به نام خدا

گزارش پروژه دوم

مبانی رایانش توزیع شده

دانشکده مهندسی برق و کامپیوتر

علی هدایی	۸۱۰۱۹۹۵۱۳
فاطمه محمدی	۸۱۰۱۹۹۴۸۹
علی عطاءاللهی	۸۱۰۱۹۹۴۶۱
محمد محجل صادقی	۸۱۰۱۹۹۴۸۳

فهرست مطالب

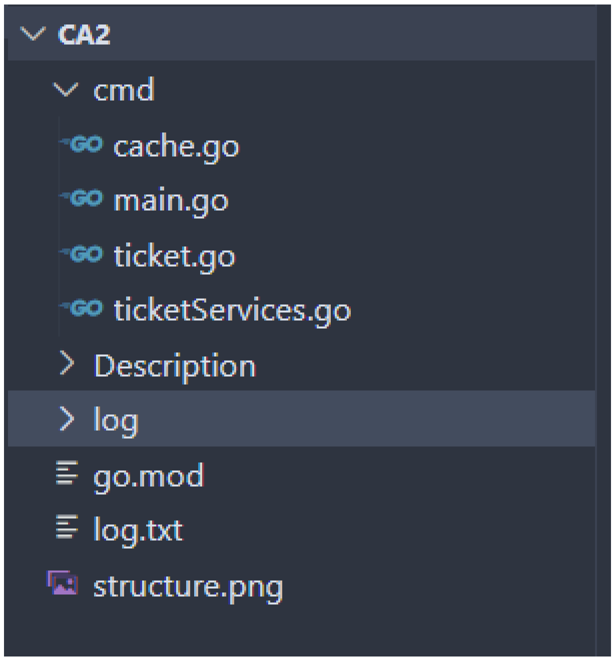
Concurrent Ticket Reservation System

Project Overview

In this project, we are to develop a concurrent ticket reservation system using the Go programming language. This system will allow clients to reserve tickets for an event concurrently while maintaining data consistency and fairness.

1. Project Setup:

- Create a new directory for our project and define the necessary data structures within this directory in main.go. Initialize the project with go mod init.



```
CA2
  cmd
    cache.go
    main.go
    ticket.go
    ticketServices.go
  Description
  log
  go.mod
  log.txt
  structure.png
```

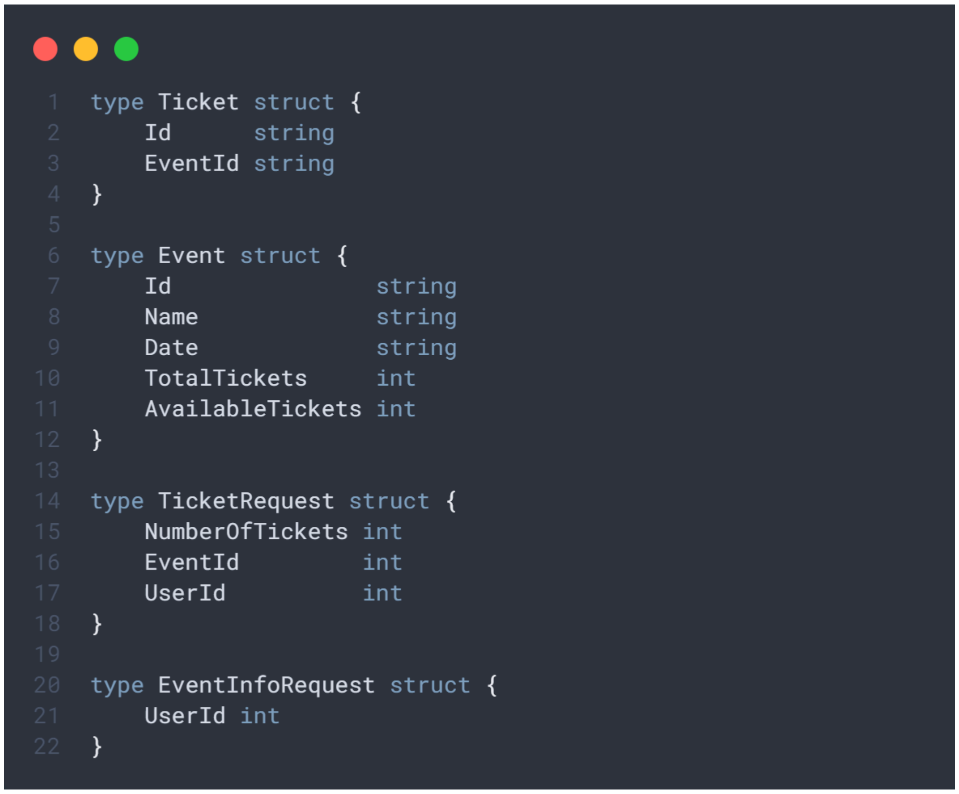
cmd: This directory contains the main application entry points or command-line tools for the project.

cache.go, **main.go**, **ticket.go**, and **ticketServices.go**: These are Go source files that contain the code for the project. Their specific purposes can be inferred from their names:

- `cache.go`: contains code related to caching functionality.
- `main.go`: This is typically the entry point of the application.
- `ticket.go`: contains code related to ticket management or a ticketing system.
- `ticketServices.go`: contains services or functionality related to the ticketing system.

Description: This is a text file or directory containing a description of the project.
log: This is either a directory for storing log files or a specific log file.
go.mod: This file is used for Go module dependency management and defines the module path and dependencies.
log.txt: This is a text file that contains log entries or output from the application.
structure.png: This is an image file, a diagram or visual representation of the project structure.

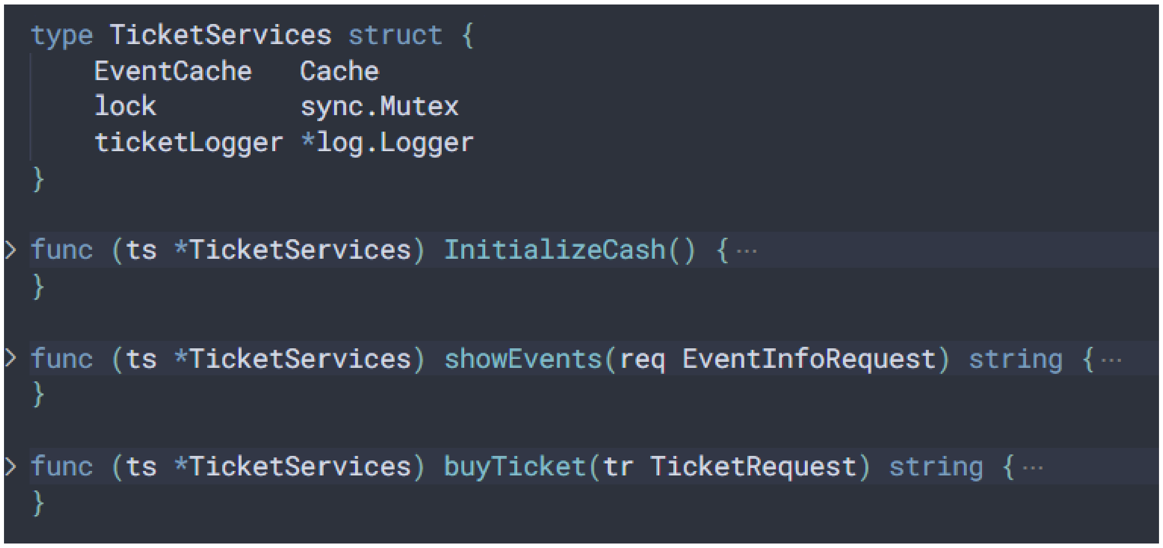
- Define structs for Event and Ticket.



```go
type Ticket struct {
    Id      string
    EventId string
}

type Event struct {
    Id               string
    Name             string
    Date             string
    TotalTickets     int
    AvailableTickets int
}

type TicketRequest struct {
    NumberOfTickets int
    EventId         int
    UserId          int
}

type EventInfoRequest struct {
    UserId int
}
```

2. Ticket Reservation Service Implementation:

- Implement the logic for managing events and ticket reservations using the TicketService struct. This method should define new events, display a list of available events, and manage ticket reservations for an event.



```go
type TicketServices struct {
    EventCache   Cache
    lock         sync.Mutex
    ticketLogger *log.Logger
}

func (ts *TicketServices) InitializeCash() {…
}

func (ts *TicketServices) showEvents(req EventInfoRequest) string {…
}

func (ts *TicketServices) buyTicket(tr TicketRequest) string {…
}
```

The `TicketServices` struct encapsulates the following components:

1. **EventCache**: A cache mechanism for storing event information.
2. **lock**: A mutex for synchronizing access to shared resources.
3. **ticketLogger**: A logger for recording ticket purchase activities.

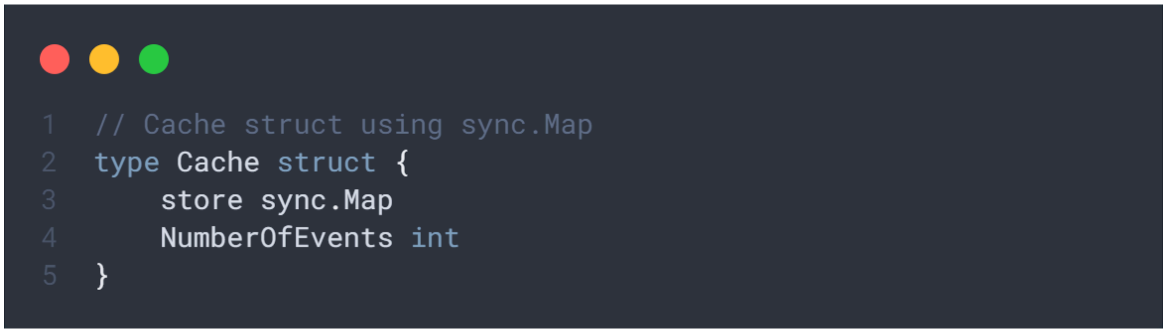
Functionality:

1. **InitializeCash():**
    - This method initializes the event cache.
    - It configures a logger to record ticket purchase activities.
    - The cache is prepared for storing event information.
2. **showEvents(req EventInfoRequest) string:**
    - This function retrieves event information from the cache.

- It iterates over the cached events and constructs a message containing event details.
- Access to the cache is synchronized using a mutex to prevent concurrent modifications.

3. **buyTicket(tr TicketRequest) string:**
   - This method facilitates the purchase of tickets for a specific event.
   - It retrieves the event from the cache based on the provided event ID.
   - If the event exists and has sufficient available tickets, it decrements the available ticket count and records the purchase activity.
   - Access to the cache and logger is synchronized using a mutex to ensure thread safety.

- Use the sync.Map data structure for concurrent storage of events and tickets.



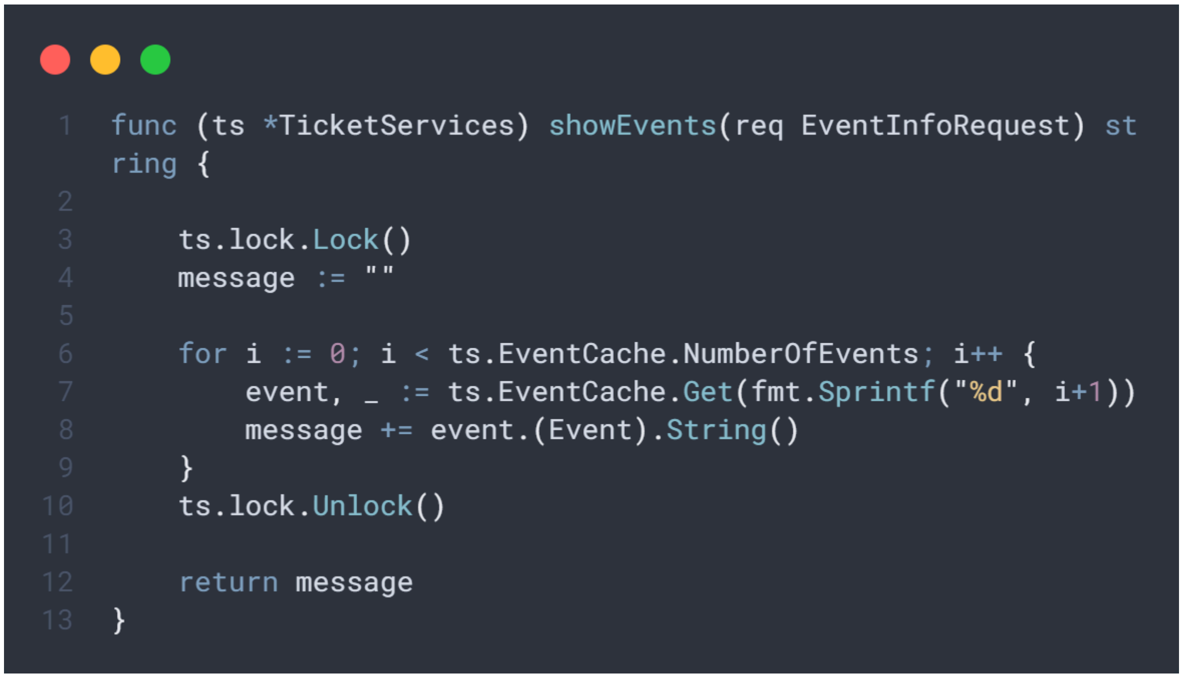
```go
// Cache struct using sync.Map
type Cache struct {
    store sync.Map
    NumberOfEvents int
}
```

The modification of the cache implementation involves substituting the conventional map structure with the `sync.Map` type provided by the `sync` package. This replacement ensures that concurrent operations on the cache are managed securely, mitigating the risk of data corruption or race conditions. Specifically, the methods responsible for setting, retrieving, and deleting entries in the cache are adapted to utilize the corresponding functionalities of `sync.Map`, enabling safe concurrent manipulation of cache contents.

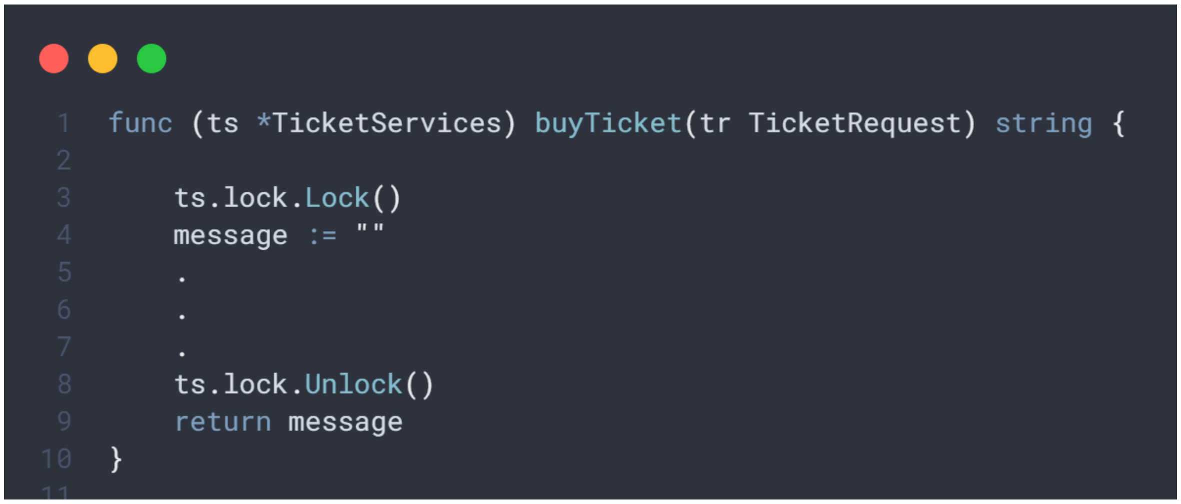
Additionally, sections of the code pertaining to concurrent cache access through goroutines and the main function are provided but commented out. These sections offer a framework for implementing concurrent cache access patterns, utilizing goroutines for parallel execution while ensuring thread safety. By integrating `sync.Map`, the cache implementation gains robustness, facilitating seamless concurrent access and modification of cache data across multiple execution threads without compromising data integrity.

3. Concurrency Control:

- Identify critical sections in your code that could lead to race conditions and data inconsistency.

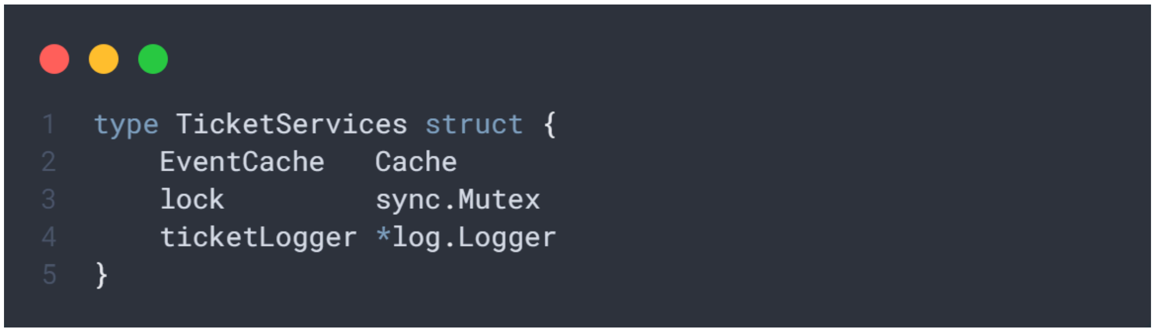


```go
func (ts *TicketServices) showEvents(req EventInfoRequest) string {

    ts.lock.Lock()
    message := ""

    for i := 0; i < ts.EventCache.NumberOfEvents; i++ {
        event, _ := ts.EventCache.Get(fmt.Sprintf("%d", i+1))
        message += event.(Event).String()
    }
    ts.lock.Unlock()

    return message
}
```

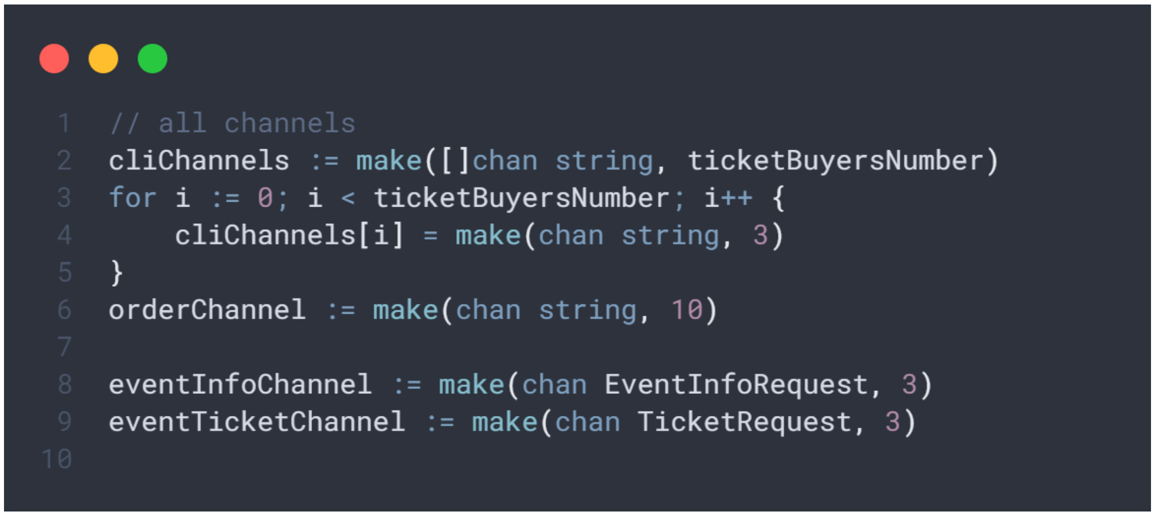


```go
func (ts *TicketServices) buyTicket(tr TicketRequest) string {

    ts.lock.Lock()
    message := ""
    .
    .
    .
    ts.lock.Unlock()
    return message
}
```

- Implement synchronization techniques using sync.Mutex, sync.RWMutex, or channels to ensure data consistency. Make sure to lock or modify ticket inventories before allowing reservations to prevent inconsistencies.



```go
type TicketServices struct {
    EventCache    Cache
    lock          sync.Mutex
    ticketLogger *log.Logger
}
```

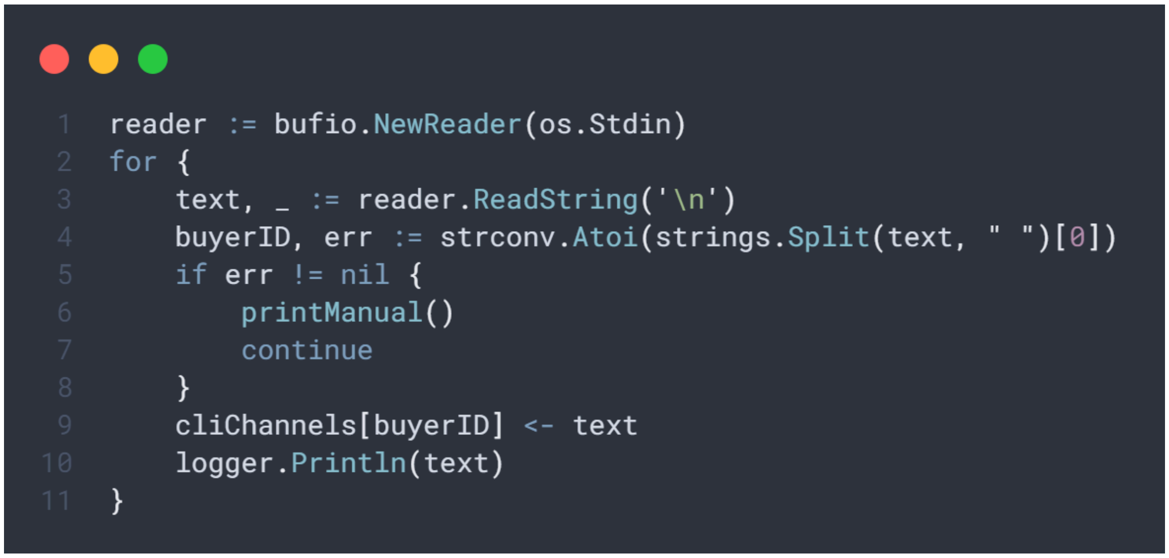


```go
// all channels
cliChannels := make([]chan string, ticketBuyersNumber)
for i := 0; i < ticketBuyersNumber; i++ {
    cliChannels[i] = make(chan string, 3)
}
orderChannel := make(chan string, 10)

eventInfoChannel := make(chan EventInfoRequest, 3)
eventTicketChannel := make(chan TicketRequest, 3)
```

Concurrency control is achieved here through the use of `sync.Mutex` for locking and channels for communication between goroutines. The `sync.Mutex` is employed to ensure exclusive access to shared resources, preventing data races. Channels facilitate communication between different parts of the program, allowing goroutines to coordinate their actions. Goroutines execute concurrently, processing tasks such as handling user input, managing ticket purchases, and distributing workload. Through this combination of locking and communication mechanisms, the program ensures safe and synchronized execution in a concurrent environment.

4. Client Interface:

- Develop a simple interface that allows users to interact with the ticket reservation system.



```go
reader := bufio.NewReader(os.Stdin)
for {
    text, _ := reader.ReadString('\n')
    buyerID, err := strconv.Atoi(strings.Split(text, " ")[0])
    if err != nil {
        printManual()
        continue
    }
    cliChannels[buyerID] <- text
    logger.Println(text)
}
```

- The interface serves as a conduit between users and the ticket reservation system, operating within a command-line environment.
- Users input commands via the command line, which are promptly captured by the interface for processing.
- **User Input Reading:** Utilizes Go's bufio package to continuously read input from the standard input (os.Stdin).
- **Command Parsing:** Parses user input to extract relevant information, primarily the buyer ID, by splitting the input string based on a space delimiter. Employs strconv.Atoi to convert the buyer ID portion into an integer.
- **Error Handling:** Handles potential conversion errors gracefully, prompting users to consult the manual for correct command formatting in case of failure.
- **Communication with Ticket Reservation System:** Sends user commands to the ticket reservation system for further processing using channels (cliChannels).

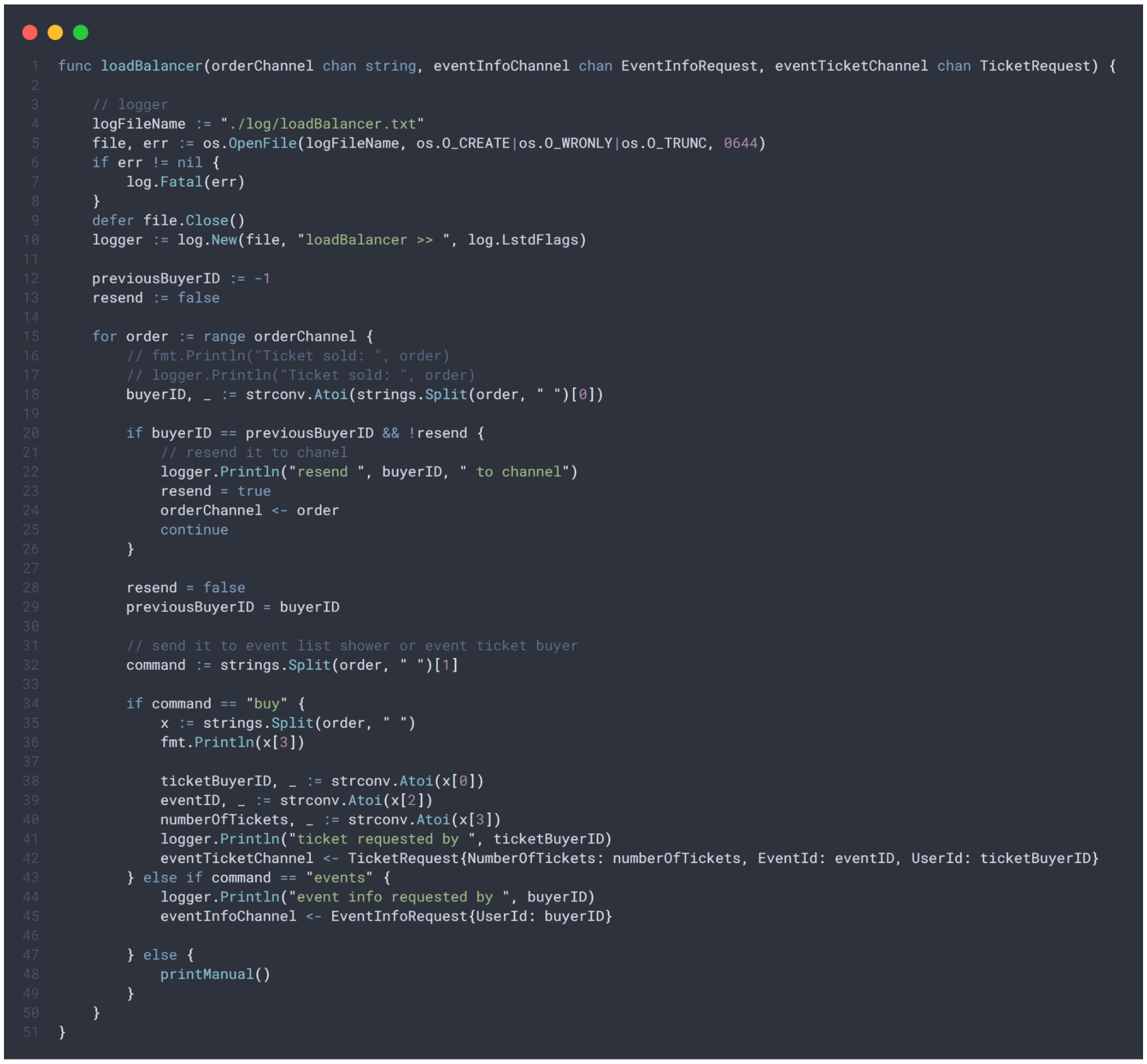
- **Logging:** Logs each user input for monitoring and debugging purposes, enhancing system transparency and aiding administrators in tracking user activity.

- Define functions necessary to display a list of available events and handle ticket reservations for an event.
  The explanation of this part was given in the second part.

- Use goroutines to handle multiple user requests simultaneously.
  The explanation of this part was given in the second part.

5. Resource Management and Fairness:

- Implement mechanisms in the ticket reservation system to ensure fairness and prevent starvation. Use methods like semaphores or the leaky bucket algorithm to limit the number of concurrent requests and prevent resource exhaustion.



```go
func loadBalancer(orderChannel chan string, eventInfoChannel chan EventInfoRequest, eventTicketChannel chan TicketRequest) {

    // logger
    logFileName := "./log/loadBalancer.txt"
    file, err := os.OpenFile(logFileName, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
    if err != nil {
        log.Fatal(err)
    }
    defer file.Close()
    logger := log.New(file, "loadBalancer >> ", log.LstdFlags)

    previousBuyerID := -1
    resend := false

    for order := range orderChannel {
        // fmt.Println("Ticket sold: ", order)
        // logger.Println("Ticket sold: ", order)
        buyerID, _ := strconv.Atoi(strings.Split(order, " ")[0])

        if buyerID == previousBuyerID && !resend {
            // resend it to chanel
            logger.Println("resend ", buyerID, " to channel")
            resend = true
            orderChannel <- order
            continue
        }

        resend = false
        previousBuyerID = buyerID

        // send it to event list shower or event ticket buyer
        command := strings.Split(order, " ")[1]

        if command == "buy" {
            x := strings.Split(order, " ")
            fmt.Println(x[3])

            ticketBuyerID, _ := strconv.Atoi(x[0])
            eventID, _ := strconv.Atoi(x[2])
            numberOfTickets, _ := strconv.Atoi(x[3])
            logger.Println("ticket requested by ", ticketBuyerID)
            eventTicketChannel <- TicketRequest{NumberOfTickets: numberOfTickets, EventId: eventID, UserId: ticketBuyerID}
        } else if command == "events" {
            logger.Println("event info requested by ", buyerID)
            eventInfoChannel <- EventInfoRequest{UserId: buyerID}

        } else {
            printManual()
        }
    }
}
```

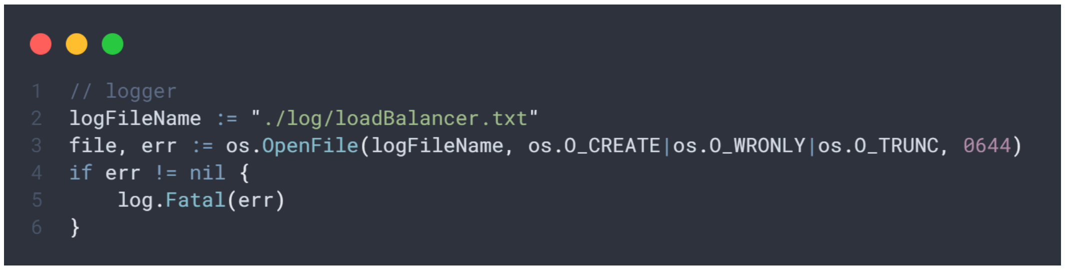
The primary goal of the load balancer is to distribute incoming requests among different components of the ticket reservation system effectively. It aims to prevent starvation, where certain requests might be continuously delayed or overlooked, ensuring fairness in processing.

- **Logging Mechanism:** The load balancer includes a logging mechanism to record important events and actions. This mechanism helps in tracking system behavior, debugging, and monitoring performance. The log file is created with the name "loadBalancer.txt" in the specified directory.
- **Request Processing:** Incoming requests are received via the `orderChannel`. Each request is processed sequentially within the loop.

- **Preventing Starvation:** To prevent starvation, the load balancer implements a resend mechanism. It detects if the current buyer ID matches the previous one and if the previous request was not resent. If conditions are met, the load balancer resends the request to the order channel, ensuring that requests from the same buyer are not consistently delayed.
- **Request Handling:** Requests are categorized based on the command specified. If the command is "buy," indicating a ticket purchase request, relevant details such as the buyer ID, event ID, and number of tickets are extracted and forwarded to the `eventTicketChannel` for processing by the ticket buyer component.
  If the command is "events," indicating a request for event information, the load balancer forwards the request to the `eventInfoChannel` for processing by the event list shower component.
  For any other command, the load balancer provides instructions to print the manual, ensuring proper usage of the system.

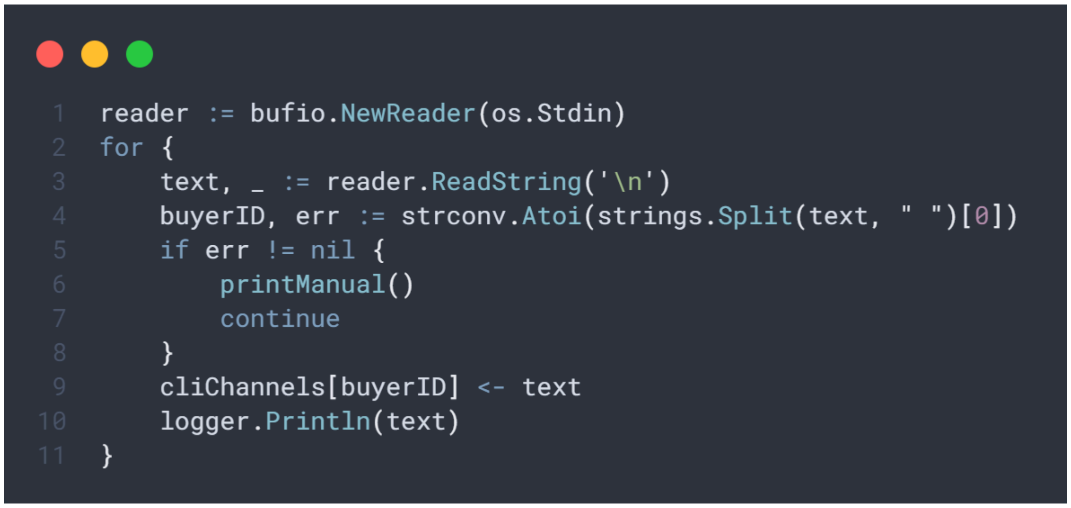
6. Error Handling and Logging:

- Use error handling mechanisms to manage potential errors during ticket reservations or user interactions.
  - **File Operations:** Error handling for opening log files in functions like `ticketBuyer`, `loadBalancer`, `eventInfoHandler`, and `eventTicketHandler`.



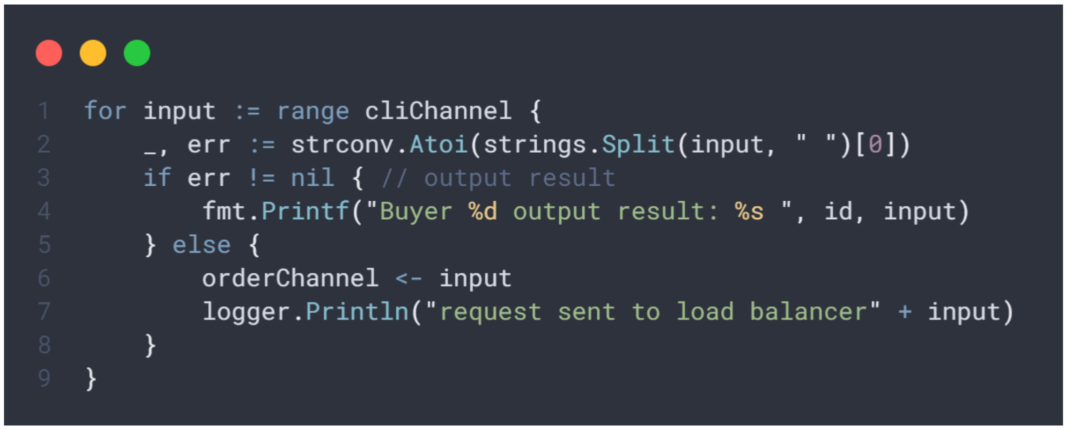
```go
// logger
logFileName := "./log/loadBalancer.txt"
file, err := os.OpenFile(logFileName, os.O_CREATE|os.O_WRONLY|os.O_TRUNC, 0644)
if err != nil {
    log.Fatal(err)
}
```

- **Parsing Input:** Error handling for parsing user input in the `main` function.



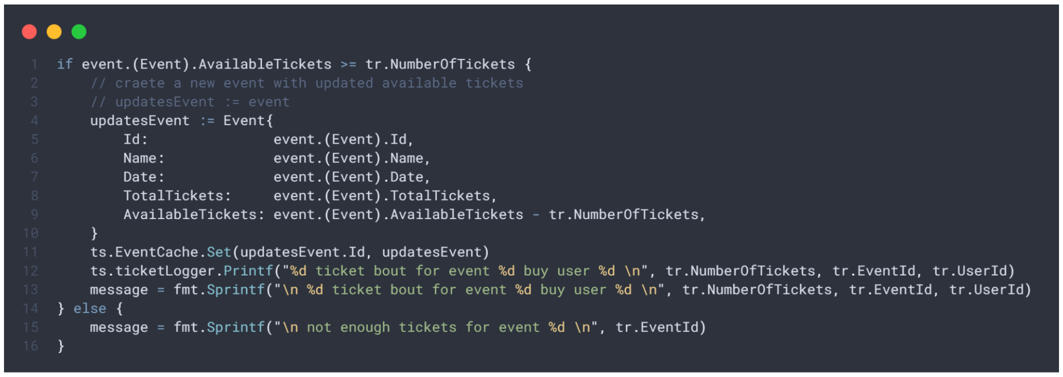
```go
reader := bufio.NewReader(os.Stdin)
for {
    text, _ := reader.ReadString('\n')
    buyerID, err := strconv.Atoi(strings.Split(text, " ")[0])
    if err != nil {
        printManual()
        continue
    }
    cliChannels[buyerID] <- text
    logger.Println(text)
}
```

- **Channel Communication:**Error handling before sending an order to the `orderChannel` in the `ticketBuyer` function.



```go
for input := range cliChannel {
    _, err := strconv.Atoi(strings.Split(input, " ")[0])
    if err != nil { // output result
        fmt.Printf("Buyer %d output result: %s ", id, input)
    } else {
        orderChannel <- input
        logger.Println("request sent to load balancer" + input)
    }
}
```

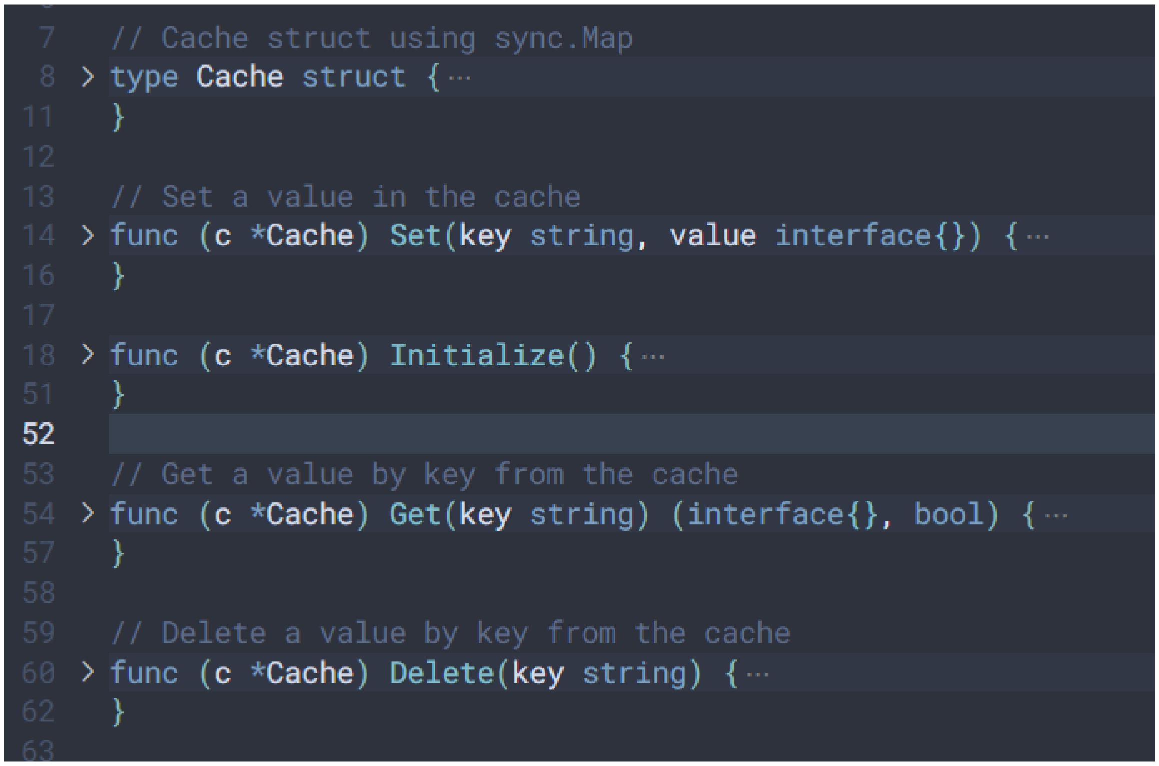
- **Ticket Purchase:** Error handling in the `buyTicket` method of the `TicketServices` struct.



```go
if event.(Event).AvailableTickets >= tr.NumberOfTickets {
    // craete a new event with updated available tickets
    // updatesEvent := event
    updatesEvent := Event{
        Id:               event.(Event).Id,
        Name:             event.(Event).Name,
        Date:             event.(Event).Date,
        TotalTickets:     event.(Event).TotalTickets,
        AvailableTickets: event.(Event).AvailableTickets - tr.NumberOfTickets,
    }
    ts.EventCache.Set(updatesEvent.Id, updatesEvent)
    ts.ticketLogger.Printf("%d ticket bout for event %d buy user %d \n", tr.NumberOfTickets, tr.EventId, tr.UserId)
    message = fmt.Sprintf("\n %d ticket bout for event %d buy user %d \n", tr.NumberOfTickets, tr.EventId, tr.UserId)
} else {
    message = fmt.Sprintf("\n not enough tickets for event %d \n", tr.EventId)
}
```

7. Caching:

- Implement a caching mechanism to improve system performance by caching events that are frequently accessed by many users.



```go
// Cache struct using sync.Map
type Cache struct {…
}

// Set a value in the cache
func (c *Cache) Set(key string, value interface{}) {…
}

func (c *Cache) Initialize() {…
}

// Get a value by key from the cache
func (c *Cache) Get(key string) (interface{}, bool) {…
}

// Delete a value by key from the cache
func (c *Cache) Delete(key string) {…
}
```

The cache is encapsulated within a struct termed `Cache`, comprising two primary attributes:

1. `store`: Utilizing a sync.Map data structure, it facilitates concurrent storage and retrieval of key-value pairs.

2. `NumberOfEvents`: An integer denoting the predetermined count of events to be initially cached.

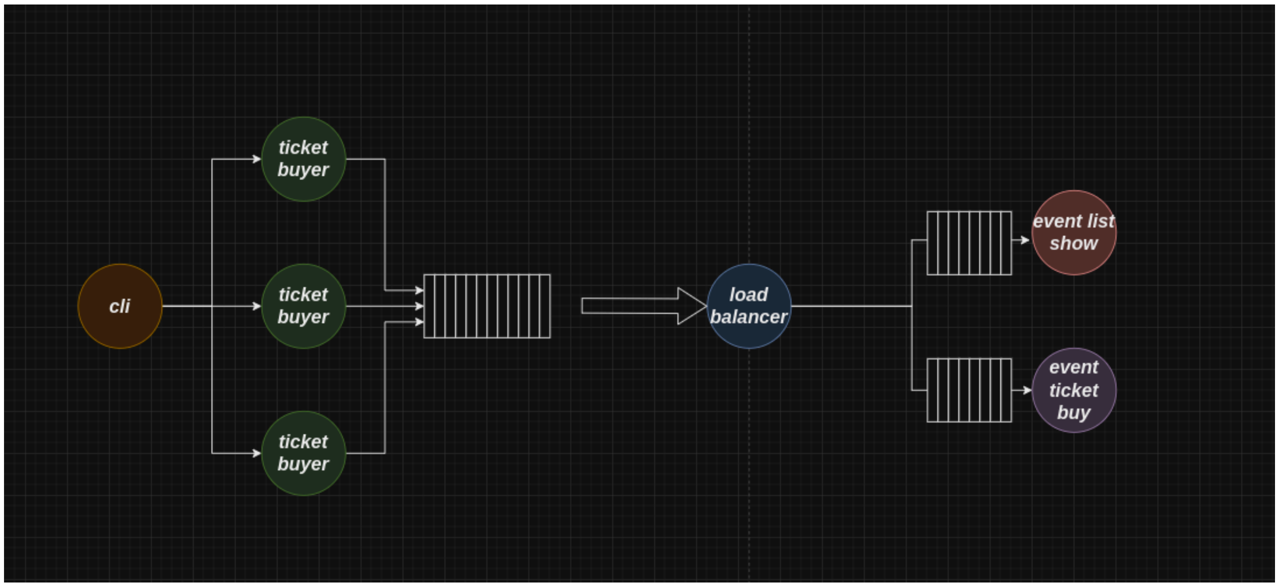
Cache Operations: Four key operations are supported by the cache:

- `Set`: Facilitates addition or modification of a key-value pair within the cache.
- `Initialize`: Prepares the cache by populating it with a specified number of sample events.
- `Get`: Retrieves the value associated with a given key from the cache, returning both the value and a boolean flag indicating its presence.
- `Delete`: Removes a key-value pair from the cache based on the provided key.

Sample Events: Upon initialization, three sample events are created and stored within the cache. Each event is represented by an instance of the `Event` struct, containing attributes like `Id`, `Name`, `Date`, `TotalTickets`, and `AvailableTickets`.

Concurrency Handling: While the code contains sections illustrating concurrent cache access by multiple goroutines, the functionality remains unused within the provided main function.

Project architecture



The main components are:

- CLI (Command Line Interface): This component serves as the user interface, allowing clients to input commands and interact with the ticket reservation system.
- Ticket Buyer: Multiple instances of the ticket buyer component handle user requests for purchasing tickets concurrently. These components communicate with the load balancer to process ticket buying requests.
- Load Balancer: This component acts as a central coordinator, receiving requests from the CLI and distributing them to the appropriate components (event list show or event ticket buy) for processing. It also implements mechanisms to ensure fairness and prevent starvation.
- Event List Show: This component is responsible for displaying the list of available events to the users upon request.
- Event Ticket Buy: This component handles the actual ticket reservation process, deducting the requested number of tickets from the available inventory for a specific event.

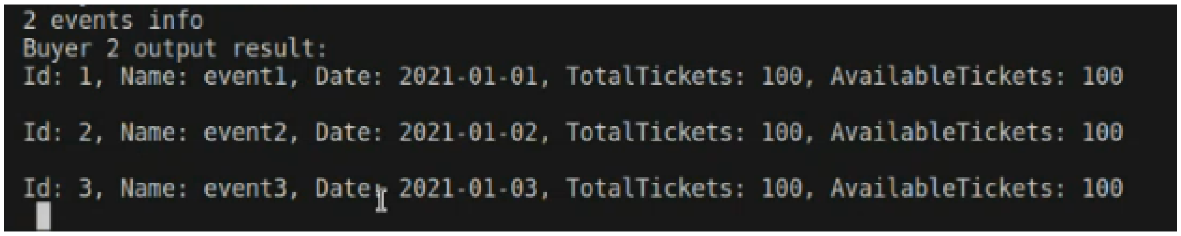
The architecture follows a modular design, with different components communicating with each other through interfaces or channels (represented by arrows in the diagram). The load balancer acts as a central hub, coordinating the flow of requests and responses between the different components.

This architecture allows for concurrent processing of user requests, as multiple ticket buyer instances can operate simultaneously. The load balancer ensures fair distribution of requests and prevents resource exhaustion or starvation.

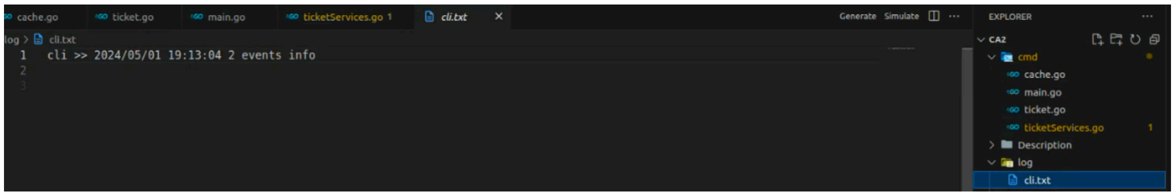
Overall, the architecture adheres to the project structure described in the report, with separate components for handling user interactions, ticket reservations, event management, and load balancing, while also incorporating concurrency control and fairness mechanisms.

Test

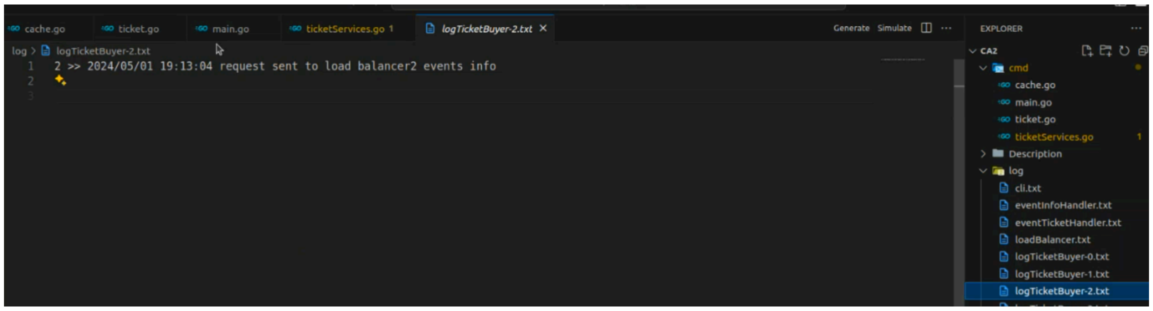
1. events info:



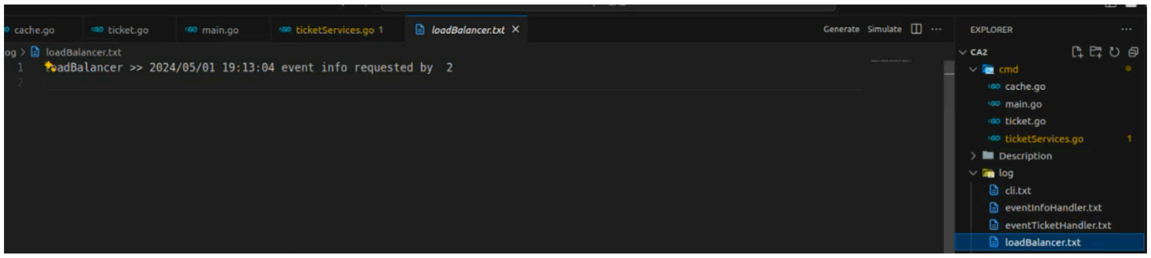
```
2 events info
Buyer 2 output result:
Id: 1, Name: event1, Date: 2021-01-01, TotalTickets: 100, AvailableTickets: 100

Id: 2, Name: event2, Date: 2021-01-02, TotalTickets: 100, AvailableTickets: 100

Id: 3, Name: event3, Date: 2021-01-03, TotalTickets: 100, AvailableTickets: 100
```



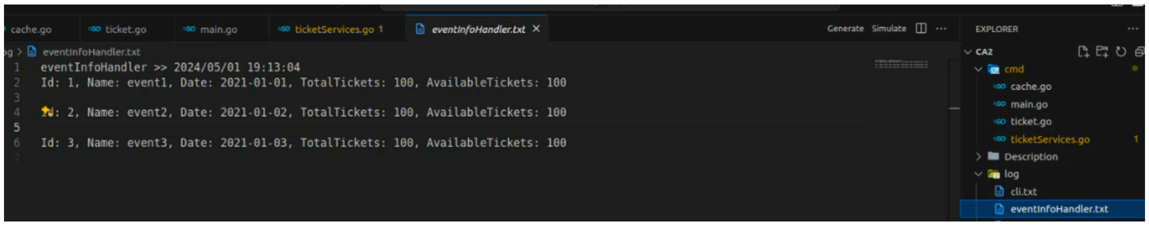
```
cli >> 2024/05/01 19:13:04 2 events info
```



```
2 >> 2024/05/01 19:13:04 request sent to load balancer2 events info
```

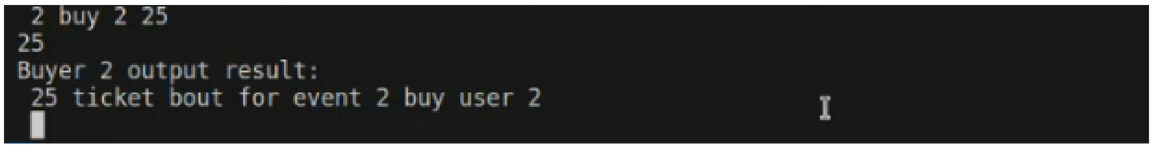


```
loadBalancer >> 2024/05/01 19:13:04 event info requested by  2
```

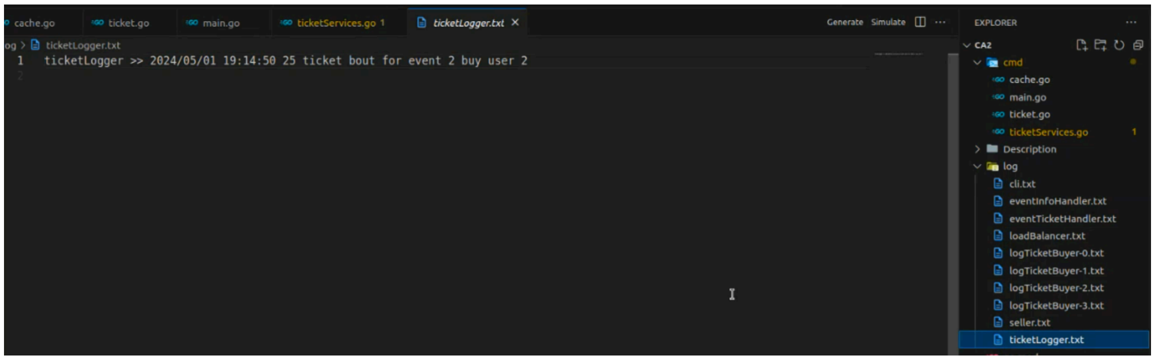


```
eventInfoHandler >> 2024/05/01 19:13:04
Id: 1, Name: event1, Date: 2021-01-01, TotalTickets: 100, AvailableTickets: 100

Id: 2, Name: event2, Date: 2021-01-02, TotalTickets: 100, AvailableTickets: 100

Id: 3, Name: event3, Date: 2021-01-03, TotalTickets: 100, AvailableTickets: 100
```

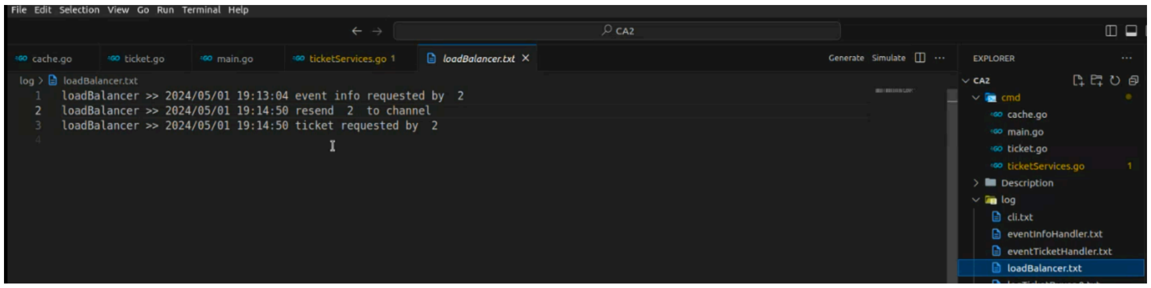
2. buy ticket:



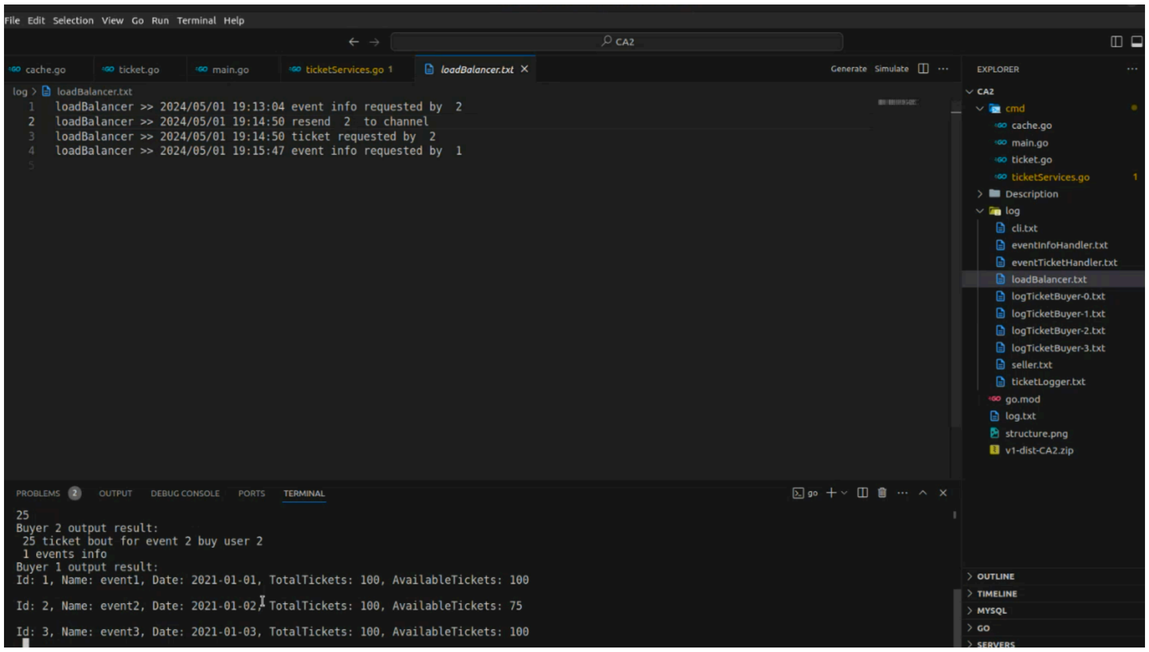
```
 2 buy 2 25
25
Buyer 2 output result:
 25 ticket bout for event 2 buy user 2
```



```
ticketLogger >> 2024/05/01 19:14:50 25 ticket bout for event 2 buy user 2
```



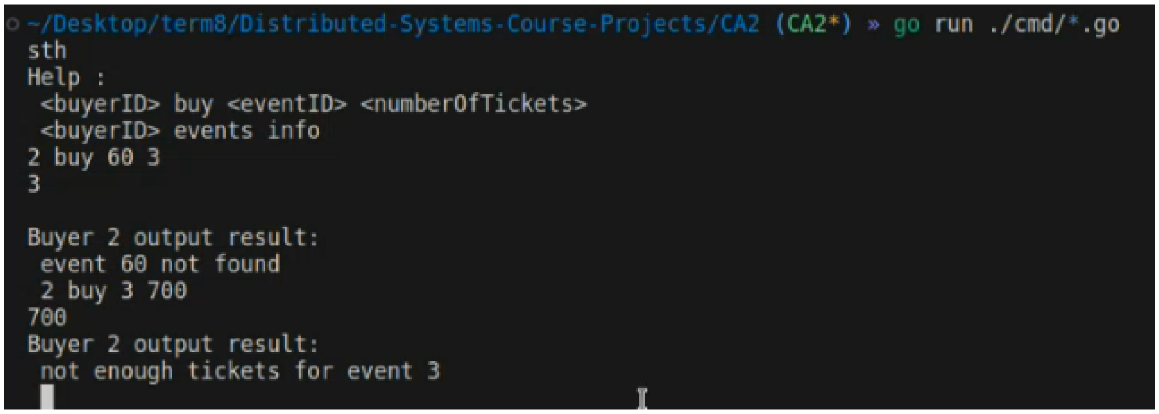
```
loadBalancer >> 2024/05/01 19:13:04 event info requested by  2
loadBalancer >> 2024/05/01 19:14:50 resend  2  to channel
loadBalancer >> 2024/05/01 19:14:50 ticket requested by  2
```



```
loadBalancer >> 2024/05/01 19:13:04 event info requested by  2
loadBalancer >> 2024/05/01 19:14:50 resend  2  to channel
loadBalancer >> 2024/05/01 19:14:50 ticket requested by  2
loadBalancer >> 2024/05/01 19:15:47 event info requested by  1
```

```
25
Buyer 2 output result:
 25 ticket bout for event 2 buy user 2
 1 events info
Buyer 1 output result:
Id: 1, Name: event1, Date: 2021-01-01, TotalTickets: 100, AvailableTickets: 100

Id: 2, Name: event2, Date: 2021-01-02, TotalTickets: 100, AvailableTickets: 75

Id: 3, Name: event3, Date: 2021-01-03, TotalTickets: 100, AvailableTickets: 100
```

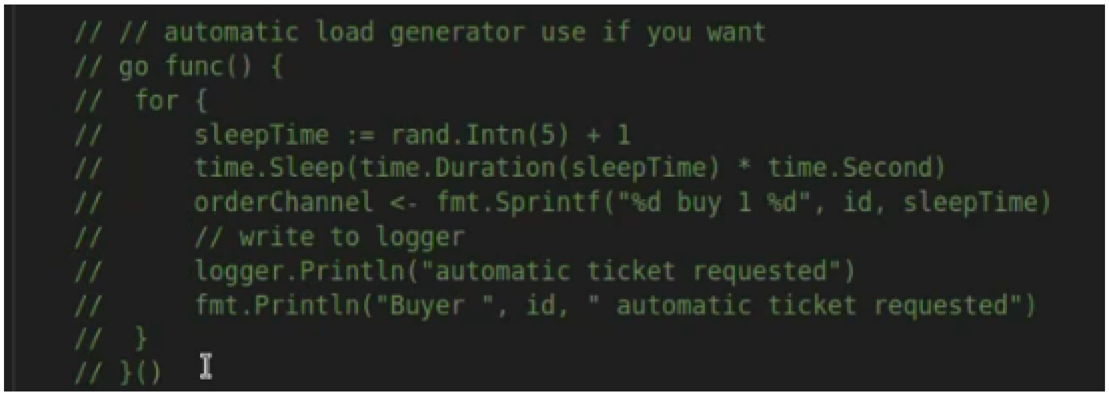
3. Error Handling Test



```
~/Desktop/term8/Distributed-Systems-Course-Projects/CA2 (CA2*) » go run ./cmd/*.go
sth
Help :
 <buyerID> buy <eventID> <numberOfTickets>
 <buyerID> events info
2 buy 60 3
3

Buyer 2 output result:
 event 60 not found
 2 buy 3 700
700
Buyer 2 output result:
 not enough tickets for event 3
```

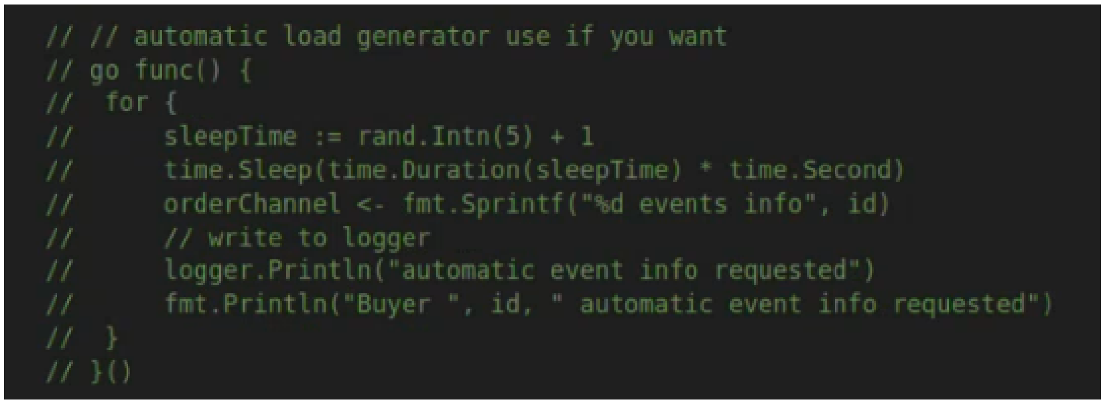
4. Generate load

generate buy:

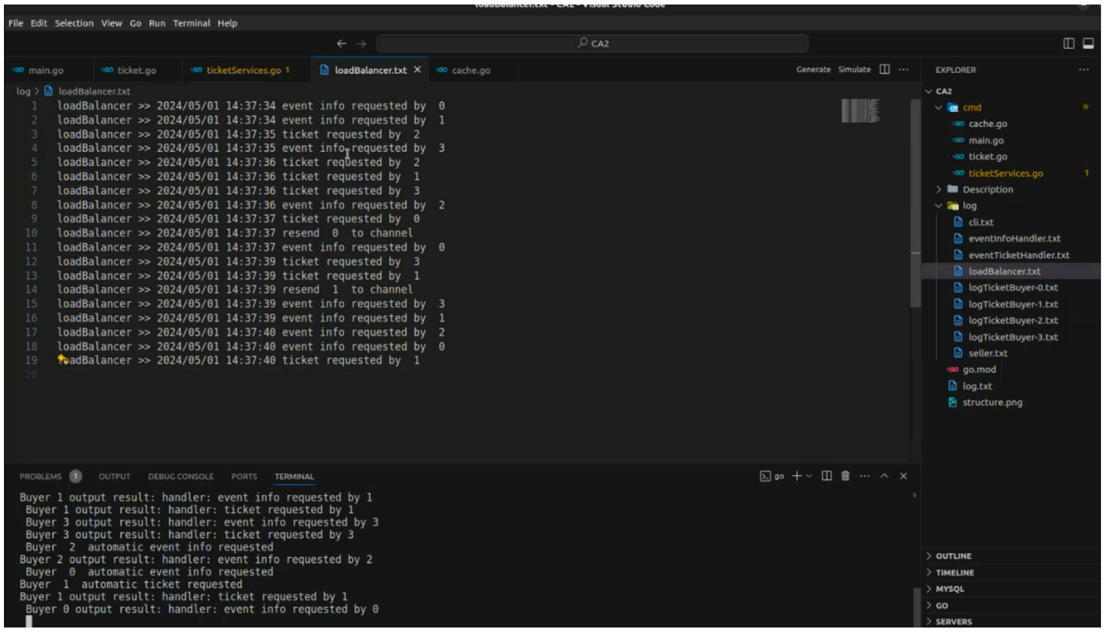


```go
// // automatic load generator use if you want
// go func() {
//   for {
//       sleepTime := rand.Intn(5) + 1
//       time.Sleep(time.Duration(sleepTime) * time.Second)
//       orderChannel <- fmt.Sprintf("%d buy 1 %d", id, sleepTime)
//       // write to logger
//       logger.Println("automatic ticket requested")
//       fmt.Println("Buyer ", id, " automatic ticket requested")
//   }
// }()
```

generate event



```go
// // automatic load generator use if you want
// go func() {
//   for {
//       sleepTime := rand.Intn(5) + 1
//       time.Sleep(time.Duration(sleepTime) * time.Second)
//       orderChannel <- fmt.Sprintf("%d events info", id)
//       // write to logger
//       logger.Println("automatic event info requested")
//       fmt.Println("Buyer ", id, " automatic event info requested")
//   }
// }()
```



```
loadBalancer >> 2024/05/01 14:37:34 event info requested by  0
loadBalancer >> 2024/05/01 14:37:34 event info requested by  1
loadBalancer >> 2024/05/01 14:37:35 ticket requested by  2
loadBalancer >> 2024/05/01 14:37:35 event info requested by  3
loadBalancer >> 2024/05/01 14:37:36 ticket requested by  2
loadBalancer >> 2024/05/01 14:37:36 ticket requested by  1
loadBalancer >> 2024/05/01 14:37:36 ticket requested by  3
loadBalancer >> 2024/05/01 14:37:36 event info requested by  2
loadBalancer >> 2024/05/01 14:37:37 ticket requested by  0
loadBalancer >> 2024/05/01 14:37:37 resend  0  to channel
loadBalancer >> 2024/05/01 14:37:37 event info requested by  0
loadBalancer >> 2024/05/01 14:37:39 ticket requested by  3
loadBalancer >> 2024/05/01 14:37:39 ticket requested by  1
loadBalancer >> 2024/05/01 14:37:39 resend  1  to channel
loadBalancer >> 2024/05/01 14:37:39 event info requested by  3
loadBalancer >> 2024/05/01 14:37:39 event info requested by  1
loadBalancer >> 2024/05/01 14:37:40 event info requested by  2
loadBalancer >> 2024/05/01 14:37:40 event info requested by  0
loadBalancer >> 2024/05/01 14:37:40 ticket requested by  1
```

```
Buyer 1 output result: handler: event info requested by 1
 Buyer 1 output result: handler: ticket requested by 1
 Buyer 3 output result: handler: event info requested by 3
 Buyer 3 output result: handler: ticket requested by 3
 Buyer  2  automatic event info requested
Buyer 2 output result: handler: event info requested by 2
 Buyer  0  automatic event info requested
Buyer  1  automatic ticket requested
Buyer 1 output result: handler: ticket requested by 1
 Buyer 0 output result: handler: event info requested by 0
```

Project Contributors and Roles

Ali Hodaei (810199513):

- Assumed the responsibility of designing the project architecture and overseeing its development. His expertise in project architecture design and meticulous attention to detail ensured the creation of a robust and scalable system. Furthermore, Ali played a crucial role in generating outputs that aligned with project objectives and requirements.

Fatemeh Mohammadi(810199489):

- Actively contributed to the project by providing assistance in writing work reports and collaborating on architectural design tasks. Her insights and contributions were invaluable in ensuring clear communication and documentation throughout the project lifecycle.

Ali Ataollahi (810199461):

- Played a vital role in the project by writing the Task Report and assisting in the division of tasks. Additionally, he provided valuable assistance in the implementation of the Purchase and Display components, ensuring their seamless integration and functionality within the project framework.

Mohammad Sadegi (810199483):

- Contributed to the project by writing different components and implementing an important part of the code. His expertise and dedication were instrumental in ensuring the functionality and efficiency of various aspects of the project.