ECGR 4090/5090 Cloud Native Application Architecture
Lab 3: Go Channels

"Do not communicate by sharing memory; instead, share memory by communicating" - The Go Way

In Go, each concurrently executing activity is called a goroutine. New routines are created by the go statement. Syntactically, a go statement is an ordinary function call prefixed by the keyword go. A go statement causes the function to be called in a newly created goroutine.

```
f() // call f(); wait for it to return
go f() // create a new goroutine that calls f(); don't' wait
```

Channels are the connections between goroutines. A channel is a communication mechanism that lets one goroutine send values to another goroutine. Each channel is a conduit for values of a particular type, called the channel's element type. A channel has two principal operations, send and receive. A send statement transmits a value from one go routine, through the channel, to another goroutine executing a corresponding receive expression. Both operations are written using the <- operator. In a send statement, the <- separates the channel and value operands. In a receive expression, <- precedes the channel operand. A receive expression whose result is not used is a valid statement.

```
ch <- x // a send statement
x <- ch // a receive expression in an assignment statement
<-ch // a receive statement; result is discarded
```

Channels support close operation, which sets a flag indicating that no more values will ever be sent on this channel. Receive operations on a closed channel yields values that have been sent until no more values are left; any receive operations thereafter complete immediately and yield the zero value of the channel type.

To close a channel, we call the built it close function.

```
close(ch)
```

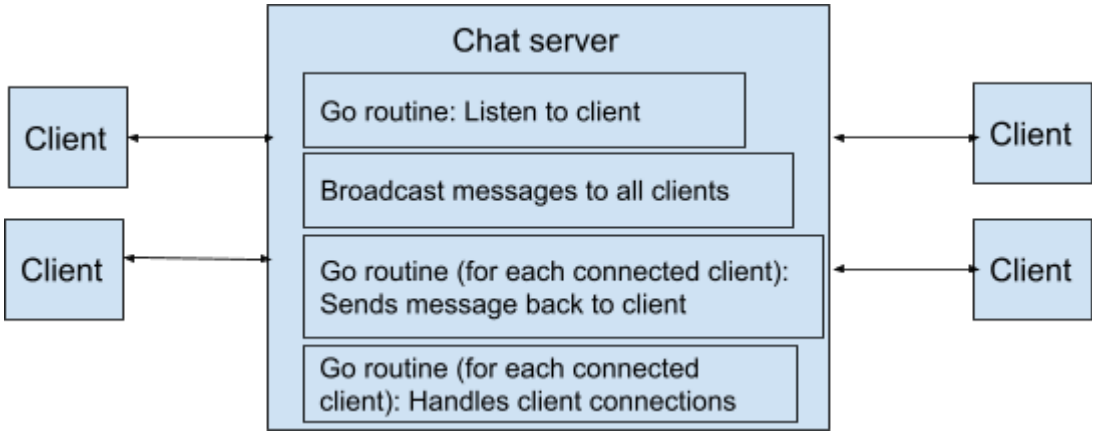
Example usage of goroutines and channels
Using goroutines and channels to concurrently sum elements in an array. Here 2 goroutines are created, that each gets one half of the array, and a channel to communicate the computed sum back to the main thread.

```
package main

import "fmt"
```

```
func sum(s []int, c chan int) {
        sum := 0
        for _, v := range s {
                sum += v
        }
        c <- sum // send sum to c
}

func main() {
        s := []int{7, 2, 8, -9, 4, 0}

        c := make(chan int)
        go sum(s[:len(s)/2], c)
        go sum(s[len(s)/2:], c)
        x, y := <-c, <-c // receive from c

        fmt.Println(x, y, x+y)
}
```

Chat server: A more sophisticated use of goroutines and channels
(from The Go Programming language book)

We'll look at a chat server where, like many cloud server applications, many operations occur concurrently. Multiple clients connect to a chat server, to post messages. A message posted by a particular client is broadcast to all other clients. Also, the server has to keep track of clients entering and leaving the server.



3 global channels for communication between goroutines
entering channel : a channel of client channels.

leaving channel: a channel of client channels
message channel: for chat messages (strings)

main(): Spawns broadcast go routine. In an infinite loop accepts client connections, creates a connector for that client, and invokes a Go routine with call to *handleConn*

broadcaster: Keep a map of clients in the system (key: client channel, value: true). A particular client's channel is extracted from the entering channel and stored in the map. Messages received on the message channel are written to each client channel stored in the map (broadcast). When a client leaves, the corresponding channel is deleted from the map.

handleConn: Creates a channel for the client (*ch*) to receive messages from the broadcaster. This channel is written to the *entering* channel which the broadcaster can then use. A goroutine (*clientWriter*) is spawned to take messages from *ch* and send to the client over the network through the connector. All messages received over the network from the client are written to the message channel which goes to the broadcaster. When no messages are left (client types Ctrl-C), its channel is written to the leaving channel, and the connection is closed.

clientWriter: Ranges over the client channel *ch*, and prints the messages (received from other clients).

Create a lab3 directory in the labs directory. We are not developing anu packages in this lab.

Download chat.go from Canvas
On one terminal launch chat server - *go run chat.go*
On a second terminal launch first chat client with netcat utility - *netcat localhost 8000*
On a third terminal launch a second chat client with netcat utility - *netcat localhost 8000*
Chat away!
Ctrl-C to close clients and server
Note: Install netcat if needed (*sudo apt-get install netca*t)

To do -
Make the broadcaster announce the current set of clients to each new arrival. This requires that the clients set their name on the entering and leaving channels.
Hint: Convert type client to a struct with two fields - channel and name