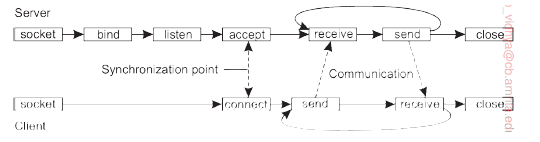
**Message oriented communication and sockets**

Message oriented communication is an alternative to remote procedure calls and remote invocations. Message oriented communication is used in the cases like when it cannot be assumed that the receiving side is executing at the time request is issued and the nature in RPC i.e., a client is blocked until its request been processed may need to replaced.

* Many distributed systems and applications are built directly on top of the simple message oriented-model offered by the transport layer i.e., transport level sockets.
* The communication using sockets is done by using the following model.



**Application:**

The idea of my application is to transport the messages from one client to another client using sockets. It starts with client 1 sending data to server and the server store the messages in a buffer and when the client 2 connects with the server it sends the messages stored in the buffer making it a persistent type of communication. The application is implemented using golang and made use of goroutines as much as possible. The code for server.go, client1.go and client2.go is given below:

**server.go:**

package main

import (

    "bufio"

    "fmt"

    "log"

    "net"

    "strings"

)

//local buffer

var messages []string

func main() {

    //lister object and error handling

    l, err := net.Listen("tcp", ":1200")

    if err != nil {

        log.Fatal(err)

    }

    //closing the listener at the end

    defer l.Close()

    //Infinite loop to handle multiple clients at once

    for {

        //connection object and error handling

        conn, err := l.Accept()

        if err != nil {

            log.Fatal(err)

        }

        //goroutine for each client to increase concureent excecution

        go func(c net.Conn) {

            //reading the 1st message from the connected client

            message, err := bufio.NewReader(conn).ReadString('\n')

            if err != nil {

                log.Printf("Error: %+v", err.Error())

                return

            }

            //casting the received message into string and removing any trailing spaces

            message = string(message)

            message = strings.TrimSpace(message)

            fmt.Fprintf(conn, "hey...\n")

            //log.Println("Message:", message, len(message))

            //checking if the connected cleint wants to send data or receive data

            if message == "sender" {

                //receiving messages from the sedning client until it stopss

                for {

                    message, err := bufio.NewReader(conn).ReadString('\n')

                    if err != nil {

                        log.Printf("Error: %+v", err.Error())

                        return

                    }

                    message = string(message)

                    message = strings.TrimSpace(message)

                    //if the client wants to stop sending messages then it sends "exit" as a message

                    if message != "exit" {

                        messages = append(messages, message)

                        fmt.Fprintf(conn, "hey...\n")

                        log.Println("Message:", message, len(message))

                    } else {

                        fmt.Fprintf(conn, "exit")

                        break

                    }

                }

            } else if message == "receiver" { //if the connected client wants receive the data then it sends "receiver" as the 1st message

                //fmt.Println(messages)

                //sending the buffered messages to the client

                for \_, val := range messages {

                    fmt.Println(val)

                    fmt.Fprintf(conn, val+"\n")

                }

                fmt.Fprintf(conn, "exit")

            }

        }(conn)

    }

}

**client.go:**

package main

import (

    "bufio"

    "fmt"

    "net"

    "os"

)

func main() {

    // if len(os.Args) != 2 {

    //  fmt.Fprintf(os.Stderr, "Usage: %s host:port ", os.Args[0])

    //  os.Exit(1)

    // }

    // service := os.Args[1]

    // resolving the tcp address and error handling

    tcpAddr, err := net.ResolveTCPAddr("tcp4", ":1200")

    checkError(err)

    // connecting to the tcp server

    conn, err := net.DialTCP("tcp", nil, tcpAddr)

    checkError(err)

    // reader := bufio.NewReader(os.Stdin)

    // For sending as many messages as client wishes

    for {

        reader := bufio.NewReader(os.Stdin)

        // sedning a message to indicate that the client wants to send meesages

        fmt.Print("Text to send: ")

        text, \_ := reader.ReadString('\n')

        // send messages to the server

        fmt.Fprintf(conn, text+"\n")

        // wait for reply

        message, \_ := bufio.NewReader(conn).ReadString('\n')

        //result, \_ := ioutil.ReadAll(conn)

        // if the meesage sent is 'exit' then server stops and again sends 'exit' which stops the client

        if message == "exit" {

            break

        }

        fmt.Print("Message from server: " + message)

    }

    os.Exit(0)

}

// a small function to handle errors

func checkError(err error) {

    if err != nil {

        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())

        os.Exit(1)

    }

}

**client2.go:**

package main

import (

    "bufio"

    "fmt"

    "net"

    "os"

    "strings"

)

func main() {

    // resolving the TCP address and error handling

    tcpAddr, err := net.ResolveTCPAddr("tcp4", ":1200")

    checkError(err)

    // connecting to the server and error handling

    conn, err := net.DialTCP("tcp", nil, tcpAddr)

    checkError(err)

    // reader := bufio.NewReader(os.Stdin)

    reader := bufio.NewReader(os.Stdin)

    // sending a message to indicate that the client wants to receive

    fmt.Print("Text to send: ")

    text, \_ := reader.ReadString('\n')

    // send to server

    fmt.Fprintf(conn, text+"\n")

    // wait for reply

    // to continously receive messages from the server buffer

    for {

        message, \_ := bufio.NewReader(conn).ReadString('\n')

        message = string(message)

        message = strings.TrimSpace(message)

        fmt.Println("Message from server: " + message)

        if message == "exit" {

            break

        }

    }

    os.Exit(0)

}

// small function to handle the errors

func checkError(err error) {

    if err != nil {

        fmt.Fprintf(os.Stderr, "Fatal error: %s", err.Error())

        os.Exit(1)

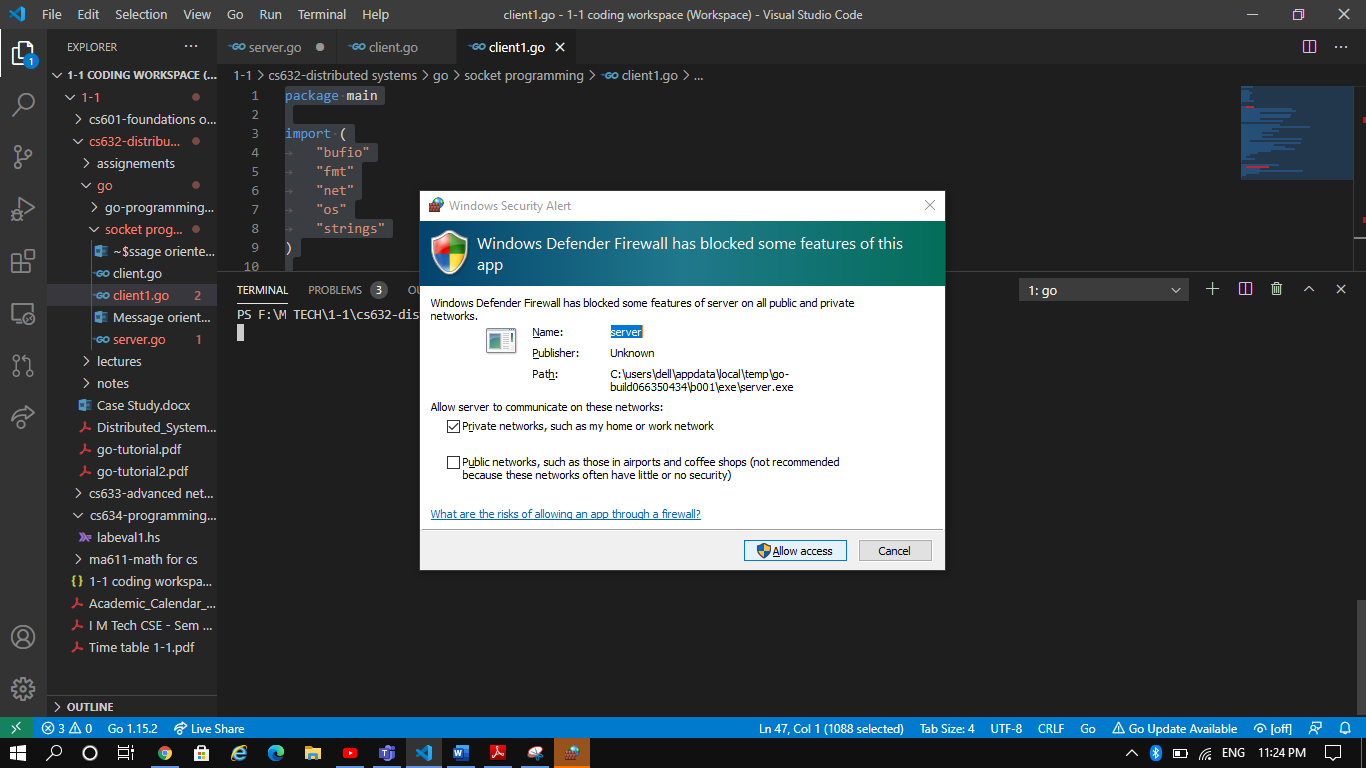
    }

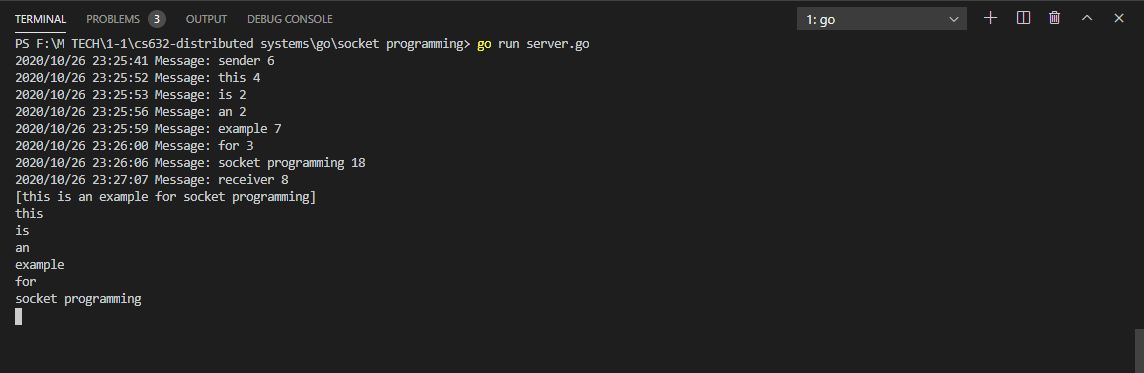
}

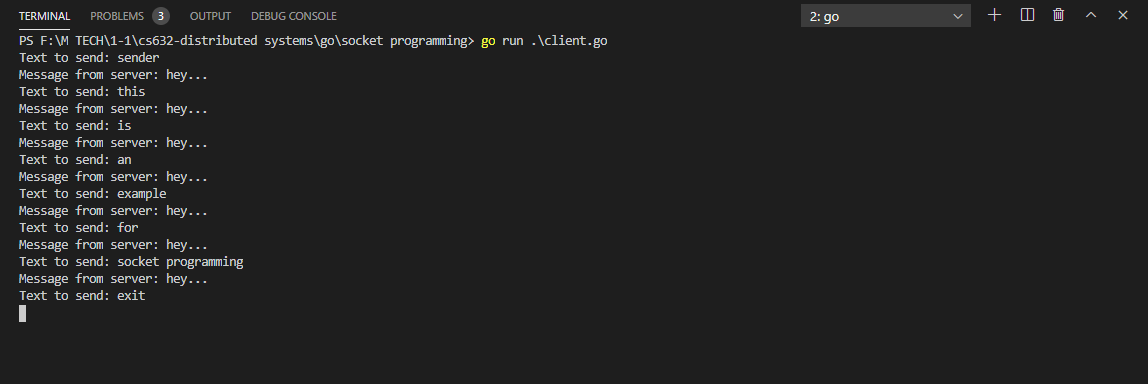
**Execution:**

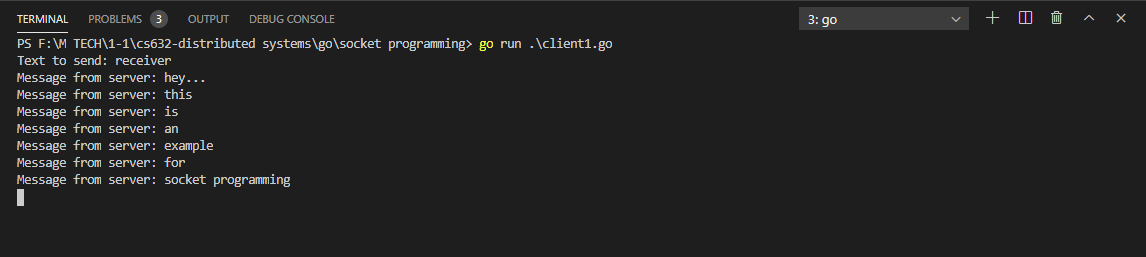
* The client that wants to send data will send “sender” as its first message, which indicates server to store the messages in the buffer
* The client that wants to receive the data will send “receiver” as its first message indicating the server to send the messages stored in the buffer
* Both clients and server are connected through the server’s socket which is “127.0.0.1:1200”
* The port number can be changed.
* At 1st the server should be executed, followed the by the client that wants to send the data
* In this example, the “client.go” is the sender program and hence it has to send the 1st message as “sender” and “client1.go” is the receiver program and it has to send the 1st message as “receiver”

**Output screenshots:**









**Advantages of message-oriented communication over RPC:**

* Clients are free to perform other operations while waiting for a response from the server
* Message Oriented Communication (MOC) allows many responses to one request or vice versa
* MOC is suitable for applications with long transaction lifetimes, such as workflow
* MOC supports priority by allowing retrieval of messages off the queue in any order
* MOC products support fault tolerance, persistent queues allow messages to be recovered when the system fails
* MOC products are designed to support legacy systems and widely dispersed systems.