WEEBCHAT

A dynamically distributed telnet chat server

First of all, **Dynamic**, the word itself means always active or changing. Second, **Distributed,** which is one, if not the best way to write a future ready server. Put them together and you have a server that can scale horizontally in *N* number of servers.

Why horizontal? Well, this helps you accommodate more users and can save you money because the marginal cost of adding (*vertical scaling*) one more core or a hard drive that does a few more I/O operations per second grows exponentially. In the long run, the cost of adding one more node to the system becomes far cheaper than the cost of additional hardware.

With those in mind, I decided to write a load balancer called "Lobby" to distribute workloads across multiple servers I call "School". Initially, the lobby does not know any school. A school introduces itself to the lobby when it comes online thus, making the number of schools dynamic.

A school is a server that can host a limited number of chatrooms depending on admin configuration. Each time a user creates a chatroom the lobby asks each active school for available slots, the first to respond gets to host the new chatroom. Now when a user tries to join a chatroom the lobby should pass the clients connection to the specific school hosting the room freeing itself from the duty of handling the client.

One of the things to consider when developing a server is the client. A lot of the server's architecture must be complimented by the client's design. Given this task that restrains me to only use **Telnet** as a client, I encountered my first major problem: "How do I properly distribute load between schools when I cannot pass the client's connection from the lobby to a school?"

The usual workflow of a client for a distributed system is:

- Client connects to load balancer.
- Load balancer selects an appropriate server and sends the server info to the client.
- Client receives server info and disconnects from the load balancer.
- Client reconnects directly to the given server.

This allows load to be properly distributed. Problem is, I do not have the liberty of coding my own client. So I must find a way to keep the architecture and comply with the restriction of only using telnet.

With the given constraint I have decided to sacrifice some proper distributed architecture in favor of the following workflow:

- Client connects to lobby.
- Client decides to join a certain room.
- Lobby figures out which server hosts the selected room.
- Lobby now calls methods using RPC on the school to handle the client's requests.

This allowed me to handle telnet clients while keeping the dynamically distributed architecture.

Now as for the code, I have decided to make every request coming from the client be a command. This allows the developer to extend the functionalities of the server by easily adding new commands:

1. Add a new command by specifying it to the list of commands found on "*/src/db/command.js*".
2. Create a new file inside the folder "*/src/commands/".*
3. The name of the file must correspond to the name of the command.
4. The command script must expose 4 important API's:
    a. Callback – the actual command handler.
    b. Structure – how the message should be parsed.
    c. Manual – how the command is used.
    d. and Permission – who can use the command.

This approach makes the code easily maintainable and extensible.

Another aspect to a production ready server is, **security**, I implemented a simple anti-flooding method to prevent users from flooding the server with zombie clients and/or connections.

One should also prevent malicious users from crashing the server. These are usually done by exploiting mistakes made by the developer. A great deal of these is caused by un-sanitized inputs. That is why I have decided to check all inputs and limit each incoming message to 100 characters. This approach can also avoid crashing the server by leaking the memory.

Another thing to consider is that a malicious user can write his/her own custom client that is aimed to find and exploit the server. This can be avoided by keeping in mind to never trust the client and keep everything strict.

Considering that the client has the power to send you anything through telnet, edge cases can happen often. To avoid unexpected behaviors I have decided to write my code in such a way that the last condition is always going to be a fallback. Successful operations are done within early returning conditions. This approach also allows us to tell the user that the operation didn't satisfy any condition and has failed.

I have also noticed another problem with the telnet client. Issuing commands like **ctrl+C** suddenly stops the client from displaying any response from the server.

With all of the above points properly implemented I am sure that this server will perform as expected and should not break from a few connected clients. I have found building this server fun and have plans to continue improving it for academic purposes.

I really look forward in an opportunity to work with Weeby as I really want to get into the industry of "servers" and "games". I hope this will be my stepping-stone into a career in online gaming (my dream).

Best,
M. Siddiqui