Disciplina ACH 2026 – Redes de Computadores
Profa.: Gisele S. Craveiro

Segunda e Terceira Atividades

O trabalho de confecção do EP 1 será desmembrado em 3 atividades: a segunda e terceira especificadas nesse documento e a quarta em outro documento próprio. Os alunos em dupla deverão confeccionar um relatório e disponibilizar o código fonte documentado e manual para compilação e execução do sistema.

–

- Data de entrega da segunda atividade (Parte A): **03/09/2016** somente pelo Tidia
- Data de entrega da terceira atividade (Parte B): **10/09/2016** somente pelo Tidia

– 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.