CS-43333/6333 Project 2

Overview

For this project we will explore the challenges associated with providing reliable delivery services over an unreliable network. Three different ARQ approaches have been described in class: stop-and-wait, concurrent logical channels, and sliding window. Your task during this project is to develop a set of Java classes capable of reliably transferring a file between two hosts over UDP (TCP already offers reliable delivery services).

Students will be provided with a *UDPSocket* class that extends the *DatagramSocket* class provided by Java to send a *DatagramPacket* over UDP. Take time to familiarize yourself with the main two classes provided by Java for UDP support and their functionality. The UDPSocket class has the additional capability of allowing programmers to modify the MTU (to model fragmentation), the packet drop rate (to model unreliable networks) and add random time delays to packets (to model out-of-order reception).

Support material

Parameters regulating message transfers using the UDPSocket class are located in a file named **unet.properties**. This file may be modified to test different scenarios (the file includes comments describing each parameter). Note that you can also use this file to enable and configure logging options (to file or standard output stream).

Unet.properties

```
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
# unet properties file
#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
#== log file name ================================
#
# log.filename = system (standard output stream)
log.filename = system
log.enable = true
#== packet drop rate =============================
#
# n<0 --- Drop random packets (n*-100) probability
# n=0 --- Do not drop any packets
# n=-1 --- Drop all packets
# n>0 --- Drop every nth packet (assuming no delay)
# n=[d,]*d --- Drop select packets e.g. packet.droprate = 4,6,7
packet.droprate = 0
#== packet delay rate ============================
#
# packet delay in milliseconds
# min==max --- in order delivery
packet.delay.minimum = 10
packet.delay.maximum = 100
#== packet maximum transmission size =============
```

```
#
# -1 unlimited (Integer.MAX_VALUE)
packet.mtu = 1500
```

A brief tutorial on how to use UDP in Java is provided in the Appendix. A supporting jar file (unet.jar) as well as the default unet.properties file are provided in this repo.

Use **TUGrader.java** to ensure that all unit tests in the provided test file(s) pass.

Assignment

Programs

A minimum of two classes will have to be submitted, **RSendUDP.java** and **RReceiveUDP.java** (do not make them part of the edu.utulsa.unet package) where each class must implement the **edu.utulsa.unet.RSendUDPI** and **edu.utulsa.unet.RReceiveUDPI** interfaces respectively (as specified below, provided in unet.jar).

```
package edu.utulsa.unet;

import java.net.InetSocketAddress;

public interface RSendUDPI {

    public String getFilename();

    public int getLocalPort();

    public int getMode();

    public long getModeParameter();

    public InetSocketAddress getReceiver();

    public long getTimeout();

    public boolean sendFile();

    public void setFilename(String fname);

    public boolean setLocalPort(int port);

    public boolean setMode(int mode);

    public boolean setModeParameter(long n);

    public boolean setReceiver(InetSocketAddress receiver);

    public boolean setTimeout(long timeout);
}
```

```java
package edu.utulsa.unet;

public interface RReceiveUDPI {

    public String getFilename();

    public int getLocalPort();

    public int getMode();

    public long getModeParameter();

    public boolean receiveFile();

    public void setFilename(String fname);

    public boolean setLocalPort(int port);

    public boolean setMode(int mode);

    public boolean setModeParameter(long n);
}
```

The *setMode* method specifies the algorithm used for reliable delivery where mode is 0 or 1 (to specify stop-and-wait and sliding window respectively). If the method is not called, the mode should default to stop-and-wait. Its companion, *getMode* simply returns an int indicating the mode of operation.

The *setModeParameter* method is used to indicate the size of the window in bytes for the sliding window mode. A call to this method when using stop-and-wait should have no effect. The default value should be 256 for the sliding window algorithm. Hint: your program will have to use this value and the MTU (max payload size) value to calculate the maximum number of outstanding frames you can send if using the Sliding Window algorithm. For instance, if the window size is 2400 and the MTU is 20 you could have up to 120 outstanding frames on the network.

The *setFilename* method is used to indicate the name of the file that should be sent (when used by the sender) or the name to give to a received file (when used by the receiver).

The *setTimeout* method specifies the timeout value in milliseconds. Its default value should be one second.

A sender uses *setReceiver* to specify IP address (or fully qualified name) of the receiver and the remote port number. Similarly, *setLocalPort* is used to indicate the local port number used by the host. The sender will send data to the specified IP and port number. The default local port number is 12987 and if an IP address is not specified then the localhost is used.

The methods *sendFile* and *receiveFile* initiate file transmission and reception respectively. Methods returning a boolean should return true if the operation succeeded and false otherwise.

Operation Requirements

RReceiveUDP

- Should print an initial message indicating the local IP, the ARQ algorithm (indicating the value of n if appropriate) in use and the UDP port where the connection is expected.
- Upon successful initial connection from a sender, a line should be printed indicating IP address and port used by the sender.
- For each received message print its sequence number and number of data bytes
- For each ACK sent to the sender, print a message indicating which sequence number is being acknowledged
- Upon successful file reception, print a line indicating how many messages/bytes were received and how long it took.

RSendUDP

- Should print an initial message indicating the local IP, the ARQ algorithm (indicating the value of n if appropriate) in use and the local source UDP port used by the sender.
- Upon successful initial connection, a line should be printed indicating address and port used by the receiver.
- For each sent message print the message sequence number and number of data bytes in the message
- For each received ACK print a message indicating which sequence number is being acknowledged
- If a timeout occurs, i.e. an ACK has been delayed or lost, and a message needs to be resent, print a message indicating this condition
- Upon successful file transmission, print a line indicating how many bytes were sent and how long it took.

Submission

The assignment is due by the end of the month. A brief report should be typed and turned in on Harvey. The report does not need to be long and should only describe your approach (and challenges) in solving the problem and any methods or variables of relevance in your class. In particular, describe the packet format in detail and provide explanations for each field. The report should also include a "feedback" section about this project.

*To submit your work, please review the **Commits** section below.*

Grading

- ☐ (100pts) Implements stop-and-wait
- ☐ (140pts) Implements sliding-window
- ☐ (60pts) Written report

Stop-and-wait and sliding window have to be implemented for full credit. Note that the value of the MTU cannot be changed programmatically when using UDPSocket. However, you can manipulate its value by editing the unet.properties file. The **getSendBufferSize** and **setSendBufferSize** provided by DatagramSocket (and consequently by UDPSocket) should be used when accessing MTU values (a call to setSendBufferSize in UDPSocket will throw an exception). At the time your submission is graded, several values of MTU will be used (by modifying unet.properties).

Appendix

Fragmentation and In-order Delivery

In-order delivery is implicitly solved by the stop-and-wait algorithm (only one outstanding packet at all times) and explicitly by the sliding-window algorithm. You are responsible for developing an operating solution that will allow your programs to reconstruct the file even if fragments were received out-of-order.

Design

Your programs are essentially operating at the application layer (just like FTP, HTTP and many other protocols) using an unreliable transport layer (UDP). Please plan your design before you start coding and define a packet format that will allow safe and consistent file transfers. Your header design must include information that will allow the receiver to detect when the file has been transferred. Furthermore, the receiver should save the received data as soon as the transfer is complete. I encourage group discussions and sharing of the packet format used for the file transfers. Implementation should be individual.

DO NOT make RSendUDP and RReceiveUDP part of any package.

Usage

Program usage will be illustrated by the following example. Assume Host A has IP address 192.168.1.23 and Host B has IP address 172.17.34.56 (note that none of these IP addresses are routable over the Internet). Host A wants to send a file named important.txt to Host B. Host A wants to use local port 23456 and Host B wants to use local port 32456 during the file transfer. Host B does not consider the file so important and it wants to save the received file as less_important.txt. Both Host A and B have agreed to use the sliding-window protocol with a window size of 512 bytes and Host A will use a timeout value of 10 seconds (the path between A and B has a rather large delay).

Sample Code

The following code running on Host A should accomplish the task of reliably sending the file:

```
RSendUDP sender = new RSendUDP();
sender.setMode(1);
sender.setModeParameter(512);
sender.setTimeout(10000);
sender.setFilename("important.txt");
sender.setLocalPort(23456);
sender.setReceiver(new InetSocketAddress("172.17.34.56", 32456));
sender.sendFile();
```

The following code running on Host B should accomplish the task of receiving the file:

```
RReceiveUDP receiver = new RReceiveUDP();
receiver.setMode(1);
receiver.setModeParameter(512);
receiver.setFilename("less_important.txt");
receiver.setLocalPort(32456);
receiver.receiveFile();
```

Sample Output

Assume our solution uses 10 bytes of header for every message, the network cannot deliver messages larger than 200 bytes and the file we are transferring is 800 bytes long. The output of your program (on the sender side) when running the stop-and-wait algorithm could look like this:

```
Sending important.txt from 192.168.1.23:23456 to 172.17.34.56:32456 with
800 bytes
Using stop-and-wait
Message 1 sent with 190 bytes of actual data
Message 1 acknowledged
Message 2 sent with 190 bytes of actual data
Message 2 acknowledged
Message 3 sent with 190 bytes of actual data
Message 3 timed-out
Message 3 sent with 190 bytes of actual data
Message 3 acknowledged
Message 4 sent with 190 bytes of actual data
Message 4 acknowledged
Message 5 sent with 40 bytes of actual data
Message 5 acknowledged
Successfully transferred important.txt (800 bytes) in 1.2 seconds
```

A similar output should be seen on the receiver side.

Compiling and Testing Code

Your IDE should provide tools to compile your code. If you're unfamiliar with that process, you can research it online or ask. Most developers compile their code from command line using a shell script, such as a **Makefile** or build script (**build.sh**). I've provided build scripts for you in *Bash*. Refer to the following directions on how to use these scripts based on the terminal that you're using. If you're on Windows, please use Windows Subsystem for Linux (WSL) or Git Bash.

Windows Users (WSL, Git Bash), Mac and Linux Users

- To compile your code: `./build.sh`
- To compile and run your code: `./build.sh run` (forwards clargs to program)
- To compile and test your code: `./build.sh test` (forwards clargs to TUGrader)
- To format your code: `./build.sh fmt`
- To sync your code: `./build.sh sync`
- To submit your code: `./build.sh submit`
- To remove class files: `./build.sh clean`

These scripts use the following commands. Note that Windows users need to replace the colon with a semicolon in the Java classpath.

- To compile a Java file: `javac -d target -cp lib/*:src <filepath>.java`
- To execute a Java file: `java -cp lib/*:target <package-path>.<filename>`

- To format a Java file: `java -jar lib/google-java-format.jar --replace --skip-javadoc-formatting <filepath>.java`
- To remove class files: `rm -r target/*`

Code Style

All code should follow the Google Java style guidelines. If you find anything in the code that does not follow the style guidelines, feel free to fix it, but you are not required to do so. Only your handwritten code will be evaluated for its style. You do not need to follow the style guidelines to the letter but egregious deviations from the style guidelines will be penalized. A submission that passes all test cases but does not use an appropriate style will not receive an A for the assignment.

For those using an IDE, such as Eclipse or VS Code, the IDE should provide a formatting tool. I've included the XML specification of the Google Java Style Guidelines at `.vscode/java-google-style.xml`. You can configure your IDE to use the provided XML as its formatting rules to format your code to the Google Java Style Guidelines, which are the industry standard.

If you're working from command-line, google-java-format is an open-source formatting tool that you can use to format your files. You can use the following commands to format your code depending on your terminal.

- `./build.sh fmt`

Commits

Commits should be made incrementally. Many commits are always better than few, and commits can always be squashed together later if there are too many. You should try to make a commit every time you've made tangible progress in the development of your code.

Every commit should have a commit message. The standard format for commit messages is to start with a verb in present-tense followed by a concise message (typically less than 50 characters) that summarizes the change that the commit is introducing to the repo. For example, "Updates README", "Implements Array", "Passes testGet".

Popular IDEs, such as Eclipse and VS Code, provide integrated Git tools. If you're on Windows, you can install Git Bash or Windows Subsystem for Linux (WSL). If you're on Mac or Linux, you already have git installed.

If you've just installed git, it will need to be configured. The easy way to configure git is from a terminal. Use the following commands.

- `git config --global user.name "<github-username-goes-here>"`
- `git config --global user.email "<github-email-goes-here>"`
- `git config --global pull.rebase true` (optional)
- `git config --global fetch.prune true` (optional)
- `git config --global diff.colorMoved zebra` (optional)

To sync changes made from another device, use the following command.

- `git fetch origin main`

- `git pull origin main`

To push commits from command line, use the following commands.

- `git add -A`
- `git commit -m "<your message goes here>"`
- `git push origin main`

You can also sync all changes and submit with the following commands depending on your terminal.

- `./build.sh submit`