COMP3310/etc – UDP Client Server

Outline

In this tutorial you will

- Compile and run a pair of programs implementing a client-server design
- Begin studying a few of the problems that can occur in Internet programs
- See how text is sent and received over the Internet

> *This tutorial assumes you have completed the previous* ***`Programming Warm Up`****.*
>
> *There is a lot to do in this tutorial. Don't worry if you cannot finish everything within the time slot, but do allocate time if necessary to finish. Programming tutorials are not marked, but they are practice for the assignments.*

1 Client Server

For this tutorial you will need two command line shells, one for running the server (**`udpServer.py`** or **`UdpServer.java`**) and one for the client (**`udpClient.py`** or **`UdpClient.java`**).

Have a quick look through the code, but don't worry about understanding every line just yet. For now, just identify the main client or server loop and the two functions or methods that the loop relies on.

Both the client and server programs have optional command line arguments. For this tutorial we will not use these.

Ideally you should have WireShark running as well, so you can see the network packets being sent back and forth. You just need to watch UDP packets on the 127.0.0.1 (loopback) interface.

Run the server program in one shell. Nothing should happen: the server is 'passive' and waits for clients to connect.

Run the client program in another shell, and type a line. This is the **`request`** sent, which should generate output in both windows. The server generates some log messages to show what it is doing, which don't affect the client but are useful for debugging. The client also has some log messages, and prints the **`reply`** or **`response`** it gets from the server.

In WireShark you should see a packet being sent in each direction.

The client program stops on terminal EOF. (Control-D or Control-Z depending on your system.) Do this. What happens in the server window? In WireShark?

Run the client again, and type a few more words.

Open a third shell, and run a second client program, so you have two connected to the same server. Try typing different words in each client shell.

> *If you study the logging messages in the server window, you should see a difference between those for the first run of the client program and those from the second. This will be explained later in the course.*

The server program does not read from input, so won't respond to EOF. Instead use Control-C to stop it. Try this *while* a client is still running. What happens to the client? What do you see in WireShark?

EOF all your clients, then run a client without any server. What happens and when?

2 Reliability

This pair of programs are using UDP, **`Unreliable Data Protocol`**. Since real network errors are very rare when both programs are running on the same computer, we will modify the server program to make it misbehave on special requests, "it" and "ni". If you're not sure how to start, the **`knight`** program from the previous tutorial has code you could use.

1. Modify the server program so that if the request is "it", the server does not reply. (But it should print a log message to the terminal saying that it has received a special message.)

> *For the next steps, try to predict what will happen before you actually type in any requests. If your prediction is right, good. If your prediction is wrong but you can understand why, that's good too.*

Run the modified server, then a client. Type just `it` as the request. Wait several seconds (read the client program code to find out exactly how long).

2. Modify the server program so that if the request is "ni" it sends three replies, not one.

Type a few requests (Enter after each) `hello`, `ni`, `world`. What happens?

Type in three or four more words.

3 Messages

The client and server programs both have a limit on the maximum size data that will be read from a socket. Read the code to find out what these values are.

This tutorial comes with a file `words.txt` for testing the limits. Open this file in a text editor or word processor: if you need to specify the encoding, UTF-8. The file contains some French and Mandarin characters: if these are not displayed properly on your system, instead appearing as small boxes, you may need to change editor or terminal settings, or try a different editor.

Start a server and a client.

From the editor with `words.txt` try copying `Test` and pasting into the client terminal. Press Enter to send the request as normal.

3. Copy the long request ('extraterrestrial') into the client. How many characters are you sending?

How long is the reply from the server? Is this greater than the `MSG_SIZE` limit inside the server program? Read the server code: how can this happen?

4. Copy and send the very long request. Now what happens, and why?

The maximum size of each request and reply should be part of the **`protocol specification`**, *so everyone agrees on what the limits are and can write programs that interoperate. Or the programs may* **`negotiate`** *settings: 'Can I send you 65535 bytes?' 'No, only 16384'.*

This tutorial shows what often happens in the real world: client and server programmers pick numbers that seem about right, and only discover problems when strange things start happening. Don't do this.

5. Copy and send the 16 characters in French, and the 10 characters in Mandarin. What happens?

Characters are not bytes and strings are complicated.

6. Control-C the server. Increase the server limit in the program code and run it again. Repeat until all the requests can be received without being truncated.

4 More Coding

Change the client **`sendRequest`** to send a UTF-16 encoded string instead of the current UTF-8. *Don't* change anything else. In Wireshark, what happens to the request and reply? Change back to UTF-8 afterwards.

Modify the client and server programs to use a shared package that contains the maximum message size instead of having a separate limit within each program.

Optional: Have a single pair of send and receive functions/methods in the shared package that are used by both the client and server.

Optional: Modify the server program to shut down nicely on Control-C: it should close the socket and print something as a log messages.
