CS60, Lab 2: A Custom TCP Protocol Server via Berkeley Sockets

Sergey Bratus, Spring 2017

Date due: Solutions to this lab will be due on Thursday April 20 before class.

Platform: Your solutions must compile and run on CS Unix systems.

Submission: Your solutions must be submitted by checking them into the CS department LabGit system at `https://gitlab.cs.dartmouth.edu/`. In your `cs60` project, create a directory called `lab2` and work there. Don't forget to give access to your project to your section's grader and TAs (they must be able to see your code to grade it!)

In Lab1, you wrote a C client that queried a custom protocol server over TCP, using the stream socket API. In this lab, you must write a C *server* for this protocol, with a few enhancements.

Your client code and our client test scripts must work with your server (see `testing.txt` for testing ideas with Netcat (`nc`).

Remember that the server may receive "evil" requests, either from compromised clients or entirely crafted and spoofed by attackers. You server must handle them without crashing (and, preferably, with useful logging of errors).

For ease of testing, your server should take one argument: the port to listen on. If another process is already bound to listen on a TCP or UDP port, or just exited (so that the OS has not "freed" the port in its internal bookkeeping), your server's `bind()` call will fail. Setting the socket option `SO_REUSEADDR` with `setsockopt` on your socket (see "man 7 socket" for documentation of this and other useful options that change OS behavior) will help with just-exited timeouts, but if another student has claimed the port for the moment, just change the port your server listens on. Remember that, as a non-root user, you can only listen on ports above 1024.

Task 1. Implement the server for the exact same protocol (version 1) as in Lab 1. You will find the data to serve in `data.zip`.

Submit: Your `tcpserv.c` and your test scripts.

Task 2. Add UDP functionality to the server: listen for requests on a UDP port, and return the answer over UDP *if* the answer fits inside a single UDP packet. If the answer would not fit, return a status code of 254, and the error string of "Response too large for UDP; try TCP".

Submit: Your `udpserv.c` and your test scripts.

Note: it is possible to combine listening on both a TCP and a UDP socket in the same server, but you would need to use the `select()` function. You may do this for extra credit.

Task 3. Implement an improved version of the protocol, Version 2, as follows, in both TCP and UDP.

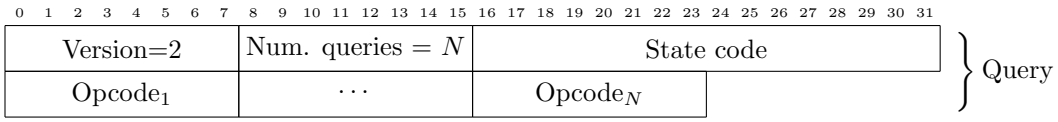
Avoid repeating your code: after ensuring that the response length isnt too long, the rest of the code for handling UDP responses should be the same code as handles TCP responses (i.e., the TCP and UDP code paths should converge, rather than being cut-and-pasted around).

Submit: Your `tcpserv2.c`, your `udpserv2.c`, your library that implements the common part of your protocol handling, and your test scripts.

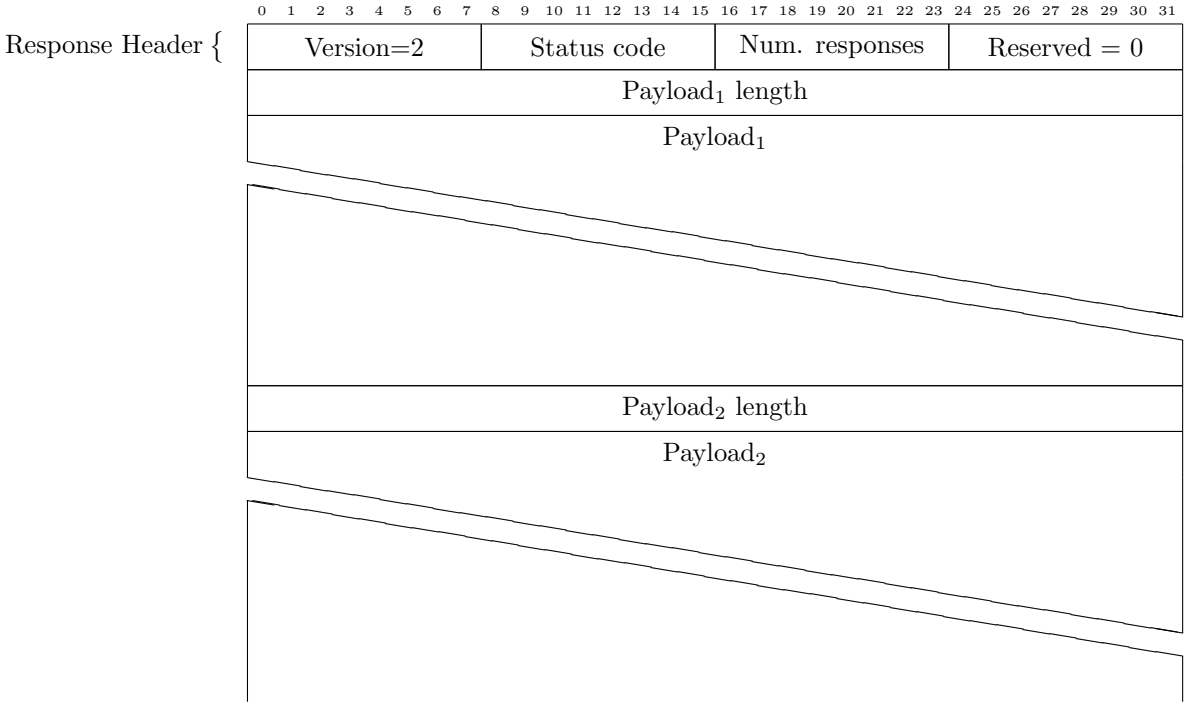
Protocol, v.2:

In version 2, the client can request several information elements about a state per connection, and the server combines them into a single response, "back to back".

A version 2 request is as follows:



A version 2 response is as follows:



Note that although the first length field is 4-byte aligned, the rest are not necessarily so aligned, since they start right after the previous payload ends![1] Thus client code should expect the 4-byte length to follow right after however many bytes were in the previous payload; your server code need not insert any padding between the end of one payload and the length of another.

In case of an error, use the status code 255, make the number of responses 1, and make the error string your one payload, with its 4-byte length being the length of the error string. The design of the error string is up to you; e.g., you could say "One of the query opcodes was invalid".

Important: Your code should not crash no matter what response(s) it receives, even illegal ones! The client may be evil, or its queries could be spoofed. We will test your code with evil invalid queries.

Terms and conditions: You are allowed to use any externals materials, printed or electronic. You are allowed to discuss problems and technical/C tricks, but the code you submit must be your own: you are not allowed to copy solutions from other students. Abide by the Honor Code; if in doubt, ask.

Late submissions: Throughout the course, you will get *two* free extensions for a late submission, each of 48 hours after a deadline. Once you've used these up, you will lose 10% points of credit for each late assignment, for each late day.

[1] Some protocols use padding bytes as needed, to ensure alignment, but not ours.