

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 exhibe 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 caractere fim-de-linha, ou em nosso caso, uma sequência de caractere 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 sequência 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><TITLE>Not Found</TITLE></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";
}

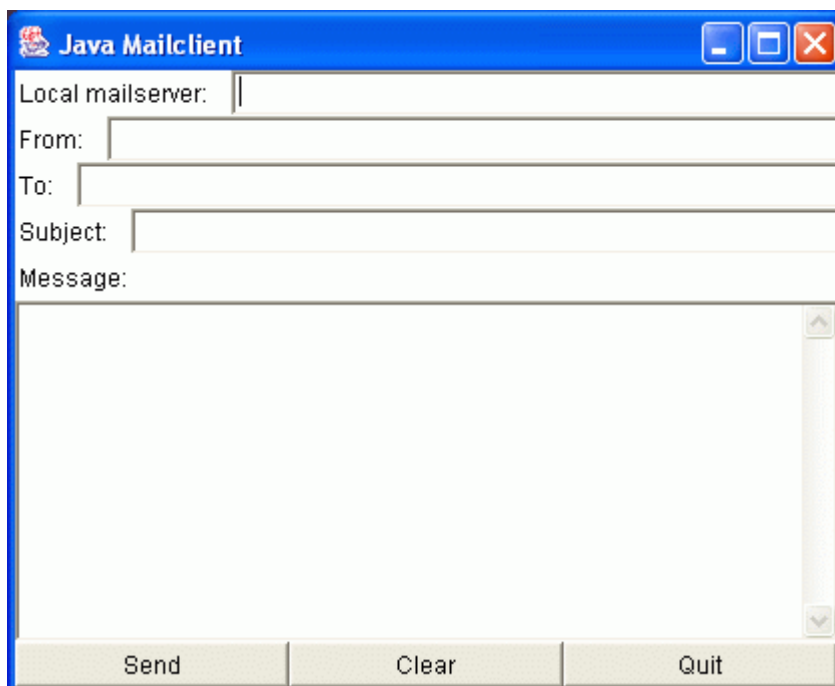
```

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:

<i>Comando</i>	<i>Código de resposta</i>
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	<code>nslookup host</code>
Normal query using a given Server	<code>nslookup host server</code>
NS-query	<code>nslookup -type=NS host</code>
MX-query	<code>nslookup -type=MX host</code>

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 close() {
    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 {
    if(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 sequência 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 entanto, 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.getOutputStream();

// 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 3: Laboratório de UDP pinger

Neste laboratório, você irá estudar um simples servidor de Ping da Internet escrito em linguagem Java e implementar um cliente correspondente. A funcionalidade provida por esses programas é similar à dos programas de Ping padrão disponíveis nos sistemas operacionais modernos, exceto aqueles que usam o UDP em vez do ICMP (Internet Control Message Protocol) para se comunicar. (Java não provê meios diretos para interagir com o ICMP.)

O protocolo Ping permite a uma máquina cliente enviar um pacote de dados para uma máquina remota, a qual retornará o dado para o cliente sem modificações (uma ação conhecida como eco). Entre outros usuários, o protocolo Ping permite aos hospedeiros determinarem o tempo de resposta de outras máquinas.

Complete o código para o servidor de Ping abaixo. Seu trabalho será escrever o cliente Ping.

Código do servidor

O código a seguir implementa por completo o servidor de Ping. Você precisará compilar e executar este código. Estude-o cuidadosamente, pois ele irá ajudá-lo a escrever seu cliente de Ping.

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

/*
 * Servidor para processar as requisições de Ping sobre UDP.
 */
public class PingServer
{
    private static final double LOSS_RATE = 0.3;
    private static final int AVERAGE_DELAY = 100; //milliseconds

    public static void main(String[] args) throws Exception
    {
        // Obter o argumento da linha de comando.
```

```

if (args.length !=1) {
    System.out.println("Required arguments: port");
    return;
}
int port = Integer.parseInt(args[0]);

// Criar um gerador de números aleatórios para uso em simulação
de perda de pacotes e atrasos na rede.
Random random = new Random();

// Criar um socket de datagrama para receber e enviar pacotes UDP
através da porta especificada na linha de comando.
DatagramSocket socket = new DatagramSocket(port);

// Loop de processamento.
while(true);
    // Criar um pacote de datagrama para comportar o pacote
    UDP // de chegada.
    DatagramPacket request = new DatagramPacket(new
    byte[1024],1024);
    // Bloquear até que o hospedeiro receba o pacote UDP.
    Socket.receive(request);

    // Imprimir os dados recebidos.
    printData(request);

    // Decidir se responde, ou simula perda de pacotes.
    if(random.nextDouble() < LOSS_RATE) {
        System.out.println("Reply not sent.");
        Continue;
    }

    // Simular o atraso da rede.
    Thread.sleep((int) (random.nextDouble) * 2 * AVERAGE_DELAY));

    // Enviar resposta.
    InetAddress clientHost = request.getAddress();
    Int clientPort = request.getPort();
    Byte[]buf = request.getData();
    DatagramPacket reply = new DatagramPacket(buf, buf.length,
    clientHost, clientPort);

```



```

        Socket.send(reply);

        System.out.println("Reply sent.");
    }
}

/*
 * Imprimir o dado de Ping para o trecho de saída padrão.
 */

private static void printData(DatagramPacket request) throws
Exception
{
    // Obter referências para a ordem de pacotes de bytes.
    byte[] buf = request.getData();

    // Envolver os bytes numa cadeia de entrada vetor de bytes, de
    modo que você possa ler os dados como uma cadeia de bytes.
    ByteArrayInputStream bais = new ByteArrayInputStream(buf);

    // Envolver a cadeia de saída do vetor bytes num leitor de
    cadeia de entrada, de modo que você possa ler os dados como uma
    cadeia de caracteres.
    InputStreamReader isr = new InputStreamReader(bais);

    // Envolver o leitor de cadeia de entrada num leitor com
    armazenagem, de modo que você possa ler os dados de caracteres
    linha a linha. (A linha é uma sequência de caracteres
    terminados por alguma combinação de \r e \n.)
    BufferedReader br = new BufferedReader(isr);

    // O dado da mensagem está contido numa única linha, então leia
    esta linha.
    String line = br.readLine();

    // Imprimir o endereço do hospedeiro e o dado recebido dele.
    System.out.println(
        "Received from" +
        request.getAddress().getHostAddress() +
        ":" +
        new String(line));
}

```

```
}  
}
```

O servidor fica num loop infinito de escuta pela chegada de pacotes UDP. Quando um pacote chega, o servidor simplesmente envia o dado encapsulado de volta para o cliente.

Perda de pacotes

O UDP provê aplicações com serviço de transporte não confiável, pois as mensagens podem se perder pela rede devido a um overflow na fila do roteador ou por outras razões. Em contraste a isso, o TCP fornece aplicações com um serviço de transporte confiável, e cada pacote perdido é retransmitido até que ele seja recebido com sucesso. Aplicações que usam o UDP para comunicação precisam implementar alguma segurança separadamente no nível de aplicação (cada aplicação pode implementar uma política diferente, de acordo com necessidades específicas).

Devido ao fato de a perda de pacotes ser rara, ou até mesmo inexistente, em uma rede típica, o servidor neste laboratório injeta perda artificial para simular os efeitos da perda de pacotes na rede. O servidor possui um parâmetro `LOSS_RATE`, que determina qual a porcentagem de pacotes deve ser perdida.

O servidor também possui outro parâmetro, `AVERAGE_DELAY`, que é usado para simular o atraso de transmissão ao enviar um pacote pela Internet. Você deve ajustar o `AVERAGE_DELAY` com um valor positivo quando o cliente e o servidor forem estar na mesma máquina, ou quando as máquinas estiverem muito perto fisicamente na rede. Você pode ajustar o `AVERAGE_DELAY` em 0 (zero) para encontrar o tempo de transmissão verdadeiro dos seus pacotes.

Compilando e executando o servidor

Para compilar o servidor, faça o seguinte:

```
javac PingServer.java
```

Para executar o servidor, faça o seguinte:

```
java PingServer port
```

onde port é o número da porta que o servidor escuta. Lembre que você deve usar um número de porta maior do que 1024, pois apenas os processos executando no modo root (administrador) possuem privilégio de usar portas menores que 1024.

Nota: Se você obtiver um erro de classe não encontrada quando executar o comando acima, você precisará dizer para o Java olhar no diretório atual para resolver as referências de classe. Nesse caso, os comandos são:

```
java -classpath . PingServer port
```

Sua tarefa: o cliente

Você deve escrever o cliente de modo que ele envie 10 requisições de Ping para o servidor, separadas por aproximadamente 1 segundo. Cada mensagem contém uma carga útil de dados que inclui a palavra PING, um número de sequência, e uma marca de tempo. Após enviar cada pacote, o cliente espera um segundo para receber a resposta. Se um segundo se passar sem uma resposta do servidor, então o cliente pode supor que esse pacote ou o pacote de resposta do servidor se perdeu pela rede.

Dica: Copie e cole o PingServer, renomeie o código para PingClient e então modifique-o.

Você deve escrever o cliente de modo que ele inicie com o seguinte comando:

```
java PingClient host port
```

onde hospedeiro é o nome do computador em que o servidor está sendo executado e port é o número da porta que ele está escutando. Note que você pode executar o cliente e o servidor em diferentes máquinas ou na mesma.

O cliente deve enviar 10 Pings para o servidor. Como o UDP é um protocolo não confiável, alguns dos pacotes enviados pelo cliente ou pelo servidor podem ser perdidos. Por essa razão, o cliente não pode esperar indefinidamente pela resposta a uma mensagem de Ping. Você deve fazer com que o cliente espere até um segundo por uma resposta; se nenhuma resposta for recebida, ele presume que o pacote foi perdido durante a transmissão. Você precisará pesquisar a API para o DatagramSocket de modo a descobrir como se ajusta o valor de tempo de expiração num socket de datagrama.

Ao desenvolver seu código, você deve executar o servidor de Ping em sua máquina e testar seu cliente enviando pacotes para o hospedeiro local (ou, 127.0.0.1). Após o

completo debug do seu código, você deve ver como sua aplicação se comunica através da rede com um servidor de Ping sendo executado por um outro membro da classe.

Formato das mensagens

As mensagens de Ping neste laboratório são formatadas de modo simples. Cada mensagem contém uma sequência de caracteres terminados por um caracter de retorno (r) e um caráter de mudança de linha (n). A mensagem contém a seguinte string:

```
PING sequence_number time CRLF
```

onde sequence_number começa em 0 (zero) e progride até 9, para cada mensagem sucessiva de Ping enviada pelo cliente; time é o tempo do momento em que o cliente enviou a mensagem e CRLF representa o retorno e linha de caracteres que finalizam a linha.

Exercícios opcionais

Quando você terminar de escrever seu código, pode tentar resolver os seguintes exercícios.

- 1) No ponto atual, o programa calcula o tempo de transmissão de cada pacote e os imprime individualmente. Modifique isso para corresponder ao modo de funcionamento do programas de Ping padrões. Você deverá informar os RTTs mínimo, máximo e médio. (fácil)
- 2) O programa básico envia um novo Ping imediatamente quando recebe uma resposta. Modifique-o de modo que ele envie exatamente 1 Ping por segundo, similar ao modo como programas de Ping padrões funcionam. Dica: Use as classes Timer e TimerTask em java.util. (difícil)
- 3) Desenvolva duas novas classes, ReliableUdpSender e ReliableUdpReceiver, que serão usadas para enviar e receber dados de maneira confiável sobre UDP. Para fazer isso, você precisará projetar um protocolo (como o TCP) em que o destinatário dos dados envia acknowledgements de volta ao remetente para indicar que o dado foi recebido. Você pode simplificar o problema apenas provendo um transporte unidirecional de dados de aplicação do remetente ao destinatário. Como seus

experimentos podem ser feitos em um ambiente com pouca ou nenhuma perda de pacotes IP, você deverá simular perda de pacotes. (difícil)

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 `DataOutputStream` 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.

Tarefa de Programação 5: Implementando um protocolo de transporte confiável

Visão geral

Neste laboratório, você escreverá o código de nível de transporte de envio e recepção, implementando um simples protocolo de transferência de dados confiável. Há duas versões deste laboratório: a versão protocolo bit alternante e a versão Go-Back-N. O laboratório será divertido, visto que sua implementação irá diferir um pouco do que seria exigido numa situação do mundo real.

Como você provavelmente não possui máquinas standalone (com um OS que pode ser modificado), seu código precisará ser executado num ambiente de hardware/software simulado. No entanto, a interface de programação fornecida para suas rotinas, por exemplo, o código que poderia chamar suas entidades de cima e de baixo é bem próximo do que é feito num ambiente UNIX atual. (Na verdade, as interfaces de software descritas neste exercício são muito mais realistas do que remetentes e destinatários de loop infinito que muitos textos descrevem.) Interrupção/acionamento de temporizadores serão também simulados, e interrupções feitas pelo temporizador farão com que seu temporizador manipule rotinas a serem ativadas.

Rotinas que você irá escrever

Os procedimentos que você escreverá são para a entidade de envio (A) e para a entidade de recepção (B). Apenas transferência unidirecional de dados (de A para B) é exigida. Naturalmente, o lado B deverá enviar pacotes ao lado A para confirmar (positiva ou negativamente) a recepção do dado. Suas rotinas devem ser implementadas na forma dos procedimentos descritos abaixo. Esses procedimentos serão chamados pelos (e chamarão) procedimentos que escrevi com um ambiente de rede emulado. Toda a estrutura do ambiente é mostrada na Figura Lab.3-1 (estrutura do ambiente emulado).

A unidade de dados que trafega entre as camadas superiores e seus protocolos é uma mensagem, que é declarada como:

```
Struct msg {  
    Char data[20];  
};
```

Essa declaração, e todas as outras estruturas de dados e rotinas de emulador, bem como rotinas de ponta (exemplo: aquelas que você precisa completar) estão no arquivo `prog2.c`, descrito mais tarde. Sua entidade de envio receberá dados em blocos de 20-bytes da camada 5; sua entidade de recepção deverá entregar blocos de 20 bytes de dados recebidos corretamente para a camada 5 do lado destinatário.

A unidade de dados que trafega entre suas rotinas e a camada de rede é o pacote, que é declarado como:

```
struct pkt {  
    int seqnum;  
    int acknum;  
    int checksum;  
    char payload[20];  
};
```

Suas rotinas irão preencher o campo carga útil do dado da mensagem vinda da camada 5. Os outros campos da mensagem serão usados por seus protocolos para assegurar entrega confiável.

As rotinas que você escreverá estão detalhadas abaixo. Conforme dito acima, tais procedimentos no mundo real seriam parte do sistema operacional e chamados por outros procedimentos no sistema operacional.

- **A_output(message)**, onde `message` é a estrutura do tipo `msg`, contendo dados que serão enviados para o lado B. Esta rotina será chamada sempre que a camada superior do lado (A) tenha uma mensagem a ser enviada. Este é o trabalho do seu protocolo para assegurar que os dados em tal mensagem sejam entregues em ordem, e corretamente, para a camada superior do lado destinatário.
- **A_input(packet)**, onde `packet` é a estrutura do tipo `pkt`. Esta rotina será chamada sempre que um pacote enviado pelo lado B (exemplo: como resultado de um `tolayer3()` feito por um procedimento do lado B) chega ao lado A. `packet` é o pacote (possivelmente corrompido) enviado pelo lado B.
- **A_timeinterrupt()** Esta rotina será chamada quando o temporizador de A expirar (gerando uma interrupção no temporizador). Você provavelmente usará esta rotina para

controlar a retransmissão dos pacotes. Veja `starttimer()` e `stoptimer()` abaixo para ver como o temporizador é acionado e interrompido.

- **A_init()** Esta rotina será chamada apenas uma vez, antes de qualquer outra rotina do lado A ser chamada. Ela pode ser usada para fazer qualquer inicialização requerida.
- **B_input(packet)**, onde `packet` é a estrutura do tipo `pkt`. Esta rotina será chamada sempre que um pacote enviado pelo lado A (exemplo: como resultado de um `tolayer3()` feito por um procedimento do lado A) chegar ao lado A. `packet` é o pacote (possivelmente corrompido) enviado pelo lado A.
- **B_init()** Esta rotina será chamada apenas uma vez, antes de qualquer outra rotina do lado B ser chamada. Ela pode ser usada para fazer qualquer inicialização requerida.

Interfaces de software

Os procedimentos descritos acima são aqueles que você irá escrever. As rotinas a seguir foram escritas e podem ser chamadas pelas suas rotinas:

- **starttimer(calling_entity, increment)**, em que `calling_entity` será “0” (para acionar o temporizador do lado A) ou “1” (para acionar o temporizador do lado B), e `increment` é um valor flutuante que indica a quantidade de tempo que se passou antes de o temporizador ser interrompido. O temporizador de A dever ser acionado (ou interrompido) apenas pelas rotinas do lado A, similarmente para o temporizador do lado B. Para dar uma idéia do valor apropriado de incremento a ser usado, um pacote enviado dentro de uma rede leva em média 5 unidades de tempo para chegar ao outro lado quando não há outras mensagens no meio.
- **stoptimer(calling_entity)**, onde `calling_entity` será “0” (para acionar o temporizador do lado A) ou “1” (para acionar o temporizador do lado B).
- **tolayer3(calling_entity, packet)**, onde `calling_entity` será “0” (para envio do lado A) ou “1” (para envio do lado B), e `packet` é a estrutura do tipo `pkt`. Ao chamar esta rotina, um pacote será enviado pela rede, destinado a outra entidade.
- **tolayer5(calling_entity, packet)**, onde `calling_entity` será “0” (entrega do lado A para a camada 5) ou “1” (entrega do lado B para a camada 5), e `message` é a estrutura do tipo `msg`. Em transferência de dados unidirecional, você poderia apenas ser chamado

com `calling_entity` igual a “1” (entrega para o lado B). Ao chamar esta rotina, os dados passarão para a camada 5.

O ambiente de rede simulado

Uma chamada ao procedimento `tolayer3()` envia pacotes para o meio (exemplo: dentro da camada de redes). Seus procedimentos `A_input()` e `B_input()` são chamados quando um pacote deve ser entregue do meio para a sua camada de protocolo.

O meio é capaz de corromper e perder pacotes. Ele não irá reordenar os pacotes. Quando você compilar seus procedimentos com os que são fornecidos aqui e executar o programa resultante, será necessário especificar os valores de acordo com o ambiente de rede simulado:

- **Número de mensagens para simular.** Meu emulador (e suas rotinas) irá parar quando esse número de mensagens tiver sido passado para a camada 5, não importando se todas as mensagens foram, ou não, entregues corretamente. Então, você não precisa se preocupar com mensagens não entregues ou não confirmadas que ainda estiverem no seu remetente quando o emulador parar. Note que se você ajustar esse valor para “1”, seu programa irá terminar imediatamente, antes de as mensagens serem entregues ao outro lado. Logo, esse valor deve ser sempre maior do que “1”.

- **Perdas.** Você deverá especificar a probabilidade de perda de pacotes. Um valor de 0,1 significa que um em cada dez pacotes (na média) será perdido.

- **Corrupção.** Você deverá especificar a probabilidade de perda de pacotes. Um valor de 0,2 significa que um em cada cinco pacotes (na média) será corrompido. Note que os campos de conteúdo da carga útil, sequência, ack, ou checksum podem ser corrompidos. Seu checksum deverá então incluir os campos de dados, sequência e ack.

- **Tracing.** Ajustar um valor de tracing de “1” ou “2” imprimirá informações úteis sobre o que está acontecendo dentro da emulação (exemplo: o que ocorre com pacotes e temporizadores). Um valor de tracing de “0” desliga esta opção. Um valor maior do que “2” exibirá todos os tipos de mensagens especiais que são próprias para a depuração do meu emulador. O valor “2” pode ser útil para você fazer o debug do seu código. Mantenha em mente que implementações reais não provêem tais informações sobre o que está acontecendo com seus pacotes.

- **Média de tempo entre mensagens do remetente da camada 5.** Você pode ajustar este valor para qualquer valor positivo diferente de zero. Note que quanto menor o valor escolhido, mais rápido os pacotes chegarão ao seu remetente.

Versão protocolo bit alternante deste laboratório

Você escreverá os procedimentos `A_output()`, `A_input()`, `A_timerinterrupt()`, `A_init()`, `B_input()` e `B_init()` que, juntos, implementarão uma transferência pare-e-espere (exemplo: o protocolo bit alternante referido como rdt3.0 no texto) unidirecional de dados do lado A para o lado B. **Seu protocolo deverá usar mensagens ACK e NACK.**

Escolha um valor bem alto para a média de tempo entre mensagens da camada 5 do remetente, assim ele nunca será chamado enquanto ainda tiver pendências, isto é, mensagens não confirmadas que ele esteja tentando enviar ao destinatário. Sugiro que se escolha um valor de 1.000. Realize também uma verificação em seu remetente para ter certeza de que quando `A_output()` é chamado, não haja nenhuma mensagem em trânsito. Se houver, você pode simplesmente ignorar (descartar) os dados que estão sendo passados para a rotina `A_output()`.

Coloque seus procedimentos em um arquivo chamado `prog2.c`. Você precisará da versão inicial deste arquivo, contendo as rotinas de emulação que escrevemos para você, e as chamadas para seus procedimentos. Para obter este programa, acesse <http://gaia.cs.umass.edu/kurose/transport/prog2.c>.

Este laboratório pode ser feito em qualquer máquina com suporte a C. Ele não faz uso de características UNIX. (Você pode simplesmente copiar o arquivo `prog2.c` em qualquer máquina e sistema operacional de sua escolha).

Recomendamos-lhe que tenha em mãos uma listagem do código, um documento de projeto e uma saída de amostra. Para sua saída de amostra, seus procedimentos podem imprimir uma mensagem sempre que um evento ocorrer no seu remetente ou destinatário (a chegada de uma mensagem/pacote, ou uma interrupção de temporizador) bem como qualquer ação tomada como resposta. Você pode querer ter uma saída para uma execução até o ponto (aproximadamente) quando 10 mensagens tiverem sido confirmadas (ACK) corretamente pelo destinatário, uma probabilidade de perda de 0,1,

e uma probabilidade de corrupção de 0,3, e um nível de trace de 2. Anote sua impressão com uma caneta colorida mostrando como seu protocolo recuperou corretamente uma perda ou corrupção de pacote.

Não deixe de ler as “dicas úteis” para este laboratório logo após a descrição da versão Go_Back-N.

Versão Go-Back-N deste laboratório

Você escreverá os procedimentos `A_output()`, `A_input()`, `A_timerinterrupt()`, `A_init()`, `B_input()` e `B_init()` e juntamente implementará uma transferência unidirecional de dados do lado A para o lado B, com tamanho de janela igual a 8. Seu protocolo deverá usar mensagens ACK e NACK. Consulte a versão protocolo bit alternante acima para informações sobre como obter e emulador de rede.

É altamente recomendável que você implemente primeiro o laboratório mais fácil (bit alternante) e então estenda seu código para implementar o mais difícil (Go-Back-N). Acredite, isso não será perda de tempo! No entanto, algumas novas considerações para o seu código Go-Back-N (que não se aplicam ao protocolo bit alternante) são:

- **A_output(message)**, onde `message` é uma estrutura do tipo `msg` contendo dados para serem enviados ao lado B.

Agora sua rotina `A_output()` será chamada algumas vezes quando houver pendências, mensagens não confirmadas no meio – implicando a necessidade de você ter um buffer para mensagens múltiplas em seu remetente. Você também precisará do buffer devido à natureza do Go-Back-N: algumas vezes seu remetente será chamado mas não será capaz de enviar a nova mensagem porque ela cai fora da janela.

Então você deverá se preocupar em armazenar um número arbitrário de mensagens. Você pode ter um valor finito para o número máximo de buffers disponíveis em seu remetente (para 50 mensagens) e pode simplesmente abortar (desistir e sair) se todos os 50 buffers estiverem em uso de uma vez (*Nota:* usando os valores acima, isso nunca deverá acontecer). No mundo real, naturalmente, deveria haver uma solução mais elegante para o problema de buffer finito.

- **A_timerinterrupt()** Esta rotina será chamada quando o temporizador de A expirar (gerando uma interrupção de temporizador). Lembre que você possui apenas um

temporizador, e pode ter muitas pendências e pacotes não confirmados no meio; então, pense um pouco em como usar este único temporizador.

Consulte a versão protocolo bit alternante para uma descrição geral do que você precisa ter em mãos. Você pode querer ter uma saída para uma execução que seja grande o bastante de modo que pelo menos 20 mensagens sejam transferidas com sucesso do remetente ao destinatário (exemplo: o remetente recebe o ACK para estas mensagens), uma probabilidade de perda de 0,2, e uma probabilidade de corrupção de 0,2, e um nível de trace de 2, e um tempo médio entre chegadas de 10. Você pode anotar sua impressão com uma caneta colorida mostrando como seu protocolo recuperou corretamente uma perda ou corrupção de pacote.

Para **crédito extra**, você pode implementar transferência bidirecional de mensagens. Nesse caso, as entidades A e B operam tanto como remetente quanto como destinatário. Você também pode sobrepor confirmações nos pacotes de dados (ou pode escolher não fazê-lo). Para fazer meu emulador entregar mensagens da camada 5 para sua rotina `B_output()`, é preciso trocar o valor declarado de `BIDIRECTIONAL` de 0 para 1.

Dicas úteis

- **Soma de verificação.** Você pode usar qualquer solução de soma de verificação que quiser. Lembre que o número de sequência e o campo ack também podem ser corrompidos. Sugerimos uma soma de verificação como o do TCP, que consiste na soma dos valores (inteiro) dos campos de sequência e do campo ack adicionada à soma caracter por caracter do campo carga útil do pacote (exemplo: trate cada caracter como se ele fosse um inteiro de 8 bits e apenas adicione-os juntos).
- Note que qualquer “estado” compartilhado entre suas rotinas deve estar na forma de variáveis globais. Note também que qualquer informação que seus procedimentos precisam salvar de uma invocação para a próxima também deve ser uma variável global (ou estática). Por exemplo, suas rotinas precisam manter uma cópia de um pacote para uma possível retransmissão. Nesse caso, seria provavelmente uma boa idéia uma estrutura de dados ser uma variável global no seu código. Note, no entanto, que se uma das suas variáveis é usada pelo seu lado remetente, essa variável não deve ser acessada pela entidade do lado destinatário, pois, no mundo real, a comunicação entre entidades conectadas apenas por um canal de comunicação não compartilha variáveis.

- Há uma variável global flutuante chamada *time*, que você pode acessar de dentro do seu código para auxiliá-lo com diagnósticos de mensagens.

- **COMECE SIMPLES.** Ajuste as probabilidades de perda e corrupção para zero e teste suas rotinas. Melhor ainda, projete e implemente seus procedimentos para o caso de nenhuma perda ou corrupção. Primeiro, faça-as funcionar; então trate o caso de uma dessas probabilidades não ser zero e, finalmente, de ambas não serem zero.

- **Debugging.** Recomendamos que você ajuste o nível de trace para 2 e coloque vários printf em seu código enquanto faz a depuração de seus procedimentos.

- **Números aleatórios.** O emulador gera perda de pacotes e erros usando um gerador de números aleatórios. Nossa experiência passada é que geradores de números aleatórios podem variar bastante de uma máquina para outra. Você pode precisar modificar o código do gerador de números aleatórios no emulador que estamos fornecendo. Nossas rotinas do emulador possuem um teste para ver se o gerador de números aleatórios em sua máquina funcionará com o nosso código. Se você obtiver uma mensagem de erro:

É provável que o gerador de números aleatórios em sua máquina seja diferente daquele que este emulador espera. Favor verificar a rotina `jmsrand()` no código do emulador. Desculpe.

então você saberá que precisa olhar como os números aleatórios são gerados na rotina `jmsrand()`; veja os comentários nessa rotina.

Q&A

Quando pensamos neste laboratório em nosso curso de introdução de redes, os estudantes enviaram várias questões. Se você estiver interessado em vê-las, acesse: http://gaia.cs.umass.edu/kurose/transport/programming_assignment_QA.htm.

Tarefa de Programação 6: Implementando um algoritmo

Visão geral

Neste laboratório, você escreverá um conjunto distribuído de procedimentos que implementam um roteamento de vetor de distâncias assíncrono distribuído para a rede mostrada na Figura Lab.4-1.

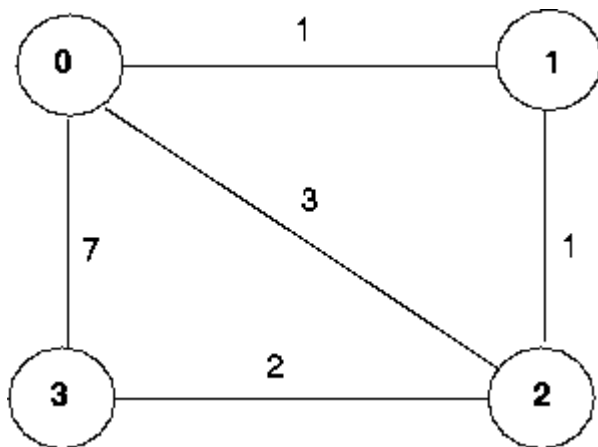


Figura Lab.4-1: Topologia de rede e custos dos enlaces para o laboratório de roteamento de vetor de distâncias

Tarefa básica

Rotinas que você irá escrever. Para a parte básica da tarefa, você escreverá as seguintes rotinas que executarão assincronamente dentro do ambiente emulado que escrevemos para esta tarefa.

Para o nó “0”, você escreverá as rotinas:

- **rtinit0()** Esta rotina será chamada uma vez no início da emulação. `rtinit0()` não possui argumentos. Ela deverá inicializar a tabela de distâncias no nó “0” para refletir os custos 1, 3 e 7 para os nós 1, 2 e 3, respectivamente. Na Figura 1, todos os links são bidirecionais, e os custos em ambas as direções são idênticos. Após inicializar a tabela de distância, e qualquer outra estrutura de dados necessária para sua rotina do nó “0”, ela deverá então enviar aos vizinhos diretamente conectados (neste caso, 1, 2, e 3) o custo dos seus caminhos de menor custo para todos os outros nós da rede. Esta

informação de custo mínimo é enviada para os nós vizinhos em um *pacote de rotina* através da chamada rotina `tolayer2()`, conforme descrita abaixo. O formato do pacote de rotina é descrito abaixo também.

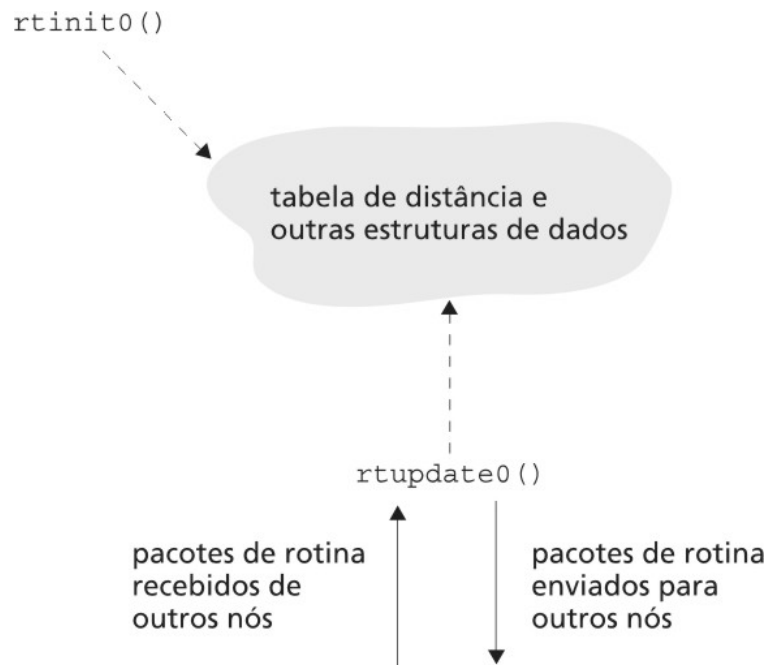
- **`rtupdate0()`** (**`struct rtpkt *rcvdpkt`**) . Esta rotina será chamada quando o nó “0” receber um pacote de rotina enviado por um de seus vizinhos diretamente conectados. O parâmetro `*rcvdpkt` é um ponteiro para o pacote que foi recebido.

`rtupdate0()` é o “coração” do algoritmo de vetor de distâncias. O valor que ele recebe num pacote de rotina vindo de algum outro nó *i* contém o custo do caminho mais curto de *i* para todos os outros nós da rede. `rtupdate0()` usa estes valores recebidos para atualizar sua própria tabela de distância (como especificado pelo algoritmo de vetor de distâncias). Se o seu próprio custo mínimo para outro nó mudar como resultado da atualização, o nó “0” informa a todos os vizinhos diretamente conectados sobre essa mudança em custo mínimo enviando para eles um pacote de rotina. Ressaltamos que no algoritmo de vetor de distâncias, apenas os nós diretamente conectados irão trocar pacotes de rotina. Portanto, os nós 1 e 2 se comunicam entre si, mas os nós 1 e 3 não.

Conforme vimos em aula, a tabela de distância dentro de cada nó é a principal estrutura de dados usada pelo algoritmo de vetor de distâncias. Você achará conveniente declarar a tabela de distância como uma disposição 4-por-4 de `int`’s, onde a entrada `[i, j]` na tabela de distância no nó “0” é o custo atual do nó “0” para o nó *i* pelo vizinho direto *j*. Se “0” não estiver diretamente conectado ao *j*, você pode ignorar essa entrada. Usaremos a convenção de que o valor inteiro 999 é infinito.

A Figura Lab.4-2 fornece uma visão conceitual do relacionamento dos procedimentos dentro do nó “0”.

Rotinas similares são definidas para os nós 1, 2 e 3. Portanto, você escreverá 8 procedimentos ao todo: `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()`, `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()`



Lab.4-2: Relação entre procedimentos dentro do nó 0.

Interfaces de software

Os procedimentos descritos acima são os que você irá escrever. Escrevemos as seguintes rotinas que podem ser chamadas pelas suas rotinas:

```
tolayer2(struct rtpkt pkt2send)
```

onde `rtpkt` é a seguinte estrutura, que já está declarada para você. O procedimento `tolayer2()` é definido no arquivo `prog3.c`

```
extern struct rtpkt {
    int sourceid; /* id do nó que está enviando o pkt, 0, 1, 2 ou
3 */
    int destid /* id do roteador para o qual o pkt está sendo
enviado (deve ser um vizinho imediato) */
    int mincost[4]; /* custo mínimo para o nó 0 ... 3 */
};
```

Note que `tolayer2()` é passado como estrutura, não como um ponteiro para uma estrutura.

```
printdt0()
```

imprimirá a tabela de distância para o nó “0”. Ela é passada como um ponteiro para uma estrutura do tipo `distance_table`. `printdt0()` e a declaração da estrutura para a tabela de distância do nó “0” está declarada no arquivo `node0.c`. Rotinas de

impressão similares estão definidas para você nos arquivos `node1.c`, `node2.c` e `node3.c`.

O ambiente de rede simulado

Seus procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` e `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` enviam pacotes de rotina (cujo formato está descrito acima) dentro do meio. O meio entregará os pacotes em ordem e sem perda para o destinatário específico. Apenas nós diretamente conectados podem se comunicar. O atraso entre o remetente e o destinatário é variável (e desconhecido).

Quando você compilar seus procedimentos e meus procedimentos juntos e executar o programa resultante, deverá especificar apenas um valor relativo ao ambiente de rede simulado:

- **Tracing.** Ajustar um valor de tracing de 1 ou 2 imprimirá informações úteis sobre o que está acontecendo dentro da emulação (exemplo: o que ocorre com pacotes e temporizadores). Um valor de tracing de “0” desliga esta opção. Um valor maior do que 2 exibirá todos os tipos de mensagens de uso próprio para depuração do meu emulador. Um valor igual a 2 pode ser útil para você fazer o debug do seu código. Mantenha em mente que implementações reais não provêem tais informações sobre o que está acontecendo com seus pacotes.

Tarefa básica

Você escreverá os procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` e `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` que juntos implementarão uma computação distribuída assíncrona das tabelas de distância para a topologia e os custos mostrados na Figura 1.

Você deverá colocar seus procedimentos para os nós 0 até 3 em arquivos chamados `node0.c`, ... `node3.c`. Não é permitido declarar nenhuma variável global que seja visível fora de um dado arquivo C (exemplo: qualquer variável global que você definir em `node0.c` pode ser acessada somente dentro do `node0.c`). Isso o forçará a cumprir convenções que deveriam ser adotadas se você fosse executar os procedimentos em

quatro nós distintos. Para compilar suas rotinas: `cc prog3.c node0.c node1.c node2.c node3`. Versões de protótipos desses arquivos encontram-se em: `node0.c`, `node1.c`, `node2.c`, `node3.c`. Você pode adquirir uma cópia do arquivo `prog3.c` em: <http://gaia.cs.umass.edu/kurose/network/prog3.c>.

Esta tarefa pode ser feita em qualquer máquina que tenha suporte a C. Ela não utiliza características UNIX.

Como sempre, a maior parte dos instrutores espera que você possua uma lista de códigos, um documento de projeto e uma saída de amostra.

Para sua saída de amostra, seus procedimentos devem imprimir uma mensagem sempre que seus procedimentos `rtinit0()`, `rtinit1()`, `rtinit2()`, `rtinit3()` ou `rtupdate0()`, `rtupdate1()`, `rtupdate2()`, `rtupdate3()` forem chamados, fornecendo o tempo (disponível pela minha variável global `clocktime`). Para `rtupdate0()`, `rtupdate1()`, `rtupdate2()` e `rtupdate3()`, você deve imprimir a identidade do remetente do pacote de rotina que está sendo passado para sua rotina, se a tabela de distância for, ou não, atualizada, o conteúdo da tabela de distância (você pode usar minha rotina de impressão) e uma descrição de todas as mensagens enviadas para os nós vizinhos como resultado de cada atualização da tabela de distância.

A saída de amostra deve ser uma listagem de saída com um valor de TRACE igual a 2. Destaque a tabela de distância final produzida em cada nó. Seu programa será executado até que não haja mais nenhum pacote de rotina em trânsito na rede. Nesse ponto, o emulador terminará.

Tarefa avançada

Você escreverá dois procedimentos, `rtlinkhandler0(int linkid, int newcost)` e `rtlinkhandler1(int linkid, int newcost)`, que serão chamados se (e quando) o custo do link entre 0 e 1 mudar. Essas rotinas devem ser definidas nos arquivos `node0.c` e `node1.c`, respectivamente. As rotinas levarão o nome (id) do nó vizinho no outro lado do link cujo custo foi mudado, e o novo custo do link. Note que quando o custo de um link muda, suas rotinas deverão atualizar a tabela de distância e podem (ou não) precisar enviar pacotes de roteamento atualizados aos nós vizinhos.

Para completar a parte avançada da tarefa, você precisará mudar o valor da constante LINKCHANGES (linha 3 no `prog3.c`) para 1. Para sua informação, o custo do link será alterado de 1 para 20 no tempo 10000 e então retornará para 1 no tempo 20000. Suas rotinas serão invocadas nesses instantes.

É altamente recomendável que você implemente primeiro a tarefa básica e então estenda seu código para implementar a tarefa avançada.

Q&A

Quando pensamos neste laboratório em nosso curso de introdução de redes, os estudantes enviaram várias questões. Se você estiver interessado em vê-las, acesse: http://gaia.cs.umass.edu/kurose/network/programming_assignment_QA.htm.

Tarefa de Programação 7: Streaming vídeo com RTSP e RTP

O código

Neste laboratório, você implementará um servidor de streaming vídeo e cliente que se comunica usando o protocolo de fluxo contínuo em tempo real (RTSP) e envia dados usando o protocolo de tempo real (RTP). Sua tarefa é implementar o protocolo RTSP no cliente e implementar o empacotamento RTP no servidor.

Forneceremos o código que implementa o protocolo RSTP no servidor, o desempacotamento RTP no cliente e trataremos de exibir o vídeo transmitido. Você não precisa mexer neste código.

Classes

Existem quatro classes nesta tarefa.

Client

Esta classe implementa o cliente e a interface de usuário que você usará para enviar comandos RTSP e que será utilizada para exibir o vídeo. Abaixo vemos como é a interface. *Você deverá implementar as ações que são tomadas quando os botões são pressionados.*

Server

Esta classe implementa o servidor que responde às requisições RTSP e encaminha o vídeo de volta. A interação RTSP já está implementada e o servidor chama as rotinas na classe `RTPpacket` para empacotar os dados de vídeo. Você não precisa mexer nesta classe.

RTPpacket

Esta classe é usada para manipular os pacotes RTP. Ela possui rotinas separadas para tratar os pacotes recebidos no lado cliente que já é dado e você não precisa modificá-lo (mas veja os Exercícios opcionais). Você deverá completar o primeiro construtor desta classe para implementar o empacotamento RTP dos dados de vídeo. O segundo construtor é usado pelo cliente para desempacotar os dados. Você não precisa modificá-lo também.

VideoStream

Esta classe é usada para ler os dados de vídeo do arquivo em disco. Você não precisa modificar esta classe.

Executando o código

Após completar o código, você pode executá-lo da seguinte forma:

Primeiro, inicie o servidor com o comando:

```
java Server server_port
```

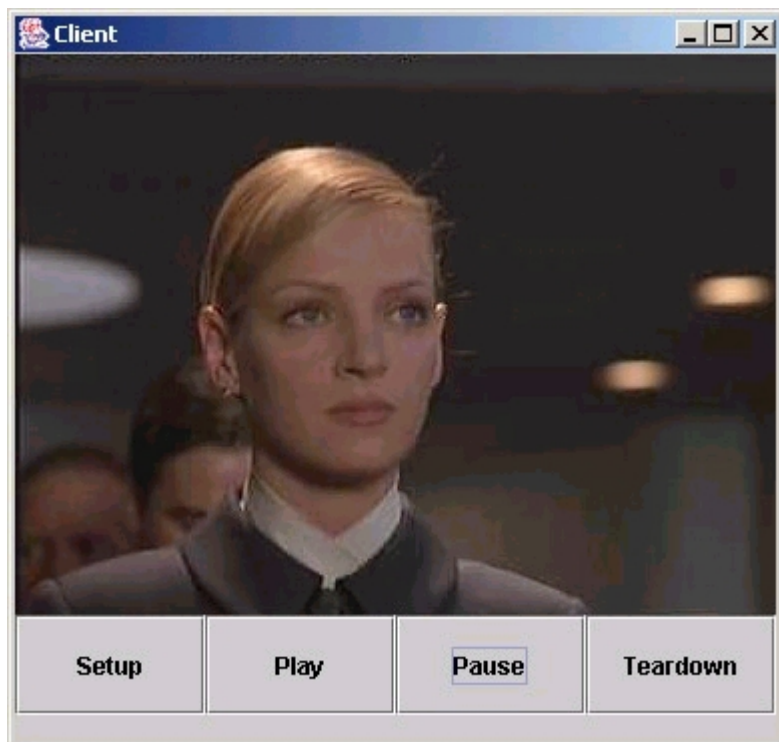
onde `server_port` é a porta onde seu servidor escuta as conexões RTSP que chegam. A porta RTSP padrão é a 554, mas você deve escolher um número de porta maior que 1024.

Então, inicie o cliente com o comando:

```
java Client server_name server_port video_file
```

onde `server_name` é o nome da máquina onde o servidor está sendo executado, `server_port` é a porta que o servidor está escutando, e `video_file` é o nome do arquivo que você quer requisitar (fornecemos um arquivo de exemplo `movie.Mjpeg`). O formato do arquivo está descrito no Apêndice.

O cliente abre uma conexão com o servidor e abre uma janela como esta:



Você pode enviar comandos RTSP para o servidor pressionando os botões. Uma interação normal RTSP acontece assim:

1. O cliente envia SETUP. Esse comando é usado para ajustar os parâmetros de sessão e de transporte.
2. O cliente envia PLAY. Isso inicia a reprodução.
3. O cliente pode enviar PAUSE se ele quiser pausar durante a reprodução.
4. O cliente envia TEARDOWN. Isso termina a sessão e fecha a conexão.

O servidor sempre responde a todas as mensagens que o cliente envia. Os códigos de resposta são exatamente os mesmos do HTTP. O código 200 indica que a requisição foi bem-sucedida. Neste laboratório, você não precisa implementar nenhum outro código de resposta. Para mais informações sobre o RTSP, veja a RFC-2326.

1. Cliente

Sua primeira tarefa é implementar o RTSP do lado cliente. Para fazer isso, você deve completar as funções que são chamadas quando o usuário clica nos botões da interface

de usuário. Para cada botão na interface, há uma função manipuladora do código. Você deve implementar as seguintes ações em cada função manipuladora.

Quando o cliente é iniciado, ele também abre o socket RTSP para o servidor. Use este socket para enviar todas as requisições RTSP.

SETUP

- Crie um socket para receber os dados RTP e ajustar o tempo de expiração no socket para 5 milissegundos.
- Envie uma requisição SETUP para o servidor. Você deve inserir o cabeçalho de Transporte onde você especificará a porta para o socket de dados RTP que você criou.
- Leia a resposta do servidor e analise o cabeçalho de Sessão na resposta para obter o ID da sessão.

PLAY

- Envie uma requisição PLAY. Você deve inserir o cabeçalho de sessão e usar o ID de sessão fornecido na resposta ao SETUP. Não coloque cabeçalho de Transporte nesta requisição.
- Leia a resposta do servidor.

PAUSE

- Envie uma requisição PAUSE. Você deve inserir o cabeçalho de Sessão e usar o ID de sessão fornecido na resposta ao SETUP. Não coloque cabeçalho de Transporte nesta requisição.
- Leia a resposta do servidor.

TEARDOWN

- Envie uma requisição TEARDOWN. Você deve inserir o cabeçalho de Sessão e usar o ID de sessão fornecido na resposta ao SETUP. Não é preciso colocar cabeçalho de Transporte neste requisição.
- Leia a resposta do servidor.

Nota: Você deve inserir o cabeçalho CSeq em cada requisição que você enviar. O valor do cabeçalho CSeq é um numero que será incrementado de 1 a cada requisição que você enviar.

Exemplo

Aqui está uma interação de amostra entre cliente e servidor. As requisições dos clientes são marcadas com C: e as respostas dos servidores com S:. Neste laboratório, tanto o cliente quanto o servidor são bem simples. Eles não precisam ter rotinas de análise sofisticadas e esperam sempre encontrar os campos do cabeçalho na ordem que você verá abaixo, ou seja, numa requisição, o primeiro cabeçalho é o CSeq, e o segundo é ou o de Transporte (para SETUP) ou o de Sessão (para todas as outras requisições). Na resposta, CSeq é novamente o primeiro e de Sessão é o segundo.

```
C: SETUP movie.Mjpeg RTSP/1.0
C: CSeq: 1
C: Transport: RTP/UDP; client_port= 25000
```

```
S: RTSP/1.0 200 OK
S: CSeq: 1
S: Session: 123456
```

```
C: PLAY movie.Mjpeg RTSP/1.0
C: CSeq: 2
C: Session: 123456
```

```
S: RTSP/1.0 200 OK
S: CSeq: 2
S: Session: 123456
```

```
C: PAUSE movie.Mjpeg RTSP/1.0
C: CSeq: 3
C: Session: 123456
```

```
S: RTSP/1.0 200 OK
S: CSeq: 3
S: Session: 123456
```

```
C: PLAY movie.Mjpeg RTSP/1.0
```

```
C: CSeq: 4
C: Session: 123456

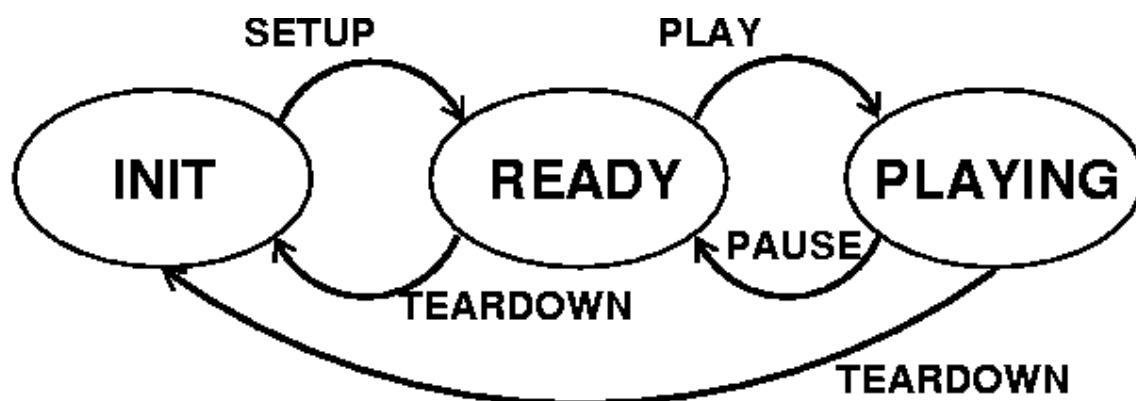
S: RTSP/1.0 200 OK
S: CSeq: 4
S: Session: 123456

C: TEARDOWN movie.Mjpeg RTSP/1.0
C: CSeq: 5
C: Session: 123456

S: RTSP/1.0 200 OK
S: CSeq: 5
S: Session: 123456
```

Estado do cliente

Uma das diferenças entre HTTP e RTSP é que no RTSP cada sessão possui um estado. Neste laboratório, você precisará manter o estado do cliente atualizado. O cliente muda de estado quando ele recebe uma resposta do servidor de acordo com o seguinte diagrama.



2. Servidor

No servidor, você precisará implementar o empacotamento dos dados de vídeo em pacotes RTP. Para isso, será necessário criar o pacote, ajustar os campos no cabeçalho do pacote e copiar a carga útil (exemplo: um quadro de vídeo) dentro do pacote.

Quando o servidor recebe a requisição **PLAY** do cliente, ele aciona um temporizador que é ativado a cada 100ms. Nesses tempos, o servidor lerá um quadro de vídeo do

arquivo e o enviará para o cliente. O servidor cria um objeto RTPpacket, que é o encapsulamento RTP do quadro de vídeo.

O servidor chama o primeiro construtor da classe RTPpacket para realizar o encapsulamento. Sua tarefa é escrever essa função. Você precisará fazer o seguinte: (as letras em parênteses se referem aos campos no formato do pacote RTP abaixo)

1. Ajuste o campo RTP-version (V). Você deve ajustá-lo para 2.
2. Ajuste os campos padding (P), extension (X), number of contributing sources (CC), e marker (M). Todos eles são ajustados em zero neste laboratório.
3. Ajuste o campo carga útil (PT). Neste laboratório, usamos MJPEG e o tipo para ele é 26.
4. Ajuste o número de sequência. O servidor fornece esse número de sequência como argumento `Framenb` para o construtor.
5. Ajuste a marca de tempo. O servidor fornece este número como argumento `Time` para o construtor.
6. Ajuste o identificador da fonte (SSRC). Este campo identifica o servidor. Você pode usar o valor inteiro que desejar.

Como não temos nenhuma outra fonte de contribuição (campo CC == 0), o campo CSRC não existe. O comprimento do cabeçalho do pacote é de 12 bytes, ou as três primeiras linhas do diagrama abaixo.

Para ajustar os bits n e $n+1$ para o valor de `foo` na variável `mybyte`:

```
mybyte = mybyte | foo << (7 - n);
```

Note que `foo` deve ter um valor que possa ser expresso com 2 bits, ou seja, 0, 1, 2 ou 3.

Para copiar um `foo` inteiro de 16-bits em 2-bytes, `b1` e `b2`:

```
b1 = foo >> 8;  
b2 = foo & 0xFF;
```

Após fazer isso, `b1` terá 8 bits de maior ordem de `foo` e `b2` terá 8 bits de menor ordem de `foo`.

Você pode copiar um inteiro de 32-bits em 4-bytes de maneira similar.

Se você não se sente confortável com o ajuste de bits, pode encontrar mais informações no Java Tutorial.

Exemplo de bit

Suponha que desejamos preencher o primeiro byte do cabeçalho do pacote RTP com os seguintes valores:

- $V = 2$
- $P = 0$
- $X = 0$
- $CC = 3$

Em binário, isso poderia ser representado como

```
1 0 | 0 | 0 | 0 0 1 1  
V=2  P  X  CC = 3  
  
2^7 . . . . . 2^0
```

Exercícios opcionais

- Em vez do servidor normal fornecido a você, use a classe chamada `FunkyServer` (faça também o download da classe `FunkyServer$1.class`), por exemplo, execute-o com `java FunkyServer server_port`. O que acontece no cliente? Explique o porquê.

- Calcule as estatísticas sobre a sessão. Você precisará calcular a taxa de perda de pacotes RTP, a taxa de dados de vídeo (em bits ou bytes por segundo) e qualquer outra estatística interessante que você conseguir encontrar.
- A interface de usuário no cliente possui 4 botões para as 4 ações. Se você compará-la a um transdutor padrão, tal como RealPlayer ou transdutor Windows, você verá que eles possuem apenas 3 botões para as mesmas ações, chamadas, PLAY, PAUSE e STOP (correspondendo exatamente ao TEARDOWN). Não há nenhum botão de SETUP disponível para o usuário. Dado que o SETUP é mandatório numa interação RTSP, como você implementaria isso? É apropriado enviar TEARDOWN quando o usuário clica no botão *stop*?
- Até aqui, o cliente e o servidor implementam apenas o mínimo necessário de interações RTSP e PAUSE. Implemente o método DESCRIBE, que é usado para passar informações sobre o *media stream*. Quando o servidor recebe uma requisição DESCRIBE, ele envia de volta um arquivo de descrição de sessão que diz ao cliente que tipos de streams estão na sessão e quais codificações estão sendo utilizadas.

Apêndice

Formato do MJPEG (Motion JPEG) proprietário do laboratório.

Neste laboratório, o servidor encaminha um vídeo codificado com um formato de arquivo proprietário MJPEG. Este formato armazena o vídeo como concatenação de imagens codificadas em JPEG, com cada imagem sendo precedida por um cabeçalho de 5-bytes que indica o tamanho em bits da imagem. O servidor analisa o bitstream do arquivo MJPEG para extrair as imagens JPEG no caminho. O servidor envia as imagens ao cliente em intervalos periódicos. O cliente então exibe as imagens JPEG individuais conforme elas vão chegando do servidor.