Redes

Conectando-se com um Servidor

Antes de escrever nosso primeiro programa de rede, vamos aprender sobre uma excelente ferramenta de depuração para programação de rede que você já tem, a saber, telnet. Os sistemas Unix sempre vêm com telnet; o Windows 95/98 e NT/2000 também vêm com um programa de telnet simples. Entretanto, ele é opcional e você pode não tê-lo instalado quando instalou o sistema operacional. Basta procurar TELNET.EXE no diretório \Windows. Se você não encontrá-lo, execute a instalação do Windows novamente.

Você pode ter usado telnet para conectar-se a um computador remoto e para verificar seu correio eletrônico, mas é possível usá-lo também para se comunicar com outros serviços fornecidos pelos hosts da Internet. Aqui está um exemplo do que você pode fazer:

1. Inicie o telnet.
2. No campo nome do host, digite time-A.timefreq.bldrdoc.gov
3. No campo porta, digite 13. (Não importa o tipo de terminal escolhido.)

Você deve obter uma linha como a seguinte:

```
53575 05-07-24 11:05:41 50 0 0 866.3 UTC(NIST) *
```

O que está acontecendo? Você se conectou com o serviço “hora do dia” que a maioria das máquinas Unix executa constantemente. O servidor em particular com que você estabeleceu uma conexão é operado pelo National Institute os Standards and Tecnology, em Boulder, Colorado, EUA e fornece a medida de um relógio atômico de césio. (É claro que a hora reportada não está completamente precisa, devido aos atrasos da rede.) Por convenção, o serviço “hora do dia”está sempre associado à porta 13.

O que está acontecendo é que o software servidor está continuamente sendo executado na máquina remota, esperando por qualquer tráfego de rede que queira falar com a porta 13. Quando o sistema operacional do computador remoto recebe um pacote de rede que contém um pedido para se conectar à porta número 13, ele ativa o processo servidor ouvidor e estabelece a conexão. A conexão permanece ativa até ser terminada por uma das partes.

Quando você iniciou a sessão de telnet com time-A.timefreq.bldrdoc.gov na porta 13, um software de rede não relacionado sabia o suficiente para converter a string “time-A.timefreq.bldrdoc.gov” em seu endereço IP correto, 132.163.4.102. Em seguida, o software enviou um pedido de conexão para esse computador, solicitando uma conexão à porta 13. Uma vez que a conexão foi estabelecida, o programa remoto enviou de volta uma linha de dados e depois fechou a conexão. Em geral, é claro, os clientes e servidores

se envolvem em um diálogo mais amplo, antes que um ou outro feche a conexão.

Aqui está outra experiência, da mesma espécie, que é um pouco mais interessante. Primeiro, ative o eco de tecla local. (No Windows 95/98, isso é feito a partir da caixa de diálogo Preferências de terminal aberta selecionando-se a opção Preferências do menu Terminal, no programa de telnet do Windows.) Em seguida, faça o seguinte:

1. Conecte-se com java.sun.com, na porta 80.
2. Digite o seguinte, exatamente como aparece, sem pressionar backspace.

```
GET / HTTP/1.0
```

3. Agora, pressione a tecla ENTER duas vezes.

A resposta que você recebeu é uma página de texto formatado em HTML, a saber, a página da Web principal do Site Web da Sun que trata da Tecnologia Java.

Esse é exatamente o mesmo processo pelo qual seu navegador da Web passa para obter uma página da Web.

Nosso primeiro programa de rede fará a mesma coisa que fizemos usando telnet para conectar-se com uma porta e imprimir o que encontrar.

Vide o código SocketTest.java.

Esse programa é extremamente simples, nas antes de analisarmos as duas linhas principais note que estamos importando a classe java.net e capturando todos os erros de entrada/saída (I/O), pois o código fica envolvido em um bloco try/catch. (Como muitas coisas podem dar errado em uma conexão de rede, a maioria dos métodos relacionados a rede ameaça lançar erros de I/O. Você deve capturá-los para que o código possa ser compilado.)

Quanto ao código em si, as linhas principais são as seguintes:

```
Socket s = new Socket("time-a.timefreq.bldrdoc.gov", 13);
BufferedReader in = new BufferedReader(new
                    InputStreamReader(s.getInputStream()));
```

A primeira linha abre um soquete, que é uma abstração para o software de rede que permite a comunicação para fora e para dentro desse programa. Passamos um endereço remoto e o número de porta para o construtor do soquete. Se a conexão falhar, então uma UnknownHostException será lançada. Se houver outro problema, então uma IOException ocorrerá. Como UnknownHostException é derivada de IOException e esse é um exemplo de programa, capturamos apenas a classe base.

Uma vez aberto o soquete, o método getInputStream em java.net.Socket retorna um objeto InputStream, que você pode usar exatamente como qualquer outro fluxo. Uma vez que você tenha pego o fluxo, esse programa simplesmente

1. Lê todos os caracteres enviados pelo servidor, usando readLine

2. Imprime cada linha na saída padrão

Esse processo continua até que o fluxo tenha terminado ou o servidor se desconecte. Você sabe que isso acontece, quando o método readLine retorna uma string null.

Claramente, a classe Socket é agradável e fácil de usar, pois a tecnologia Java oculta as complexidades do estabelecimento de uma conexão de rede e do envio de dados por ela. O pacote java.net fornece basicamente a mesma interface de programação que você usaria para trabalhar com um arquivo.

Tempo limite de espera (timeout) de Soquete

Em programas reais, você não quer apenas ler de um soquete, pois os métodos de leitura serão bloqueados até que os dados estejam disponíveis. Se o host não pode ser atingido, então seu aplicativo espera por um longo tempo e você está à mercê do sistema operacional para, eventualmente, causar uma interrupção.

Em vez disso, você deve decidir que tempo de limite de espera é razoável para seu aplicativo em particular. Então chame o método setSoTimeout para definir um tempo de limite de espera (em milissegundos).

```
Socket s = new Socket(...);
s.setSoTimeout(10000); // tempo limite de espera: 10 segundos
```

Se o valor de tempo limite de espera foi configurado para um soquete, assim todas as operações de leitura subseqüentes lançarão uma InterruptedIOException, quando o tempo limite de espera tiver sido atingido, antes que a entrada esteja disponível. Você pode capturar essa exceção e reagir ao tempo limite de espera.

```
try {
  String line;
  while((line = in.readLine()) != null) {
    processa line
  }
} catch(InterruptedIOException e) {
  reage a tempo limite de espera esgotado
}
```

Há um único problema. Você precisa ter um objeto soquete para chamar o método setSoTimeout, mas o próprio construtor de Socket pode ser bloqueado indefinidamente, até que uma conexão inicial com o host seja estabelecida. Atualmente a plataforma Java não oferece uma solução conveniente para esse problema, Se isso for uma preocupação, você precisará construir o soquete em uma linha de execução separado e esperar que a linha de execução termine ou esgote o tempo limite de espera.

O próximo exemplo mostra como conseguir isso. Você usa o método estático SocketOpener.openSocket para abrir um soquete com determinado tempo limite de espera:

```
Socket s = new SocketOpener.openSocket(host, port, timeout);
```

Esse método inicia uma nova linha de execução, cujo método run simplesmente tenta construir o soquete.

```
public void run() {
  try {
    socket = new Socket(host,port);
  } catch(IOException) {}
}
```

O método openSocket inicia a linha de execução e chama o método join com um parâmetro tempo limite de espera (timeout). O método join retorna quando a linha de execução morre ou quando o tempo limite de espera tiver-se esgotado, o que acontecer primeiro.

```
t.start();


try {
  t.join(timeout);
} catch(InterruptedException e) {}
```

Você pode simplesmente usar a classe SocketOpener em seus próprios programas, se quiser limitar o tempo de espera da construção de um soquete.

Vide o cógido SocketOpenerTest.java.

Endereços de Internet

Normalmente, você não precisa se preocupar muito com os endereços da Internet – os endereços de host em formato numérico tais como 132.163.4.102. Entretanto, você pode usar a classe InetAddress, se precisar converter entre nomes de host e endereços da Internet.

O método estático getByName retorna um objeto InerAddress de um host. Por exemplo:

```
InetAddress address = InerAddress.getByName( “java.sun.com”);
```

O método acima retorna um objeto InetAddress que encapsula a seqüência de quatro bytes 209.249.116.141. Você pode acessar os bytes com o método getBytes.

```
Byte[] addressBytes = address.getBytes();
```

Alguns nomes de host com muito tráfego correspondem a múltiplos endereços de

internet, para facilitar a distribuição da carga de acessos. Por exemplo, o nome do host www.google.com corresponde a três endereços de Internet diferentes. Um deles é escolhido aleatoriamente quando o host é acessado. Você pode obter todos os hosts com o método getAllByName.

```
InetAddress[] addresses = InerAddress.getAllByName(host);
```

Finalmente, às vezes você precisa do endereço do host local. Se você simplesmente solicitar o endereço de localhost, sempre obterá o endereço 127.0.0.1. Em vez disso, use o método estático getLocalHost para obter o endereço de seu host local.

```
InerAddress address = InerAddress.getLocalHost();
```

O programa InetAddressTest é um programa simples que imprime o endereço de Internet de seu host local, caso você não especifique quaisquer parâmetros de linha de comandos, ou todos os endereços de Internet de outro host, caso especifique o nome de host na linha de comando.

Vide o código InerAddressTest.java.

Implementando Servidores

Agora que já implementamos um cliente de rede básico que recebe dados da Internet, vamos implementar um servidor simples que pode enviar informações para a Internet. Quando você inicia o programa servidor, ele espera que algum cliente se conecte com sua porta. Escolhemos o número de porta 8189, que não é usado por nenhum dos serviços padrão. A classe ServerSocket é usada para estabelecer um soquete. Em nosso caso, o comando que estabelece um servidor que monitora a porta 8189 é:

```
ServerSocket s = new ServerSocket(8189);
```

O comando que diz ao programa para que espere indefinidamente até que um cliente se conecte com essa porta é:

```
Socket incoming = s.accept();
```

Quando alguém se conectar com essa porta, enviando o pedido correto através da rede, esse método retorna um objeto Socket que representa a conexão feita. Você pode usar esse objeto para obter um leitor de entrada e um escritor de saída desse soquete, como se vê no código a seguir:

```
BufferedReader in = new BufferedReader(new
                    InputStreamReader(incoming.getInputStream()));
PrinterWriter out = new
                    PrinterWriter(incoming.getOutputStream(), true
                    /* descarga automática */);
```

Tudo que o servidor envia para o fluxo de saída do servidor se torna a entrada do programa cliente e toda a saída do programa cliente acaba no fluxo de entrada do servidor.

Em todos os exemplos, transmitiremos texto através de soquetes. Portando, transformaremos os fluxos em leitores e escritores. Então, podemos usar o método readLine(definido em BufferedReader, mas não em InputStream) e o método print (definido em PrinterWriter, mas não em OutputStream). Se quiséssemos transmitir dados binários, transformaríamos os fluxos em DataInputStream e DataOutputStream. Para transmitir objetos serializados, usaríamos ObjectInputStream e ObjectOutputStream.

Vamos enviar uma saudação ao cliente:

```
out.println(“Hello! Enter BYE to exit.”);
```

Quando você usar telnet para se conectar com esse programa servidor na porta 8189, verá a saudação acima na tela do terminal.

Nesse servidor simples, lemos apenas a entrada do cliente, uma linha por vez e a ecoamos. Isso demostra que o programa recebe a entrada do cliente. Um servidor real obviamente calcularia e retornaria uma resposta dependente da entrada.

```
String line = in.readLine();
if(line != null) {
  out.println(“Echo: ” + line);
  if(line.trim().equals(“BYE”))
    done = true;
} else
  done = true;
```

No final, fechamos o soquete de entrada.

```
Incoming.close();
```

Isso é tudo. Todo programa servidor, como um servidor http da Web, continua a executar este laço:

1. Ele recebe um comando do cliente (“solicito esta informação”), através de um fluxo de dados de entrada.
2. De algum modo, ele busca a informação.
3. Ele envia as informações para o cliente, através do fluxo de dados de saida.

Vide o código EchoServer.java.

Execute o programa EchoServer e use o telnet para se conectar com o servidor no endereço 127.0.0.1 na porta 8189.

O endereço 127.0.0.1 é especial e se chama endereço de retorno (loopback) local, denotando a máquina local. Como você está executando o servidor de echo de forma local, é onde quer se conectar.

Na verdade, qualquer pessoa no mundo pode acessar seu servidor de echo, desde

que ele estaja em funcionamento e ela saiba seu endereço IP e o número da porta.

Quando você se conectar com a porta, obterá a mensagem:

```
Hello! Enter BYE to exit.
```

Digite qualquer coisa e veja o echo de entrada em sua tela. Digite BYE (todas as letras maiúsculas) para se desconectar. O programa servidor também terminará.

Atendendo a vários clientes

Há um problema no servidor simples do exemplo anterior. Suponha que queiramos permitir que vários clientes se conectem com nosso servidor e clientes de toda a Internet podem usar o servidor ao mesmo tempo. A rejeição de múltiplas conexões permite que qualquer cliente monopolize o serviço, conectando-se com ele por um longo tempo. Podemos fazer muito melhor, através do uso de linhas de execução.

Sempre que soubermos que o programa estabeleceu uma nova conexão de soquete, isto é, quando a chamada a accept teve êxito, ativaremos uma nova linha de execução para cuidar da conexão entre o servidor e esse cliente. O programa principal apenas voltará e esperará pela próxima conexão. Para que isso aconteça, o laço principal do servidor deve ser como segue:

```
while(true) {
  Socket incoming = s.accept();
  Thread t = new ThreadEchoHandler(incoming);
  t.start();
}
```

A classe ThreadEchoHandler é derivada de Thread e contém o laço de comunicação com o cliente em seu método run.

```
class ThreadEchoHandler extends Thread {
  ...
  public void run() {
    try {
      BufferedReader in = new BufferedReader(new
                InputStreamReader(incoming.getInputStream()));
      PrinterWriter out = new
                PrinterWriter(incoming.getOutputStream(), true
                /* descarga automática */);
      String line;
      while((line = in.readLine()) != null) {
        processa line
      }
```

```
        incoming.close();
      } catch(Exception e) {
        trata a exceção
      }
    }
  }
```

Como cada conexão inicia uma nova linha de execução, vários clientes podem se conectar ao servidor simultaneamente. Você pode verificar esse fato facilmente. Compile e execute o programa servidor ThreadEchoServer. Abra várias janelas telnet. Você pode se comunicar através de todas elas simultaneamente. O programa servidor nunca encerra sua execução. Use Ctrl+C para terminá-lo.

Vide o código ThreadEchoServer.java.