Programming Assignment 2: UDP based Web server and client(Solution)

In this programming assignment, you will develop both a UDP-based Web Server and a UDP-based Web Client. Unlike TCP, UDP is connectionless and does not guarantee the delivery of packets, making the implementation and error handling more complex. However, it can provide faster communication for applications where speed is crucial and occasional data loss is acceptable. You will start by [M1] converting an existing TCP Web Server to use UDP and then implement a UDP Web Client that can interact with your server.

Generally, when a client makes a request, it is sent to the web server. The web server then processes the request and sends back a response message to the requesting client. In this lab, you will implement this process using the UDP protocol.

The UDP based Web Server and Web client will [M1] initialize a UDP socket and [M2] create an HTTP request message, convert it to a datagram packet, and send it to the web server. The web server, upon receiving the request, will process it and then [M3] create an HTTP response message, convert it to a datagram packet, and send it back to the client.

Optionally, you will enhance your server and client to ensure reliable data transfer using the stop-and-wait protocol, which involves sequence numbers, acknowledgments, checksums, and retransmissions.

- **Mission 0: Before Start the Following Missions**

  Draw Block diagram of UDP Web Server & Web Client

  Draw Flow chart for UDP Web Server & Web Client
- **Prepare your TCP Web Server and Web Client**:

  Convert an existing TCP Web Server to use UDP.
- **Mission 1: Establish a Connection**:

  (Web Server) UDP Socket initialize for the web server: DatagramSocket

  (Web Server) Multi-Threaded UDP Web Server, Constructor Changed

  (Web Client) send datagram Packet with UDP socket to Web Server
- **Mission 2: UDP HTTP Request Handling**:

  (Web Server) Translate Data to handle datagram

  (Web Client) Create HTTP Requset MSG and translate to datagram Packet.
- **Mission 3: UDP HTTP Response Message**

  (Web Server) Create response Message and translate to datagram to send with UDP

  (Web Client) receive datagramPacket and display.

Tasks: The routines you will write

A. UDP Web Server: Running the Server

Put an HTML file (e.g., HelloWorld.html) in the same directory that the server is in. Run the server program. Determine the IP address of the host that is running the server (e.g., 128.238.251.26). From another host, open a browser and provide the corresponding URL. For example:

http://128.238.251.26:6789/HelloWorld.html

'HelloWorld.html' is the name of the file you placed in the server directory. Note also the use of the port number after the colon. You need to replace this port number with whatever port you have used in the server code. In the above example, we have used the port number 6789. The browser should then display the contents of HelloWorld.html. If you omit ":6789", the browser will assume port 80 and you will get the web page from the server only if your server is listening at port 80.

Then try to get a file that is not present at the server. You should get a "404 Not Found" message.

- **Listening for Requests:** The server will listen on a specified port for incoming UDP packets containing HTTP requests.
- **Processing Requests:** Upon receiving a packet, the server will process the HTTP request and determine the appropriate response.
- **Sending Responses:** The server will send back the requested resources, or an error message if the resource is not found, using UDP packets.
- **Concurrency Handling:** Since UDP does not inherently support connections, handling multiple simultaneous requests requires careful management of incoming and outgoing packets.

(Check point) You may fork off your UDP server from your existing TCP server. UDP server will receive exactly the same HTTP messages back and forth as before, except using in UDP segments instead of TCP segments.

B. UDP Web Client: Running the Server

- **Sending Requests:** The client will send HTTP GET requests to the server over UDP, specifying the desired resource.
- **Receiving Responses:** The client will receive the response packets from the server and assemble them to display the requested resource or error message.
- (optional) **Timeout and Retransmission:** To handle potential packet loss, the client will implement timeout and retransmission mechanisms.

Hint: You are required to implement in java a stripped down and simplified web downloader client. Before starting to panic, keep in mind that a web client is in essence a very simple program. It simply sends a request, the contents of a named file and read the response. That's it! (Also, keep in mind that your text is a good reference and a section is provided in Chapter 2 with some simple Java code for doing exactly this!) For this assignment, you only need to worry about implementing the GET request of HTTP/1.0. In other words, your client will issue a request like: ***GET /index.html HTTP/1.0***. Your UDP Web server will open this file relative to its current directory, read the contents, and send back the results. The Client will display the file on the window. When you are done, you should be able to use your UDP web browser to talk to your UDP Web server, or use your web client to retrieve documents off of another UDP web site.

C. Testing

- Test the server by sending various HTTP requests from the client and verifying the responses.
- Simulate packet loss and delays to test the reliability and robustness of the client's timeout and retransmission mechanisms.
- Ensure your client and server support the HTTP/1.0 GET command.
- Handle different HTTP commands and response codes such as command: POST, PUT, HEAD, response: 200 OK, 404 Not Found, 505 Version Not Supported, and 501 Method Not Implemented.

This assignment will enhance your understanding of UDP communication, concurrent request handling, and the challenges of implementing reliable communication over an unreliable protocol. By the end of this project, you will have a fully functional UDP-based web server and client capable of handling basic web requests and responses.

Hint; Your client needs to support the GET command of HTTP/1.0. Your client may send other options to the request, if your server may handle them. In response to a GET command, your server should respond with a proper HTTP response header when the file exists and can be sent. The first line should be a valid status line like ***HTTP/1.0 200 OK***. You should also include a header line specifying the content length; i.e., the size of the file. You should also provide a content type header line for ***GIF*** and ***JPEG*** files, to allow browsers to view embedded images and so on. Essentially, you just need to look for a ***.gif*** or ***.jpg*** or ***.jpeg*** file extension at the end of a file name, and add the content type line if the extension matches. Otherwise, do nothing, and the browser will assume it is an HTML file. You are welcome to add content type lines for other kinds of files, if you so choose. You may also add additional header values to return, such as dates, and so on, but they are not required. If the file requested in a GET request does not exist, you should return a 404 message to the client. The first line should be a status line like ***HTTP/1.0 404 Not Found***. As a response body, it should include a simple HTML message describing the problem. If the client uses a version in its request other than HTTP/1.0, you should return a 505 message to the client. The first line should be a status line like ***HTTP/1.0 505 Version Not Supported***. As a response body, it should include a simple HTML message describing the problem. If the client issues something other than a GET request, but still claims to be using HTTP/1.0, you should return a 501 message to the client. The first line should be a status line like ***HTTP/1.0 501 Method Not Implemented***. As a response body, it should include a simple HTML message describing the problem. If the web downloader client received a reply from the server other than a 200 message indicating everything is fine, the client should print the entire message to the screen.

From a UDP perspective, you no longer need to do any connection handshaking; instead, your client can start out and immediately send its request in a UDP segment to the server, and the server can respond immediately. You now have message boundaries to make your life easier, as each UDP message is logically separate from the rest, and are not all part of the same stream. The same HTTP messages as in the TCP code will be sent back and forth. It is important to note that since you no longer have a connection, your server will need to figure out where to send data back to the client. (Since it has to address packets back, it needs to know the return address!) To do so, you should be using sendto() and recvfrom() in your client and server. Your server can simply pull the client's address out of the address parameters passed out of its call to recvfrom() and use that when it needs to call sendto() to send data back. It is also important to note that you no longer have a connection for the server to close after sending the file, so you might need to do something special to let the client

know the file is finished. There are several possibilities: 1. If your client was using the content length field from your previous assignment to tell when to stop reading, you can use the same general idea here. 2. You can send a special segment with special text from the server to the client to indicate the end of the file. (Make sure the text is special enough so you it likely will not happen in the middle of files transferred!) Each time the client receives a UDP segment, it checks to see if it is the end of file segment. An interesting twist would be to try having the server send an empty UDP segment (size of 0) ... some UDP implementations allow you to do this. (This could have reading from the UDP socket return a 0, as was the case when a TCP connection was closed in the previous assignment.)

Sample codes

The server will listen on a specific UDP port for incoming requests, process these requests in separate threads, and send appropriate responses back to the clients. The client will be capable of sending HTTP requests to the server and receiving responses.

Sample Code for UDP Web Server

Main Server Class

`UDPWebServer.java`

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

public final class UDPWebServer {
   public static void main(String argv[]) throws Exception {
      //Mission 1. Fill in #1 Create DatagramSocket.
      DatagramSocket socket = /* Fill in */;// Changed ServerSocket to DatagramSocket

      // Process HTTP service requests in an infinite loop.
      while (true) {
     //Mission  1, Fill in #2 Init receiveData
     // Fill in #2 Construct an object to process the HTTP request message.(Changed to DatagramPacket)

         byte[] receiveData = /* Fill in */;
         DatagramPacket receivePacket = /* Fill in */;

         // Fill in #3 Listen for a UDP packet.
         // Fill in #3 Construct an object to process the HTTP request message(From changed constructor)
         /* Fill in */;
         UDPHttpRequest request = new UDPHttpRequest(/* Fill in */);  // Changed constructor
         // Create a new thread to process the request.
         Thread thread = new Thread(request);

         // Start the thread.
         thread.start();
      }
   }
}
```

Request Handler Class

`UDPHttpReqest.java`

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

```
final class UDPHttpRequest implements Runnable {
  final static String CRLF = "\r\n";
  // Mission 1. Fill in #4 Changed to DatagramSocket
  //  Fill in #4 Added to store the received packet
/* Fill in */;

  // Fill in #5 Constructor should be changed. Socket and packet information should be transferred
  public UDPHttpRequest(/* Fill in */) throws Exception {
    /* Fill in */;
  }

  // Implement the run() method of the Runnable interface.
  public void run() {
    try {
      processRequest();
    } catch (Exception e) {
      System.out.println(e);
    }
  }

  private void processRequest() throws Exception {
    // Mission 2. #Fill in #6 Get the request data from the packet
    // Received DatagramPacket → Byte → byteArrayInputStream → InputStream → BufferedReader
    /* Fill in */;

    // Get the request line of the HTTP request message.
    String requestLine = br.readLine();

    // Extract the filename from the request line.
    StringTokenizer tokens = new StringTokenizer(requestLine);
    String method = tokens.nextToken();
    String fileName = tokens.nextToken();

    // Prepend a "." so that file request is within the current directory.
    fileName = "." + fileName;

    // Open the requested file.
    FileInputStream fis = null;
    boolean fileExists = true;
    try {
      fis = new FileInputStream(fileName);
    } catch (FileNotFoundException e) {
      fileExists = false;
    }

    // Construct the response message.
    String statusLine = null;
    String contentTypeLine = null;
    String contentLengthLine = null;
    String entityBody = null;

    if (fileExists) {
      statusLine = "HTTP/1.0 200 OK" + CRLF;
      contentTypeLine = "Content-Type: " + contentType(fileName) + CRLF;
      contentLengthLine = "Content-Length: " + getFileSizeBytes(fileName) + CRLF;
    } else {
      statusLine = "HTTP/1.0 404 Not Found" + CRLF;
      contentTypeLine = "Content-Type: text/html" + CRLF;
      entityBody = "<HTML><HEAD><TITLE>Not Found</TITLE></HEAD><BODY>Not Found</BODY></HTML>";
    }

    // Mission 3. Fill in #7 Create the header line as bytes with get Bytes
    // Fill in #7 create and write to ByteArrayOutputStream to send header line
    /* Fill in */;
```

```
      // transfer entity to ByteArrayOutputStream
      if (fileExists) {
         sendBytes(fis, baos);
         fis.close();
      } else {
         baos.write(entityBody.getBytes());
      }

      // Mission 3. Fill in #8 Create the byte to send stream(header and entity body)
      // Fill in #8 send Datagram with UDP Socket.
      // (ByteArrayOutputStream → ByteArray →  DatagramPacket)
      /* Fill in */;

      // Close streams and socket.
      br.close();
   }

/**
 * Method which sends the context
 * @param fis FileInputStream to transfer
 * @param os outputstream to client
 */
   private static void sendBytes(FileInputStream fis,
                                                 OutputStream os) throws Exception {
            // Construct a 1K buffer to hold bytes on their way to the socket.
            byte[] buffer = new byte[1024];
            int bytes = 0;

            // Copy requested file into the socket's output stream.
            while ((bytes = fis.read(buffer)) != -1) {
               os.write(buffer, 0, bytes);
            }
   }

/**
 * Method to return appropriate
 * @param fileName
 */
private static String contentType(String fileName) {
            if(fileName.endsWith(".htm") || fileName.endsWith(".html")) {
               return "text/html";
            }
/**
 * create an HTTP response message consisting of the requested file preceded by header lines
 * Now, you are just handling text/html, is there any more context-types? Find and make codes for it.
 */
            if(fileName.endsWith(".ram") || fileName.endsWith(".ra")) {
               return "audio/x-pn-realaudio";
            }
            if(fileName.endsWith(".jpg") || fileName.endsWith(".jpeg")) {
               return "image/jpeg";
            }
            return "application/octet-stream" ;
   }

/**
 * Get the File name, and through the file name, get the size of the file.
 *.@param fileName
 */
private static long getFileSizeBytes(String fileName) throws IOException {
      File file = new File(fileName);
      return file.length();
```

```
  }
  // This method returns the size of the specified file in bytes.
}
```

Sample Code for UDP Web Client

Main Client Class

`UDPWebClient.java`

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

/** WebClient class implements a simple web client.
* Its primary responsibilities include:
* 1. Initializing the udp connection to web server
* 2. send HTTP request and receive HTTP response
*/
public class UDPWebClient {
   public static void main(String[] args) {
      String host = "localhost";
      int port = 8888;
      String resource = "/index.html";

      try {
         // Mission 1. Fill in #11 define InetAddress with host ip and initial DatagramSocket
         InetAddress address = /* Fill in */;
         DatagramSocket socket = /* Fill in */;

/**
* Improve your HTTP Client to provide other request Methods(POST, DELETE, ...)
* and also improve to handle headers(Content-Type, User-Agent, ...)
*/
         // Mission 2. Fill in #10 Create Request MSG
         //Fill in #10 Transfter to Byte and put in datagramPacket, and set it.
         /* Fill in */;



         // Mission 3. Fill in #11 Initialize byte and datagrampacket to receive data
         /* Fill in */;


         // Fill in #12 Get string from datagrampacket to display
         /* Fill in */;

         socket.close();
      } catch (IOException e) {
         e.printStackTrace();
      }
   }
}
```

Instructions

1. Compile the server and client programs:

   ```
   bash
   javac UDPWebServer.java
   javac UDPWebClient.java
   ```

2. Run the server:

   ```
   bash
   ```

```
java UDPWebServer
```

3. Run the client:

```
bash
java UDPWebClient
```

4. The client will send an HTTP GET request to the server, and the server will respond with the contents of the requested file if it exists, or an error message if it does not.

This example demonstrates how to implement a simple UDP-based web server and client in Java. The server processes HTTP GET requests and responds with the appropriate file contents or an error message. The client sends HTTP GET requests to the server and prints the server's response.

Helpful Hints and the like

- **State Management**: Use global variables for shared state among routines. Note that any shared "state" among your routines needs to be in the form of global variables. Note also that any information that your procedures need to save from one invocation to the next must also be a global (or static) variable. For example, your routines will need to keep a copy of a packet for possible retransmission. It would probably be a good idea for such a data structure to be a global variable in your code. Note, however, that if one of your global variables is used by your sender side, that variable should **NOT** be accessed by the receiving side entity, since in real life, communicating entities connected only by a communication channel can not share global variables.
- There is a float global variable called *time* that you can access from within your code to help you out with your diagnostics msgs.
- **START SIMPLE.** Set the probabilities of loss and corruption to zero and test out your routines. Better yet, design and implement your procedures for the case of no loss and no corruption, and get them working first. Then handle the case of one of these probabilities being non-zero, and then finally both being non-zero.
- **Debugging.** We'd recommend that you set the tracing level to 2 and put LOTS of printf's in your code while your debugging your procedures.

Q&A

By completing this assignment, you will gain practical experience in network programming, understanding and implementing reliability mechanisms over UDP, and working with HTTP protocols. This project simulates real-world scenarios and challenges you to design and implement robust network applications.

- What is different with TCP Web Server & TCP Web Client? (with wireshark?)
- What is Different with QUIC? And UDP Web Server & UDP Web Client?
  - What is achieved?
  - And What is not achieved?
    - Reliability(rdt 3.0), Security(tls?), … and what?
      → Following Exercise? Assignment? Honor Project Proposal?

Meaning:

- **Understanding UDP vs. TCP**:
  - Unlike TCP, UDP does not guarantee message delivery, order, or error-checking.
  - Implementing reliability mechanisms manually provides deep insights into how TCP achieves reliable communication.
- **Practical Socket Programming**:
  - Working with sockets and network programming principles.
  - Learning to handle low-level network operations and data transmission.
- **Protocol Design and Implementation**:
  - Designing and implementing a reliable data transfer protocol over UDP.
  - Learning the challenges and solutions involved in creating robust network protocols.
- **HTTP Protocol Familiarity**:
  - Understanding HTTP/1.0 protocol basics, including request and response formats.
  - Implementing and testing HTTP communication over UDP.
- **Problem-Solving and Debugging**:
  - Handling real-world issues like packet loss, corruption, and timeouts.
  - Developing debugging skills and using diagnostic tools to ensure correct functionality.

This assignment will provide hands-on experience in network programming, protocol design, and the practical challenges of ensuring reliable communication over an unreliable protocol like UDP.

By completing this assignment, you will gain practical experience in network programming, understanding and implementing reliability mechanisms over UDP, and working with HTTP protocols. This project simulates real-world scenarios and challenges you to design and implement robust network applications.

Optional Exercises

- **Ensure secure data transfer**: Typically implemented with HTTPS, inherently supporting encrypted connections via SSL/TLS, enhancing security
- **Ensure reliable data transfer**: Build Reliability on Your UDP Web Server & Client
  - Implement a reliable data transfer protocol on top of UDP to handle packet loss, errors, and ensure data integrity.
  - Implement additional protocol processing to ensure reliability using the stop-and-wait protocol (RDT 3.0) and Go-Back-N Version over UDP.
  - Use sequence numbers, acknowledgments, checksums (for error detection), and timeout (for retransmissions).

Assignment: After the upon missions:

Project Example #2: Name:
Student ID:
Organization: Department of computer science and engineering, Hanyang Univ. (Seoul, Republic of Korea)

Introduction

Contents of the attachment

Instructions: How to run the program

Wireshark

How the program works