CPSC 441: Computer Networks

Professor Carey Williamson

Winter 2022

Assignment 2: Vowelizer (40 marks)

Due: **Friday, February 18, 2022** (4:00pm)

Learning Objectives

The purpose of this assignment is to learn about client-server network applications, TCP and UDP protocols, and data representation. In particular, you will use a combination of TCP and UDP to implement a wacky client-server program (in C/C++) that provides different ways of representing ASCII text messages.

Background Scenario

TCP (Transmission Control Protocol) and UDP (User Datagram Protocol) are transport-layer protocols used for sending data over the Internet. TCP is a connection-oriented protocol that establishes a logical connection between a sender and receiver before data can be sent, and sends all data reliably, with acknowledgements. UDP is a connection-less protocol that just sends data, without establishing any connection first. As a result, its data transfers are sometimes unreliable.

Some network applications (e.g., BitTorrent, Zoom) use a combination of TCP and UDP for exchanging data. In this assignment, you will build a new network application that uses both TCP and UDP. Your application is called the "vowelizer", and will be designed to split plain ASCII text messages into separate streams for vowels and non-vowels, and then later merge them back together properly. More specifically, the split operator (also called "devoweling") will transform normal English text (sent via TCP) into vowels (sent on UDP) and non-vowels (sent on TCP), while the merge operator (also called "envoweling") will combine vowels (sent on UDP) with non-vowels (sent on TCP) back into a normal English text message (sent using TCP). Two different algorithms for splitting and merging will be implemented.

For the assignment, you can assume that there are exactly five vowels in English, namely "a", "e", "i", "o", and "u". We will ignore "y", which sometimes behaves like a consonant, and sometimes like a vowel. Your code should

handle both upper-case and lower-case text, preserving the capitalization whenever possible.

Your Task

Your primary task in this assignment is to build a network-based application that supports the text splitting and merging operations described above.

The **client** end of your application will first initiate a TCP connection to the server to establish a session. Within this interactive session, the client can choose what commands to perform (e.g., split, merge, quit), enter the text that is to be processed, view the results from the server, etc.

The **server** end of your application will passively listen on a TCP port number of your own choosing for connection attempts from one or more clients. For each new client, it establishes an interactive session, perhaps indicating a simple menu of commands that the server supports. Incoming data from the client(s) is then either split or merged, depending on the operations requested, and the results sent back to the requesting client. Note that some of this work involves TCP, and some of it UDP.

Two different versions of split/merge operations should be supported. In the **simple** case, 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, and easy to merge later. Note also that any punctuation, numbers, or real blank spaces in the original text should be preserved (i.e., in the non-vowel string), so that you can put things back together properly later. In the **advanced** case, incoming text like "Hello there!" is split into two parts "Hll thr!" (on TCP) and "1e2o3e1e" (on UDP), with (single-digit, non-negative) **integer values** denoting how many letters to skip before placing the next indicated vowel when merging the text. You can assume that the relative displacement value in normal English text never exceeds 9. Note that this advanced encoding for the text may have different lengths for the TCP and UDP data. Also, the punctuation, numbers, and real blank spaces appear only in the TCP text data, so that the UDP carries vowel information only.

Grading Scheme

The grading scheme for the assignment is as follows:

- **14 marks** for the design and implementation of the client (6 marks) and server (8 marks) solution to this problem. Your implementation should include proper use of TCP and UDP socket programming in C or C++, and reasonably commented code.
- **4 marks** for a proper implementation of the **simple** split and merge operations
- **8 marks** for a proper implementation of the **advanced** split and merge operations
- **4 marks** for a clear and concise user manual (at most 1 page) that describes how to compile, configure, and use your solution. Also, describe where and how your testing was done (e.g., home, university, office), what works, and what does not. Be honest!
- **10 marks** for a suitable demonstration of your solution to your TA in your tutorial section, or to your professor at a mutually convenient time. All demos will be done during the week of March 7.

When you are finished, please submit your assignment solution via D2L. Your submission should include your source code and user manual documentation.

Bonus (optional): Up to 4 bonus marks are available if you can devise an even more advanced split/merge operation that is much more economical in its text encodings. That is, the total number of bytes sent over TCP and UDP are (on average) much smaller than with the approaches given above, yet the original text can always be reconstituted correctly. Make sure to mention this to your TA during the demo, and document it in your user manual.

Tips

- Start with something like a TCP-based echo server, and build up from there.
- When developing your solution, 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 is also more likely to generate some encouraging output for you along the way.
- You will need a port number for your TCP socket, and a port number for your UDP socket. They can actually use the same number, since they are 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.
- On a single machine, you can do your initial testing using the loopback interface.
- Once you are testing on a real network interface, you may find Wireshark particularly useful to look at the network packets being sent and received by your application (i.e., ports, size, data content).
- Please be cognizant of 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 some of 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.
- 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.