پارسا کوت‌زری
۸۱۰۱۹۷۵۷۰

گزارش پروژه ۱ (message broker)

طراحی کلی:

این پروژه شامل ۳ ماژول است: broker, client, server

ماژول سرور همان پابلیش کننده در صف و ماژول کلاینت سابسکرایب کننده به صف است. ماژول بروکر مسئول مدیریت صف‌ها و ارسال مسیج‌هاست. این ماژول خود شامل ۲ فایل broker.go و brokerclient.go است. فایل broker.go شامل پیاده سازی اصلی بروکر است که در آن صف‌ها و مسیج‌ها و نحوه ارسال و نگه‌داری آنها پیاده‌سازی شده است و در فایل brokerclient.go توابع و اینترفیس‌هایی جهت استفاده از broker پیاده سازی شده است و پراسس‌ها مختلف از این توابع برای ارتباط با broker استفاده میکنند.

طراحی کلی broker به این صورت است که یک ساختار message دارد.

```
type Message struct {
    recievers  []*Client
    message    string
    message_Id int
}
```

این ساختار اطلاعات هر مسیج شامل متن پیام، آیدی پیام و اطلاعات گیرندگان پیام را ذخیره میکند. آیدی پیام برای acknowledge کردن آن استفاده می شود.

همچنین ساختار صف به شکل زیر است.

```
type Queue struct {
    name      string
    max_len   int
    messages  []Message
    recievers []*Client
}
```

که در این ساختار هر صف یک اسم دارد که با آن شناسایی میشود. همچنین مقدار حداکثر ظرفیت دارد. در نهایت هم آرایه‌ای از مسیج‌ها که باید ارسال شودند و لیستی از اطلاعات گوش‌دهندگان به این صف در ساختار قرار دارد. هر زمان مسیجی وارد یک صف میشود لیست دریافت کنندگان پیام برابر گوش‌دهندگان صفی که به آن وارد شده مقداردهی میشود.

ساختار client که اطلاعات پراسس های در ارتباط با broker را نگه داری میکند به این شکل است.

```
type Client struct {
    Host           string
    Port           string
    ConnectionType string
}
```

هر کلاینتی یک host, port دارد که آدرس آن را مشخص میکند و همچنین نحوه کانکشن که tcp است در این ساختار نگه‌داری میشود. برای شناسایی و تفکیک کلاینت‌ها از دو فیلد host و port استفاده میشود.

همچنین ساختار صف ۲ طرفه وجود دارد که به این صورت پیاده سازی میشود.

```
type TwoWayQueue struct {
    name          string
    one_way_queus [2]Queue
}
```

در این ساختار یک صف دوطرفه شامل اسم برای شناسایی آن و ۲ تا صف یک طرفه است که هر صف برای یک پراسس برای گرفتن مسیج و یک صف برای پابلیش کردن مسیج استفاده میشود.

تابع main بروکر به این صورت است.

```
func main() {
    init_queues()
    go send_messages_in_queues()
    start_server()
}
```

در اینجا برای کاهش پیچیدگی صف های موجود و اطلاعات آنها در broker به صورت hard code قرار داده شده است و صف‌ها در تابع init_queues ساخته می شوند.

```go
func init_queues() {
    queues = append(
        queues,
        Queue{
            name:      "owqueue",
            max_len:   100,
            messages:  make([]Message, 0, 100),
            recievers: []*Client{},
        },
    )

    queues = append(
        queues,
        Queue{
            name:      "zerocapacityqueue",
            max_len:   0,
            messages:  make([]Message, 0, 0),
            recievers: []*Client{},
        },
    )

    two_way_queues = append(
        two_way_queues,
        TwoWayQueue{
            name: "twqueue",
            one_way_queus: [2]Queue{
                {max_len: 100, messages: make([]Message, 0, 100), recievers: []*Client{}},
                {max_len: 100, messages: make([]Message, 0, 100), recievers: []*Client{}},
            },
        },
    )
}
```

همانطور که میبینید در این تابع ۲ صف یک طرفه با نام‌های owqueue و zerocapacityqueue ساخته شده اند که ظرفیت یکی از آنها ۱۰۰ و ظرفیت دیگری ۰ است.

همچنین یک صف دو طرفه با نام twqueue ساخته شده است.

دو تابع دیگر main وجود دارد. تابع send_messages_in_queues به صورت یک ترد جداگانه اجرا میشود و هدف آن چک کردن صف‌ها و فرستادن مسیج‌های آنها به گیرندگان است. در این تابع برای هر صف (چه یک طرفه و چه دو طرفه) یک ترد جدید ایجاد میشود که این ترد هر زمان که در صف مسیجی موجود باشد آن را ارسال میکند و اگر خالی باشد یک ثانیه sleep میشود و سپس دوباره صف را چک میکند. به این صورت هر زمان پیام جدیدی وارد صف شود به این شکل از صف برداشته شده و به ارسال میشود.

```go
func send_messages_in_queues() {
    for i := 0; i < len(queues); i++ {
        go send_queue_messages_till_its_empty(&queues[i])
    }
    for i := 0; i < len(two_way_queues); i++ {
        go send_queue_messages_till_its_empty(&two_way_queues[i].one_way_queus[0])
        go send_queue_messages_till_its_empty(&two_way_queues[i].one_way_queus[1])
    }
}

func send_queue_messages_till_its_empty(queue *Queue) {
    for {
        message := get_message_from_queue(queue)
        if message != nil {
            send_message(*message)
        } else {
            time.Sleep(1 * time.Second)
        }
    }

}
```

در تابع send_message مسیج به تمام گیرندگانی که اطلاعاتشان در لیست گیرندگان مسیج آمده باشند ارسال میشود و منتظر رسیدن ack هر مسیج میشود.

```go
func send_message(message Message) {
    for _, client := range message.recievers {
        connection, err := net.Dial(client.ConnectionType, client.Host+":"+client.Port)
        if err != nil {
            panic(err)
        }
        _, err = connection.Write([]byte(message.message))
        if err != nil {
            fmt.Println("failed to send the message", err)
        }
        fmt.Printf("Sent '%s' to consumer\n", message.message)
        buffer := make([]byte, 1024)
        mLen, err := connection.Read(buffer)
        if err != nil {
            fmt.Println("Error reading:", err.Error())
        }
        fmt.Printf("Acknowleged: %s with message_Id: %d\n", string(buffer[:mLen]), message.message_Id)
        mark_acknowleged_message(message.message_Id)
        connection.Close()
    }
}
```

تابع start server مربوط به ایجاد یک سرور روی پورت 9999 است که پراسس‌های مختلف بتوانند درخواست‌های خود را از این طریق به دست broker برسانند.

```go
func start_server() {
    fmt.Println("Running Broker...")

    server, err := net.Listen(brokerclient.BROKER_TYPE, brokerclient.BROKER_HOST+":"+brokerclient.BROKER_PORT)
    if err != nil {
        fmt.Println("Error listening:", err.Error())
        os.Exit(1)
    }
    defer server.Close()
    fmt.Println("Listening on " + brokerclient.BROKER_HOST + ":" + brokerclient.BROKER_PORT)
    fmt.Println("Waiting for connections...")
    for {
        connection, err := server.Accept()
        if err != nil {
            fmt.Println("Error accepting: ", err.Error())
            os.Exit(1)
        }
        go processClient(connection)
    }
}
```

به طور کلی هر پراسس میتواند ۴ نوع درخواست به بروکر ارسال کند. هر درخواست در فرمتی به شکل زیر قرار میگیرد و سپس به صورت بایت درمی‌آید و از پراسس مورد نظر به سمت بروکر ارسال میشود. سپس در بروکر دوباره به دیکود میشود و از روی اطلاعات موجود در این ساختار درخواست مورد نظر رسیدگی میشود.

```go
type RequestType int

const (
    SUBSCRIBE_TO_QUEUE RequestType = iota
    CONNECT_TO_TWO_WAY_QUEUE
    SEND_MESSAGE_TO_ONE_WAY_QUEUE
    SEND_MESSAGE_TO_TWO_WAY_QUEUE
)

type SendMessageRequest struct {
    TypeOfRequest RequestType
    Message       []byte
    QueueName     string
    ClientData    Client
}
```

درخواست subscribe_to_queue مربوط به گوش دادن به یک صف یک طرفه است و نام صف به همراه اطلاعات گیرنده (مثل host و port که مسیج باید به آنها ارسال شود) در این ساختار قرار میگیرد(فیلد message در این نوع درخواست استفاده نمیشود) با داشتن این اطلاعات بروکر این کلاینت رو به لیست گوش دهندگان صف نام برده شده اصافه میکند. لازم به ذکر است حالت‌هایی مثل وجود نداشتن صف ذکر شده که باعث ارور میشود با پیام مناسب به اطلاع درخواست دهنده میرسد.

```go
func handle_subscribe_to_queue_request(request brokerclient.SendMessageRequest, connection net.Conn) {
    fmt.Printf(
        "subscribe request from client with address %s:%s for queue '%s'\n",
        request.ClientData.Host,
        request.ClientData.Port,
        request.QueueName,
    )
    client := add_client(Client(request.ClientData))
    err := add_client_to_queue_recievers(client, request.QueueName)
    if err != nil {
        fmt.Println("Error: ", err)
        connection.Write([]byte("Error: " + err.Error()))
        return
    }
    connection.Write([]byte("successfully subscribed to queue: " + request.QueueName))
}
```

نوع درخواست بعدی اتصال به یک صف دو طرفه است. در این حالت نیز مانند قبلی اطلاعات کلاینت به همراه نام صفی که نیاز است به آن متصل شود از طرف پراسس به بروکر ارسال میشود. در اینجا علاوه بر اینکه چک میشود صف با چنین نامی وجود داشته باشد باید چک شود که تنها یک پراسس در هر طرف صف دو طرفه مشغول به گوش دادن باشد. بعد از چک کردن موارد بالا پراسس به یک طرف صف دو طرفه که هنوز پراسسی به آن اساین نشده است اساین میشود و از این پس میتواند در این صف پیام ارسال کند.

```go
func handle_connect_to_two_way_queue(request brokerclient.SendMessageRequest, connection net.Conn) {
    fmt.Printf(
        "connection request from client with address %s:%s for two way queue '%s'\n",
        request.ClientData.Host,
        request.ClientData.Port,
        request.QueueName,
    )
    client := add_client(Client(request.ClientData))
    err := add_client_to_two_way_queue(client, request.QueueName)
    if err != nil {
        fmt.Println("Error: ", err)
        connection.Write([]byte("Error: " + err.Error()))
        return
    }
    connection.Write([]byte("successfully connected to two way queue: " + request.QueueName))
}
```

نوع درخواست بعدی درخواست ارسال پیام هب صف یک طرفه است. در این نوع پیام صرفا نام صف و متن پیام ذکر میشود و نیازی نیست اطلاعات clientdata پر شود چراکه اطلاعات گیرندگان پیام در هر صف نگه‌داری میشود و بروکر با داشتن اسم صف میتواند تشخیص دهد که پیام را به چه پراسس‌هایی باید ارسال کند و نیازی به دانستن اطلاعات پابلیش کننده پیام ندارد. در زمان رسیدگی به این درخواست ابتدا وجود داشتن صف و پر نبودن آن چک میشود. سپس پیام در صف مورد نظر قرار میگیرد. حال برای ack کردن پیام از این روش استفاده میشود.

```go
const MESSAGE_STATE_SIZE int = 2048

var messages_state [MESSAGE_STATE_SIZE]int

// if messages_state[i] is equal to zero it means message with id i is acknowleged
// if it is equal to j it means j acknowlegement must come in order to insure all
// its recievers got the message
```

این ساختار برای بررسی درست رسیدن میسج‌ها استفاده میشود. در ابتدا تمام خانه‌های ارایه messages_state صفر است. زمانی که به یک مسیج آیدی تخصیص داده میشود اگر مقدار $messagestate[id]$ آن صفر باشد یعنی این مسیج اکنالج شده است. اگر مقدار آن بزرگتر از صفر باشد (مثلا برابر x باشد) یعنی x تا گیرندگانی که باید این مسیج را دریافت کنند هنوز دریافت نکرده‌اند. حال در زمان رسیدن یک مسیج جدید مقدار کوچکترین index به طوری که مقدار $messagestate[index]$ صفر است به عنوان ایدی مسیج انتخاب میشود. سپس مقدار این خانه از ارایه message_state به اندازه تعداد گیرندگان این مسیج اضافه میشود و در تابع send_message هر زمان گیرنده‌ای مسیج را دریافت کند و ack آن را بفرستد یکی از مقدار $messagestate[messagID]$ کم میشود و زمانی که این مقدار صفر شود یعنی تمام گیرندگان این پیام آن را دریافت کرده‌اند و به فرستنده اطلاع داده میشود که پیام acknowledge شده است.

تابع هندل کردن ارسال پیام به صورت زیر است

```go
func handle_send_message_request(request brokerclient.SendMessageRequest, connection net.Conn) {
    fmt.Printf("message: '%s' for queue: '%s'\n", string(request.Message), request.QueueName)
    queue := find_queue_by_name(request.QueueName)
    if queue == nil {
        fmt.Println("no such queue exists!")
        connection.Write([]byte("Error: no such queue exists!"))
        return
    }
    message_Id := make_an_unacknowleged_message(len(queue.recievers))
    if message_Id == -1 {
        fmt.Println("Maximum number of unacknowleged reached!")
        connection.Write([]byte("Error: Maximum number of unacknowleged reached!"))
        return
    }
    err := add_message_to_queue(queue, request.Message, message_Id)
    if err != nil {
        fmt.Println("Error: " + err.Error())
        connection.Write([]byte("Error: " + err.Error()))
        return
    }
    wait_for_message_to_be_acknowleged(message_Id)
    connection.Write([]byte("message '" + string(request.Message) + "' acknowleged!"))
}
```

هندل کردن ارسال sync یا async در خود بروکر هندل نمیشود و به جای آن در توابعی که برای ارتباط به بروکر در فایل brokerclient.go قرار دارد هندل میشود. اگر نوع ارسال sync باشد پراسس مورد نظر همانجا منتظر میماند تا پیام ack از طرف بروکر برایش ارسال شود اما اگر نوع ارسال async زمانی که پراسس مسیج را برای بروکر ارسال میکند اگر ارسال با موفقیت انجام شود پیام دریافت مسیج توسط بروکر رو به کاربر نشان میدهد سپس در یک ترد جدید منتظر ارسال ack آن میماند و زمانی که ack برسد از طریق یک چنل ان را به اطلاع پراسس میرساند.

```go
func (publisher *SocketPublisher) SendSyncMessage(message []byte, queue_name string) ([]byte, error) {
    connection, err := net.Dial(publisher.broker_type, publisher.broker_host+":"+publisher.broker_port)
    if err != nil {
        return nil, err
    }
    defer connection.Close()
    _, err = connection.Write(encode_send_message_request(message, queue_name))
    if err != nil {
        fmt.Println("failed to send the message", err)
        return nil, err
    }
    fmt.Printf("Sent: %s to queue: %v\n", string(message[:]), queue_name)
    buffer := make([]byte, 1024)
    mLen, err := connection.Read(buffer)
    if err != nil {
        fmt.Println("Error reading:", err.Error())
        return nil, err
    }
    return buffer[:mLen], nil
}
```

```go
func (publisher *SocketPublisher) SendAsyncMessage(message []byte, queue_name string) ([]byte, chan string, error) {
    connection, err := net.Dial(publisher.broker_type, publisher.broker_host+":"+publisher.broker_port)
    if err != nil {
        return nil, nil, err
    }
    _, err = connection.Write(encode_send_message_request(message, queue_name))
    if err != nil {
        fmt.Println("failed to send the message", err)
        return nil, nil, err
    }
    fmt.Printf("Sent: %s to queue: %v\n", string(message[:]), queue_name)
    reply_channel := make(chan string)
    go recieve_reply(connection, reply_channel)
    return []byte("Message recieved by the broker!"), reply_channel, nil
}
```

لازم به ذکر است ساختار sockerPublisher تنها اطلاعات مربوط به host و port بروکر را نگه داری میکند چراکه برای فرستادن مسیج نیازی به فرستادن اطلاعات فرستده پیام نیست.

این در صورتست که برای دریافت پیام یا ارسال پیام دوطرفه (از طریق صف دو طرفه) باید اطلاعات خود و اطلاعات بروکر را نگه‌داری کرد چراکه که برای ارسال پیام به بروکر به اطلاعات بروکر و برای شناساندن خود به بروکر به عنوان گیرنده پیام لازم است اطلاعات خود را همراه با پیام به بروکر ارسال کرد.

```go
type Consumer interface {
    ReceiveFromBrokerForEver(handler func(net.Conn))
    SubscribeToQueue(queue_name string) ([]byte, error)
    SubscribeToTwoWayQueue(queue_name string) ([]byte, error)
    SendToTwoWayQueue(queue_name string, message []byte) ([]byte, error)
}

type SockerConsumer struct {
    reciever_host string
    reciever_port string
    reciever_type string

    broker_host string
    broker_port string
    broker_type string
}

func InitConsumer(reciever_type string, reciever_host string, reciever_port string) Consumer {
    new_client := SockerConsumer{
        reciever_host: reciever_host,
        reciever_port: reciever_port,
        reciever_type: reciever_type,
        broker_host:   BROKER_HOST,
        broker_port:   BROKER_PORT,
        broker_type:   BROKER_TYPE,
    }
    return &new_client
}
```

```go
const (
    BROKER_HOST = "localhost"
    BROKER_PORT = "9999"
    BROKER_TYPE = "tcp"
)

type Publisher interface {
    SendSyncMessage(message []byte, queue_name string) ([]byte, error)
    SendAsyncMessage(message []byte, queue_name string) ([]byte, chan string, error)
}

type SocketPublisher struct {
    broker_host string
    broker_port string
    broker_type string
}

func InitPublisher() Publisher {
    return &SocketPublisher{
        broker_host: BROKER_HOST,
        broker_port: BROKER_PORT,
        broker_type: BROKER_TYPE,
    }
}
```

برای ارسال پیام به صف دو طرفه نیز مراحلی مشابه با صف یک طرفه انجام میشود فقط اینبار علاوه بر چک کردن جا داشتن صف و وجود صف با نام ذکر شده این موضوع نیز چک میشود که این پراسس حتما به صف کانکت شده باشد و در طرف دیگر صف پراسسی برای گوش کردن به صف وجود داشته باشد. در صورتی که شرایط محیا باشد مسیج به یکی از صف‌های یک طرفه موجود در صف دو طرفه اضافه میشود و منتظر ارسال به گیرنده میماند.

```go
func handle_send_message_to_two_way_queue_request(request brokerclient.SendMessageRequest, connection net.Conn) {
    fmt.Printf("message: '%s' for two way queue: '%s'\n", string(request.Message), request.QueueName)
    two_way_queue := find_two_way_queue_by_name(request.QueueName)
    if two_way_queue == nil {
        fmt.Println("no such queue exists!")
        connection.Write([]byte("Error: no such queue exists!"))
        return
    }
    queue, err := find_one_way_queue_to_send_message_in_two_way_queue(two_way_queue, Client(request.ClientData))
    if err != nil {
        fmt.Println("Error: ", err)
        connection.Write([]byte("Error: " + err.Error()))
        return
    }
    message_Id := make_an_unacknowleged_message(len(queue.recievers))
    if message_Id == -1 {
        fmt.Println("Maximum number of unacknowleged reached!")
        connection.Write([]byte("Error: Maximum number of unacknowleged reached!"))
        return
    }
    err = add_message_to_queue(queue, request.Message, message_Id)
    if err != nil {
        fmt.Println("Error: " + err.Error())
        connection.Write([]byte("Error: " + err.Error()))
        return
    }
    wait_for_message_to_be_acknowleged(message_Id)
    connection.Write([]byte("message '" + string(request.Message) + "' acknowleged!"))
}
```

برای دریافت پیام از بروکر لازم است ابتدا به صف‌های مورد نظر سابسکرایب کرد سپس با استفاده از تابع RecieveFromBrokerForEver در یک حلقه بی‌نهایت به کنکشن‌های بروکر گوش میدهد و در صورت رسیدن یک پیام جدید آن را به یک تابع handler میدهد.

سعی شده است تا حد ممکن با است از لاگ گذاری مناسب روند اجرای پروژه مشخص و قابل درک باشد.

نحوه تست و ران کردن:

در کدهای کلاینت و سرور ۵ سناریو متناسب با سوالات موجود در صورت پروژه وجود دارد. برای اجرای هرکدام ابتدا در دایرکتوری بروکر کامند . go run باید اجرا شود. سپس این کامند باید ابتدا در دایرکتوری client و سپس در دایرکتوری server اجرا شود و حال که هر ۳ پراسس مورد نظر در حال اجرا هستند میتوان یکی از سناریوها را هم در کلاینت و هم در سرور اجرا و تست کرد.

توضیح سناریوها:

در سناریو اول در کلاینت یک پراسس ساده داریم که ابتدا به صف owqueue سابسکرایب میکند و سپس به بروکر گوش میدهد. در سرور هم یه حلقه بینهایت داریم که منتظر ورودی کاربر میماند تکست وارد شده را در صف owqueue به صورت sync پابلیش میکند و منتظر ack پیامش میشود.

در سناریو دوم مانند قبل یک کلاینت را به صف owqueue سابسکرایب میکنیم و سپس به بروکر گوش میدهیم. در سرور این بار هر پیام را که از کاربر دریافت میکنیم به صورت async میفرستیم. همانطور که میبینند ابتدا پیام message recieved by broker مشاهده میشود و مقداری بعد از آن پیام ack می‌آید.

(اگر مسیج wait n seconds به کلاینت ارسال شود. کلاینت مقدار n ثانیه منتظر میماند و سپس پیام ack را ارسال میکند که این ترفند در تست این ویژگی استفاده میشود) خروجی مشاهده شده در سرور به این صورت خواهد بود:

```
❯ go run .
select on of these senarios:
1. send sync message from server to client.
2. send async message from server to client.
3. send sync message from server to client into a queue with zero capacity to check queue overflow
4. send and recieive message via a two way queue.
5. initiate 3 clients and subscribe them all to same queue and see how when server publishes a message
all of them will get the message.
2
Ready to send message to the broker...
Enter a message to send to the broker...
wait 10 seconds
Sent: wait 10 seconds to queue: owqueue
Message recieved by the broker!
message 'wait 10 seconds' acknowleged!
```

در سناریو سوم مانند سناریو اول عمل میکنیم فقط این بار پیام را به یک صف با ظرفیت صفر میفرستیم تا مکانیز کنترل queue overflow را مشاهده کنیم. لاگ سرور به صورت زیر خواهد بود.

```
❯ go run .
select on of these senarios:
1. send sync message from server to client.
2. send async message from server to client.
3. send sync message from server to client into a queue with zero capacity to check queue overflow
4. send and recieive message via a two way queue.
5. initiate 3 clients and subscribe them all to same queue and see how when server publishes a message
all of them will get the message.
3
Ready to send message to the broker...
Enter a message to send to the broker...
test message
Sent: test message to queue: zerocapacityqueue
Error: maximum capacity of queue reached
```

در سناریو چهارم میخواهیم یک صف دو طرفه ایجاد کنیم. بنابراین هم در کلاینت و هم در سرور یک کد مشابه را ران میکنیم. در هر دوی آنها ابتدا به صف twqueue اتصال برقرار میکنیم. سپس در یک coroutine با استفاده از دستور

RecieveFromBrokerForEver به پیام‌های بروکر گوش میدهیم و همچنین منتظر ورودی از کاربر میشویم تا تکست وارد شده را به صر دیگر صف دو طرفه ارسال کنیم.

لاگ هر یک از پراسس‌های دو سر صف به این صورت خواهد شد:

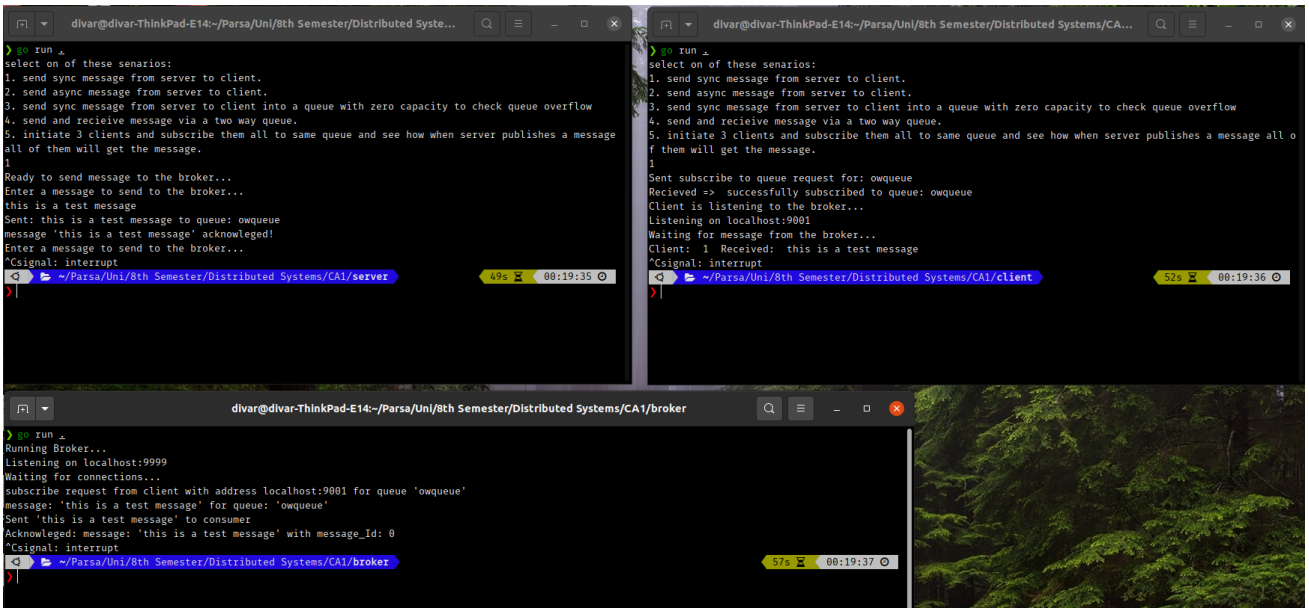
```
❯ go run .
select on of these senarios:
1. send sync message from server to client.
2. send async message from server to client.
3. send sync message from server to client into a queue with zero capacity to check queue overflow
4. send and recieive message via a two way queue.
5. initiate 3 clients and subscribe them all to same queue and see how when server publishes a message
all of them will get the message.
4
Sent subscribe to two way queue request for: twqueue
Received:  successfully connected to two way queue: twqueue
Client  1 is listening to the broker...
Listening on localhost:9005
Waiting for message from the broker...
Enter a text to send to the other side of queue or enter 'exit' to quit the program!!
message form process 1
Sent: message form process 1 to two way queue: twqueue
message 'message form process 1' acknowleged!
Enter a text to send to the other side of queue or enter 'exit' to quit the program!!
Client:  1  Received:  message from process 2
```

```
❯ go run .
select on of these senarios:
1. send sync message from server to client.
2. send async message from server to client.
3. send sync message from server to client into a queue with zero capacity to check queue overflow
4. send and recieive message via a two way queue.
5. initiate 3 clients and subscribe them all to same queue and see how when server publishes a message all o
f them will get the message.
4
Sent subscribe to two way queue request for: twqueue
Received:  successfully connected to two way queue: twqueue
Client  1 is listening to the broker...
Listening on localhost:9001
Waiting for message from the broker...
Enter a text to send to the other side of queue or enter 'exit' to quit the program!!
Client:  1  Received:  message form process 1
message from process 2
Sent: message from process 2 to two way queue: twqueue
message 'message from process 2' acknowleged!
Enter a text to send to the other side of queue or enter 'exit' to quit the program!!
```

در سناریو پنجم میخواهیم به صورت sync به چندکلاینت که همه آنها به سک صف سابسکرتیب کرده‌اند مسیج بفرستیم. برای این کار در سرور مانند قبل یک تکست از کاربر میگیریم و در صف owqueue پابلیش میکنیم. اما در کلاینت با استفاده از coroutine به تعداد ۳ کلاینت در پورت های مختلف اجرا میکنیم و هر ۳ آنها را به صف owqueue سابسکرایب میکنیم و سپس در هر ۳ آنها به بروکر گوش میدهیم. لاگ کلاینت در این سناریو مانند زیر خواهد شد:

```
3. send sync message from server to client into a queue with zero capacity to check queue overflow
4. send and recieive message via a two way queue.
5. initiate 3 clients and subscribe them all to same queue and see how when server publishes a message all o
f them will get the message.
5
Sent subscribe to queue request for: owqueue
Sent subscribe to queue request for: owqueue
Sent subscribe to queue request for: owqueue
Client  3  =>  successfully subscribed to queue: owqueue
Client  3 is listening to the broker...
Client  1  =>  successfully subscribed to queue: owqueue
Client  1 is listening to the broker...
Listening on localhost:9003
Waiting for message from the broker...
Client  2  =>  successfully subscribed to queue: owqueue
Listening on localhost:9001
Waiting for message from the broker...
Client  2 is listening to the broker...
Listening on localhost:9002
Waiting for message from the broker...
Client:  3  Received:  message from server
Client:  1  Received:  message from server
Client:  2  Received:  message from server
```

یک نمونه از اجرای همزمان هر ۳ پراسس و اجرای سناریو ۱:



علت استفاده از message queue وجود shared memory:

ساختار shared memory ساختار بسیار ساده و سریع برای انتقال اطلاعات است ولی هیچ مکانیزم قفل گذاری برای جلوگیری از دسترسی همزمان به صورت پیشفرض ندارد و این کار توسعه دهنده از این سیستم را سخت میکند چراکه

باید نگران دسترسی همزمان و اصالت داده‌های موجود در shared memory باشد. در مقابل این روش استفاده از messag queue با اینکه اندکی کندتر است ولی مکانیز ساده‌تری برای ارتباط بین پراسس‌ها ارائه میدهد و کار برنامه نویس را راحت‌تر میکند چرا که دیگر نیازی نیست نگران قفل گذاری روی داده‌ها یا دسترسی همزمان باشد چراکه مطمئن است یا مسیجی توسط یک پراسس دریافت نمیشود یا اگر دریافت میشود بدون مشکل و به صورت کامل دریافت خواهد شد بنابراین از مزیت‌های بزرگ message queue ارائه api ساده و پوشاندن جزئیات است. همچنین استفاده از shared memory برای کارهایی که در آن تعداد زیادی پراسس لازم است به یک حافظه مشترک دسترسی پیدا کنند و باید از قفل گذاری استفاده کنند میتواند سربار هزینه زیادی به همراه داشته باشد چراکه زمانی که یک پراسس بخش مشترک حافظه را قفل کند پراسس‌های دیگر بلاک میشوند.