Video 5.5

Pipelines: Build the Hydra chat system part 2

|  |
| --- |
| **Metadata**: Spot the problem, highlight it, and design the solution in 3 core steps  (To be covered in the video) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Problem / Solution (Not more than 50 words)** | **Step 1 (Not more than 10 words)** | **Step 2 (Not more than 10 words)** | **Step 3 (Not more than 10 words)** |
| The aim of the video is to cover pipeline patterns in Go  We will cover the definition of a pipeline pattern, then will continute writing a chat system using it. | Write the Hydrachat system TCP server | Write a chat client application | Run and verify the code |

|  |
| --- |
| **Script** the Video – Plan your narration (viewers will see and hear this) |

|  |  |  |  |
| --- | --- | --- | --- |
| **Introduction** | | |  |
| **No.** | **Action on Screen  (Code / Screenshots / One line about the action occurring on screen)** | **Narration**  **(The corresponding explanation to the Action on Screen)** | |
| 1 |  | **Video Introduction**  In the pervious video, we started our effort to write a chat system for the Hydra spaceship using our new knowledge in Go’s concurrency patterns. In this video we will finish building the chat system then run it. |  |
| 2 |  | In this video, we will write the remaining peaces of the Hydra’s chat system code. We will then write a sample client application to test our server code.  We will conclude by shedding more light on the concepts of Fan-In and Fan-Out which are popular concurrency concepts in Go. |  |

|  |  |  |
| --- | --- | --- |
| **Steps or Tasks**  (Refer to the Writing Guidelines- Script Best Practices) | | |
| **No.** | **Action on Screen  (Code / Screenshots / One line about the action occurring on screen)** | **Narration**  **(The corresponding explanation to the Action on Screen)** |
|  |  | Now let's fill the hydrachat.go file, this is where the chat tcp server will reside.  We will import several packages as shown, which we will use one by one. We will then initiate a new instance of the hydra logger we wrote in section 4. |
|  |  | We now write a run function which will act as the tcp chat server. For now, we will hardcode port 2100. We will use the createRoom function we wrote to create a chat room.  Will do typical error checks. |
|  |  | Now we write our tcp listener in it’s own goroutine so that run can return.  Writing a tcp listener in Go is very straight forward as shown. We covered the net package before in section 3.  Every time we receive a connection, we will pass it to the handle connection goroutine to handle it concurrently. |
|  |  | Inside handleconnection, we just add a client to the room we created earlier. |
|  |  | We will conclude by writing logic to detect if the program is in the process of exiting.This is done by monitoring the o.s. signals to our program via the signal.Notify function.  The way Notify work is that we give it a channel of type os.Signal as well as the type of signals we are looking for. Once the os signals trigger, the channel receives a value.  We use the INT and TERM signals which either should trigger when our application is closing. |
|  |  | Once the os.signal triggers, we close our tcp connection, then the quit channel.  Closing the quit channel will act as a broadcast to all the goroutines trying to read from it. This will trigger all the clients to exit and the room closing.  if we exit the program right after closing the quit channel, we probably won't get enough time to clean up all the active clients.  It should make sense to listen and wait on the room message channel, which should close when all the clients close as shown from the room type code.  However, if the client count is already 0, we shouldn’t wait on r.Msgch because no one is in the chat room.  We will need to add a handy method to the room struct which we will use to figure out the number of clients or people currently in the room. We then use this method to decide whether we should wait on the MsgCh or not. |
|  |  | We then run the chat code from the Hydra main function, and report errors if any come up. |
|  |  | It’s now time to write the chat tcp client. Let’s add a folder for the client with a main.go file |
|  |  | This will be a simple executable so it will be the main package. We will import a bunch of packages for our usage.  We will start by creating a name string with the word anonymous and a random number. The client will then simply ask the user for their name.  If for some reason, we weren’t able to scan the name, we will use the name anonymous name we setup earlier instead.  The name string will be attached to any message we sent in order for us to be known. |
|  |  | It will then connect via TCP to the chat server using net.Dial.  We will add : to the name since they will always be attached when sending messages.  We will use a scanner to read any message getting received on the tcp buffer as shown. This is made possible because net.Conn supports the io.Reader interface, which is what the scanner uses. |
|  |  | We now create another scanner to read from the standard input. Whenever the user enters a message, we will attach their name to the message then send it to the chat server by writing to the tcp buffer.  We attach the name to the message so that when the message gets communicated to other chat clients, the users will know who sent the message.  Fprintf will work because conn implements the io.writer interface, which is what Fprintf uses. |
|  |  | Now let's install the project and run it. |
|  |  | When we start the client, it will ask us for our name, let’s say we pick Ryan. When we type a message, it will be sent with our name to the chat server which will send it to everyone including us. |
|  |  | Let’s run multiple client sessions with the server session.  As seen, we are fully able to communicate between clients and server. Another piece of operation Hydra is accomplished. |
|  |  | Now let's discuss Fan-Ins and Fan-Outs, which are important to understand in Go for writing powerful concurrent programs. We will start with Fan-In. Which is basically when we take many Go channels and put their data on a single output channel which can be easily processed elsewhere in our code.  This is useful when we have too many inputs and we want to process them in one place, it makes sense to consolidate them into a single output channel, then process this channel elsewhere. |
|  |  | The opposite of Fan-In is Fan-Out which is when we take data from a single channel then distribute it amongst multiple channels. This is a powerful pattern to use when we need to distribute our data load on multiple channels where each piece of data will be processed individually.  We used that technique in our chat program in order to distribute a message to the different clients connected to the chat server via the broadcastMsg method. |
|  |  | Since we haven’t seen an example for Fan-In of channels. Let’s see an example from <https://blog.golang.org/pipelines> . This function is called merge, it takes any number of int channels , then keeps outputing their data into a single channel.  The function take variadic arguments. This means, when we use the … the caller can include any number of int channels as arguments.  We will see how we’ll use the wait group wg in a little bit. We create a channel called out, which we will use as our output channel. |
|  |  | Next we write a function called output which can take a channel, then keep listening to it using a for range. Whenever a value comes on the channel, we will write it to the out channel. We will set the wait group as done after the for loop exits, which will signal that channel c was closed.  We will then add the number of channels we received as arguments to the wait group. The wait group will be used to signal when all the channels we listen to get closed. This is because when we receive wait group done signals that correspond to the number of argument channels, then we have finished. |
|  |  | We then write another for loop. This for loop is a normal one that will just loop on the argument channels list. It will take each argument channel then call the output function on them on different goroutines.  We then write a new goroutine which will wait on our wait group. Once the wait group is done, which will happen when all the output goroutine exit, then we will close the output channel.  Outside of the goroutine we return the out channel so that the caller can use it. |
|  |  | We can then use the output of the merge in a for range loop to consume it. In the example here, we merged two channels c1 and c2, any data coming from either channel will be forwarded by the merge function to us. So we basically fanned in c1 and c2 to a single output channel. |

|  |  |
| --- | --- |
| **Summary** | |
|  | In this video, we finished our fun journey of writing the Hydra chat system. We now have full functional chat system that makes good use of Go’s features and design patterns. |
| [Mandatory slide] – Next Video | In the next video, we will tackle a new master topic in the Go universe, which is reflection. We will discuss the 3 laws that we have to be fully aware of when using the power of reflection in the language. |