Video 5.4

Pipelines: Build the Hydra chat system part 1

|  |
| --- |
| **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 start writing a chat system using it. | Define pipeline patterns | Write Hydra chat room type | Write Hydra chat client type |

|  |
| --- |
| **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 previous video, we talked about Go channel generators design pattern. In this video, we will expand further on Go’s concurrency patterns by exploring pipelines and cancellation patterns. |  |
| 2 |  | We will start by defining what a pipeline pattern is, then we will take deep code dive to write a chat system for the Hydra space ship using Go’s concurrency muscle. We will use several concepts covered so far in this section to build the Hydra chat system. |  |

|  |  |  |
| --- | --- | --- |
| **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)** |
| 3 |  | So what is the pipeline design pattern? A detailed article covering the pattern can be found in the link shown here.  Pipeline is a design pattern that makes heavy use of Go’s concurrency. It divides code into series of stages that can communicate using Go channels. Each stage is a group of goroutines performing some function.  Each stage receives values via channels, perform some work on them, then send them down using channels to the next stage.  First stage can be also called the producer, whereas the last stage can be called the consumer.  Pipelines exist in many other languages beside Go and are popular for efficient software designs. |
| 4 |  | We will design our Hydra chat system using a data pipeline.  There will be a chat server and multiple chat clients. The data flow will start from the clients to the server, and then back.  After a bunch of clients connect to the server, a client will send a message to a chat room in the chat server. After that, the server will broadcast or fan out the client message to all the clients connected to the chat server on the same chat room. |
|  |  | Let's see how we can use the pipeline pattern to design our chat application: Let’s assume the pipeline pattern starts after some clients are connected to the chat server. The first stage is when we get a message from a client, this message will go to a shared channel called msgCh.  In the second stage, Data on msgCh will then be copied to multiple channels representing the clients currently connected to the chat server. This operation is called fan-out.  On the third stage, each one of those channels will send data to the client it represents. |
| 5 |  | We will create a project folder called hydra chat under project Hydra. Underneath, we will add three files for now, client.go , hydrachat.go , and rooms.go |
| 6 |  | Let’s start by room.go which will host the chat room code. The package name will be hydrachat. We will import three packages: fmt, net , and sync. We will create a struct type called room.  In there, we will create fields to cover the room name, a channel to host messages passing through the chat room, a clients map which will represent the people currently talking in the chat room, a Quit channel that we will use to signal that the room is quitting and all clients need to disconnect. We will also embed an RWMutex to use for protecting our client map from concurrent writes.  Notice how the map values are just empty structs? That’s because we will use this map as a SET type similar to the exercise in video 2.5.  The map keys represent string send only channels which we can use in order to pass messages to clients. Clients will listen to these channels. We can call them client message channels. |
| 7 |  | Now let’s write the code to create the room via a struct literal, we will initialize all the fields in order to be able to use them right away.  Before we return the room pointer type to the caller, we will call run on the room so that the room can start waiting for people to connect. We will discuss the run method next. |
| 8 |  | The Run method will first output a log line announcing the chat room is starting. We will then create a go routine which will use the for range with the MsgCh channel in order to capture any messages that get passed to the channel. msgch represents the message channel for the room.  We will write our code such that all client chat messages end up being delivered to Msgch. Once we capture a client chat message, we need to broadcast it to all connected clients. This logic was placed in a goroutine in order to avoid it from blocking Run() from returning. |
|  |  | Now let's explore the broadcastMsg method. This method will take the message string we obtained from the msgch channel and will send it to all the client message channels that we have in our clients map. We will use read locks because we are not writing to the map in this code. The client channels are called wc because they represent write channels to which we can write to the clients.  We will send each message on a separate goroutine for faster execution. This also ensures that if there is a problem with one of the clients receiving the messages, it won’t affect the other clients. |
| 9 |  | Now let's look at the code for adding a client. For the purpose of this exercise, we will assume clients connect via tcp to the chat server. This will make the chat server a tcp server while the chat clients tcp clients.  In here will use the write lock to protect our clients map while we add items to it. We will call a channel generator function called StartClient. It will take the room object message channel, as well as the quit channel at which quit signals will be sent when the room is closing.  StartClient will return two channels, one which will be used to pass messages to that specific client, the second channel will be a signal we receive when the client no longer part of the chat room. |
|  |  | The second half of the AddClient method will start a goroutine which will block up till the point the done signal gets triggered. When it does, we will remove this client as it is no longer relevant.  This is a nice design approach in Go which allows us to clean up after the client is done without having to write complex logic elsewhere to do that. All what’s left is figuring out when to signal the done channel. |
| 10 |  | Now let’s discuss the remove client method. We will use the write lock here too because we are modifying the clients map here too.  We will delete the map, then close the client channels.  We will then use the magic of Go’s select statement to check whether the room quit channel had been signaled or not. If so, and there are no more clients in the room, then we know this is the last client to leave the room so we clean the room by closing the message channel of the room.  We use a default as in the select to ensure this code doesn’t block the RemoveClient method. If r.Quit is not ready then that means it’s not closed yet.  When we close the room message channel, the room object will stop monitoring the channel and the respective for range goroutine will return. |
| 11 |  | Now let's write code in the clients.go file. This is where we write the behavior of the chat clients.  We start by creating a client struct which takes a bufio reader, a bufio writer , and the message channel of the client. |
| 12 |  | Time to explore the code of the StartClient method we used in the room code. We will start by creating a new client, then we will set the reader and writer of the client object to be buffered readers and writers of the network connection buffer. This will allow us to efficiently send and read messages on the network connection.  We will create the client channel as shown. Then we will create a new channel to represent the done signal which we will trigger when the client is not longer relevant in order to inform the room object that it needs to clean it up. |
|  |  | Next we will invoke a new goroutine where we will setup a scanner on the network connection buffer. The Scan method will be true when the underlying buffer is healthy, and false when it is not. If it is not then the for loop will break, and we send the done signal.  Whenever we receive a new message in the buffer, we pass it to the room message channel. |
|  |  | We then set the write monitor which will take care of writing to the underlying network buffer whenever a message needs to be consumed by the client. We will discuss the method shortly.  We will then create another goroutine to watch for the room quit channel. Whenever the quit channel gets signaled, we will close the underlying network connection effectively dropping the client.  The only case where we would ignore the quit channel is if we already done since this will mean our connection is closing. We use the select statement for this check. We can leave the done case blank since no action is needed other than exiting the goroutine when done triggers.  At the end, we return the client channel and the client done signal. |
|  |  | In the write monitor, we will create a new goroutine which will listen on the client channel using the for range technique. Once a message is ready, we will write it to the underlying client network buffer. |

|  |  |
| --- | --- |
| **Summary** | |
|  | In this video, we used a lot of our new knowledge to write the first piece of the Hydra chat system. We covered pipelines and how it fits in software design. |
|  | In the next video, we will continue in our journey building the Hydra chat system. |