Tarefa de Programação 1: Construindo um servidor Web multithreaded

Neste laboratório, será desenvolvido um servidor Web em duas etapas. No final, você terá construído um servidor Web multithreaded, que será capaz de processar múltiplas requisições de serviços simultâneas em paralelo. Você deverá demonstrar que seu servidor Web é capaz de enviar sua home page ao browser Web.

Implementaremos a versão 1.0 do HTTP, definido na RFC-1945, onde requisições HTTP separadas são enviadas para cada componente da página Web. Este servidor deverá manipular múltiplas requisições simultâneas de serviços em paralelo. Isso significa que o servidor Web é multithreaded. No thread principal, o servidor escuta uma porta fixa. Quando recebe uma requisição de conexão TCP, ele ajusta a conexão TCP através de outra porta e atende essa requisição em um thread separado. Para simplificar esta tarefa de programação, desenvolveremos o código em duas etapas. No primeiro estágio, você irá escrever um servidor multithreaded que simplesmente exibe o conteúdo da mensagem de requisição HTTP recebida. Assim que esse programa funcionar adequadamente, você adicionará o código necessário para gerar a resposta apropriada.

Enquanto você desenvolve o código, pode testar seu servidor a partir de um browser Web. Mas lembre que o servidor não está atendendo pela porta padrão 80; logo, é preciso especificar o número da porta junto à URL que você fornecer ao browser Web. Por exemplo, se o nome da sua máquina é `host.someschool.edu`, seu servidor está escutando a porta 6789, e você quer recuperar o arquivo index.html, então deve especificar ao browser a seguinte URL:

```
http://host.someschool.edu:6789/index.html
```

Se você omitir ":6789", o browser irá assumir a porta 80, que, provavelmente, não terá nenhum servidor à escuta.

Quando o servidor encontra um erro, ele envia uma mensagem de resposta com a fonte HTML apropriada, de forma que a informação do erro seja exibida na janela do browser.

Servidor Web em Java: Parte A

Nas etapas seguintes, veremos o código para a primeira implementação do nosso servidor Web. Sempre que você vir o sinal "?", deverá fornecer o detalhe que estiver faltando.

Nossa primeira implementação do servidor Web será multithreaded, e o processamento de cada requisição de entrada terá um local dentro de um thread separado de execução. Isso permite ao servidor

atender a múltiplos clientes em paralelo, ou desempenhar múltiplas transferências de arquivo a um único cliente em paralelo. Quando criamos um novo thread de execução, precisamos passar ao construtor de threads uma instância de algumas classes que implementa a interface `Runnable`. Essa é a razão de se definir uma classe separada chamada `HttpRequest`. A estrutura do servidor Web é mostrada a seguir:

```
import java.io.* ;
import java.net.* ;
import java.util.* ;

public final class WebServer
{
      public static void main(String arvg[]) throws Exception
      {
            .  .  .
      }
}

final class HttpRequest implements Runnable
{
      .  .  .
}
```

Normalmente, servidores Web processam requisições de serviço recebidas através da conhecida porta 80. Você pode escolher qualquer porta acima de 1024, mas lembre-se de usar o mesmo número de porta quando fizer requisições ao seu servidor Web a partir do seu browser.

```
Public static void main(String arvg[]) throws Exception
{
      // Ajustar o número da porta.
      int port = 6789;
      .  .  .
}
```

A seguir, abrimos um socket e esperamos por uma requisição de conexão TCP. Como estaremos atendendo a mensagens de requisição indefinidamente, colocamos a operação de escuta dentro de um laço infinito. Isso significa que precisaremos terminar o servidor Web digitando ^C pelo teclado.

```
// Estabelecer o socket de escuta.
```

```
      ?
// Processar a requisição de serviço HTTP em um laço infinito.
While (true)  {
      // Escutar requisição de conexão TCP.
      ?
      .  .  .
}
```

Quando uma requisição de conexão é recebida, criamos um objeto `HttpRequest`, passando ao seu construtor uma referência para o objeto `Socket` que representa nossa conexão estabelecida com o cliente.

```
//Construir um objeto para processar a mensagem de requisição HTTP.
HttpRequest request = new HttpRequest ( ? );

// Criar um novo thread para processar a requisição.
Thread thread = new Thread(request);

//Iniciar o thread.
Thread.start();
```

Para que o objeto `HttpRequest` manipule as requisições de serviço HTTP de entrada em um thread separado, criamos primeiro um novo objeto `Thread`, passando ao seu construtor a referência para o objeto `HttpRequest`, então chamamos o método `start()` do thread.

Após o novo thread ter sido criado e iniciado, a execução no thread principal retorna para o topo do loop de processamento da mensagem. O thread principal irá então bloquear, esperando por outra requisição de conexão TCP, enquanto o novo thread continua rodando. Quando outra requisição de conexão TCP é recebida, o thread principal realiza o mesmo processo de criação de thread, a menos que o thread anterior tenha terminado a execução ou ainda esteja rodando.

Isso completa o código em `main()`. Para o restante do laboratório, falta o desenvolvimento da classe `HttpRequest`.

Declaramos duas variáveis para a classe HttpRequest: `CRLF` e `socket`. De acordo com a especificação HTTP, precisamos terminar cada linha da mensagem de resposta do servidor com um *carriage return* (CR) e um *line feed* (LF), assim definimos a `CRLF` de forma conveniente. A variável `socket` será usada para armazenar uma referência ao socket de conexão. A estrutura da classe HttpRequest é mostrada a seguir:

```
final class HttpRequest implements Runnable
{
      final static String CRLF = "\r\n";
      Socket socket;

      // Construtor

      public HttpRequest(Socket socket) throws Exception
      {
            this.socket = socket;
      }

      // Implemente o método run() da interface Runnable.
      Public void run()
      {
            .  .  .
      }

      private void processRequest() throws Exception
      {
            .  .  .
      }
}
```

Para passar uma instância da classe `HttpRequest` para o construtor de `Threads`, a `HttpRequest` deve implementar a interface `Runnable`. Isso significa simplesmente que devemos definir um método público chamado `run()` que retorna `void`. A maior parte do processamento ocorrerá dentro do `processRequest()`, que é chamado de dentro do `run()`.

Até este ponto, apenas repassamos as exceções em vez de apanhá-las. Contudo, não podemos repassá-las a partir do `run()`, pois devemos aderir estritamente à declaração do `run()` na interface `Runnable`, a qual não repassa exceção alguma. Colocaremos todo o código de processamento no `processRequest()`, e a partir daí repassaremos as exceções ao `run()`. Dentro do `run()`, explicitamente recolhemos e tratamos as exceções com um bloco try/catch.

```
// Implementar o método run() da interface Runnable.
Public void run()
{
      try {
            processRequest();
      } catch (Exception e) {
```

```
            System.out.println(e);
      }
}
```

Agora, vamos desenvolver o código de dentro do `processRequest()`. Primeiro obtemos referências para os trechos de entrada e saída do socket. Então colocamos os filtros `InputStreamReader` e `BufferedReader` em torno do trecho de entrada. No entanto, não colocamos nenhum filtro em torno do trecho de saída, pois estaremos escrevendo bytes diretamente no trecho de saída.

```
Private void processRequest() throws Exception
{
      // Obter uma referência para os trechos de entrada e saída do socket.
      InputStream is = ?;
      DataOutputStream os = ?;
      // Ajustar os filtros do trecho de entrada.
      ?
      BufferedReader br = ?;
      .  .  .
}
```

Agora estamos preparados para capturar mensagens de requisição dos clientes, fazendo a leitura dos trechos de entrada do socket. O método `readLine()` da classe `BufferedReader` irá extrair caracteres do trecho de entrada até encontrar um caracter fim-de-linha, ou em nosso caso, uma seqüência de caracter fim-de-linha CRLF.

O primeiro item disponível no trecho de entrada será a linha de requisição HTTP. (Veja Seção 2.2 do livro-texto para a descrição disso e dos seguintes campos.)

```
// Obter a linha de requisição da mensagem de requisição HTTP.
String requestLine = ?;
//  Exibir a linha de requisição.
System.out.println();
System.out.println(requestLine);
```

Após obter a linha de requisição do cabeçalho da mensagem, obteremos as linhas de cabeçalho. Desde

que não saibamos antecipadamente quantas linhas de cabeçalho o cliente irá enviar, podemos obter essas linhas dentro de uma operação de looping.

```
// Obter e exibir as linhas de cabeçalho.
String headerLine = null;
While ((headerLine = br.readLine()).length() != 0) {
      System.out.println(headerLine);
}
```

Não precisamos das linhas de cabeçalho, a não ser para exibi-las na tela; portanto, usamos uma variável string temporária, `headerLine`, para manter uma referência aos seus valores. O loop termina quando a expressão

```
(headerLine = br.readLine()).length()
```

chegar a zero, ou seja, quando o `headerLine` tiver comprimento zero. Isso acontecerá quando a linha vazia ao final do cabeçalho for lida.

Na próxima etapa deste laboratório, iremos adicionar o código para analisar a mensagem de requisição do cliente e enviar uma resposta. Mas, antes de fazer isso, vamos tentar compilar nosso programa e testá-lo com um browser. Adicione as linhas a seguir ao código para fechar as cadeias e conexão de socket.

```
// Feche as cadeias e socket.
os.close();
br.close();
socket.close();
```

Após compilar o programa com sucesso, execute-o com um número de porta disponível, e tente contatá-lo a partir de um browser. Para fazer isso, digite o endereço IP do seu servidor em execução na barra de endereços do seu browser. Por exemplo, se o nome da sua máquina é `host.someschool.edu`, e o seu servidor roda na porta 6789, então você deve especificar a seguinte URL:

```
http://host.someschool.edu:6789/
```

O servidor deverá exibir o conteúdo da mensagem de requisição HTTP. Cheque se ele está de acordo com o formato de mensagem mostrado na figura do HTTP Request Message da Seção 2.2 do livro-

texto.

Servidor Web em Java: Parte B

Em vez de simplesmente encerrar a thread após exibir a mensagem de requisição HTTP do browser, analisaremos a requisição e enviaremos uma resposta apropriada. Iremos ignorar a informação nas linhas de cabeçalho e usar apenas o nome do arquivo contido na linha de requisição. De fato, vamos supor que a linha de requisição sempre especifica o método GET e ignorar o fato de que o cliente pode enviar algum outro tipo de requisição, tal como HEAD o POST.

Extraímos o nome do arquivo da linha de requisição com a ajuda da classe `StringTokenizer`. Primeiro, criamos um objeto `StringTokenizer` que contém a string de caracteres da linha de requisição. Segundo, pulamos a especificação do método, que supusemos como sendo "GET". Terceiro, extraímos o nome do arquivo.

```
// Extrair o nome do arquivo a linha de requisição.
StringTokenizer tokens = new StringTokenizer(requestLine);
tokens.nextToken(); // pular o método, que deve ser "GET"

String fileName = tokens.nextToken();

// Acrescente um "." de modo que a requisição do arquivo esteja dentro do diretório atual.
fileName = "." + fileName;
```

Como o browser precede o nome do arquivo com uma barra, usamos um ponto como prefixo para que o nome do caminho resultante inicie dentro do diretório atual.

Agora que temos o nome do arquivo, podemos abrir o arquivo como primeira etapa para enviá-lo ao cliente. Se o arquivo não existir, o construtor `FileInputStream()` irá retornar a `FileNotFoundException`. Em vez de retornar esta possível exceção e encerrar a thread, usaremos uma construção try/catch para ajustar a variável booleana `fileExists` para falsa. A seguir, no código, usaremos este flag para construir uma mensagem de resposta de erro, melhor do que tentar enviar um arquivo não existente.

```
// Abrir o arquivo requisitado.
FileInputStream fis = null;
Boolean fileExists = true;
```

```
try {
      fis = new FileInputStream(fileName);
} catch (FileNotFoundException e) {
      fileExists = false;
}
```

Existem três partes para a mensagem de resposta: a linha de status, os cabeçalhos da resposta e o corpo da entidade. A linha de status e os cabeçalhos da resposta são terminados pela de seqüência de caracteres CRLF. Iremos responder com uma linha de status, que armazenamos na variável `statusLine`, e um único cabeçalho de resposta, que armazenamos na variável `contentTypeLine`. No caso de uma requisição de um arquivo não existente, retornamos *404 Not Found* na linha de status da mensagem de resposta e incluímos uma mensagem de erro no formato de um documento HTML no corpo da entidade.

```
// Construir a mensagem de resposta.
String statusLine = null;
String contentTypeLine = null;
String entityBody = null;
If (fileExists) {
      statusLine = ?;
      contentTypeLine = "Content-type: " +
            contentType( filename ) + CRLF;
} else {
      statusLine = ?;
      contentTypeLine = ?;
      entityBody = "<HTML>" +
            "<HEAD><TITTLE>Not Found</TITTLE></HEAD>" +
            "<BODY>Not Found</BODY></HTML>";
```

Quando o arquivo existe, precisamos determinar o tipo MIME do arquivo e enviar o especificador do tipo MIME apropriado. Fazemos esta determinação num método privado separado chamado `contentType()`, que retorna uma string que podemos incluir na linha de tipo de conteúdo que estamos construindo.

Agora podemos enviar a linha de status e nossa única linha de cabeçalho para o browser escrevendo na cadeia de saída do socket.

```
// Enviar a linha de status.
os.writeBytes(statusLine);

// Enviar a linha de tipo de conteúdo.
os.writebytes(?);

// Enviar uma linha em branco para indicar o fim das linhas de cabeçalho.
os.writeBytes(CRLF);
```

Agora que a linha de status e a linha de cabeçalho com delimitador CRLF foram colocadas dentro do trecho de saída no caminho para o browser, é hora de fazermos o mesmo com o corpo da entidade. Se o arquivo requisitado existir, chamamos um método separado para enviar o arquivo. Se o arquivo requisitado não existir, enviamos a mensagem de erro codificada em HTML que preparamos.

```
// Enviar o corpo da entidade.
If (fileExists) {
      sendBytes(fis, os);
      fis.close();
} else {
      os.writeBytes(?);
}
```

Após enviar o corpo da entidade, o trabalho neste thread está terminado; então fechamos as cadeias e o socket antes de encerrarmos.

Ainda precisamos codificar os dois métodos que referenciamos no código acima, ou seja, o método que determina o tipo MIME, `contentType()` e o método que escreve o arquivo requisitado no trecho de saída do socket. Primeiro veremos o código para enviar o arquivo para o cliente.

```
private static void sendBytes(FileInputStream fis, OutputStream os)
throws Exception
{
      // Construir um buffer de 1K para comportar os bytes no caminho para o
socket.
      byte[] buffer = new byte[1024];
      int bytes = 0;
      // Copiar o arquivo requisitado dentro da cadeia de saída do socket.
      While((bytes = fis.read(buffer)) != -1 ) {
            os.write(buffer, 0, bytes);
      }
```

```
}
```

Ambos `read()` e `write()` repassam exceções. Em vez de pegar essas exceções e manipulá-las em nosso código, iremos repassá-las pelo método de chamada.

A variável, `buffer`, é o nosso espaço de armazenamento intermediário para os bytes em seu caminho desde o arquivo para a cadeia de saída. Quando lemos os bytes do `FileInputStream`, verificamos se `read()` retorna menos um (–1), indicando que o final do arquivo foi alcançado. Se o final do arquivo não foi alcançado, `read()` retorna o número de bytes que foi colocado dentro do `buffer`. Usamos o método `write()` da classe `OutputStream` para colocar estes bytes na cadeia de saída, passando para ele o nome do vetor dos bytes, `buffer`, o ponto inicial nesse vetor, 0, e o número de bytes no vetor para escrita, `bytes`.

A parte final do código necessária para completar o servidor Web é um método que examina a extensão de um nome de arquivo e retorna uma string que representa o tipo MIME. Se a extensão do arquivo for desconhecida, podemos retornar o tipo `application/octet-stream`.

```
Private static String contentType(String fileName)
{
      if(filename.endsWith(".htm") || fileName.endsWith(".html")) {
            return "text/html";
      }
      if(?) {
            ?;
      }
      of(?) {
            ?;
      }
      return "application/octet-stream";
}
```

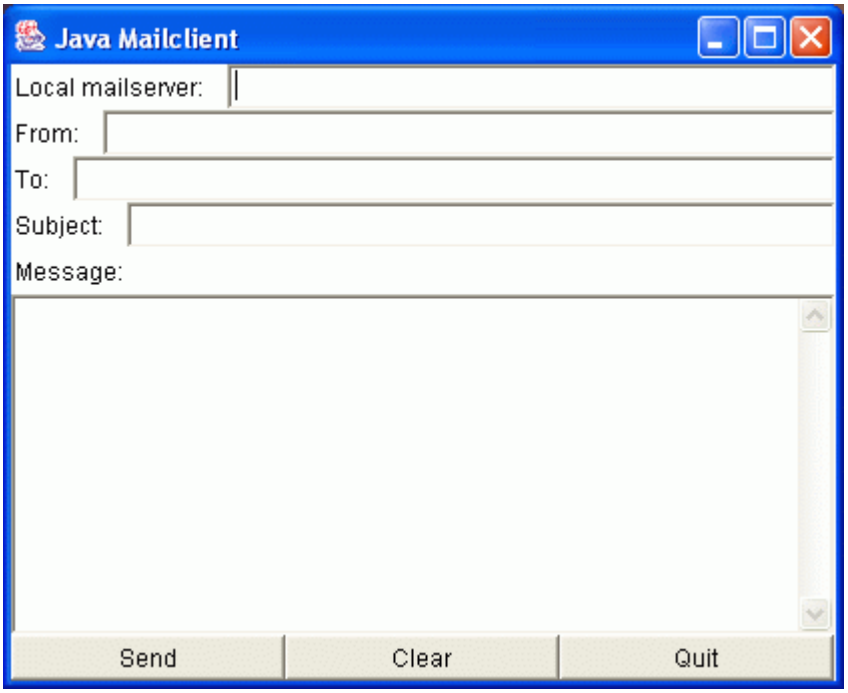
Está faltando um pedacinho deste método. Por exemplo, nada é retornado para arquivos GIF ou JPEG. Você mesmo poderá adicionar os tipos de arquivo que estão faltando, de forma que os componentes da sua home page sejam enviados com o tipo de conteúdo corretamente especificado na linha de

cabeçalho do tipo de conteúdo. Para GIFs, o tipo MIME é `image/gif` e, para JPEGs, é `image/jpeg`.

Isso completa o código para a segunda fase de desenvolvimento do nosso servidor Web. Tente rodar o servidor a partir do diretório onde sua home page está localizada e visualizar os arquivos da home page com o browser. Lembre-se de incluir um especificador de porta na URL, de forma que o browser não tente se conectar pela porta 80. Quando você se conectar ao servidor Web em execução, examine as requisições GET de mensagens que o servidor Web receber do browser.

Tarefa de Programação 2: Agente usuário de correio em Java

Neste laboratório, você implementará um agente usuário de correio que envia e-mail para outros usuários. Sua tarefa é programar a interação SMTP entre o MUA e o servidor SMTP local. O cliente provê uma interface gráfica de usuário, a qual deve conter campos para os endereços do remetente e do destinatário, para o assunto da mensagem e para a própria mensagem. A interface de usuário deve ser parecida com:



Com essa interface, quando se quer enviar um e-mail, é preciso preencher os endereços completos do remetente e do destinatário (exemplo: `user@someschool.edu`, e não simplesmente `user`). Você poderá enviar e-mail para apenas um destinatário. Também será necessário informar o nome (ou endereço IP) do seu servidor de correio local. Veja Querying the DNS abaixo para mais informações sobre como obter o endereço do servidor de correio local.

Quando tiver encerrado a composição do e-mail, pressione *Send* para enviá-lo.

O código

O programa consiste de quatro classes:

MailClient	A interface do usuário
Message	A mensagem de e-mail
Envelope	Envelope SMTP que envolve a mensagem
SMTPConnection	Conexão ao servidor SMTP

Você precisará completar o código na classe `SMTPConnection` de modo que no fim você tenha um programa capaz de enviar e-mail para qualquer destinatário. O código para a classe `SMTPConnection` está no fim desta página. O código para outras três classes é fornecido nesta página.

Os locais onde você deverá completar o código estão marcados com comentários `/* Fill in */`. Cada um deles requer uma ou mais linhas de código.

A classe `MailClient` provê a interface do usuário e chama as outras classes quando necessário. Quando o botão *Send* é pressionado, a classe `MailClient` constrói um objeto de classe `Message` para comportar a mensagem de correio. O objeto `Message` comporta os cabeçalhos e o corpo da mensagem atual. Ele constrói o envelope SMTP usando a classe `Envelope`. Essa classe compreende a informação SMTP do remetente e do destinatário, o servidor SMTP de domínio do remetente e o objeto `Message`. Então o objeto `MailClient` cria o objeto `SMTPConnection`, que abre uma conexão para o servidor SMTP, e o objeto `MailClient` envia a mensagem através dessa conexão. O envio do e-mail acontece em três fases:

1. O objeto `MailClient` cria o objeto `SMTPConnection` e abre a conexão para o servidor SMTP.

2. O objeto `MailClient` envia a mensagem usando a função `SMTPConnection.send()`.

3. O objeto `MailClient` fecha a conexão SMTP.

A classe `Message` contém a função `isValid()`, que é usada para verificar os endereços do remetente e do destinatário para se certificar de que há apenas um único endereço e que este contém o sinal @. O código fornecido não realiza nenhum outro tipo de verificação de erro.

Códigos de resposta

Para o processo básico de envio de mensagem, é necessário implementar apenas uma parte do SMTP.

Neste laboratório, você precisará implementar somente os seguintes comandos SMTP:

Comando	*Código de resposta*
DATA	354
HELO	250
MAIL FROM	250
QUIT	221
RCPT TO	250

A tabela acima também lista os códigos de resposta aceitos para cada um dos comandos SMTP que você precisará implementar. Para simplificar, pode-se presumir que qualquer outra resposta do servidor indica um erro fatal e aborta o envio da mensagem. Na realidade, o SMTP distingue entre erros transientes (códigos de resposta 4xx) e permanentes (códigos de resposta 5xx), e o remetente é autorizado a repetir os comandos que provocaram um erro transiente. Veja o Apêndice E da RFC-821 para mais detalhes.

Além disso, quando você abre uma conexão para o servidor, ele irá responder com o código 220.

Nota: A RFC-821 permite o código 251 como resposta ao comando RCPT TO para indicar que o destinatário não é um usuário local. Você pode verificar manualmente com o comando `telnet` o que o seu servidor SMTP local responde.

Dicas

A maior parte do código que você precisará completar é similar ao código que você escreveu no laboratório de servidor Web. Você pode usá-lo aqui para ajudá-lo.

Para facilitar o debug do seu programa, não inclua logo de início o código que abre o socket, mas as seguintes definições: `fromServer` e `toServer`. Desse modo, seu programa envia os comandos para o terminal. Atuando como um servidor SMTP, você precisará fornecer os códigos de resposta correto. Quando seu programa funcionar, adicione o código que abre o socket para o servidor.

```
fromServer = new BufferedReader(new InputStreamReader(System.in));
toServer = System.out;
```

As linhas para abrir e fechar o socket, por exemplo, as linhas `connection = ...` no construtor e a linha `connection.close()` na função `close()`, foram excluídas como comentários por default.

Começaremos completando a função `parseReply()`, que será necessária em vários lugares. Na função `parseReply()`, você deve usar a classe `StringTokenizer` para analisar as strings de resposta. Você pode converter uma string em um inteiro da seguinte forma:

```
Int i = Integer.parseInt(argv[0]);
```

Na função `sendCommand()`, você deve usar a função `writeBytes()` para escrever os comandos para o servidor. A vantagem de usar `writeBytes()` em vez de `write()` é que a primeira converte automaticamente as strings em bytes, que são o que o servidor espera. Não se esqueça de encerrar cada comando com a string CRLF.

Você pode repassar exceções assim:

```
throw new Exception();
```

Você não precisa se preocupar com os detalhes, desde que as exceções neste laboratório sejam usadas apenas para sinalizar um erro, e não para dar informação detalhada sobre o que está errado.

Exercícios opcionais

Tente fazer os seguintes exercícios opcionais para tornar seu programa mais sofisticado. Para estes exercícios, será necessário modificar outras classes também (MailClient, Message, e Envelope).

- **Verificar o endereço do remetente.** A classe `System` contém informações sobre o nome de usuário, e a classe `InetAddress` contém métodos para encontrar o nome do hospedeiro local. Use-as para construir o endereço do remetente para o Envelope em vez de usar o valor fornecido pelo usuário no campo From do cabeçalho.
- **Cabeçalhos adicionais.** Os e-mails gerados possuem apenas quatro campos no cabeçalho, From, To, Subject e Date. Adicione outros campos de cabeçalho de acordo com a RFC-822, por exemplo, Message-ID, Keywords. Verifique a RFC para definições dos diferentes campos.
- **Múltiplos destinatários.** Até este ponto, o programa permite apenas enviar e-mail a um único destinatário. Modifique a interface de usuário para incluir um campo Cc e modifique o programa para enviar e-mail a dois destinatários. Para aumentar o desafio, modifique o programa para enviar e-mail a um número arbitrário de destinatários.
- **Maior verificação de erros.** O código fornecido presume que todos os erros que ocorrem durante a conexão SMTP são fatais. Adicione código para distinguir entre erros fatais e não

fatais e adicione um mecanismo para sinalizá-los ao usuário. Cheque o RFC para saber o significado dos diferentes códigos de resposta. Este exercício pode requerer grandes modificações nas funções `send( ), sendCommand( ) e parseReply( ).`

Consultando o DNS

O DNS (Domain Name System) armazena informações em registros de recursos. Os nomes para mapeamentos de endereço IP são armazenados nos registros de recursos do tipo A (Address). Os registros do tipo NS (NameServer) guardam informações sobre servidores de nomes e os registros do tipo MX (Mail eXchange) dizem qual servidor está manipulando a entrega de correio no domínio.

O servidor que você precisa encontrar é o que está manipulando o correio para o domínio da sua escola. Primeiramente, é preciso encontrar o servidor de nomes da escola e então consultá-lo pelo MX-hospedeiro. Supondo que você está na Someschool e que seu domínio é someschool.edu, você pode fazer o seguinte:

1. Encontrar o endereço de um servidor de nomes para o domínio do maior nível `.edu` (NS query)

2. Consultar o servidor de nomes de `.edu` , que buscará o servidor de nomes do domínio `someschool.edu` , para conseguir o endereço do servidor de nomes da Someschool. (NS query)

3. Perguntar ao servidor de nomes da Someschool pelos registros-MX para o domínio `someschool.edu`. (MX query)

Pergunte ao administrador do seu sistema local sobre como realizar manualmente perguntas ao DNS.

No UNIX, você pode perguntar manualmente ao DNS com o comando `nslookup`. A sintaxe desse comando encontra-se a seguir. Note que o argumento `host` pode ser um domínio.

Normal query	`nslookup host`
Normal query using a given Server	`nslookup host server`
NS-query	`nslookup –type=NS host`
MX-query	`nslookup –type=MX host`

A resposta para o MX-query pode conter múltiplas contas de e-mail. Cada uma delas é precedida por um número que será o valor preferencial para este servidor. Valores menores de preferência indicam servidores preferidos de modo que você possa usar o servidor com o valor mais baixo de preferência.

SMTPConnection.java

Este é o código para a classe SMTPConnection que você precisará completar. O código para as outras três classes é fornecido neste ponteiro.

```
Import java.net.*;
Import java.io.*;
Import java.util.*;

/**
 * Abre uma conexão SMTP para o servidor de correio e envia um e-mail.
 *
 */
public class SMTPConnection {
   /* O socket para o servidor */
   private Socket connection;

   /* Trechos para leitura e escrita no socket */
   private BufferedReader fromServer;
   private DataOutputStream toServer;

   private static final int SMTP_PORT = 25;
   private static final String CRLF = "\r\n";
   /* Estamos conectados? Usamos close() para determinar o que fazer.*/
   private boolean isConnected = false;

   /* Crie um objeto SMTPConnection. Crie os sockets e os
      trechos associados. Inicie a conexão SMTP. */
   public SMTPConnection(Envelope envelope) throws IOException {
         // connection = /* Preencher */;
         fromServer = /* Preencher */;
         toServer =   /* Preencher */;

         /* Preencher */
      /* Ler uma linha do servidor e verificar se o código de
         resposta é 220. Se não for, lance uma IOException. */
         /* Preencher */
```

```
      /* SMTP handshake. Precisamos do nome da máquina local.
         Envie o comando handskhake do SMTP apropriado. */
         String localhost = /* Preencher */;
         sendCommand( /* Preencher*/ );
         isConnected = true;
   }

   /* Envie a mensagem. Escreva os comandos SMTP corretos na ordem
      correta. Não verifique de erros, apenas lance-os ao chamador. */
   public void send(Envelope envelope) throws IOException {
         /* Preencher */
         /* Envie todos os comandos necessários para enviar a mensagem.
         Chame o sendCommand() para fazer o trabalho sujo. Não apanhe a exceção
         lançada pelo sendCommand(). */
         /* Preencher */
   }

   /* Feche a conexão. Primeiro, termine no nível SMTP, então feche o socket. */
   public void cloce() {
         isConnected = false;
         try {
               sendCommand( /* Preencher */ );
               // connection.close();
         } catch (IOException e) {
               System.out.println("Unable to lose connection: " + e);
               isConnected = true;
         }
   }

/* Envie um comando SMTP ao servidor. Cheque se o código de resposta está de acordo
   com o RFC 821. */
   /* Preencher */
   /* Escrever o comando do servidor e ler a resposta do servidor. */
   /* Preencher */

   /* Preencher */
   /* Cheque se o código de resposta do servidor é o mesmo do parâmetro rc. Se
   não, envie um IOException. */
   /* Preencher */
```

```
    /* Analise a linha de resposta do servidor. Retorne o código de resposta. */
    private int parseReply(string reply) {
          /* Preencher */
    }

    /* Destructor. Fecha a conexão se algo de ruim acontecer. */
    protected void finalize() throws Throwable {
          is(isConnected) {
                close();
          }
          super.finalize();
    }
}
```

Agente usuário de correio: versão simplificada

Este laboratório está dividido em duas partes. Na primeira, você deverá usar o telnet para enviar e-mail manualmente através de um servidor de correio SMTP. Na segunda, você terá de escrever o programa Java que realiza a mesma ação.

Parte 1: Enviando e-mail com telnet

Tente enviar um e-mail para você mesmo. Isso significa que você precisa saber o nome do hospedeiro do servidor de correio do seu próprio domínio. Para encontrar essa informação, você pode consultar o DNS para buscar o registro MX que mantém informações sobre seu domínio de correio. Por exemplo, bob@someschool.edu possui o domínio de correio *someschool.edu*. O comando a seguir consulta o DNS para encontrar os servidores de correio responsáveis pela entrega de correio neste domínio:

```
nslookup –type=MX someschool.edu
```

Para a resposta a este comando, pode haver vários servidores de correio que entregam e-mail para as caixas de correio no domínio *someschool.edu*. Suponha que o nome de um deles é *mx1.someschool.edu*. Nesse caso, o seguinte comando estabelecerá a conexão TCP a este servidor de correio. (Note que a porta número 25 é especificada na linha de comando.)

```
telnet mx1.someschool.edu 25
```

Neste ponto, o programa telnet permitirá a você entrar com os comando SMTP e exibirá as respostas do servidor de correio. Por exemplo, a seqüência de comandos a seguir envia um e-mail da Alice para o Bob.

```
HELO alice
MAIL FROM: <alice@crepes.fr>
RCPT TO: <bob@someschool.edu>
DATA
SUBJECT: hello

Hi Bob, How’s the weather? Alice.
.
```

```
QUIT
```

O protocolo SMTP foi originalmente projetado para permitir às pessoas interagirem manualmente com os servidores de correio em modo de conversação. Por essa razão, se você digitar um comando com sintaxe incorreta, ou com argumentos inaceitáveis, o servidor retornará uma mensagem reportando isso e permitirá que você tente novamente.

Para completar esta parte do laboratório, você deve enviar uma mensagem de e-mail para você mesmo e verificar se ela chegou.

Parte 2: Enviando e-mail com Java

A Java fornece uma API para interagir com o sistema de correio da Internet, que é chamado JavaMail. No entando, não usaremos esta API, pois ela esconde os detalhes do SMTP e da programação de sockets. Em vez disso, você escreverá um programa Java que estabelece uma conexão TCP com um servidor de correio através da interface de socket e enviará uma mensagem de e-mail.

Você pode colocar todo seu código dentro do método principal de uma classe chamada *EmailSender*. Execute seu programa com o simples comando a seguir:

```
java EmailSender
```

Isso significa que você incluirá em seu código os detalhes da mensagem de e-mail que você está tentando enviar.

Aqui está um esqueleto do código que você precisará para escrever:

```
import java.io.*;
import java.net.*;

public class EmailSender
{
    public static void main(String() args) throws Exception
    {
        // Estabelecer uma conexão TCP com o servidor de correio.

        // Criar um BufferedReader para ler a linha atual.
        InputStream is = socket.getInputStream();
```

```
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        // Ler os cumprimentos do servidor.
        String response = br.readLine();
        System.out.println(response);
        if (!response.startsWith("220")) {
           Throw new Exception("220 reply not received from server.");
        }

        // Pegar uma referência para o trecho de saída do socket.
        OutputStream os = socket.getOuputStream();

        // Enviar o comando HELO e pegar a resposta do servidor.
        String comand = "Helo Alice\r\n";
        System.out.println(command);
        os.write(command.getBytes("US-ASCII"));
        response = br.readLine();
        System.out.println(response);
        if (!response.startsWith("250")) {
           throw new Exception("250 reply not received from server.");
        }

        // Enviar o comando MAIL FROM.

        // Enviar o comando RECP TO.

        // Enviar o comando DATA.

        // Enviar o dados da mensagem.

        // Terminar com uma linha de um único período.

        // Enviar o comando QUIT.
      }
}
```

Tarefa de Programação 4: Proxy cache

Neste laboratório, você desenvolverá um pequeno servidor Web proxy que também será capaz de fazer cache de páginas Web. Este será um servidor proxy bem simples, que apenas entenderá requisições GET simples, mas será capaz de manipular todos os tipos de objetos, não apenas páginas HTML, mas também imagens.

Código

O código está dividido em três classes:

- `ProxyCache` – compreende o código de inicialização do proxy e o código para tratar as requisições.
- `HttpRequest` – contém as rotinas para analisar e processar as mensagens que chegam do cliente.
- `HttpResponse` – encarregada de ler as respostas vindas do servidor e processá-las.

Seu trabalho será completar o proxy de modo que ele seja capaz de receber requisições, encaminhá-las, ler as respostas e retorná-las aos clientes. Você precisará completar a classes `ProxyCache`, `HttpRequest`, e `HttpResponse`. Os lugares onde você precisará preencher o código estarão marcados com /* Preencher */. Cada lugar pode exigir uma ou mais linhas de código.

Nota: Conforme será explicado abaixo, o proxy utiliza DataOutputStreams para processar as respostas dos servidores. Isso ocorre porque as respostas são uma mistura de texto e dados binários, e a única cadeia de entrada em Java que permite trabalhar com ambos ao mesmo tempo é a DataOutputStream. Para obter o código a ser compilado, você deve usar o argumento –deprecation para o compilador, conforme segue:

```
javac –deprecation *.java
```

Se você não usar o flag –deprecation, o compilador irá se recusar a compilar seu código.

Executando o proxy

Para executar o proxy faça o seguinte:

```
java ProxyCache port
```

onde *port* é o número da porta que você quer que o proxy escute pela chegada de conexões dos clientes.

Configurando seu browser

Você também vai precisar configurar seu browser Web para usar o proxy. Isso depende do seu browser Web. No Internet Explorer, você pode ajustar o proxy em "Internet Options" na barra Connection em LAN settings. No Netscape (e browsers derivados, como Mozilla), você pode ajustar o proxy em Edit → Preferences e então selecionar Advanced e Proxies.

Em ambos os casos, você precisará informar o endereço do proxy e o número da porta que você atribuiu a ele quando foi inicializado. Você pode executar o proxy e o browser na mesma máquina sem nenhum problema.

Funcionalidades do proxy

O proxy funciona da seguinte forma:

1. O proxy escuta por requisições dos clientes.
2. Quando há uma requisição, o proxy gera um novo thread para tratar a requisição e cria um objeto HttpRequest que contém a requisição.
3. O novo thread envia a requisição para o servidor e lê a resposta do servidor dentro de um objeto HttpResponse.
4. O thread envia a resposta de volta ao cliente requisitante.

Sua tarefa é completar o código que manipula o processo acima. A maior parte dos erros de manipulação em proxy é muito simples, e o erro não é informado ao cliente. Quando ocorrem erros, o proxy simplesmente pára de processar a requisição e o cliente eventualmente recebe uma resposta por causa do esgotamento da temporização.

Alguns browsers também enviam uma requisição por vez, sem usar conexões paralelas. Especialmente em páginas com várias imagens, pode haver muita lentidão no carregamento delas.

Caching

Realizar o caching das respostas no proxy fica como um exercício opcional, pois isso demanda uma

quantidade significativa de trabalho adicional. O funcionamento básico do caching é descrito a seguir:

1. Quando o proxy obtém uma requisição, ele verifica se o objeto requisitado está no cache; se estiver, ele retorna o objeto do cachê, sem contatar o servidor.

2. Se o objeto não estiver no cache, o proxy traz o objeto do servidor, retorna-o ao cliente e guarda uma cópia no cache para requisições futuras.

Na prática, o proxy precisa verificar se as respostas que possui no cache ainda são válidas e se são a resposta correta à requisição do cliente. Você pode ler mais sobre caching e como ele é tratado em HTTP na RFC-2068. Para este laboratório, isto é suficiente para implementar a simples política acima.

Dicas de programação

A maior parte do código que você precisará escrever está relacionada ao processamento de requisições e respostas HTTP, bem como o tratamento de sockets Java.

Um ponto notável é o processamento das respostas do servidor. Em uma resposta HTTP, os cabeçalhos são enviados como linhas ASCII, separadas por seqüências de caracteres CRLF. Os cabeçalhos são seguidos por uma linha vazia e pelo corpo da resposta, que pode ser dados binários no caso de imagens por exemplo.

O Java separa as cadeias de entrada conforme elas sejam texto ou binário. Isso apresenta um pequeno problema neste caso. Apenas DataInputstreams são capazes de tratar texto e binário simultaneamente; todas as outras cadeias são puro texto (exemplo: BufferedReader), ou puro binário (exemplo: BufferedInputStream), e misturá-los no mesmo socket geralmente não funciona.

O DataInputstream possui um pequeno defeito, pois ele não é capaz de garantir que o dado lido possa ser corretamente convertido para os caracteres corretos em todas as plataformas (função DataInputstream.readLine()). No caso deste laboratório, a conversão geralmente funciona, mas o compilador verá o método DataInputstream.readLine() como “deprecated” (obsoleto) e se recusará a compilá-lo sem o flag –deprecation.

É altamente recomendável que você utilize o DataInputStream para ler as respostas.

Exercícios opcionais

Quando você terminar os exercícios básicos, tente resolver os seguintes exercícios opcionais.

1. Melhor tratamento de erros. No ponto atual, o proxy não possui nenhum tratamento de erro. Isso pode ser um problema especialmente quando o cliente requisita um objeto que não está disponível, desde que a resposta usual “404 Not Found” não contenha corpo de resposta e o proxy presuma que existe um corpo e tente lê-lo.

2. Suporte para o método POST. O proxy simples suporta apenas o método GET. Adicione o suporte para POST incluindo o corpo de requisição enviado na requisição POST.

3. Adicione caching. Adicione o funcionamento simples de caching descrito acima. Não é preciso implementar nenhuma política de substituição ou validação. Sua implementação deve ser capaz de escrever respostas ao disco (exemplo: o cache) e trazê-las do disco quando você obtiver um encontro no cache. Para isso, você precisa implementar alguma estrutura interna de dados no proxy para registrar quais objeto estão no cache e onde eles se encontram no disco. Você pode colocar essa estrutura de dados na memória principal; não é necessário fazê-la persistir após processos de desligamento.