James Grant
Fred Barnes
CO600: Engine Implementation

Engine Implementation

Document Contents

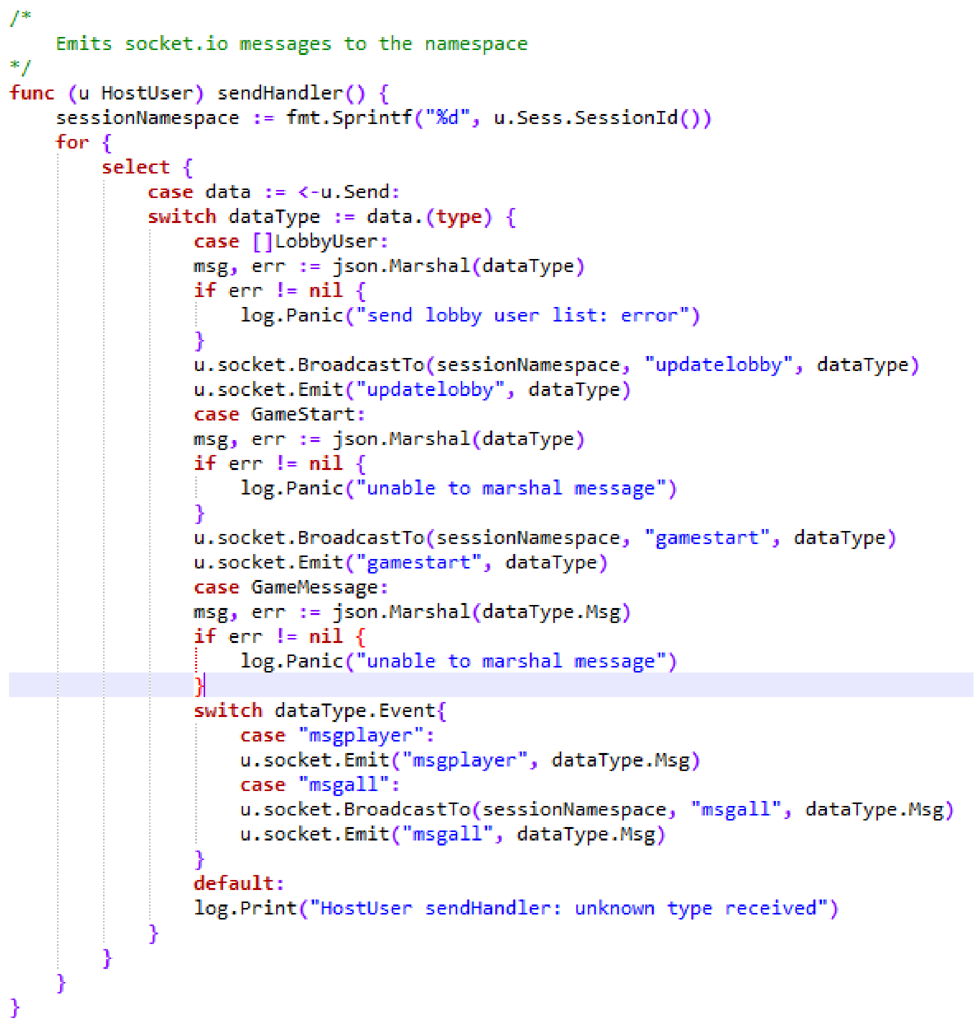
This document aims to describe and document the implementation process that took place in creating the ANEXD backend engine. It will also describe the programming techniques used, including architectural decisions such as data structure, analysis of the efficiency of code and justify why certain techniques were used.

1. Implementation 1

My first attempt at programming the backend engine was functionally comprehensive, however ultimately failed to meet requirements when it came to testing.

a. Architecture Summary

My first implementation used message channels for different processes such as individual users, the lobby itself and the application server. In this implementation however, I made the channels of interface{} type – Go's equivalent of a generic type that needs to be casted to the underlying data. Fortunately Go has a *type switch* statement, which allows a switch case like functionality to determine the underlying type. I originally used these type switch statements to check between all of the possible messages received, and had the same thread process the operations once it found the message type.



```
/*
    Emits socket.io messages to the namespace
*/
func (u HostUser) sendHandler() {
    sessionNamespace := fmt.Sprintf("%d", u.Sess.SessionId())
    for {
        select {
            case data := <-u.Send:
            switch dataType := data.(type) {
                case []LobbyUser:
                msg, err := json.Marshal(dataType)
                if err != nil {
                    log.Panic("send lobby user list: error")
                }
                u.socket.BroadcastTo(sessionNamespace, "updatelobby", dataType)
                u.socket.Emit("updatelobby", dataType)
                case GameStart:
                msg, err := json.Marshal(dataType)
                if err != nil {
                    log.Panic("unable to marshal message")
                }
                u.socket.BroadcastTo(sessionNamespace, "gamestart", dataType)
                u.socket.Emit("gamestart", dataType)
                case GameMessage:
                msg, err := json.Marshal(dataType.Msg)
                if err != nil {
                    log.Panic("unable to marshal message")
                }
                switch dataType.Event{
                    case "msgplayer":
                    u.socket.Emit("msgplayer", dataType.Msg)
                    case "msgall":
                    u.socket.BroadcastTo(sessionNamespace, "msgall", dataType.Msg)
                    u.socket.Emit("msgall", dataType.Msg)
                }
                default:
                log.Print("HostUser sendHandler: unknown type received")
            }
        }
    }
}
```

First of all this made the code difficult to maintain, while also requiring a type switch for every single message being sent and processed, which was unnecessary overhead and of O(n) time complexity over the total number of potential message types.

In addition to this, the lobby Manager structure contained two maps:

```go
type Manager struct {
    sync.RWMutex
    sessions map[int]*Session
    hosts map[int]*HostUser //map of active session hosts
}
```

One stored the lobby Session pointers, and the other pointed to the desktop Hosts with a Session currently open. Firstly, in this iteration the sessions map used an int type Key for the map, which has potential for inconsistency when the six-digit lobby id was below 100000, as some 0s will be removed from the beginning. It would be much more logical if this was done with a string.

Secondly, I was using an Interface type for User, and had a separate struct for HostUser and AnonUser.

Additionally, I was using two data structures to store users for different methods of instant access – a Map (userMap) for retrieval via a string username, and a Slice (PlayerMap) for the majority of access via player number (i.e player 1, 2, 3, etc).

```go
userMap map[string]*AnonUser //int map changed
PlayerMap []User
Send chan []byte
```

While it was possible to keep them both synchronized using mutual exclusion locks, this was a failure because the PlayerMap had to store both HostUser and AnonUser types, I made the slice data type the User interface. Unfortunately, I could not use a pointer type here, as you cannot point to a type that hasn't yet implemented the interface type.

```go
type User interface {
    SetSession(s *Session)
    SetSocket(sock socketio.Socket)
    Player() int
    SetPlayer(p int)
    Setup()
    sendHandler()
    receiveHandler()
}

type HostUser struct {
    userId int
    username string
    Sess *Session
    player int
    Send chan interface{}
    Receive chan interface{}
    socket socketio.Socket
    socketId string
}

type AnonUser struct {
    Nickname string
    Ready bool
    Sess *Session
    player int
    Send chan interface{}
    Receive chan interface{}
    socket socketio.Socket
    socketId string
}
```

This combined with my lack of experience with Go’s typing and pointers (slices are known as a *reference type*, which means that accessing them directly returns the original object), meant that while I believed the User stored within would be consistent when called upon, for AnonUser stored within the userMap (which used pointers), they were in fact a completely separate copy instance the instance that the pointers pointed to. This lead to disastrous inconsistencies in data stored and retrieved, depending on which data structure they were retrieved from.

Additionally, all of the functions that I wrote were written with a Non-Pointer method receiver, which means that whenever a function is called on a struct (i.e. the Session), it would actually create a copy of the instance and run the function on the copy. While this is fine for functions where you are just reading data from a struct instance, it meant that all modifications to the data fields within would be lost when the function returned.

b. Reasons for Failure

i. Inconsistencies

As seen from several of the explanations above, some of the functionality was aiming to cater to inconsistent external input (i.e. accessing users via int or string input). This was due to poor communication with other components of the ANEXD platform, where different parts of the system required inconsistent functionality. This could have been solved if a more collaborative approach was taken in planning the system. Additionally, the mistakes I made with pointers lead to inconsistently stored data.

ii. Programming Habits

Additionally, my experience in concurrent programming was primarily with the use of threads in Java. For this reason, despite using channels in Go, I was only running two threads for each entity within the system (one for incoming, and one for outgoing channels). While this sounds like a reasonable level of concurrency, it does not fully utilise the potential of Go’s concurrency with lightweight goroutine threads.

Upon further knowledge, I learned that these can be run into the millions concurrently, theoretically without causing any additional overhead as the language concurrency is extremely scalable.

Additionally, my preferences for strongly typed languages lead to inconsistencies when processing JSON data, which is decoded into Go's data types using the encoding/JSON library provided by the language. Of course, JavaScript is not strongly typed, so some minor issues arose when passing data between the two, such as expecting an int type when receiving a Number type from JavaScript, but the JSON specification defining it as a float64 – something I took into account when designing my data structures during the second implementation.

iii. Initial lack of Experience with Go

As I had just learned Go for this project, the ANEXD engine was in fact my first major program developed in the language after learning it. For this reason, I was not aware of many little intricacies in the language specification. While developing this iteration, the further I progressed and learned more, it lead to me backtracking and changing things that I had implemented earlier to improve things, however this often caused other dependent functions to develop some unforeseen quirks in functionality.

However, once I decided to do a full overhaul of the engine when the aforementioned failures occurred, I had learned a considerable amount about the language in the process, and so I had a much clearer plan on how to implement it the second time around. Ultimately this experience lead to a far better implementation.

c. Aims for New Implementation

For the new implementation, the main points I wanted to achieve were:

- Consistent use of Pointer method receivers when calling more persistent data structures such as Users, Lobbies, Socket.IO connections, etc.
- A single data structure for each entity – other components of the ANEXD platform will have to adapt slightly to fit the requirements of the engine rather than compromising or over-complicating the engine.
- The use of interface implementation for messages sent between channels, all of which will be executed in a new goroutine.
- An extremely modular implementation that is easier to maintain and is more decoupled from other functionality.
- Efficient and *constant* O(1) time complexity where possible (especially where real-time functionality is expected) to ensure high scalability when under heavy load.

2. Implementation 2: Total Overhaul and Final Version

This section documents the notable techniques that I used in implementing the final version to ensure that it exceeds the requirements.

a. Architecture

i. Data Structures

For the Manager struct, I only used one Map data structure, storing pointers to Lobby instances running:

```go
type Manager struct {
    sync.RWMutex
    lobbies map[string]*Lobby
}
```

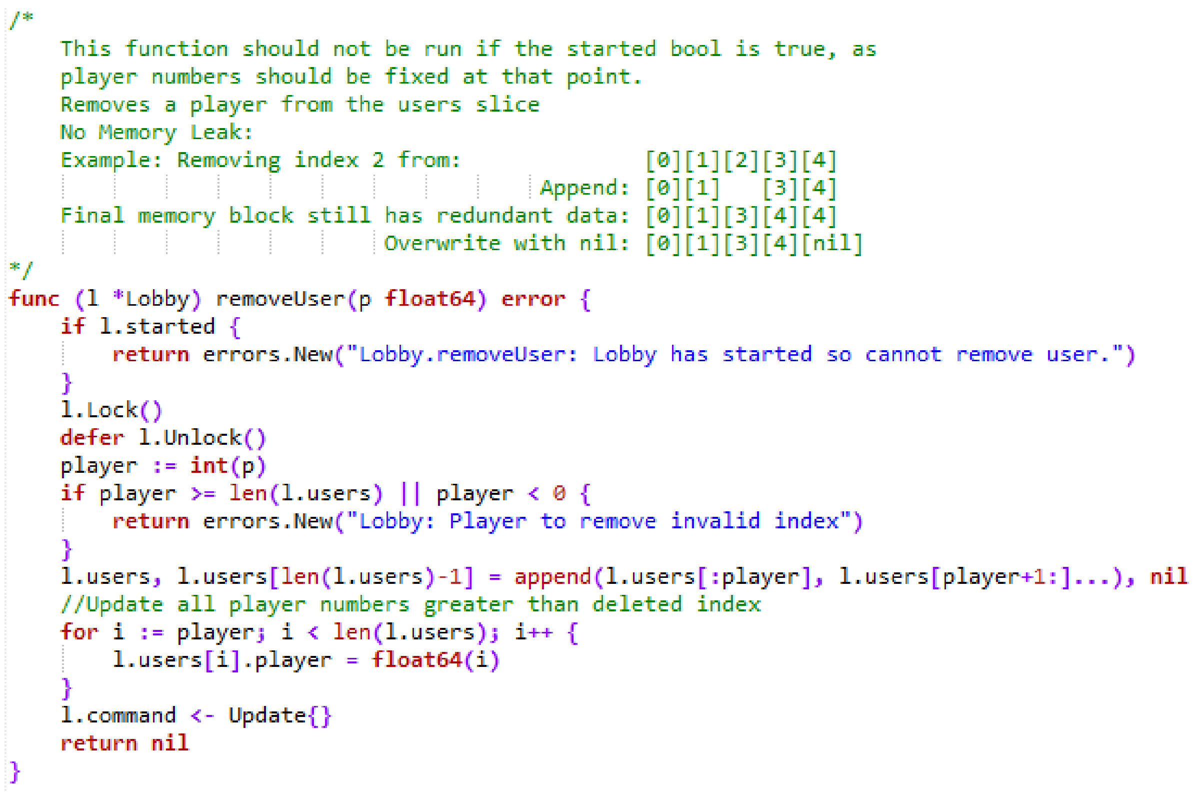
This means that the frontend now requires a lobby ID to access a running Lobby instance (the use of Host ID is also not possible), and additionally it is now of string type – meaning that other components storing this value as a numeric type will have to perform a type conversion first to fit the requirements.

As shown, Manager inherits the Go sync.RWMutex type, which allows the use of Locks to block concurrent access to the data structure, ensuring that data is consistent.

```go
type Lobby struct {
    sync.RWMutex
    lobbyId string
    game Game
    started bool
    users []*User
    send chan MessageServer
    command chan Command
    quit chan int
    timeout chan bool
    tcpConn *net.TCPConn
    socket *socketio.Socket
}
```

```go
type User struct {
    player float64 //json decodes to float64
    username string
    ready bool
    lobby *Lobby
    send chan MessageUser
    quit chan bool
    socket *socketio.Socket
}
```

The Lobby instances now just use a single Slice data structure, storing pointers to a User type – which is now a single struct, meaning HostUser and AnonUser from the previous implementation are now just instances of the same data structure.



```
/*
    This function should not be run if the started bool is true, as
    player numbers should be fixed at that point.
    Removes a player from the users slice
    No Memory Leak:
    Example: Removing index 2 from:              [0][1][2][3][4]
                                        Append: [0][1]   [3][4]
    Final memory block still has redundant data: [0][1][3][4][4]
                           Overwrite with nil: [0][1][3][4][nil]
*/
func (l *Lobby) removeUser(p float64) error {
    if l.started {
        return errors.New("Lobby.removeUser: Lobby has started so cannot remove user.")
    }
    l.Lock()
    defer l.Unlock()
    player := int(p)
    if player >= len(l.users) || player < 0 {
        return errors.New("Lobby: Player to remove invalid index")
    }
    l.users, l.users[len(l.users)-1] = append(l.users[:player], l.users[player+1:]...), nil
    //Update all player numbers greater than deleted index
    for i := player; i < len(l.users); i++ {
        l.users[i].player = float64(i)
    }
    l.command <- Update{}
    return nil
}
```

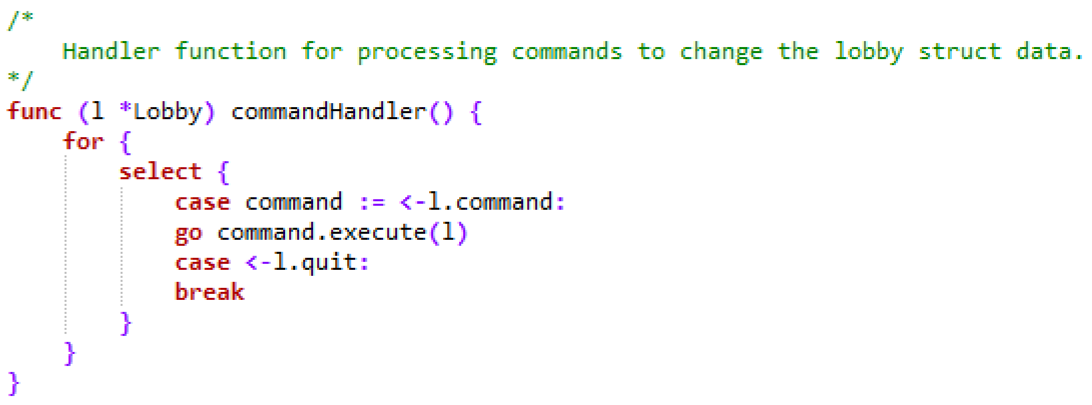
The above example of manipulating the slice data structure shows that extra effort was taken in ensuring that are no memory leaks in the underlying array storage.

ii. Interface Type Message Channels (CSP Concurrency)

In addition to this, it should be noted that the *send* and *command* channel types are no longer of the generic interface{} type, but now have MessageServer and Command respectively, two new interface types that I have defined. The reasons and advantages for this are explained later.

I have also implemented a quit channel to prevent memory leaks for running goroutines when a Lobby/User is removed from the memory, these work by exiting any infinite loops running in separate threads such as channel listener functions.

There is also a global timeout channel, this is used to ensure that certain responses are received within a certain time limit to prevent certain goroutines from running infinitely when no response is received.

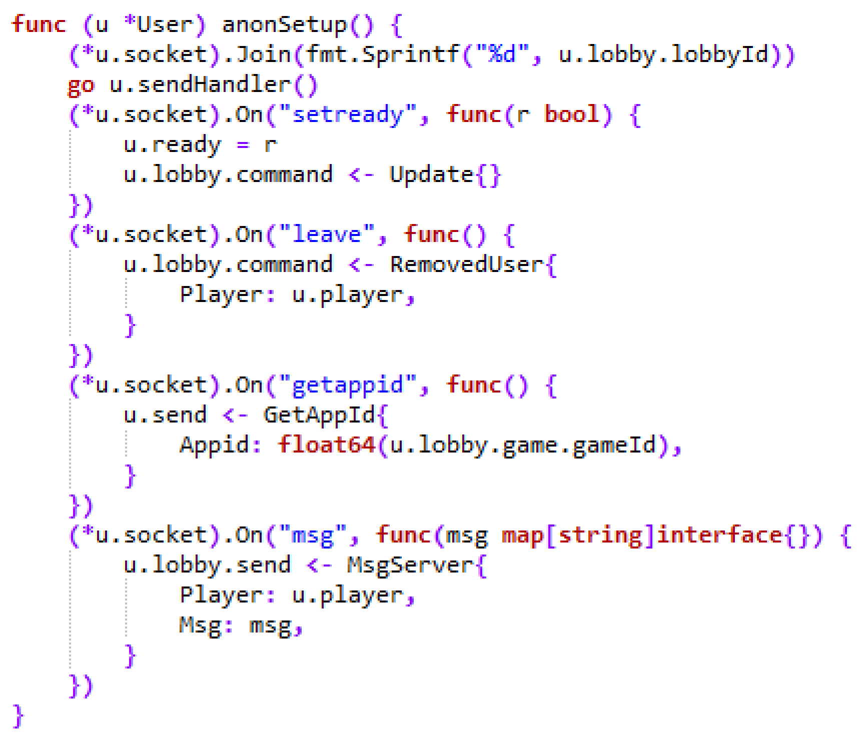


```go
/*
    Handler function for processing commands to change the lobby struct data.
*/
func (l *Lobby) commandHandler() {
    for {
        select {
            case command := <-l.command:
            go command.execute(l)
            case <-l.quit:
            break
        }
    }
}
```

As shown above, this single example of the many channel handler functions are now very much simplified, and rather than using a type switch statement to determine the functionality to execute, now use **run time polymorphism** to determine the underlying type's function. This implementation improves the time complexity from O(n) in my first implementation to O(1), which is ideal for the requirements. For this example, this means that any command that operates on the Lobby struct simply needs to implement the Command interface, making code far more modular and easy to maintain.

We can also see that the underlying command is executed in a new goroutine thread, this means that not only does the channel handler itself have its own thread, but the functions executed from within it are offloaded to a new thread, removing any processing overhead on the channel receiver thread. This allows all operations to be executed independently in real-time when received by a channel, and do not need to wait on a queue within the same thread. Once the new goroutine has finished executing it's underlying function, there are no more remaining pointers to the message struct that was sent to the channel, allowing it to be garbage collected by Go, meaning that this implementation cause any redundant memory usage.

Additionally, the Socket.IO library that I used for the project also uses its own thread to manage incoming events, meaning that the incoming data over WebSocket connections behaves with a very similar level of concurrency to my architecture.



```go
func (u *User) anonSetup() {
    (*u.socket).Join(fmt.Sprintf("%d", u.lobby.lobbyId))
    go u.sendHandler()
    (*u.socket).On("setready", func(r bool) {
        u.ready = r
        u.lobby.command <- Update{}
    })
    (*u.socket).On("leave", func() {
        u.lobby.command <- RemovedUser{
            Player: u.player,
        }
    })
    (*u.socket).On("getappid", func() {
        u.send <- GetAppId{
            Appid: float64(u.lobby.game.gameId),
        }
    })
    (*u.socket).On("msg", func(msg map[string]interface{}) {
        u.lobby.send <- MsgServer{
            Player: u.player,
            Msg: msg,
        }
    })
}
```

This function is an example of adding event listeners to the Socket.IO connection, which attaches a function to be executed when an event is received.

Additionally to ensure consistency this time around in defining socket events, I have always used a pointer to the original socket object, as shown above, the (*u.socket) function call now returns the original socket instance to attach the event to.

The next page shows a diagram on the CSP model used within the engine.

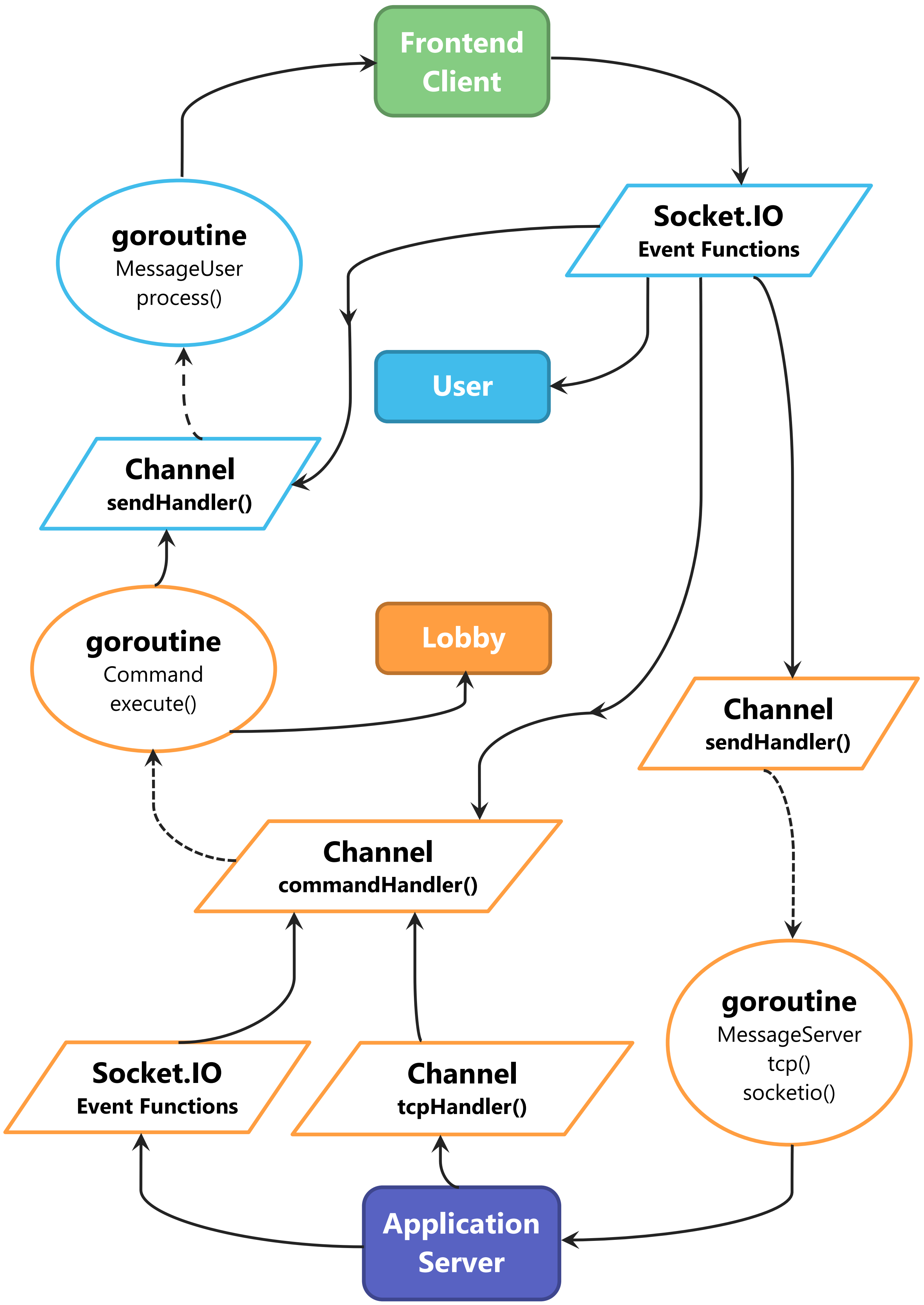
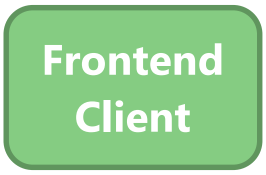
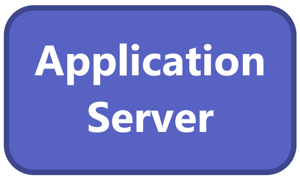

Frontend Client
Socket.IO Event Functions
goroutine MessageUser process()
User
Channel sendHandler()
goroutine Command execute()
Lobby
Channel sendHandler()
Channel commandHandler()
goroutine MessageServer tcp() socketio()
Socket.IO Event Functions
Channel tcpHandler()
Application Server

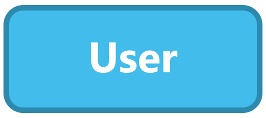
Diagram Key:

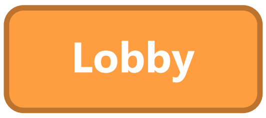
External Components:



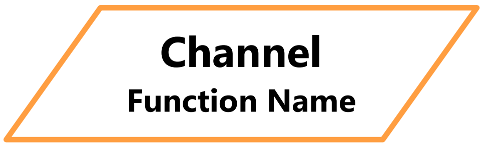


Data Structures:

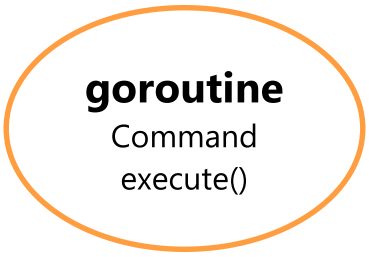




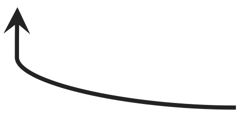
Channel Handlers:



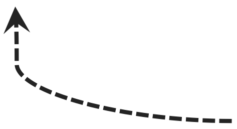
goroutines (temporary threads for executing commands):



Connector (single connection instance):



Connector (multiple connection instances):



Channel Message Interfaces

```go
type MessageServer interface {
    tcp(l *Lobby)
    socketio(l *Lobby)
}

type Command interface {
    execute(l *Lobby)
}

type MessageUser interface {
    process(u *User)
}
```

All messages communicated between channels must implement the respective interface of the channel receiving it. Implementing an interface is implicit in Go, and is done when a struct type has a method receiver function that matches the described functions of the interface.

An example of a MsgServer struct, which implements the tcp() and socketio() functions as defined in the MessageServer interface:

```go
type MsgServer struct {
    Event string                    `json:"event"`
    Player float64                  `json:"player"`
    Msg map[string]interface{}      `json:"msg"`
}

func (m MsgServer) tcp(l *Lobby) {
    jsonMsg, err := json.Marshal(m)
    if err != nil {
        log.Print(err)
        return
    }
    (*l.tcpConn).Write(jsonMsg)
}

func (m MsgServer) socketio(l *Lobby) {
    (*l.socket).Emit("in", m)
}
```

As shown, the struct has the JSON variables mapped to the data within, and using the JSON Marshal/Unmarshal functions in the encoding/JSON library within Go, the JSON data can be mapped between the two languages easily. This way, when a particular JSON Object is received from the frontend, it can be instantly mapped to a respective Go struct for processing.

Graceful termination of Channels

When a lobby ends and needs to be destroyed, a quit channel has been implemented to end all running goroutines, so the channels can be garbage collected:

```
func (u *User) terminate() {
    close(u.quit)
}
```

b. Time Complexity of Functions

One of the function requirements of the system was that the system must be highly scalable for handling sessions with potentially thousands of simultaneous users. This means that having efficient code is vital, and having an Order of constant time complexity is extremely favourable where possible.

This page looks into the time complexity of key functionality:

Process:	Efficiency Requirement	Order	Meets Requirements?
Manipulating and Accessing the Manager Data Structure	Linear O(n)	O(1)	Yes
Adding and Removing a User from the Lobby data structure	Linear O(n)	O(n)	Yes
Accessing a User from the Lobby data slice	Constant O(1)	O(1)	Yes
Updating the list of Users in the Lobby	Linear O(n)	O(n)	Yes
Processing messages received through channels	Constant O(1)	O(1)	Yes
Manipulating and Accessing User Data Structure	Constant O(1)	O(1)	Yes
Manipulating and Accessing Lobby Data Structure	Constant O(1)	O(1)	Yes

Conclusion of Implementation

Despite the failure of the first implementation, it taught me a lot of advanced level knowledge about the Go language, which I am much more confident in programming with now. In gaining more knowledge on the language, I feel like the final implementation surpasses the expectations that I originally had for the engine. For example, the level of concurrency it utilizes is far greater than I imagined, coming from languages such as Java.

As exhibited in this document, my final implementation uses many of Go's strengths to create a very efficient end product. I feel like my code meets most of my standards, including considerations for things like preventing memory leaks and garbage collection, clean, commented and efficient code, and a well thought out architecture.