3. Projeto e implementação de Servidores

3.1 Introdução

Este capítulo discute questões fundamentais relacionadas ao projeto de software servidor, incluindo acesso com conexão vs. sem conexão a servidores e implementações iterativos vs. concorrentes de servidores.

3.2 O Algoritmo conceitual do servidor

Conceitualmente, um servidor segue um algoritmo simples: cria um soquete e o liga a uma porta conhecida onde espera conexões. O servidor entra então em um ciclo infinito no qual aceita o próximo pedidos de um cliente, processa o pedido, formula a resposta e a envia ao cliente.

Infelizmente, este algoritmo é válido apenas para um pequeno conjunto de servidores triviais. Considere, por exemplo, um serviço de transferência de arquivos que necessita de um tempo substancial para tratar cada pedido. Suponha que um primeiro cliente peça a transferência de um arquivo grande (por exemplo 800 megabytes), enquanto que um segundo cliente peça a transferência de um arquivo pequeno (por exemplo 20 bytes). Se o servidor esperar até transmitir completamente o primeiro arquivo antes de iniciar a transferência do segundo, o segundo cliente deverá esperar um tempo excessivamente longo para conseguir a transferência de um pequeno arquivo. Como o arquivo desejado é pequeno, o segundo usuário espera que o servidor responda imediatamente. Na prática, os servidores manipulam pedidos pequenos rapidamente, porque eles tratam mais do que um pedido ao mesmo tempo.

3.3 Servidores concorrentes versus *servidores iterativos.*

O termo servidor iterativo é usado para descrever implementações que processam um único pedido de cada vez, e o termo servidor concorrente para descrever implementações que manipulam múltiplos pedidos ao mesmo tempo. Neste contexto, o termo servidor concorrente está relacionado à capacidade do servidor de manipular várias requisições concorrentemente e não à sua implementação usar vários processos ou linhas de execução (*threads*). O importante é que, sob a perspectiva do cliente, o servidor parece se comunicar com vários clientes de forma concorrente.

Em geral, servidores concorrentes são mais difíceis de projetar e construir, e o código resultante é mais complexo e mais difícil de modificar. Entretanto, a maioria dos projetistas escolhe implementar servidores concorrentes, porque servidores iterativos introduzem atrasos desnecessários em aplicações distribuídas e podem gerar gargalos de desempenho que afetam a muitas aplicações clientes.

3.4 Acesso orientado a conexão versus *acesso sem conexão*

No conjunto de protocolos TCP/IP, o TCP fornece um serviço de transporte orientado a conexão enquanto que o UDP fornece um serviço sem conexão. Por definição, os servidores que usam o TCP são servidores orientados a conexão e os servidores que usam o protocolo UDP são servidores sem conexão[1].

A escolha entre um serviço orientado a conexão e um serviço sem conexão depende do protocolo de aplicação. Um protocolo de aplicação projetado para usar um serviço de transporte orientado a conexão pode funcionar de forma incorreta ou ineficiente se usar um protocolo de transporte sem conexão.

3.5 Servidores orientados a conexão

A grande vantagem da abordagem orientada a conexão está na facilidade de programação. Como o protocolo de transporte trata automaticamente problemas como a perda de pacotes e a entrega fora de ordem, o servidor não precisa se preocupar com isso. Em vez disso, um servidor orientado a conexão administra e usa conexões. Ele aceita pedidos de conexões de clientes, comunica-se através das conexões abertas e as fecha após ter completado a interação.

Enquanto uma conexão permanece aberta, o TCP fornece todos os requisitos de confiabilidade. Ele retransmite dados perdidos, verifica se os dados chegaram sem erro de transmissão e, quando necessário, reordena os pacotes de entrada. Quando um cliente envia um pedido, o TCP o entrega corretamente ou informa ao cliente que a conexão foi interrompida. Da mesma forma, o servidor pode contar com o TCP para entregar as respostas ao cliente ou para ser informado de que a entrega não é possível.

Servidores orientados a conexão também possuem desvantagens. Servidores orientados a conexão necessitam de um soquete separado para cada conexão, enquanto que servidores sem conexão permitem a comunicação com múltiplos hospedeiros usando um único soquete. A alocação de soquetes e a administração de conexões podem ser especialmente importantes para os sistemas operacionais, já que eles devem rodar permanentemente sem exaurir os recursos disponíveis. Para aplicações triviais, a sobrecarga do protocolo de apresentação em três vias usado para estabelecer e terminar uma conexão faz o TCP oneroso se comparado ao UDP. A desvantagem mais importante é que o TCP não envia pacotes através de uma conexão ociosa. Suponha que um cliente estabelece uma conexão com um servidor, troca pedidos e respostas e então cai: assim que o servidor tiver respondido a todos os pedidos recebidos ele não enviará mais dados ao cliente, provocando uma má utilização de seus recursos. O servidor tem estruturas de dados (incluindo espaço em *buffer*) alocadas para a

conexão, e estes recursos não serão mais usados. Um servidor é projetado para rodar continuamente e se seus clientes caírem repetidamente ficará sem recursos (ou seja, sem soquetes, espaço em *buffer*, conexões TCP) e não poderá atender a novos pedidos de conexão.

3.6 Servidores sem conexão

Servidores sem conexão também possuem vantagens e desvantagens. Se por um lado os servidores sem conexão não sofrem do problema da diminuição (ou esgotamento) de recursos, por outro lado, não podem depender da camada de transporte subjacente para obter a entrega confiável dos dados. Um dos lados deve assumir a responsabilidade pela entrega confiável. Normalmente, os clientes se responsabilizam por retransmitir os pedidos cujas respostas não chegarem. Se o servidor necessitar dividir a sua resposta em vários pacotes de dados, ele também pode precisar implementar um mecanismo de retransmissão.

Conseguir confiabilidade através de mecanismo de retransmissão e de tempos de espera (*timeouts*) pode ser extremamente difícil, de modo que programadores novatos devem procurar usar um serviço de transporte orientado a conexão.

A escolha de um projeto orientado a conexão ou não orientado a conexão depende também de a aplicação requerer comunicação do tipo *broadcast* ou *multicast*, já que esse tipo de comunicação é oferecido apenas pelo UDP.

3.7 Servidores com e sem estado

A informação que um servidor mantém sobre as interações correntes com os clientes é chamada de *informação de estado*. Servidores que guardam informação de estado são chamados de *servidores com estado* e os outros de *servidores sem estado.* A busca de eficiência motiva projetistas a manterem informação de estado nos servidores. A informação de estado pode reduzir o tamanho das mensagens que clientes e servidores trocam, e pode permitir que os servidores respondam rapidamente. Essencialmente, informação de estado permite que um servidor lembre-se do que o cliente requisitou anteriormente e compute uma resposta incremental à medida que cada novo pedido chegar. |Por outro lado, a motivação para não manter estado está associada à confiabilidade dos protocolos: a informação de estado em um servidor pode ficar incorreta se mensagens forem perdidas, duplicadas ou entregues fora de ordem, ou se o cliente cair e reiniciar a sua execução.

---

[1] A interface *socket* permite que uma aplicação conecte um *socket* UDP a um ponto remoto, mas a conexão com o servidor não é efetivamente realizada.

3.8 Quatro tipos básicos de servidores

Servidores podem ser iterativos ou concorrentes, e podem usar transporte orientado a conexão ou transporte sem conexão. A Fig. 2 mostra que essas propriedades agrupam os servidores em quatro categorias gerais.

sem conexão iterativo	orientado a conexão iterativo
sem conexão concorrente	orientado a conexão concorrente

Figura 1: As quatro categorias gerais definidas pelo oferecimento de concorrência e pelo uso de serviço de transporte orientado a conexão.

3.9 Tempo de processamento de uma requisição

Em geral, servidores iterativos são adequados apenas a protocolos de aplicação triviais. O teste para saber se uma implementação iterativa será suficiente foca o tempo de resposta necessário, que pode ser medida local e globalmente.

Define-se o *tempo de processamento de uma requisição* pelo servidor como sendo o tempo total que o servidor leva para tratar um pedido isolado, e o *tempo de resposta observado* pelo cliente como sendo o atraso total entre o instante em que o pedido é enviado e o instante em que a resposta do servidor chega. Evidentemente, o *tempo de resposta observado* nunca poderá ser menor que o *tempo de processamento de uma requisição*, entretanto, se o servidor mantêm uma fila de pedidos esperando atendimento, o tempo de resposta observado pode ser muito maior que o tempo de processamento de uma requisição.

Servidores iterativos manipulam um único pedido por vez. Se um pedido chega enquanto o servidor estiver ocupado manipulando um pedido anterior, o sistema enfileira o novo pedido. Quando o servidor termina o processamento de um pedido, ele consulta a fila. Se N denota o comprimento médio da fila de pedidos, o tempo de resposta para um pedido de chegada será aproximadamente igual a N/2 + 1 vezes o tempo que o servidor leva para processar um pedido. Como o tempo de resposta observado é proporcional a N, a maioria das implementações restringe N a um valor pequeno (por exemplo 5) e usa-se servidores concorrentes quanto uma fila pequena não for suficiente.

A carga total tratada pelo servidor também deve ser considerada na decisão de implementação de servidores iterativos ou concorrentes. Um servidor projetado para manipular K clientes, cada qual

enviando R pedidos por segundo, deve ter um tempo de processamento menor que 1/KR segundos por pedido. Se o servidor não puder tratar os pedidos a essa taxa, ocorrerá o transbordamento na fila de espera. Nesse tipo de situação, o projetista deverá considerar a implementação de um servidor concorrente.

3.10 Algoritmos para servidores iterativos

Um servidor iterativo é fácil de projetar, programar, depurar e modificar. Por isso, a maioria dos programadores escolhe projetar servidores iterativos sempre que a sua execução fornece respostas suficientemente rápidas para a carga esperada. Normalmente, servidores iterativos são mais adequados a serviços simples, acessados por protocolos sem conexão. Entretanto é possível implementar servidores iterativos com conexão e sem conexão.

3.11 Um algoritmo para servidor iterativo orientado a conexão

O **Algoritmo 1** apresenta o algoritmo para um servidor iterativo acessado através do protocolo orientado a conexão TCP. Os passos do algoritmo são discutidos a seguir com mais detalhes.

Algoritmo 1
1. Criar um soquete e ligá-lo (*bind*) ao endereço conhecido para o serviço oferecido.
2. Colocar o soquete no modo passivo, tornando-o pronto para ser usado pelo servidor.
3. Aceitar o próximo pedido de conexão ao soquete e obter um novo soquete para a conexão.
4. Repetidamente, ler pedidos do cliente, formular uma resposta e enviar a resposta de volta ao cliente de acordo com o protocolo de aplicação.
5. Quando terminar de atender ao cliente, fechar a conexão e retornar ao passo 3 para aceitar uma nova conexão.

Algoritmo 1: Servidor iterativo orientado a conexão. Um único processo manipula conexões de clientes, um por vez.

3.12 Ligando um servidor a um endereço usando INADDR_ANY

Um servidor precisa criar um soquete e ligá-lo a uma porta conhecida associada ao serviço oferecido. Como os clientes, os servidores usam o procedimento *getportbyname()* para mapear o nome de um serviço em um número de porta conhecida. Por exemplo, o TCP/IP define um serviço de ECHO. Um servidor que implementa o serviço de ECHO usa *getportbyname()* para mapear a cadeia "echo" à porta 7.

Quando a primitiva *bind()* especifica um ponto de conexão final para um soquete, ela usa a estrutura *sockaddr_in*, que contém tanto o endereço IP como o número da porta do protocolo. Logo

bind() não pode especificar um número de porta de protocolo sem especificar um endereço IP. Infelizmente, selecionar o endereço IP do servidor onde serão aceitas conexões pode ser difícil. Roteadores e hospedeiros *multi-homed* podem ter vários endereços IP. Se o servidor especificar um endereço IP particular, o soquete não aceitará comunicações de clientes que enviarem pedidos para um outro IP da máquina.

Para resolver este problema, a interface *socket* define uma constante especial, INADDR_ANY, que pode ser usada no lugar de um endereço IP. INADDR_ANY especifica um *endereço coringa* que pode assumir qualquer um dos endereços IPs do hospedeiro. INADDR_ANY torna possível a um mesmo servidor aceitar qualquer comunicação endereçada a qualquer endereço IP do hospedeiro.

3.13 Colocando o soquete no modo passivo

Um servidor TCP chama *listen()* para colocar o servidor no modo passivo. *listen()* também recebe um argumento que especifica o comprimento da fila interna de requisições para o soquete. A fila de pedidos guarda os pedidos de conexão TCP que chegaram de clientes e ainda não foram aceitas.

3.14 Aceitando conexões e usando-as.

O servidor TCP chama *accept()* para obter o próximo pedido de conexão (ou seja, para retirá-lo da fila de pedidos de conexão). A chamada retorna o descritor de um soquete a ser usado pela nova conexão. Uma vez aceita uma nova conexão, o servidor usa *recv()* ou *read()* para obter pedidos do protocolo de aplicação do cliente e *send()* ou *write()* para enviar respostas. Finalmente, ao terminar o atendimento ao cliente, o servidor chama *close()* para liberar o soquete.

3.15 Algoritmo de servidor iterativo sem conexão

Servidores iterativos são mais adequados para serviços que têm um baixo tempo de processamento de requisição. A maioria dos servidores iterativos usa protocolos não orientados a conexão (v. Algoritmo 2), como o UDP, pois os protocolos de transporte orientados a conexão, como o TCP, apresentam maiores custos (*overhead*).

Algoritmo 2
1. Criar um soquete e ligá-lo a um endereço conhecido para o serviço sendo oferecido.
2. Repetidamente, ler a próxima requisição do cliente, formular uma resposta e enviá-la ao cliente de acordo com o protocolo de aplicação.

Algoritmo 2: Servidor iterativo sem conexão. Uma única linha de execução trata pedidos (datagramas) de clientes, um por vez.

O procedimento de criação de um soquete para um servidor iterativo sem conexão é o mesmo usado para o servidor orientado a conexão. Neste caso, porém, o soquete do servidor permanece desconectado e pode aceitar datagramas de entrada de qualquer cliente.

3.16 Formando o endereço de resposta em um servidor sem conexão

A interface *socket* fornece duas formas de especificar um ponto terminal de comunicação remoto. Conforme visto anteriormente, uma dessas formas é usando *connect()*. Entretanto, um servidor sem conexão não pode usar *connect()* para encontrar o endereço remoto, pois procedendo dessa maneira restringiria o servidor a comunicar-se com um único endereço IP e número de porta. Portanto, um servidor sem conexão usa um soquete desconectado. Ele gera endereços de resposta explicitamente e usa *sendto()* para especificar tanto o datagrama a ser enviado como o endereço para onde o datagrama deverá ser enviado. A primitiva *sendto()* tem a seguinte forma:

```
retcode = sendto(s, message, len, flags, toaddr, toaddrlen);
```

onde `s` é um soquete desconectado, `message` é o endereço de um *buffer* que contém os dados que devem ser enviados, `len` especifica o número de bytes no buffer, `flags` especifica opções de controle e depuração, `toaddr` é um ponteiro para uma estrutura *sockaddr_in* que contém o endereço do destinatário da mensagem e `toaddrlen` é um inteiro que especifica o tamanho da estrutura de endereço.

A interface *socket* fornece uma maneira fácil para que servidores sem conexão obtenham o endereço do cliente. A chamada *recvfrom()* tem dois argumentos que especificam dois *buffers*. O sistema coloca o datagrama recebido em um *buffer* e o endereço do remetente no outro *buffer*. A chamada *recvfrom()* tem a seguinte forma:

```
retcode = recvfrom (s, buf, len, flags, from, fromlen);
```

onde `s` especifica um soquete, `buf` especifica o *buffer* onde o sistema irá colocar o próximo datagrama, `len` especifica o espaço disponível (bytes) no buffer, `flags` controla casos especiais (p.ex. verificar se chegou um datagrama sem retirá-lo do soquete), `from` especifica o segundo *buffer* onde o sistema colocará o endereço da fonte, `fromlen` especifica o endereço de um inteiro. Inicialmente, o inteiro para onde `fromlen` aponta especifica o tamanho do buffer `from`. Quando a chamada retorna, `fromlen` conterá o tamanho do endereço da fonte que o sistema colocará no *buffer* `from`. Para gerar uma resposta o servidor usa o endereço que *recvfrom()* armazena no buffer `from` quando o pedido chega.

3.17 Algoritmos de servidores concorrentes

A principal razão para introduzir concorrência em servidores é a necessidade de fornecer resposta rápida a múltiplos clientes. A concorrência diminui o tempo de resposta sob as seguintes condições:

- Para formar uma resposta é necessária uma quantidade significativa de operações E/S;
- O tempo de processamento varia consideravelmente entre os pedidos; ou
- O servidor é executado em um computador com vários processadores.

No primeiro caso, permitir que o servidor calcule respostas concorrentemente significa que ele pode sobrepor o uso do processador com o uso de dispositivos periféricos, mesmo que a máquina tenha apenas uma CPU. Enquanto o processador trabalha para formar uma resposta, os dispositivos de E/S podem transferir para a memória os dados necessários para outras respostas.

No segundo caso, o mecanismo de compartilhamento de tempo permite que um único processador manipule pedidos que necessitem pequena quantidade de processamento, sem que esses pedidos tenham que esperar pelo atendimento de outros mais demorados.

No terceiro caso, a execução paralela em um computador com múltiplos processadores permite que um processador processe uma resposta, enquanto outro processador se encarrega de outra resposta. A maioria dos servidores se adapta automaticamente ao *hardware* disponível - quanto maior a quantidade de recursos de *hardware* (p. ex. mais processadores) melhor o desempenho dos servidores.

3.18 Linhas de execução mestre e escravas

Embora seja possível a um servidor propiciar algum nível de concorrência usando uma única linha de execução (*thread*), a maioria dos servidores concorrentes usa várias linhas de execução..

As linhas de execução de um servidor concorrente podem ser divididas em dois tipos: mestra e escrava. Uma linha conhecida como *mestra* inicia a execução. A linha mestra abre um soquete associado à porta conhecida para o serviço, espera pelo próximo pedido e cria uma *escrava* (possivelmente em um novo processo) para tratar o pedido. A mestra nunca se comunica diretamente com o cliente, ela passa esta responsabilidade para a escrava. Cada escrava trata da comunicação com um cliente. Após formar uma resposta e enviá-la ao cliente a escrava termina a sua execução.

3.19 Algoritmo de servidor concorrente sem conexão

A versão mais direta de servidor concorrente sem conexão é apresentada pelo algoritmo 3. Deve-se lembrar que embora o custo exato de criar uma nova *thread* ou processo dependa do sistema operacional e da arquitetura subjacente, a operação pode ser dispendiosa. No caso de protocolos sem

conexão deve-se considerar cuidadosamente se o custo da concorrência não é maior que o ganho em velocidade. De fato, como criar um processo ou *thread* é dispendioso, poucos servidores sem conexão têm implementações concorrentes.

Algoritmo 3	
Mestra 1.	Criar um soquete e ligá-lo ao endereço conhecido para o serviço que está sendo oferecido. Deixar o soquete desconectado.
Mestra 2.	Repetir: chamar *recvfrom()* para receber o próximo pedido de um cliente e criar uma nova escrava (possivelmente em um novo processo) para manipular a resposta.
Escrava 1.	Começa com uma requisição específica passada pela Mestra assim como com acesso ao soquete.
Escrava 2.	Formar uma resposta de acordo com o protocolo de aplicação e enviá-la ao cliente usando *sendto()*.
Escrava 3.	Sair (ou seja, a escrava termina após tratar um pedido).

Algoritmo 3 : Servidor Concorrente e sem conexão. A servidora mestra aceita pedidos de entrada (datagramas) e cria escravas (processos/threads) para manipular cada um deles.

3.20 Algoritmo de servidor concorrente orientado a conexão

Protocolos de aplicação orientados a conexão usam uma conexão como paradigma básico para a comunicação. Estes protocolos permitem que os clientes estabeleçam uma conexão com um servidor, comuniquem-se através da conexão e finalmente descartem a conexão.

O Algoritmo 4 especifica os passos que um servidor concorrente usa para um protocolo orientado a conexão.

Algoritmo 4	
Mestra 1.	Criar um soquete e ligá-lo a uma porta popular associada ao serviço oferecido. Deixar o soquete desconectado.
Mestra 2.	Colocar o soquete no modo passivo, deixando-o pronto para ser usado pelo servidor.
Mestra 3.	Repetir: chamar *accept()* para receber o próximo pedido de um cliente, e cria uma nova *thread* ou processo escravo para tratar da resposta.
Escrava 1.	Começa com uma conexão passada pela Mestra (ou seja, um soquete para a conexão).
Escrava 2.	Interagir com o cliente usando a conexão: ler os pedidos e enviar as respostas de volta.
Escrava 3.	Fechar a conexão e sair. A escrava sai após tratar todos os pedidos de um cliente.

Algoritmo 4: Servidor concorrente orientado a conexão. A linha de execução mestra aceita uma conexão, cria uma escrava para tratá-la. Uma vez que a escrava tenha terminado, ela fecha a conexão.

Como no caso sem conexão, a linha servidora mestra nunca se comunica com o cliente diretamente. Enquanto a escrava interage com o cliente, a mestra espera por outras conexões.

3.21 Implementação de concorrência em servidores

Como o Linux (e o Unix) oferecem duas formas de concorrência, processos e *threads,* duas implementações do paradigma mestre-escravo são possíveis. Na primeira, o servidor cria vários processos, cada um com uma única linha de execução. Na outra, o servidor cria várias linhas de execução (*threads*) dentro do mesmo processo.

3.22 Usando programas separados como escravos

O Algoritmo 4 mostra como um servidor concorrente cria uma novo escrava para cada conexão. Na implementação baseada em processos, isso é feito pela servidora mestra usando a chamada da primitiva de sistema *fork()*. Para protocolos de aplicação simples, um único programa servidor pode conter todo o código necessário para a mestra e a escrava. Após a chamada ao *fork()*, o processo original volta para aceitar o próximo pedido de conexão, enquanto o novo processo torna-se o escravo e manipula a conexão. Em alguns casos, é mais conveniente que o processo escravo execute o código de um programa que tenha sido escrito e compilado independentemente. Sistemas como o Linux e o UNIX podem tratar disso facilmente porque eles permitem que o processo escravo chame *execve()* após a chamada de *fork(). Execve()* sobrecarrega o processo escravo com o código de um novo programa.

3.23 Concorrência aparente usando uma única linha de execução

Em alguns casos, pode ser interessante usar uma única linha de execução para tratar as requisições de clientes concorrentemente. Em particular, em alguns SOs a criação de *threads* ou processos é tão dispendiosa que os servidores não podem criar uma nova *thread* para cada requisição ou conexão. Mais importante, muitos protocolos de aplicação requerem que o servidor compartilhe informação entre as conexões.

O sistema X Window é um exemplo típico de uso de servidor com *concorrência aparente* e uma única linha de execução. O X permite que vários clientes insiram texto ou gráficos em janelas que aparecem em uma tela mapeada por *bits* (*bit-mapped*). Cada cliente controla uma janela. enviando pedidos que atualizam seu conteúdo. Cada cliente opera independentemente, e pode passar muito tempo sem mudar a tela, ou pode atualizá-la freqüentemente. Um servidor para o sistema X integra informações obtidas dos clientes em uma sessão de memória única e contígua chamada *display buffer*. Os dados que chegam dos clientes contribuem para uma estrutura de dados única e compartilhada.

Embora seja possível obter a desejada concorrência usando *threads* que compartilham memória, também é possível obter *concorrência aparente* se a carga total dos pedidos apresentados ao servidor não exceder a sua capacidade para tratá-los. Para fazer isso, o servidor opera como um *thread* único que usa a chamada ao sistema *select()* para operações de E/S assíncronas. O Algoritmo 5

descreve um servidor com uma única linha de execução que manipula várias conexões concorrentemente.

Algoritmo 5
1. Criar um soquete e ligá-lo à porta bem-conhecida para o serviço. Adicionar o soquete a uma lista de soquetes através dos quais a E/S é possível.
2. Usar *select()* para esperar por E/S nos soquetes existentes.
3. Se o soquete original estiver pronto, usar *accept()* para obter a próxima conexão e adicionar o novo socket à lista de soquetes onde E/S é possível.
4. Se algum soquete diferente do original estiver pronto, usar *recv()* ou *read()* para obter o próximo pedido, formar a resposta e usar *send()* ou *write()* para enviar a resposta de volta ao cliente.
5. Continuar no passo 2 acima.

Algoritmo 5: Servidor concorrente orientado a conexão implementado por uma única thread. A thread espera pelo próximo descritor que estiver pronto, o que pode significar ser a chegada de uma nova conexão que um cliente enviou uma mensagem através, de uma conexão já existente.

3.24 O problema da ocorrência de impasse (deadlock) no servidor

Muitas implementações compartilham uma falha importante: o servidor pode estar sujeito à ocorrência de impasses (*deadlock*[2]). Considere um servidor iterativo orientado a conexão. Suponha que uma aplicação cliente mal comportada faça uma conexão com o servidor mas nunca envie pedidos. O servidor aceitará a nova conexão e chamará *read()* para extrair os pedidos, ficando bloqueado numa chamada ao sistema, esperando por uma requisição que nunca chegará.

A condição de impasse no servidor pode ser muito mais sutil se o cliente se comportar mal não consumindo recursos. Por exemplo, suponha que um cliente estabeleça uma conexão a um servidor, envie-lhe uma seqüência de requisições, mas nunca leia as respostas. O servidor recebe as requisições, gera as respostas e as envia ao cliente. No lado servidor, o protocolo TCP transmite os primeiros *bytes* para o cliente através da conexão. Eventualmente, o mecanismo de controle de fluxo do TCP irá perceber que o *buffer* de recepção do cliente está cheio e parará de enviar dados. Se o programa de aplicação servidor continuar gerando respostas, o *buffer* local do TCP, que armazena dados de saída para a conexão, irá se encher e o processo servidor ficará bloqueado.

Impasses acontecem porque os processos ficam bloqueados quando o sistema operacional não pode satisfazer uma chamada ao sistema. Em particular, uma chamada a *write()* bloqueará o processo que a invocou se o TCP não tiver espaço no *buffer* local para onde os dados serão enviados; uma chamada a *read()* bloqueará o processo que a invocou até o TCP receber dados. Em servidores

[2] O termo impasse, ou *deadlock,* refere-se a uma condição na qual um programa ou conjunto de programas não pode prosseguir porque estão bloqueados esperando por um evento que nunca acontecerá. No caso tratado, a ocorrência de um impasse leva a que servidor pare de responder a pedidos.

concorrentes, ficará bloqueado apenas a *thread* escrava associada ao cliente que não envia requisições ou não recebe as respostas. Entretanto, em implementações que usam uma única *thread*, o servidor central ficará bloqueado, não podendo tratar outras conexões. A questão importante aqui é que qualquer servidor que use apenas uma *thread* estará sujeito à ocorrência de impasses.

3.25 Resumo

Foram apresentados algoritmos e discutidos alguns aspectos importantes a considerar quando do projeto de *software* servidor.

Bibliografia

COMER, D. E., STEVENS, D. L., Internetworking With TCP/IP Volume III: Client-Server Programming and Applications, Linux/POSIX Socket Version, Prentice-Hall International 2001, *Capítulo 8*.

STEVENS,W.R.; "UNIX NETWORK PROGRAMMING - Networking APIs: Sockets and XTI" - Volume 1 - Second Edition - Prentice Hall - 1998

Obs. Resumo preparado por Magda Patrícia Caldeira Arantes e revisado por Juan Manuel Adán Coello.