**SCHOOL OF INFORMATION TECHNOLOGY AND ENGINEERING**

**ADDIS ABABA INSTITUTE OF TECHNOLOGY, ADDIS ABABA UNIVERSITY**

Topic: Message Passing with Go Channels

## Activity 1: Message Passing with Go Channels

### Objective:

Understand message passing within a single Go application using channels, which are Go's native message-passing mechanism for goroutines.

### Activities:

1. Basic Message Passing with Channels:
   * Task: Write a Go program that launches two goroutines: one to produce messages and one to consume them. Use an unbuffered channel to pass a simple message (like a string or integer) between them.
   * Instructions:
     + Use go keyword to create goroutines.
     + Set up a chan string channel to pass string messages.
     + The producer goroutine sends messages like "Hello from producer!" every second.
     + The consumer goroutine prints the received message.
   * Expected Outcome: The consumer goroutine should print messages as they are passed by the producer.
2. Buffered Channels:
   * Task: Modify the program to use a buffered channel. Experiment with different buffer sizes and observe how it affects the behavior of the producer and consumer goroutines.
3. Channel Synchronization:
   * Task: Implement a "shutdown signal" using a bool channel to gracefully terminate both goroutines after a fixed time (e.g., 10 seconds).
   * Expected Outcome: Both goroutines should terminate cleanly after the specified time.

Codes:

unbuffered.go

package main

import (

"fmt"

"time"

)

func producer(ch chan<- string) {

for i := 1; i <= 5; i++ {

ch <- fmt.Sprintf("Message %d", i)

time.Sleep(1 \* time.Second)

}

close(ch) // Close the channel after sending all messages

}

func consumer(ch <-chan string) {

for msg := range ch {

fmt.Println("Received:", msg)

}

}

func main() {

ch := make(chan string)

go producer(ch)

consumer(ch) // Run the consumer on the main thread to keep it open until all messages are received

}

buffered.go

package main

import (

"fmt"

"time"

)

func producer(ch chan<- string) {

for i := 1; i <= 5; i++ {

fmt.Printf("Sending: Message %d\n", i)

ch <- fmt.Sprintf("Message %d", i)

}

close(ch)

}

func consumer(ch <-chan string) {

for msg := range ch {

fmt.Println("Received:", msg)

time.Sleep(2 \* time.Second) // Simulate processing time

}

}

func main() {

ch := make(chan string, 3) // Buffered channel with a capacity of 3

go producer(ch)

consumer(ch)

}

synchronization.go

package main

import (

"fmt"

"time"

)

func producer(ch chan<- string, quit <-chan bool) {

for i := 1; ; i++ {

select {

case <-quit:

fmt.Println("Producer shutting down")

return

case ch <- fmt.Sprintf("Message %d", i):

fmt.Printf("Produced: Message %d\n", i)

time.Sleep(1 \* time.Second)

}

}

}

func consumer(ch <-chan string, quit chan<- bool) {

for i := 0; i < 10; i++ {

fmt.Println("Consumed:", <-ch)

}

quit <- true

}

func main() {

ch := make(chan string)

quit := make(chan bool)

go producer(ch, quit)

go consumer(ch, quit)

<-quit // Wait for shutdown signal

fmt.Println("Main shutting down")

}

## Activity 2: Using RabbitMQ for Message Passing in Go

### Objective:

Set up a RabbitMQ server to implement message passing in a distributed context using Go.

### Prerequisites:

* Install RabbitMQ.
* Install the Go RabbitMQ client (github.com/streadway/amqp).

### Activities:

1. Setting Up RabbitMQ:
   * Task: Start the RabbitMQ server on your local machine and use the RabbitMQ management console to view message queues.
   * Outcome: Verify that RabbitMQ is running and accessible.
2. Sending Messages to RabbitMQ:
   * Task: Write a Go program to act as a producer that sends messages to a RabbitMQ queue named task\_queue.
   * Instructions:
     + Import the amqp package.
     + Connect to the RabbitMQ server and declare a queue (task\_queue).
     + Publish a message (e.g., "Hello, world!") to the queue.
   * Expected Outcome: Messages should appear in the task\_queue on the RabbitMQ management console.
3. Consuming Messages from RabbitMQ:
   * Task: Write a separate Go program that acts as a consumer to receive messages from the task\_queue.
   * Instructions:
     + Connect to RabbitMQ and declare (or access) the same task\_queue.
     + Use a Deliveries channel to receive messages, and print each received message.
   * Expected Outcome: The consumer program should print each message sent by the producer.
4. Simulating a Distributed System:
   * Task: Modify the producer to send multiple messages (e.g., a list of tasks) and run multiple instances of the consumer program.
   * Reflection Questions:
     + How does RabbitMQ handle load balancing between multiple consumers?
     + What happens when a consumer disconnects?
5. Acknowledgment and Reliability:
   * Task: Implement acknowledgment (ack) in the consumer program to confirm message processing.
   * Instructions:
     + Modify the consumer to manually acknowledge messages after they are processed.
     + Use the RabbitMQ management console to monitor unacknowledged messages if any consumer fails.
   * Expected Outcome: Messages should not be removed from the queue until acknowledged by a consumer.

Code:

Install RabbitMQ Go client:

go get github.com/streadway/amqp

producer.go

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

}

}

func main() {

conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")

failOnError(err, "Failed to connect to RabbitMQ")

defer conn.Close()

ch, err := conn.Channel()

failOnError(err, "Failed to open a channel")

defer ch.Close()

q, err := ch.QueueDeclare(

"task\_queue",

true,

false,

false,

false,

nil,

)

failOnError(err, "Failed to declare a queue")

body := "Hello, RabbitMQ!"

err = ch.Publish(

"",

q.Name,

false,

false,

amqp.Publishing{

DeliveryMode: amqp.Persistent,

ContentType: "text/plain",

Body: []byte(body),

})

failOnError(err, "Failed to publish a message")

fmt.Println("Sent:", body)

}

consumer.go

package main

import (

"fmt"

"log"

"github.com/streadway/amqp"

)

func failOnError(err error, msg string) {

if err != nil {

log.Fatalf("%s: %s", msg, err)

}

}

func main() {

conn, err := amqp.Dial("amqp://guest:guest@localhost:5672/")

failOnError(err, "Failed to connect to RabbitMQ")

defer conn.Close()

ch, err := conn.Channel()

failOnError(err, "Failed to open a channel")

defer ch.Close()

q, err := ch.QueueDeclare(

"task\_queue",

true,

false,

false,

false,

nil,

)

failOnError(err, "Failed to declare a queue")

msgs, err := ch.Consume(

q.Name,

"",

true,

false,

false,

false,

nil,

)

failOnError(err, "Failed to register a consumer")

go func() {

for d := range msgs {

fmt.Printf("Received a message: %s\n", d.Body)

}

}()

log.Println("Waiting for messages...")

select {}

}

## Activity 3: Exploring Publish-Subscribe with NATS

### Objective:

Use NATS, a lightweight, high-performance message queue, to understand publish-subscribe message-passing patterns.

### Prerequisites:

* Install NATS Server.
* Install the Go NATS client (github.com/nats-io/nats.go).

### Activities:

1. Setting Up NATS:
   * Task: Start a local NATS server and explore basic commands to connect to it.
   * Expected Outcome: NATS server should be up and running locally.
2. Implementing a Publisher:
   * Task: Write a Go program to act as a publisher that sends messages to a specific NATS subject (updates).
   * Instructions:
     + Import the nats.go package.
     + Connect to the NATS server.
     + Publish a message to the updates subject.
   * Expected Outcome: Messages should be published successfully to the updates subject.
3. Implementing Subscribers:
   * Task: Write another Go program to subscribe to the updates subject and print received messages.
   * Instructions:
     + Connect to the NATS server.
     + Subscribe to the updates subject and log incoming messages.
   * Experiment: Run multiple instances of the subscriber to simulate a pub-sub system.
   * Expected Outcome: All subscribers should receive every message published to updates.
4. Message Filtering:
   * Task: Implement different subjects (e.g., updates.error, updates.info) to filter messages for different subscribers.

Code:

Install NATS Go client:

go get github.com/nats-io/nats.go

publisher.go

package main

import (

"fmt"

"log"

"github.com/nats-io/nats.go"

)

func main() {

nc, err := nats.Connect(nats.DefaultURL)

if err != nil {

log.Fatal(err)

}

defer nc.Close()

subject := "updates"

message := "Hello, NATS!"

if err := nc.Publish(subject, []byte(message)); err != nil {

log.Fatal(err)

}

fmt.Println("Sent:", message)

}

subscriber.go

package main

import (

"fmt"

"log"

"github.com/nats-io/nats.go"

)

func main() {

nc, err := nats.Connect(nats.DefaultURL)

if err != nil {

log.Fatal(err)

}

defer nc.Close()

subject := "updates"

\_, err = nc.Subscribe(subject, func(m \*nats.Msg) {

fmt.Printf("Received a message: %s\n", string(m.Data))

})

if err != nil {

log.Fatal(err)

}

fmt.Println("Subscribed to updates. Waiting for messages...")

select {} // Keep the program running

}

## **Submission Requirements:**

1. Code Submission:
   * Submit both all code for activity 1-3.
2. Testing Evidence:
   * Provide screenshots showing interaction.
3. Reflection Report:
   * How does the buffer size affect the frequency and timing of message passing?
   * What happens when the buffer is full?
   * How does RabbitMQ handle load balancing between multiple consumers?
   * What happens when a consumer disconnects?
   * How does NATS handle different subjects?
   * What advantages does this give in message organization?