após os nós b_i e b_j estarem posicionados.

O parâmetro oculto na minimização da função erro agregado é a dimensão m . É óbvio que $L > m$ sempre, mas nada nos impede de escolher um valor para m que seja muito menor do que L . Nesse caso, um nó P mede sua distância até cada um dos L marcos e calcula suas coordenadas minimizando

$$\sum_{i=1}^L \left[\frac{d(b_i, P) - \hat{d}(b_i, P)}{d(b_i, P)} \right]^2$$

Ocorre que, com marcos bem escolhidos, m pode ser um valor tão pequeno como 6 ou 7, sendo que $\hat{d}(P, Q)$ é diferente da real latência $d(P, Q)$ por um fator não maior do que 2 para nós arbitrários P e Q (Szymaniak et al., 2004).

Um outro modo de atacar esse problema é considerar o conjunto de nós como um enorme sistema no qual os nós são ligados uns aos outros por molas. Nesse caso, $|d(P, Q) - \hat{d}(P, Q)|$ indica até que ponto os nós P e Q estão deslocados em relação à situação em que o sistema de molas estaria em descanso. Permitindo que cada nó altere, ligeiramente, sua posição, pode-se mostrar que, a certa altura, o sistema convergirá para uma organização ótima na qual o erro agregado é mínimo. Essa aborda-

gem foi seguida em Vivaldi, cujos detalhes podem ser encontrados em Dabek et al. (2004a).

6.5 Algoritmos de Eleição

Muitos algoritmos distribuídos requerem que um processo aja como coordenador, iniciador ou, então, desempenhe algum papel especial. Em geral, não importa qual processo assume essa responsabilidade especial, mas um deles tem de fazê-lo. Nesta seção, estudaremos algoritmos para eleger um coordenador. Usaremos esse nome como genérico para o processo especial.

Se todos os processos forem exatamente iguais, sem nenhuma característica distintiva, não haveria nenhum modo de selecionar um deles para ser especial. Em decorrência, consideraremos que cada processo tem um número exclusivo, por exemplo, seu endereço de rede (para simplificar, consideraremos um processo por máquina). Em geral, algoritmos de eleição tentam localizar o processo que tenha o número de processo mais alto e designá-lo como coordenador. Os modos como os algoritmos fazem essa localização são diferentes.

Além do mais, vamos considerar também que todo processo sabe qual é o número de processo de todos os outros. O que os processos não sabem é quais estão funcionando e quais estão inativos no momento considerado. A meta de um algoritmo de eleição é garantir que, quando uma eleição começar, ela concluirá todos os processos concordando com o novo coordenador escolhido. Há muitos algoritmos e variações, e vários dos mais importantes são discutidos em livros técnicos por Lynch (1996) e Tel (2000), respectivamente.

6.5.1 Algoritmos de eleição tradicionais

Começamos estudando dois algoritmos de eleição tradicionais para dar uma idéia do que grandes grupos de pesquisadores vêm fazendo nas últimas décadas. Nas seções subsequentes, damos atenção a novas aplicações do problema da eleição.

Algoritmo do valentão

Como primeiro exemplo, considere o **algoritmo do valentão** inventado por Garcia-Molina (1982). Quando qualquer processo nota que o coordenador não está mais respondendo às requisições, ele inicia uma eleição. Um processo, P , convoca uma eleição como segue:

1. P envia uma mensagem *ELEIÇÃO* a todos os processos de números mais altos.
2. Se nenhum responder, P vence a eleição e se torna coordenador.
3. Se um dos processos de número mais alto responder, ele toma o poder e o trabalho de P está concluído.

A qualquer momento, um processo pode receber uma mensagem *ELEIÇÃO* de um de seus colegas de números mais baixos. Quando tal mensagem chega, o receptor envia uma mensagem *OK* de volta ao remetente para indicar que está vivo e tomará o poder. Então, o receptor convoca uma eleição, a menos que já tenha convocado uma. A certa altura, todos os processos desistem, exceto um, e este é o novo coordenador. Ele anuncia sua vitória enviando a todos os processos uma mensagem informando que a partir daquele instante ele é o novo coordenador.

Se um processo que antes estava inativo voltar, convoca uma eleição. Se acaso ele for o processo de número mais alto que está executando naquele instante, ganhará a eleição e assumirá a tarefa de coordenador. Assim, o indivíduo mais poderoso da cidade sempre ganha, daí o nome ‘algoritmo do valentão’.

Na Figura 6.19 podemos ver um exemplo do funcionamento do algoritmo do valentão. O grupo consiste em oito processos, numerados de 0 a 7. Antes, o processo 7 era o coordenador, mas ele acabou de cair. O processo 4 é o primeiro a notar isso, portanto envia mensagens *ELEIÇÃO* a todos os processos mais altos do que ele, ou seja, 5, 6 e 7, como mostra a Figura 6.19(a). Ambos os processos, 5 e 6, respondem com *OK*, como mostra a Figura 6.19(b). Ao receber a primeira dessas respostas, 4 sabe que sua tarefa está encerrada. Ele sabe que um daqueles figurões tomará o poder e se tornará coordenador e fica só esperando para ver quem será o vencedor (embora nesse ponto ele já possa fazer uma boa idéia).

Na Figura 6.19(c), ambos, 5 e 6, convocam eleições, cada um enviando somente mensagens aos processos mais altos do que ele. Na Figura 6.19(d), o processo 6 informa ao 5 que ele próprio tomará o poder. Nesse ponto, 6 sabe que 7 está morto e que ele próprio (6) é o vencedor. Se houver informações de estado a colher do disco ou de qualquer outro lugar que informe onde o antigo coordenador parou, agora 6 deve fazer o que for necessário. Quando estiver pronto para tomar o poder, 6 anuncia esse fato com o envio de uma mensagem *COORDENADOR* a todos os processos em execução. Quando 4 recebe essa mensagem, pode continuar com a operação que estava tentando executar quando descobriu que 7 estava morto, porém, desta vez, usando 6 como coordenador. Desse modo a falha de 7 é resolvida e o trabalho pode continuar.

Se acaso o processo 7 for reiniciado, ele apenas enviará a todos os outros uma mensagem *COORDENADOR* e os fará se submeter à força.

Algoritmo de anel

Um outro algoritmo de eleição é baseado na utilização de um anel. Diferente de alguns algoritmos de

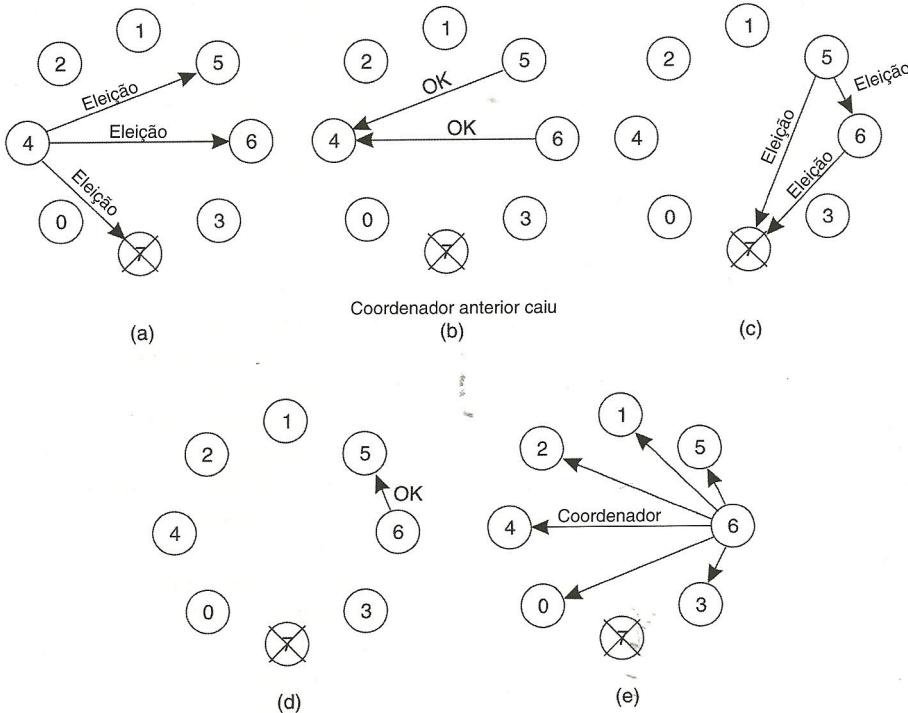


Figura 6.19 Algoritmo de eleição do valentão. (a) O processo 4 convoca uma eleição. (b) Os processos 5 e 6 respondem e mandam 4 parar. (c) Agora, cada um, 5 e 6, convoca uma eleição. (d) O processo 6 manda 5 parar. (e) O processo 6 vence e informa a todos.

anel, esse não usa uma ficha. Adotamos como premissa que os processos estão ordenados por ordem física ou por ordem lógica, de modo que cada processo sabe quem é seu sucessor. Quando qualquer processo nota que o coordenador não está funcionando, monta uma mensagem *ELEIÇÃO* que contém seu próprio número de processo e envia a mensagem a seu sucessor. Se o sucessor tiver caído, o remetente pula o sucessor e vai até o próximo membro ao longo do anel, ou até o próximo depois deste, até localizar um processo em funcionamento. A cada etapa ao longo do caminho, o remetente adiciona seu próprio número de processo à lista na mensagem, o que o torna efetivamente um candidato a ser eleito como coordenador.

A certa altura, a mensagem volta ao processo que começou tudo. Esse processo reconhece esse evento quando recebe uma mensagem de entrada que contém seu próprio número de processo. Nesse ponto, o tipo de mensagem é mudado para *COORDENADOR* e circulado novamente, desta vez para informar a todos quem é o coordenador (o membro da lista que tem o número mais alto) e quem são os membros do novo anel. Quando essa mensagem circulou uma vez, é removida e todos voltam a trabalhar.

Na Figura 6.20, vemos o que acontece se dois processos, 2 e 5, descobrem ao mesmo tempo que o coordenador anterior, o processo 7, caiu. Cada um deles monta uma mensagem *ELEIÇÃO* e cada um deles começa a cir-

cular sua mensagem, independentemente do outro. A certa altura, ambas as mensagens terão percorrido todo o caminho, e ambos, 2 e 5, as converterão em mensagens *COORDENADOR*, com exatamente os mesmos membros e na mesma ordem. Quando ambas tiverem percorrido o anel mais uma vez, ambas serão removidas. Não há problema nenhum em ter mensagens extras em circulação; na pior das hipóteses, elas consomem um pouco de largura de banda, mas isso não é considerado desperdício.

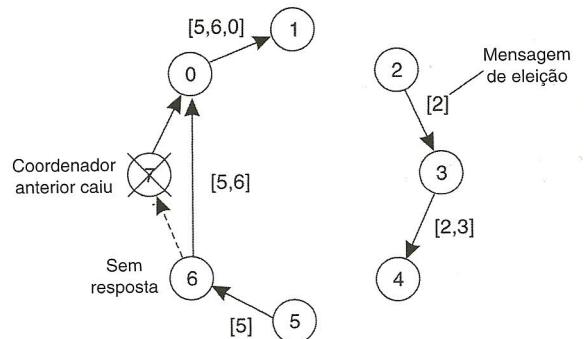


Figura 6.20 Algoritmo de eleição que usa um anel.

6.5.2 Eleições em ambientes sem fio

Algoritmos de eleição tradicionais em geral são baseados em premissas que não são realistas em ambientes

sem fio. Por exemplo, eles consideram que a troca de mensagens é confiável e que a topologia da rede não muda. Essas premissas são falsas para a maioria dos ambientes sem fio, em especial para os de redes móveis *ad hoc*.

Foram desenvolvidos apenas alguns protocolos de eleição que funcionam em redes *ad hoc*. Vasudevan et al. (2004) propõem uma solução que pode manipular nós que falham e participação de redes. Uma propriedade importante da solução desses autores é que se pode eleger o *melhor* líder em vez de apenas um líder aleatório, como era mais ou menos o caso nas soluções que já discutimos. A seguir, descreveremos como funciona o protocolo proposto por eles. Para simplificar nossa discussão, vamos nos concentrar somente em redes *ad hoc* e ignoraremos que os nós podem se mover.

Considere uma rede *ad hoc* sem fio. Para eleger um líder, qualquer nó da rede, denominado fonte, pode iniciar uma eleição enviando uma mensagem *ELEIÇÃO* a seus vizinhos imediatos (isto é, os nós que estão no seu alcance). Quando um nó recebe uma mensagem *ELEIÇÃO* pela primeira vez, designa o remetente como seu pai e, na seqüência, envia uma mensagem *ELEIÇÃO* a todos os seus vizinhos imediatos, com exceção do pai. Quando um nó recebe uma mensagem *ELEIÇÃO* de um nó que não é seu pai, ele se limita a reconhecer o recebimento.

Quando o nó *R* designou o nó *Q* como seu pai, ele repassa a mensagem *ELEIÇÃO* a seus vizinhos imediatos (excluindo *Q*) e espera que os reconhecimentos cheguem antes de reconhecer a mensagem *ELEIÇÃO* de *Q*. Essa espera tem consequência importante. Em primeiro lugar, note que os vizinhos que já selecionaram um pai responderão imediatamente a *R*. Mais especificamente, se todos os vizinhos já têm pai, *R* é um nó-folha e poderá se reportar de volta a *Q* rapidamente. Ao fazer isso, ele também reportará informações tais como o tempo de vida útil de sua bateria e outras capacidades de recursos.

Essas informações permitirão que, mais tarde, *Q* compare as capacidades de *R* com as de outros nós abaixo dele e selecione o mais qualificado para a liderança. Claro que *Q* tinha enviado uma mensagem *ELEIÇÃO* só porque seu próprio pai, *P*, também o fizera. Por sua vez, quando, a certa altura, *Q* reconhece a mensagem *ELEIÇÃO* enviada anteriormente por *P*, ele passará o nó mais qualificado a *P* também. Desse modo, o fonte acabará sabendo qual nó é o melhor para ser selecionado como líder e, depois disso, transmitirá essa informação em broadcast a todos os outros nós.

Esse processo é ilustrado na Figura 6.21. Os nós foram rotulados *a* a *j*, com suas perspectivas capacidades. O nó *a* inicia uma eleição enviando uma mensagem *ELEIÇÃO* em broadcast aos nós *b* e *j*, como mostra a Figura 6.21(b). Após essa etapa, mensagens *ELEIÇÃO* são propagadas para todos os nós, terminando com a situação mostrada na Figura 6.21(e), na qual omitimos o último

broadcast pelos nós *f* e *i*. Dali em diante, cada nó reporta a seu pai o nó que tem a melhor capacidade, como mostra a Figura 6.21(f). Por exemplo, quando o nó *g* recebe os reconhecimentos de seus filhos, *e* e *h*, ele perceberá que *h* é o melhor nó e propagará [*h*, 8] a seu próprio pai, o nó *b*. No final, o fonte notará que *h* é o melhor líder e transmitirá essa informação em broadcast a todos os outros nós.

Quando são iniciadas várias eleições, cada nó decidirá se juntar a uma só eleição. Com essa finalidade, cada fonte rotula sua mensagem *ELEIÇÃO* com um único identificador. Nós participarão somente na eleição que tiver o identificador mais alto, interrompendo qualquer participação em curso em outras eleições.

Com alguns pequenos ajustes, pode-se mostrar que esse protocolo funciona também quando há partições de rede e quando nós se juntam à rede ou saem dela. Os detalhes podem ser encontrados em Vasudevan et al. (2004).

6.5.3 Eleições em sistemas de grande escala

Os algoritmos que discutimos até aqui se aplicam, de modo geral, a sistemas distribuídos relativamente pequenos. Além disso, os algoritmos se concentram na seleção de um único nó. Há situações em que, na verdade, vários nós devem ser selecionados, como no caso de **superpares** em redes peer-to-peer, que discutimos no Capítulo 2. Nesta seção, vamos nos concentrar especificamente no problema de selecionar superpares.

Lo et al. (2005) identificaram os seguintes requisitos que precisam ser cumpridos para a seleção de superpar:

1. Nós normais devem ter baixa latência de acesso a superpares.
2. Superpares devem estar uniformemente distribuídos pela rede de sobreposição.
3. Deve haver uma porção predefinida de superpares em relação ao número total de nós na rede de sobreposição.
4. Cada superpar não deve precisar atender mais do que um número fixo de nós normais.

Felizmente, esses requisitos são relativamente fáceis de cumprir na maioria dos sistemas peer-to-peer, dado o fato de que a rede de sobreposição ou é estruturada (como em sistemas baseados em DHT) ou é aleatoriamente não-estruturada (como, por exemplo, pode ocorrer em soluções baseadas em gossiping). Vamos estudar soluções propostas por Lo et al. (2005).

No caso de sistemas baseados em DHT, a idéia básica é reservar uma fração do espaço de identificadores para superpares. Lembre-se de que, em sistemas baseados em DHT, cada nó recebe um identificador de *m* bits aleatório e uniformemente designado. Agora, suponha que reservemos os primeiros *k* bits (isto é, os da extrema

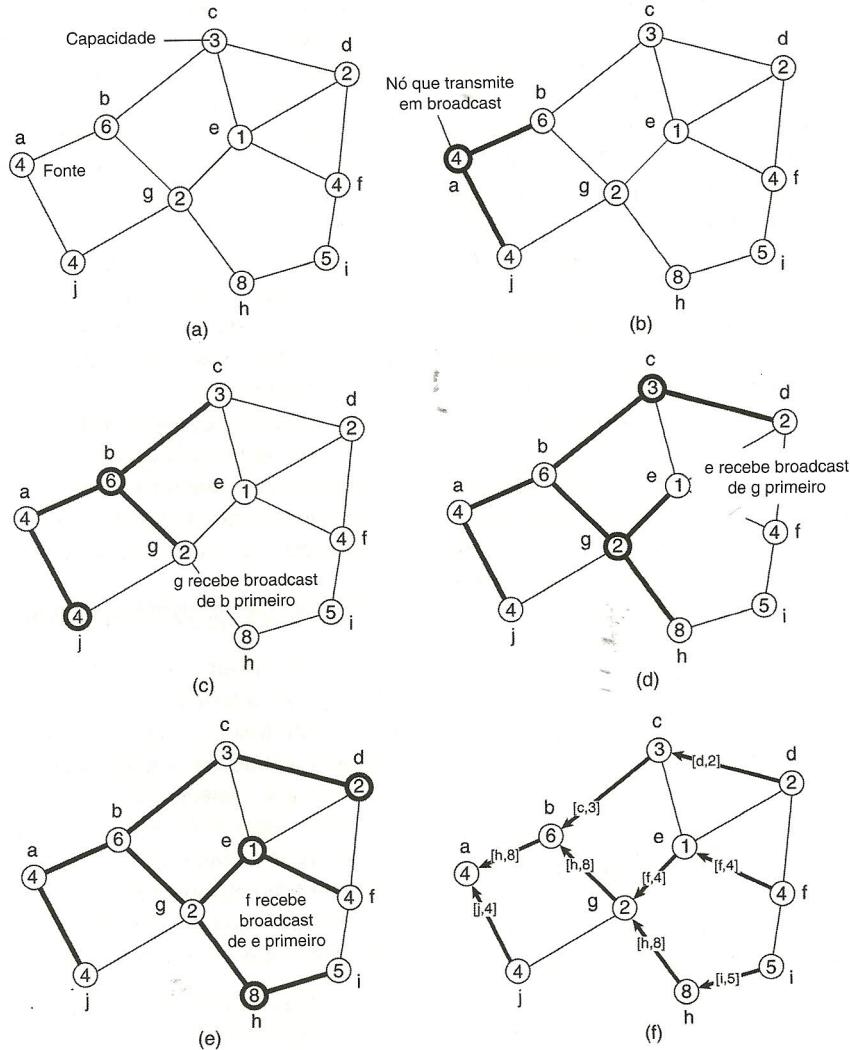


Figura 6.21 Algoritmo de eleição em uma rede sem fio, com o nó *a* como o fonte. (a) Rede inicial. (b)–(e) A fase de montagem da árvore (última etapa do broadcast pelos nós *f* e *i* não é mostrada). (f) Reportando o melhor nó ao fonte.

esquerda) para identificar superpares. Por exemplo, se precisarmos de N superpares, então os primeiros $\lceil \log_2(N) \rceil^*$ bits de qualquer *chave* podem ser usados para identificar esses nós.

Para explicar, vamos supor que temos um (pequeno) sistema Chord com $m = 8$ e $k = 3$. Quando consultamos o nó responsável por uma chave específica p , podemos decidir primeiro rotear a requisição de consulta até o nó responsável pelo padrão

$p \text{ AND } 11100000$

que então é tratado como o superpar. Note que cada nó *id* pode verificar se ele é um superpar consultando

$\text{id AND } 11100000$

para ver se essa requisição é roteada para ele mesmo. Contanto que identificadores de nós sejam designados uniformemente a nós, pode-se ver que, com um total de N nós, o número de superpares é, na média, igual a $2^{k-m} N$.

Uma abordagem completamente diferente é baseada no posicionamento de nós em um espaço geométrico m -dimensional, como já discutimos. Nesse caso, suponha que precisemos colocar N superpares *uniformemente* por toda a extensão da sobreposição. A idéia básica é simples: um total de N fichas é distribuído por N nós escolhidos aleatoriamente. Nenhum nó pode ter mais do que uma ficha. Cada ficha representa uma força de repulsão da qual uma outra ficha está inclinada a se afastar. O efeito líquido é que, se todas as fichas exercerem a mesma força de repulsão, elas se afastarãoumas das outras e se espalharão uniformemente no espaço geométrico.

* O símbolo ' $\lceil \rceil$ ' é usado para representar o inteiro imediatamente superior a um número (N. do R.T.).

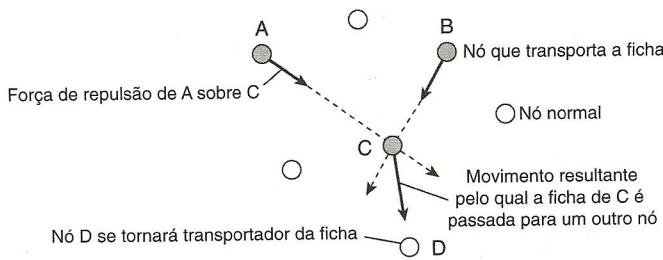


Figura 6.22 Movimentação de fichas em um espaço bidimensional que utiliza forças de repulsão.

Essa abordagem requer que nós que transportem uma ficha saibam da existência de outras fichas. Com essa finalidade, Lo et al. propõem utilizar um protocolo de gossiping pelo qual a força de uma ficha é disseminada por toda a extensão da rede. Se um nó descobrir que a força total que está agindo sobre ele excede um patamar, ele moverá a ficha na direção das forças combinadas, como mostra a Figura 6.22.

Quando uma ficha é transportada por um nó por dado período de tempo, esse nó se promoverá a superpar.

6.6 Resumo

Uma questão intimamente ligada com comunicação entre processos é como os processos em sistemas distribuídos sincronizam. Sincronização quer dizer fazer a coisa certa na hora certa. Um problema em sistemas distribuídos e redes de computadores em geral é que não há nenhuma idéia de um relógio globalmente compartilhado. Em outras palavras, processos em máquinas diferentes têm sua própria idéia do que é o tempo.

Há vários modos de sincronizar relógios em um sistema distribuído mas, em essência, todos os métodos são baseados em troca de valores de relógio considerando simultaneamente o tempo que leva para enviar e receber mensagens. Variações em atrasos de comunicação e o modo como essas variações são tratadas determinam, em grande parte, a precisão de algoritmos de sincronização de relógios.

Relacionado com esses problemas de sincronização está o posicionamento de nós em uma sobreposição geométrica. A idéia básica é designar a cada nó coordenadas de um espaço m -dimensional de modo tal que a distância geométrica possa ser utilizada como medida precisa para a latência entre dois nós. O método de atribuir coordenadas é muito parecido com o aplicado para determinar a localização e a hora em GPS.

Em muitos casos, não é necessário saber a hora absoluta. O que conta é que os eventos relacionados em processos diferentes aconteçam na ordem correta. Lamport mostrou que, ao introduzir uma noção de relógios lógicos, é possível que um conjunto de processos chegue a um acordo global sobre a ordenação correta de eventos. Em

essência, a cada evento e , tal como enviar ou receber uma mensagem, é designada uma marca de tempo lógica globalmente exclusiva $C(e)$ tal que, quando o evento a aconteceu antes de b , $C(a) < C(b)$. As marcas de tempo de Lamport podem ser estendidas para marcas de tempo vetoriais: se $C(a) < C(b)$, sabemos até que o evento a precedeu b por causalidade.

Uma classe importante de algoritmos de sincronização é a da exclusão mútua distribuída. Esses algoritmos asseguram que, em um conjunto de processos distribuídos, pelo menos um processo por vez tem acesso a um recurso compartilhado. Pode se conseguir exclusão mútua distribuída com facilidade se utilizarmos um coordenador que monitora de quem é a vez. Também existem algoritmos totalmente distribuídos, mas eles têm a desvantagem de ser, de modo geral, mais suscetíveis a falhas de comunicação e de processo.

Sincronização entre processos muitas vezes requer que um processo aja como um coordenador. Nos casos em que o coordenador não é fixo, é necessário que os processos em um sistema distribuído de computação decidam quem será esse coordenador. Tal decisão é tomada por meio de algoritmos de eleição. Algoritmos de eleição são usados primordialmente em casos em que o coordenador pode cair. Contudo, eles também podem ser aplicados para a seleção de superparas em sistemas peer-to-peer.

Problemas

1. Cite no mínimo três fontes de atraso que podem ser introduzidas entre a transmissão da hora em broadcast WWV e o ajuste, pelos processadores, de seus relógios internos em um sistema distribuído.
2. Considere o comportamento de duas máquinas em um sistema distribuído. Ambas têm relógios que devem pulsar 1.000 vezes por milissegundo. Um deles realmente pulsa a essa taxa, mas o outro pulsa somente 990 vezes por milissegundo. Se as atualizações UTC chegam uma vez por minuto, qual será a máxima defasagem entre os relógios?
3. Um dos dispositivos modernos que se instalaram (silenciosamente) em sistemas distribuídos são os

receptores GPS. Dê exemplos de aplicações distribuídas que possam utilizar informações GPS.

4. Quando um nó sincroniza seu relógio com o de outro nó, em geral é uma boa idéia também levar em conta medições anteriores. Por quê? Dê um exemplo de como essas leituras anteriores podem ser levadas em conta.
5. Adicione uma nova mensagem à Figura 6.9 que seja concorrente com a mensagem *A*, isto é, que não acontece antes de *A* ou não acontece depois de *A*.
6. Para conseguir multicast totalmente ordenado com marcas de tempo Lamport, é estritamente necessário que cada mensagem seja reconhecida?
7. Considere uma camada de comunicação na qual as mensagens são entregues somente na ordem em que foram enviadas. Dê um exemplo no qual até mesmo essa ordenação é desnecessariamente restritiva.
8. Muitos algoritmos distribuídos requerem a utilização de um processo coordenador. Até que ponto esses algoritmos realmente são considerados distribuídos? Comente sua resposta.
9. Na abordagem centralizada da exclusão mútua (Figura 6.14), ao receber uma mensagem de um processo que está liberando seu acesso exclusivo aos recursos que estava usando, o coordenador normalmente concede permissão ao primeiro processo na fila. Cite um outro algoritmo possível para o coordenador.
10. Considere novamente a Figura 6.14. Suponha que o coordenador caia. Isso sempre derruba o sistema? Se não derrubar, sob quais circunstâncias isso acontece? Há algum modo de evitar o problema e fazer com que o sistema seja capaz de tolerar quedas de coordenador?
11. O algoritmo de Ricart e Agrawala apresenta o seguinte problema: se um processo falhou e não responde a uma requisição de um outro processo para acessar um recurso, a falta de resposta será interpretada como uma recusa de permissão. Sugerimos que todas as requisições sejam respondidas imediatamente para facilitar a detecção de processos que falharam. Há algumas circunstâncias em que até esse método é insuficiente? Discuta sua resposta.
12. Como as entradas na Tabela 6.1 mudariam se admitíssemos que os algoritmos podem ser implementados sobre uma LAN que suporta broadcast por hardware?
13. Um sistema distribuído pode ter vários recursos independentes. Imagine que o processo 0 quer acessar o recurso *A* e o processo 1 quer acessar o recurso *B*. O algoritmo de Ricart e Agrawala pode resultar em deadlocks? Explique sua resposta.
14. Suponha que dois processos detectem a morte do coordenador simultaneamente e ambos decidam convocar uma eleição que utilize o algoritmo do valentão. O que acontecerá?
15. Na Figura 6.20 temos duas mensagens *ELEIÇÃO* que circulam simultaneamente. Embora não haja problema em ter duas delas, seria mais elegante se uma fosse eliminada. Proponha um algoritmo para fazer isso sem afetar a operação do algoritmo de eleição básico.
16. (Tarefa de laboratório) Sistemas Unix oferecem muitas facilidades para manter computadores em sincronia; em particular, a combinação da ferramenta *crontab* (que permite o escalonamento automático das operações) e vários comandos de sincronização são poderosos. Configure um sistema Unix que mantém a precisão do horário local dentro da faixa de um único segundo. Da mesma maneira, configure uma facilidade automática de apoio pela qual uma quantidade de arquivos cruciais seja transferida automaticamente para uma máquina remota uma vez a cada 5 minutos. Sua solução deve ser eficiente no que se refere à utilização de largura de banda.