COMP3310/etc – TCP Client Server

Outline

In this tutorial you will

- Study the differences between UDP and TCP program design
- Study how data is transmitted and received over TCP
- Study and change a simple protocol

This tutorial is very similar to the previous UDP client-server, so you should know what to do. First make sure everything is working: run the server in one window, the client in another. Whatever you type in the client window will be sent to the server and echoed back. EOF in the client window to stop, Control-C for the server.

For this tutorial you should have WireShark (or `tcpdump`) running on the loopback interface, port 3310.

1 Program Design

Open all the source code files (for your preferred language) in your editor or IDE.

The TCP client and server both use a shared library for reading and writing lines of text, **`sockLine.py`** or **`SockLine.java`**. Have a look at the code.

Sending a line is easy: append a newline special character and then encode into UTF-8. It is only a couple of lines, but using a library means we cannot forget either step.

Reading a line is very different. The UDP programs received packets, and each package was a single request or response. TCP hides the packets, so a TCP socket behaves like a sequential file of bytes. The library code reads one byte at a time, but this does not mean that every byte is a packet. WireShark will show you what is actually being sent.

The TCP client uses the shared library, so the code for reading and writing is shorter. One difference is that the client sends a final message on EOF: this is a simple example of **`protocol`** design so the server can tell whether a client deliberately ended the session or just crashed.

The TCP server code is structured very differently. A UDP server responds to client packets and does not really need any state. A TCP server receives connection requests from a client, and each connection is a **`session`** dedicated to that client. The server now has two loops, one for client connections and one for requests by a single client.

2 Data Over TCP

In the UDP tutorial you could start a client with no server running. Try the same with the TCP client.

Two UDP clients could run at the same time (in different terminal windows), both sending requests to a single server. Try the same with the TCP server and two clients. What happens?

> *A real world server that only one person can use at a time would not be very useful, but in this course we are studying computer networks, not Site Reliability Engineering. Do* not *use threads, timeouts, async...in your assignments without first asking if you really need to.*

1. Modify the server `handleRequest` code so that it uses the special **`slowSend`** in the socket line library instead of `writeLine`. With a new server running, try sending requests of varying length from the client. What does WireShark show? How many responses does the client receive?

Control-C the server and change the code back to using `writeLine`. (If you don't, you will be waiting a very long time for the next step to complete.)

2. The tutorial includes a text file `loremIpsum.txt` which is a single 16K long line of randomly generated words. Use this as a client request by running the client program with standard input from this file, not the terminal.

```
python tcpClient.py < loremIpsum.txt
```

or

```
java TcpClient < loremIpsum.txt
```

> *This is an example of 'fuzzing', sending unexpected input to a program and seeing what happens.*

3 Protocol

3. In the 'protocol' for this client server system a linefeed is added to each line when transmitted, but it isn't being removed when received. This makes it harder for either program to test if a line is empty ("") or has a particular value, and inserts blank lines into the log output.

Modify the code for `readLine` in the shared library to remove the linefeed (and any extra spaces) from the text once received and decoded.

> *The general rule for layered network design is that any headers or trailers added to encapsulate data at one end should be removed by the equivalent layer at the other end. Here the LF is the protocol marker for a message boundary.*

4. The client sends a BYE message when it shuts down. In the server, modify `serverLoop` so receiving this message exits the loop and closes the socket.

5. Modify the server to send an extra empty line at the end of `handleRequest`, and modify the client `readReply` to keep reading lines until it receives the empty line. (Which should not be logged.)

Run the new client and server and verify that everything still works with simple one line messages and replies.

When you add new features to a program, don't break what already works!

6. Now that the client can loop until an empty line, modify the server so that if the request is "it", there is no reply, just the empty line.

7. Modify the server so that if the request is "ni", it sends back three lines. *Optional*: send back a random number of lines.

4 Experiments

8. Using the `words.txt` file from the UDP tutorial, make sure your client and server can handle non-ASCII text.

The TCP server can echo the input from other programs, not just the client. Open a web browser and type `localhost:3310` as the URL. Or try **`telnet`** or **`nc`** as the client.

Optional: Rewrite the server with a client session class, so that each `serverLoop` could be run as a thread to handle multiple client connections in parallel. (You don't *have* to actually make the server multithreaded, although this is a good way to learn about threads if you haven't done so before.)
