Project 0: Implementing a multi-client echo server

Due Thursday, January 30, 2014 at 11:59pm

The goal of this assignment is to get up to speed on the Go programming language and to help remind you about the basics of socket programming that you learned in 15-213. In this assignment you will implement a simple multi-client echo server in Go: every message sent by a client should be echoed to all connected clients.

Server Characteristics

Your multi-client echo server must have the following characteristics:

1. The server must manage and interact with its clients concurrently using goroutines and channels. Multiple clients should be able to connect/disconnect to the server simultaneously.
2. The server should assume that all messages are line-oriented, separated by newline (`\n`) characters. When the server reads a newline-terminated message from a client, it must respond by writing that exact message (up to *and including* the newline character) to all connected clients, including the client that sent the message.
3. The server must be responsive to slow-reading clients. To better understand what this means, consider a scenario in which a client does not call `Read` for an extended period of time. If during this time the server continues to write messages to the client's TCP connection, eventually the TCP connection's output buffer will reach maximum capacity and subsequent calls to `Write` made by the server will block.

   To handle these cases, your server should keep a queue of at most 100 outgoing messages to be written to the client at a later time. Messages sent to a slow-reading client whose outgoing message buffer has reached the maximum capacity of 100 should simply be dropped. If the slow-reading client starts reading again later on, the server should make sure to write any buffered messages in its queue back to the client. (Hint: use a buffered channel to implement this property).

Requirements

This project is intentionally open-ended and there are many possible solutions. That said, your implementation must meet the following four requirements:

1. The project must be done individually. You are not allowed to use any code that you have not written yourself.
2. Your code *may not* use locks and mutexes. All synchronization must be done using goroutines, channels, and Go's channel-based `select` statement (not to be confused with the low-level socket select that you might use in C, which is also not allowed).
3. You may only use the following packages: `bufio`, `fmt`, `net`, and `strconv`.
4. You must format your code using `go fmt` and must follow Go's standard naming conventions. See the Formatting and Names sections of Effective Go for details.

We don't expect your solutions to be overly-complicated. As a reference, our sample solution is a little over 100 lines including sparse comments and whitespace. We do, however, *highly recommend* that you familiarize yourself with Go's concurrency model before beginning the assignment. For additional resources, check out the course lecture notes and the Go-related posts on Piazza.

Instructions

The starter code for this project is hosted as a read-only repository on GitHub (you can view the repository online here). To clone a copy, execute the following Git command:

```
git clone https://github.com/cmu440/p0.git
```

The starter code consists of three source files located in the `src/github.com/cmu440/p0` directory:

1. `server_impl.go` is the only file you should modify, and is where you will add your code as you implement your multi-client echo server.
2. `server_api.go` contains the interface and documentation for the `MultiEchoServer` you will be implementing in this project. You should **not** modify this file.
3. `server_test.go` contains the tests that we will run to grade your submission.

For instructions on how to build, run, test, and submit your server implementation, see the `README.md` file in the project's root directory.