Assignment 3: TCP and UDP (10%)

Fall 2023 – CPSC 441

Due at 23:59, Nov. 12 on D2L

Assignment policy:

- This is an **individual** assignment, so the work you hand in must be your own. Any external sources used must be properly cited (see below).
- Extensions will not be granted to individual students. Requests on behalf of the entire class will only be considered if made more than 24h before the original deadline.
- Some tips to avoid plagiarism in your programming assignments:

  1. Cite all sources of code that you hand in that are not your original work. You can put the citation into comments in your program. For example, if you find and use code found on a web site, include a comment that says, for example:

     ```
     # the following code is from https://www.quackit.com/hello_world.cfm.
     ```

     Use the complete URL so that the marker can check the source.
  2. Citing sources avoids accusations of plagiarism and penalties for academic misconduct. However, you may still get a low grade if you submit code that is not primarily developed by yourself.
  3. Discuss and share ideas with other programmers as much as you like, but make sure that when you write your code that it is your own. A good rule of thumb is to wait 20 minutes after talking with somebody before writing your code. If you exchange code with another student, write code while discussing it with a fellow student, or copy code from another person's console, then this code is not yours.
  4. Collaborative coding is strictly prohibited. Your assignment submission must be entirely your code. Discussing anything beyond assignment requirements and ideas is a strictly forbidden form of collaboration. This includes sharing code, discussing code itself, or modelling code after another student's algorithm. You cannot use another student's code, even with citation.
  5. We will be looking for plagiarism in all code submissions, possibly using automated software designed for the task.
  6. Remember, if you are having trouble with an assignment, it is always better to go to your TA and/or instructor to get help than it is to plagiarize.

Background:

The goal of this assignment is for you to get some experience with client-server socket programming over both TCP and UDP. You are all familiar with how TCP sockets provide reliable data delivery (within some constraints), while UDP is a lightweight, connectionless protocol that provides minimal guarantees. Your program in this assignment will require reliable data delivery over both TCP and UDP. The TCP sockets can provide these services at the transport layer, but any desired reliability protocols for UDP would have to be implemented at the application layer (if at all). Note that this is not uncommon practice – applications like BitTorrent and Zoom also use a combination of TCP and UDP.

In addition to both types of socket programming, you'll work a bit with data representation. Remember that in any networking application, network communication is a valuable resource, so it is in your interest as a programmer to find an efficient way to represent the data you need to transmit. This becomes particularly important with e.g. video streaming or other compressible media. Here, you will be given a simple scheme to compress vowel data for English text to implement in your assignment code.

Instructions:

Your task in this assignment is to create a "vowelizer" program that splits text into consonants and vowels, transmits these separately across a network, and then combines them at the other end. You can assume for this assignment that there are five vowels: a, e, i, o, and u. Your code should work on both uppercase and lowercase letters, and should preserve capitalization, spacing, and punctuation.

To do this, you will need to create both a client and a server program. These should work as follows. Note that all communication should be over a TCP channel, except for the task of sending the vowel string.

The client end of your program should initiate a TCP connection to communicate with your server. It should then offer a menu including the following options:

- Split text (basic encoding)
- Merge text (basic encoding)
- Split text (advanced encoding)
- Merge text (advanced encoding)
- Quit

If the user selects a split option, the client program will prompt for a string of text. The client program will then send this string to the server, which will split it into consonants and vowels according to the selected encoding. The server should then send the result back in two separate strings to the client, with the consonant string sent over TCP and the

vowel string sent over UDP. Upon receipt, the client should display these strings to the user.

Similarly, if the user selects a merge option, the client program should prompt them for two strings of text according to the selected encoding. The consonant string should be sent to the server over TCP, and the vowel string should be sent to the server over UDP. The server should then merge the strings and echo back the result over TCP, at which point it is displayed to the user.

Two different versions of split/merge operations should be supported. In the basic encoding, incoming text like "Hello there!" is split into two parts "H ll th r !" (on TCP) and " e o e e " (on UDP), with blank spaces as placeholders to keep both text strings the same length. This should make it easy to merge later. Note also that any punctuation, numbers, or real blank spaces in the original text should be preserved in the consonant string, so that you can put everything back together later.

In the advanced encoding, incoming text like "Hello there!" should be split into two parts: "Hll thr!" (on TCP) and "1e2o3e1e" (on UDP). Here, single-digit, non-negative integer values are used to capture how many letters to skip before placing the next indicated vowel when merging the text. You can assume that this value never exceeds 9. Keep in mind that this advanced encoding for the text is more character efficient, but may have different lengths for the TCP and UDP data.

As mentioned earlier, you will have to keep in mind that UDP is not a reliable transport protocol. Pay attention to this as you implement the UDP component. You should design your program so that the client and server do not crash or freeze if a UDP segment is not received. It is recommended that you use a timeout on your UDP server sockets so that your program doesn't freeze waiting for input. If a UDP segment is not received in this time, then your program should display an appropriate error message and return to the main menu.

Finally, remember that the TAs will be testing your code on the university servers, so it is your responsibility to make sure your code runs in that environment. The C++ socket libraries are different for windows and linux, so it is important you plan this from the start.

Rubric (40 points total):

- **12 marks** for a suitable implementation of a client interface, with proper menu display, TCP socket use, user input collection, and reasonably commented code.
- **6 marks** for a suitable server interface, including proper socket setup/closure, port binding, and reasonably commented code.
- **8 marks** for proper handling of the UDP sockets, including approach for handling packet loss.

- **4 + 6 marks** for proper implementation of the simple + advanced split and merging operations.
- **4 marks** for a clear and concise user manual (at most 1 page) that describes how to compile, configure, and use your code. Also briefly describe where and how your testing was done (e.g., home, university, office), what works, and what does not.

Tips:

- Note that all string split/merge functionality is handled at the server. Tasks like getting user input, running the menu loop, and displaying results are assigned to the client.
- Start with a TCP-based echo server, and build up from there.
- It is probably easier to initially build all of the communication functionality using TCP only, and then later make the small modifications required to transmit vowels via UDP instead. This approach will reduce the number of sockets you need to deal with at first, and it is more enjoyable to program when you see something working along the way.
- You should use a single TCP socket connection for the entire client/server session.
- Remember that UDP is a connectionless protocol. Each of the client and the server programs will need both a UDP server and a UDP client socket.
- You will need a port number for your TCP server socket and port numbers for your UDP server sockets. All sockets can technically use the same number, since they are different hosts and/or different transport-layer protocols. However, having different values for these port numbers is fine, and might simplify your life (e.g., conceptualization, coding, debugging, bind errors, Wireshark).
- Focus on getting the simple encoding working before you try the advanced one. Having equal-length strings and blank spaces as placeholders makes things easier. The advanced one may have different string lengths to deal with, plus some data type conversions to worry about. Fortunately, these modifications only involve the server code, not the client code.
- Pay attention to the data types that you are working with. For example, the messages that you send back and forth are likely to be ASCII strings, but processing these may involve manipulating numerical values. Make sure you know how to convert between different data types when needed.
- UDP sometimes loses packets, which may cause your client or your server to fail (e.g., hang), without providing a response. You may want to use a timeout on your UDP socket to detect and handle such a situation.

- You can easily test whether your program handles missing UDP data gracefully by commenting out the line of code at the UDP client where you send data.
- When testing your encodings, you may see your server producing gibberish when the wrong UDP text is applied to the TCP text. Watch out for this, and have fun debugging it!
- When handling multiple clients, you may need to keep track of which TCP text belongs with which UDP text. For this purpose, some type of unique identifier might be helpful. However, if you have a well-designed multi-threaded server, you may not need this at all.

---

author: jcleahy@ucalgary.ca