Addis Ababa Institute of Technology

Reflection Report

Fundamentals of Distributed Systems

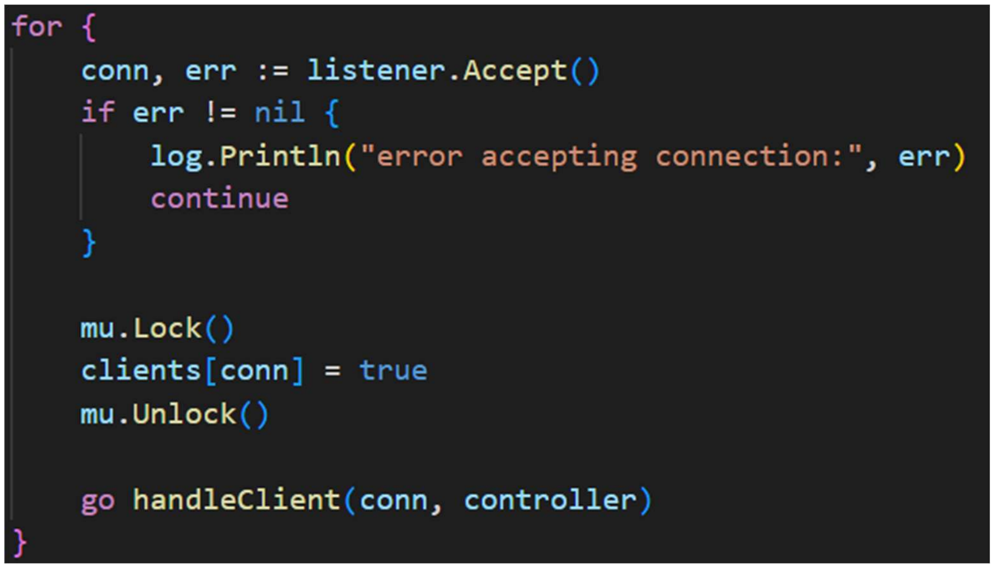
Nathan Mesfin Shiferaw
UGR/0534/14
Software Stream

Reflection Report

How concurrency is implemented to handle multiple clients

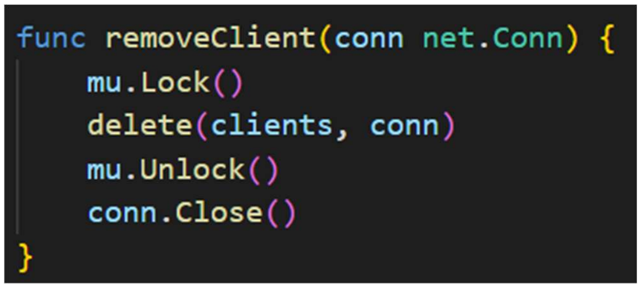
To handle multiple clients, a new goroutine is created (with context deadlines). A map of the connections is kept maintaining a list of available connections, and upon disconnection, either through manual client disconnection or due to timeout, the following function is called to remove the connection from the map and close it gracefully.

Accepting new clients:



```go
for {
    conn, err := listener.Accept()
    if err != nil {
        log.Println("error accepting connection:", err)
        continue
    }

    mu.Lock()
    clients[conn] = true
    mu.Unlock()

    go handleClient(conn, controller)
}
```

Removing clients:



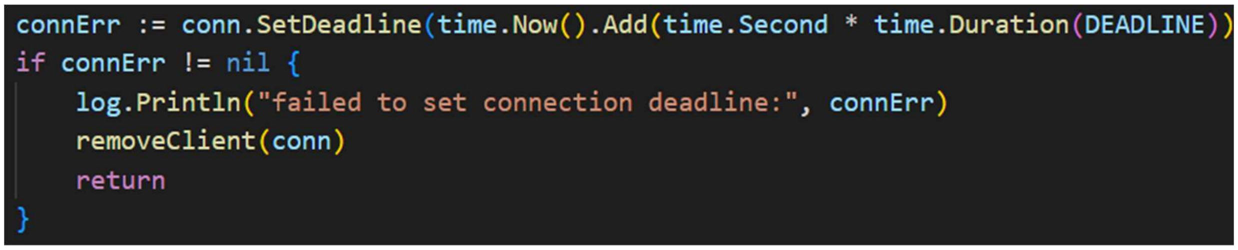
```go
func removeClient(conn net.Conn) {
    mu.Lock()
    delete(clients, conn)
    mu.Unlock()
    conn.Close()
}
```

Details on the ***handleClient()*** function can be found in *server/initialize/init_server.go*.

Challenges faced and how they were solved

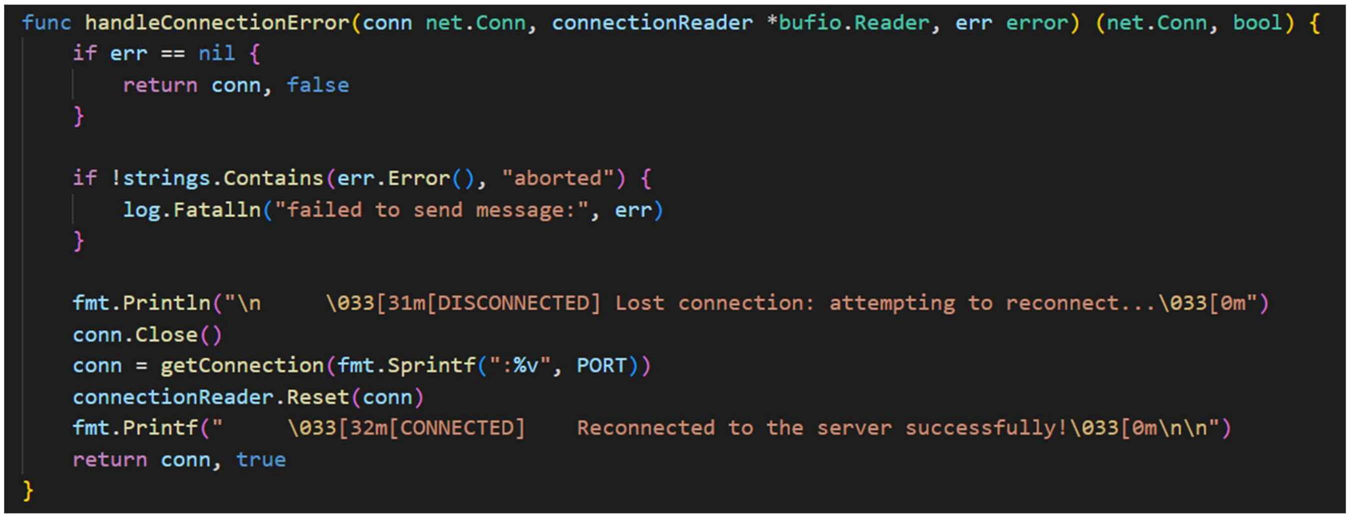
The first challenge came in the form of setting timeouts for the client connections (to lower the load on the server when clients are idle). This had been solved by setting a deadline for each connection on the server side and allowing the client to try reconnecting once if it tries to call the server after idling for a brief period.

Server code:



```go
connErr := conn.SetDeadline(time.Now().Add(time.Second * time.Duration(DEADLINE)))
if connErr != nil {
    log.Println("failed to set connection deadline:", connErr)
    removeClient(conn)
    return
}
```

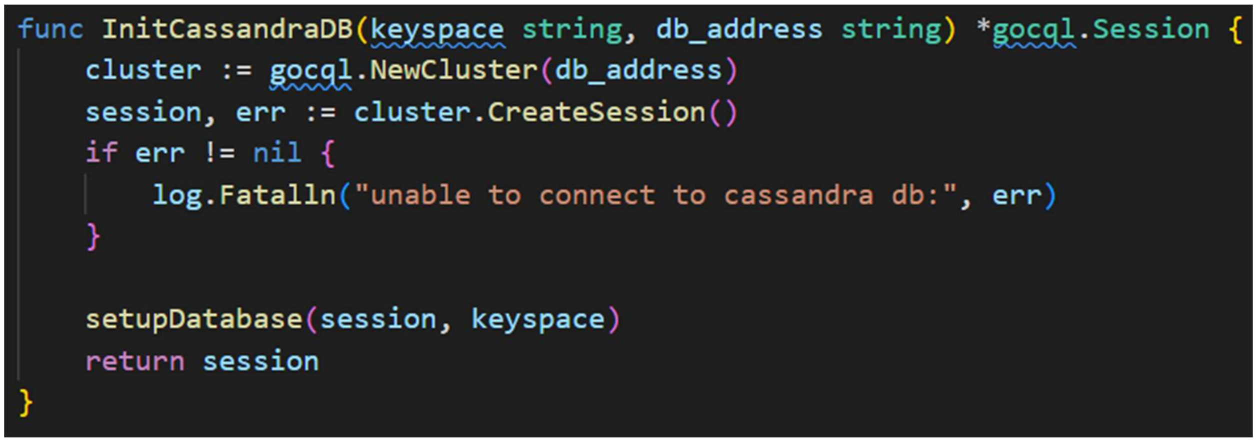
Client code:



```go
func handleConnectionError(conn net.Conn, connectionReader *bufio.Reader, err error) (net.Conn, bool) {
    if err == nil {
        return conn, false
    }

    if !strings.Contains(err.Error(), "aborted") {
        log.Fatalln("failed to send message:", err)
    }

    fmt.Println("\n     \033[31m[DISCONNECTED] Lost connection: attempting to reconnect...\033[0m")
    conn.Close()
    conn = getConnection(fmt.Sprintf(":%v", PORT))
    connectionReader.Reset(conn)
    fmt.Printf("     \033[32m[CONNECTED]    Reconnected to the server successfully!\033[0m\n\n")
    return conn, true
}
```

The second challenge I faced is related to persisting with the data. Cassandra was used to store key-value pairs.

Server code for initializing Cassandra DB:



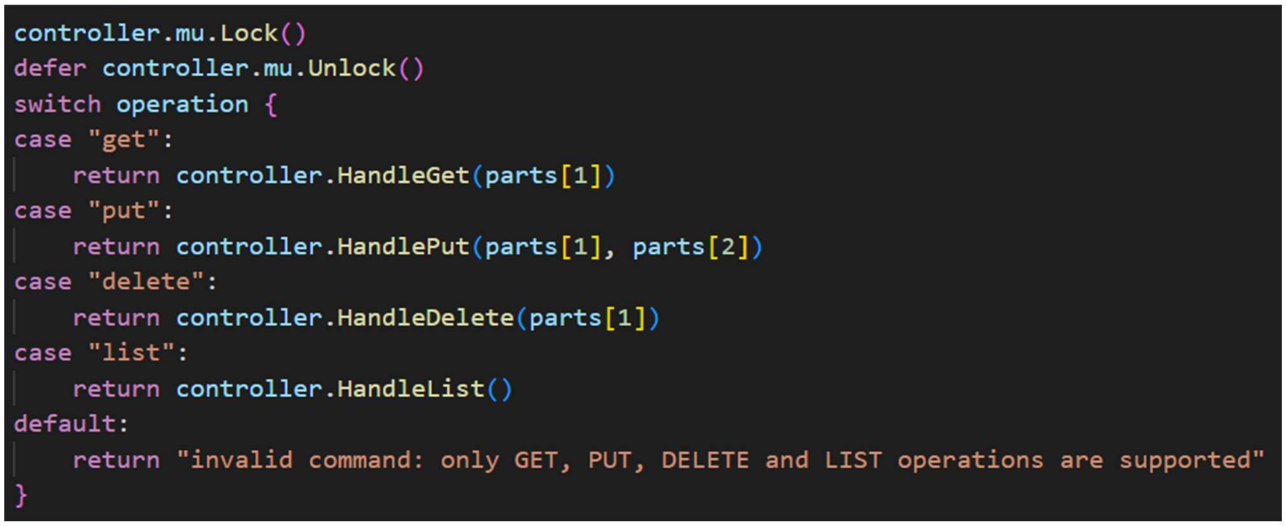
```go
func InitCassandraDB(keyspace string, db_address string) *gocql.Session {
    cluster := gocql.NewCluster(db_address)
    session, err := cluster.CreateSession()
    if err != nil {
        log.Fatalln("unable to connect to cassandra db:", err)
    }

    setupDatabase(session, keyspace)
    return session
}
```

The details of how the server interacts with the database can be found in *service/service.go*.

How was consistency ensured when multiple clients accessed the store concurrently?

To ensure consistency, the built-in *sync.Mutex* had been used before each request. This made it possible to avoid any race conditions by guaranteeing mutual exclusion for the segment of the code that calls the service functions, and by extension the database itself.

Server code handling consistency:



```go
controller.mu.Lock()
defer controller.mu.Unlock()
switch operation {
case "get":
    return controller.HandleGet(parts[1])
case "put":
    return controller.HandlePut(parts[1], parts[2])
case "delete":
    return controller.HandleDelete(parts[1])
case "list":
    return controller.HandleList()
default:
    return "invalid command: only GET, PUT, DELETE and LIST operations are supported"
}
```