VIETNAM INTERNATIONAL UNIVERSITY – HO CHI MINH CITY

MangaHub Project – Final Report

By
Phan Khắc Hoàng Anh - Student ID: ITITWE22144
Đỗ Hoàng Minh - Student ID: ITITWE22142

Course: Net-centric programming

T

I. INTRODUCTION

1. About the Project Team

MangaHub is developed as an academic software project to apply web application development and networking concepts in a practical environment. The development team focuses on backend systems, real-time communication, and service-oriented architecture using the Go programming language.

Project Name: MangaHub
Domain: Online Manga Library and Community Platform
Technology Focus: Go, REST API, TCP, UDP, WebSocket, gRPC

2. The Product's Information

Manga reading has become increasingly popular among students and young adults. Readers often use multiple platforms and devices to track their reading progress, discover new titles, and interact with other readers. However, most existing platforms focus on either content delivery or community discussion, not both.

MangaHub is designed to solve this problem by providing an integrated system where users can:

- Manage their personal manga library
- Track reading progress
- Receive notifications for new chapters
- Synchronize progress across devices
- Participate in real-time discussions

The system simulates a real-world distributed application by combining multiple communication protocols in a single platform.

3. Work Breakdown Structure

The project is divided into the following major functional modules:

- User Account Management
- Manga Management and Search
- Reading Progress Tracking
- Real-time Chat System
- Notification System
- Internal Service Communication

Each module is developed and tested incrementally following an agile development approach.

4. Development Process

The Scrum-based agile methodology is applied throughout the project lifecycle. Development is organized into short iterations, allowing continuous integration and testing of new features.

After each iteration:

- New functionalities are integrated
- Existing features are refined

- System stability and performance are evaluated

This approach ensures flexibility and timely delivery of a functional system.

5. Development Environment

The MangaHub system is implemented using the following technologies:

- **Programming Language:** Go (Golang)
- **Database:** SQLite
- **Protocols:** HTTP/REST, TCP, UDP, WebSocket, gRPC
- **Authentication:** JWT
- **Data Format:** JSON, Protocol Buffers

The system architecture follows a layered design separating controllers, services, and data access layers.

II. REQUIREMENT ANALYSIS AND DESIGN

1. Requirement Analysis

Actors

- **User (Manga Reader):** Uses the system to search manga, manage library, track reading progress, and join chat rooms
- **Administrator:** Manages system notifications and monitors services
- **TCP Client:** Receives real-time reading progress synchronization
- **UDP Client:** Receives broadcast notifications
- **WebSocket Client:** Participates in real-time chat
- **Internal Service:** Communicates via gRPC

Use Case 1: Register a New Account

Identifier: UC1
Actor: User

Inputs: Username, password
Outputs: User account created / error message

Basic Course:

1. User accesses the registration page
2. User enters required personal information
3. User submits the form

4. System validates the information
5. System stores user data in the database
6. System confirms successful registration

Precondition: User is not registered
Postcondition: User account exists in the system

User Story: As a new user, I want to register an account so that I can use MangaHub features.

Use Case 2: User Login

Identifier: UC2
Actor: User

Inputs: Username, password
Outputs: Authentication token / login failure message

Basic Course:

1. User opens login page
2. User enters credentials
3. System verifies credentials
4. System generates JWT token
5. User is redirected to homepage

Precondition: User account exists
Postcondition: User is authenticated

User Story: As a user, I want to log in so that I can securely access my library.

Use Case 3: Search Manga

Identifier: UC3
Actor: User

Inputs: Search keywords, filters
Outputs: List of matching manga

Basic Course:

1. User enters search keywords
2. System queries database
3. System returns matching results

Precondition: None
Postcondition: Search results displayed

Use Case 4: Add Manga

Identifier: UC4
Actor: Admin

Inputs: Manga ID
Outputs: Confirmation message

Basic Course:

1. Admin choose to add a manga to the database
2. Admin input manga info
3. check manga id is different to other
4. System stores manga database

Precondition: Admin logged in
Postcondition: Manga added to database

Use Case 5: Delete Manga

Identifier: UC4
Actor: Admin

Inputs: Manga ID
Outputs: Confirmation message

Basic Course:

1. Admin choose to delete a manga from the database
2. Admin input manga ID
3. check manga id
4. System delete manga in database

Precondition: Admin logged in
Postcondition: Manga deleted from database

Use Case 6: Update Reading Progress

Identifier: UC5
Actor: User

Inputs: Manga ID, chapter number
Outputs: Updated progress

Basic Course:

1. User updates chapter number
2. System validates progress
3. System updates database
4. System triggers synchronization

Precondition: Manga exists in user library
Postcondition: Progress updated

Use Case 7: TCP Progress Synchronization

Identifier: UC6
Actor: TCP Client

Inputs: Progress update event
Outputs: Synchronized updates to clients

Basic Course:

1. Client establishes TCP connection
2. Server listens for updates
3. Server broadcasts updates to clients

Use Case 8: Notification Broadcast

Identifier: UC7
Actor: Administrator

Inputs: Notification message
Outputs: Broadcast message

Basic Course:

1. Administrator triggers notification
2. System broadcasts via UDP

Use Case 9: gRPC Internal Communication

Identifier: UC9
Actor: Internal Service

Inputs: gRPC request
Outputs: gRPC response

Basic Course:

1. Service sends gRPC request
2. Server processes request
3. Response returned

A. Functional Requirements

- User authentication and authorization
- Manga search and library management
- Reading progress tracking
- TCP synchronization
- UDP notifications
- WebSocket chat
- gRPC internal services

B. Non-Functional Requirements

- Performance: Support concurrent users
- Security: JWT, encrypted communication
- Scalability: Modular design
- Reliability: Fault tolerance
- Usability: Simple APIs

III. IMPLEMENTATION

1. User Account Management

Users can register and log in through RESTful APIs. Passwords are hashed using bcrypt, and JWT tokens are used for session management.

Register:

-CLI:

```go
func registerUser() {
    clearScreen()
    printHeader("REGISTER")

    username := input("Enter username: ")
    password := input("Enter password: ")

    payload := map[string]string{
        "username": username,
        "password": password,
    }
    body, _ := json.Marshal(payload)

    res, err := http.Post(HTTP_API+"/auth/register", "application/json", bytes.NewReader(body))
    if err != nil {
        fmt.Println("Error connecting to server:", err)
        return
    }
    defer res.Body.Close()

    var reply map[string]interface{}
    json.NewDecoder(res.Body).Decode(&reply)

    if res.StatusCode == http.StatusCreated || res.StatusCode == http.StatusOK {
        fmt.Println("Account created successfully!")
        // some server return tokens on register; try extract
        if t, ok := reply["access_token"].(string); ok {
            accessToken = t
            if rt, ok2 := reply["refresh_token"].(string); ok2 {
                refreshToken = rt
            }
            currentUser = getUserIDFromJWT(accessToken)
            fmt.Println("Logged in as:", username)
        }
    } else {
        fmt.Println("Register failed:", reply["error"])
    }
}
```

-api-server:

```go
authGroup.POST("/register", auth.RegisterHandler(db))
```

-auth/handlers.go:

```go
func RegisterHandler(db *sql.DB) gin.HandlerFunc {
    return func(c *gin.Context) {
        var req struct {
            Username string `json:"username"`
            Password string `json:"password"`
        }

        if err := c.BindJSON(&req); err != nil {
            c.JSON(400, gin.H{"error": "invalid input"})
            return
        }

        hash, _ := bcrypt.GenerateFromPassword([]byte(req.Password), bcrypt.DefaultCost)

        res, err := db.Exec(`
            INSERT INTO users (username, password_hash, role)
            VALUES (?, ?, 'user')
        `, req.Username, hash)

        if err != nil {
            c.JSON(409, gin.H{"error": "username exists"})
            return
        }

        id, _ := res.LastInsertId()
        userID := fmt.Sprintf("%d", id)

        access, _ := CreateAccessToken(userID, req.Username, "user")
        refresh, _ := CreateRefreshToken(db, userID)

        c.JSON(201, gin.H{
            "access_token":  access,
            "refresh_token": refresh,
        })
    }
}
```

Login:

CLI:

```
func loginUser() {
    clearScreen()
    printHeader("LOGIN")

    username := input("Username: ")
    password := input("Password: ")

    payload := map[string]string{
        "username": username,
        "password": password,
    }
    body, _ := json.Marshal(payload)

    res, err := http.Post(HTTP_API+"/auth/login", "application/json", bytes.NewReader(body))
    if err != nil {
        fmt.Println("Error connecting to server:", err)
        return
    }
    defer res.Body.Close()

    var reply map[string]interface{}
    json.NewDecoder(res.Body).Decode(&reply)

    // support both {"token": "..."} and {"access_token":"..."}
    if t, ok := reply["access_token"].(string); ok {
        accessToken = t
        if rt, ok2 := reply["refresh_token"].(string); ok2 {
            refreshToken = rt
        }
        currentUser = getUserIDFromJWT(accessToken)

        fmt.Println("Login successful!")
    } else if t2, ok := reply["token"].(string); ok {
        accessToken = t2
        currentUser = getUserIDFromJWT(accessToken)
        fmt.Println("Login successful!")
    } else {
        fmt.Println("Login failed:", reply["error"])
    }
}
```

api-server:

```
authGroup.POST("/register", auth.RegisterHandler(db))
```

auth/handlers.go:

```go
func LoginHandler(db *sql.DB) gin.HandlerFunc {
    return func(c *gin.Context) {
        var req struct {
            Username string `json:"username"`
            Password string `json:"password"`
        }

        if err := c.BindJSON(&req); err != nil {
            c.JSON(400, gin.H{"error": "invalid input"})
            return
        }

        var id int
        var hash, role string

        err := db.QueryRow(`
            SELECT id, password_hash, role
            FROM users
            WHERE username = ?
        `, req.Username).Scan(&id, &hash, &role)

        if err != nil || bcrypt.CompareHashAndPassword([]byte(hash), []byte(req.Password)) != nil {
            c.JSON(401, gin.H{"error": "invalid credentials"})
            return
        }

        userID := fmt.Sprintf("%d", id)

        access, _ := CreateAccessToken(userID, req.Username, role)
        refresh, _ := CreateRefreshToken(db, userID)

        c.JSON(200, gin.H{
            "access_token":  access,
            "refresh_token": refresh,
        })
    }
}
```

Token:

```go
func CreateAccessToken(userID, username, role string) (string, error) {
    claims := &Claims{
        UserID:   userID,
        Username: username,
        Role:     role,
        RegisteredClaims: jwt.RegisteredClaims{
            ExpiresAt: jwt.NewNumericDate(time.Now().Add(accessTokenTTL)),
            IssuedAt:  jwt.NewNumericDate(time.Now()),
        },
    }
    t := jwt.NewWithClaims(jwt.SigningMethodHS256, claims)
    return t.SignedString(jwtSecret)
}
```

```go
func CreateRefreshToken(db *sql.DB, userID string) (string, error) {
    token := uuid.NewString()
    expires := time.Now().Add(refreshTokenTTL)

    _, err := db.Exec(`
        INSERT INTO refresh_tokens (token, user_id, expires_at)
        VALUES (?, ?, ?)
    `, token, userID, expires)

    return token, err
}
```

```go
func ValidateRefreshToken(db *sql.DB, token string) (string, error) {
    var userID string
    var expiresAt time.Time

    err := db.QueryRow(`
        SELECT user_id, expires_at
        FROM refresh_tokens
        WHERE token = ?
    `, token).Scan(&userID, &expiresAt)

    if err == sql.ErrNoRows {
        return "", errors.New("invalid refresh token")
    }
    if err != nil {
        return "", err
    }
    if time.Now().After(expiresAt) {
        _ = RevokeRefreshToken(db, token)
        return "", errors.New("refresh token expired")
    }

    return userID, nil
}
```

```go
func RevokeRefreshToken(db *sql.DB, token string) error {
    _, err := db.Exec(`DELETE FROM refresh_tokens WHERE token = ?`, token)
    return err
}
```

```go
func ParseAccessToken(tokenStr string) (*Claims, error) {
    token, err := jwt.ParseWithClaims(tokenStr, &Claims{}, func(t *jwt.Token) (interface{}, error) {
        return jwtSecret, nil
    })
    if err != nil {
        return nil, err
    }
    claims, ok := token.Claims.(*Claims)
    if !ok || !token.Valid {
        return nil, jwt.ErrTokenInvalidClaims
    }
    return claims, nil
}
```

2. Manga Management

The system provides APIs for searching manga, viewing details, and managing personal libraries. SQLite is used for persistent storage.
**The database is from AniList.*

cmd/import-json/main.go:

-Manga attribute:

```go
type Manga struct {
    ID            string   `json:"id"`
    Title         string   `json:"title"`
    Author        string   `json:"author"`
    Genres        []string `json:"genres"`
    Status        string   `json:"status"`
    TotalChapters int      `json:"total_chapters"`
    Description   string   `json:"description"`
}
```

```go
    // 1. Open Database
    db, err := sql.Open("sqlite3", "../../mangahub.db")
    if err != nil {
        panic(err)
    }
    defer db.Close()

    // 2. Load JSON
    data, err := os.ReadFile("manga.json")
    if err != nil {
        panic(err)
    }
```

```
    // 3. Parse JSON → []Manga
    var items []Manga
    if err := json.Unmarshal(data, &items); err != nil {
        panic(err)
    }

    // 4. Prepare SQL insert
    stmt, err := db.Prepare(`
        INSERT INTO manga (id, title, author, genres, status,
total_chapters, description)
        VALUES (?, ?, ?, ?, ?, ?, ?)
    `)
    if err != nil {
        panic(err)
    }
    defer stmt.Close()

    // 5. Insert each manga
    for _, item := range items {

        // convert []string → JSON string
        genresJson, _ := json.Marshal(item.Genres)

        _, err := stmt.Exec(
            item.ID,
            item.Title,
            item.Author,
            string(genresJson), // <-- store JSON text
            item.Status,
            item.TotalChapters,
            item.Description,
        )
        if err != nil {
            fmt.Println("Error inserting", item.ID, ":", err)
            continue
        }
    }

    fmt.Println("Import complete.")
}
```

***Search manga:**

cmd/cli/main.go:

```
func searchMangaGRPC(client pb.MangaServiceClient) {
    clearScreen()
    printHeader("MANGA SEARCH (gRPC)")

    query := input("Enter keyword (empty = all): ")
    genre := input("Genre filter (empty = ignore): ")
    status := input("Status filter (empty = ignore): ")
    limit := input("Limit (default 50): ")

    lim := int32(50)
    if limit != "" {
        n, _ := strconv.Atoi(limit)
        lim = int32(n)
    }

    req := &pb.SearchRequest{
        Query:  query,
        Genre:  genre,
        Status: status,
        Limit:  lim,
    }

    ctx, cancel := context.WithTimeout(context.Background(),
5*time.Second)
    defer cancel()

    resp, err := client.SearchManga(ctx, req)
    if err != nil {
        fmt.Println("gRPC error:", err.Error())
        return
    }

    clearScreen()
    printHeader("SEARCH RESULTS")

    if len(resp.Results) == 0 {
        fmt.Println("No results found.")
        time.Sleep(1 * time.Second)
        return
    }

    for _, m := range resp.Results {
```

```
        fmt.Printf("[%s] %s (%s)\n", m.Id, m.Title, m.Status)
    }

    fmt.Println("\nOptions:")
    fmt.Println("1) MANGA INFO")
    fmt.Println("2) MAIN MENU")

    choice := input("> ")

    switch choice {
    case "1":
        lastMangaID = input("Enter manga ID: ")
        mangaInfoGRPC(client) // pass through
    }
}
```

internal/grpc/grpc.go:

```
func (s *GRPCMangaServer) SearchManga(ctx context.Context, req *pb.SearchRequest) (*pb.SearchResponse, error) {

    rows, err := s.DB.Query(`
        SELECT id, title, author, status
        FROM manga
        WHERE (title LIKE '%' || ? || '%' OR ? = '')
        AND (genres LIKE '%' || ? || '%' OR ? = '')
        AND (status = ? OR ? = '')
        LIMIT ?
    `, req.Query, req.Query,
        req.Genre, req.Genre,
        req.Status, req.Status,
        req.Limit,
    )
    if err != nil {
        return nil, err
    }
    defer rows.Close()

    resp := &pb.SearchResponse{}
    for rows.Next() {
        var r pb.SearchResult
        rows.Scan(&r.Id, &r.Title, &r.Author, &r.Status)
        resp.Results = append(resp.Results, &r)
    }

    return resp, nil
}
```

proto/manga/manga_grpc.pb.go:

```
func _MangaService_SearchManga_Handler(srv interface{}, ctx context.Context, dec func(interface{}) error, interceptor grpc.UnaryServerInterceptor) (interface{}, error) {
    in := new(SearchRequest)
    if err := dec(in); err != nil {
        return nil, err
    }
    if interceptor == nil {
        return srv.(MangaServiceServer).SearchManga(ctx, in)
    }
    info := &grpc.UnaryServerInfo{
        Server:     srv,
        FullMethod: MangaService_SearchManga_FullMethodName,
    }
    handler := func(ctx context.Context, req interface{}) (interface{}, error) {
        return srv.(MangaServiceServer).SearchManga(ctx, req.(*SearchRequest))
    }
    return interceptor(ctx, in, info, handler)
}
```

****Admin****

***Add manga:**

cmd/admin-cli/main.go:

```go
func addManga() {
    payload := map[string]interface{}{
        "id":             input("ID: "),
        "title":          input("Title: "),
        "author":         input("Author: "),
        "genres":         strings.Split(input("Genres (comma): "), ","),
        "status":         input("Status: "),
        "total_chapters": mustInt(input("Total chapters: ")),
        "description":    input("Description: "),
    }

    data, _ := json.Marshal(payload)
    req, _ := http.NewRequest("POST", API+"/admin/manga", bytes.NewBuffer(data))

    resp, err := doAuthRequest(req)
    if err != nil {
        fmt.Println("Request failed:", err)
        return
    }
    defer resp.Body.Close()

    fmt.Println("Add manga status:", resp.Status)
}
```

internal/manga/manga.go:

```go
r.POST("/manga", func(c *gin.Context) {
    var m Manga
    if err := c.BindJSON(&m); err != nil {
        c.JSON(400, gin.H{"error": "Invalid JSON"})
        return
    }

    _, err := db.Exec(`
        INSERT INTO manga (id, title, author, genres, status, total_chapters, description)
        VALUES (?, ?, ?, ?, ?, ?, ?)`,
        m.ID, m.Title, m.Author, m.Genres, m.Status, m.TotalChapters, m.Description,
    )

    if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }

    c.JSON(200, gin.H{"message": "Manga added successfully"})

    // AUTO UDP NOTIFICATION
    note := map[string]interface{}{
        "type":    "manga_added",
        "title":   m.Title,
        "message": "New manga added: " + m.Title,
    }

    data, _ := json.Marshal(note)
    http.Post("http://127.0.0.1:9094/broadcast", "application/json", bytes.NewBuffer(data))
})
```

***Delete manga:**

cmd/admin-cli/main.go:

```go
func deleteManga() {
    id := input("Manga ID to delete: ")

    req, _ := http.NewRequest("DELETE", API+"/admin/manga/"+id, nil)

    resp, err := doAuthRequest(req)
    if err != nil {
        fmt.Println("Request failed:", err)
        return
    }
    defer resp.Body.Close()

    fmt.Println("Delete status:", resp.Status)
}
```

internal/manga/manga.go:

```go
r.DELETE("/manga/:id", func(c *gin.Context) {
    id := c.Param("id")

    _, err := db.Exec("DELETE FROM manga WHERE id = ?", id)
    if err != nil {
        c.JSON(500, gin.H{"error": err.Error()})
        return
    }
    c.JSON(200, gin.H{"message": "Manga deleted"})
})
```

3. Reading Progress Synchronization

A TCP server maintains persistent connections with clients and broadcasts progress updates in real time using Go routines and channels.

cmd/cli/main.go:

```go
func mangaInfoGRPC(client pb.MangaServiceClient) {
    clearScreen()

    id := lastMangaID
    if id == "" {
        id = input("Enter manga ID: ")
        lastMangaID = id
    }

    req := &pb.GetMangaRequest{Id: id}

    ctx, cancel := context.WithTimeout(context.Background(),
5*time.Second)
    defer cancel()

    m, err := client.GetManga(ctx, req)
    if err != nil {
        fmt.Println("gRPC error:", err.Error())
        return
    }

    // FETCH PROGRESS
    progReq := &pb.GetProgressRequest{
        UserId:  currentUser,
        MangaId: m.Id,
```

```
    }

    progResp, _ := client.GetProgress(ctx, progReq)

    clearScreen()
    printHeader("MANGA INFO")

    fmt.Println("ID:", m.Id)
    fmt.Println("Title:", m.Title)
    fmt.Println("Author:", m.Author)
    fmt.Println("Genres:", m.Genres)
    fmt.Println("Status:", m.Status)
    fmt.Println("Total Chapters:", m.TotalChapters)

    fmt.Println("\nDescription:")
    fmt.Println(m.Description)

    if progResp.Exists {
        fmt.Println("\nYour Progress: Chapter",
progResp.CurrentChapter)
    }

    fmt.Println("\nOptions:")
    fmt.Println("1) UPDATE PROGRESS")
    fmt.Println("2) MAIN MENU")

    choice := input("> ")

    switch choice {
    case "1", "progress":
        updateProgressHTTP()
        lastMangaID = "" // reset after update
    default:
        lastMangaID = "" // reset when returning to menu
        return
    }

}
func mangaInfoGRPC(client pb.MangaServiceClient) {
    clearScreen()

    id := lastMangaID
    if id == "" {
```

```
        id = input("Enter manga ID: ")
        lastMangaID = id
    }

    req := &pb.GetMangaRequest{Id: id}

    ctx, cancel := context.WithTimeout(context.Background(),
5*time.Second)
    defer cancel()

    m, err := client.GetManga(ctx, req)
    if err != nil {
        fmt.Println("gRPC error:", err.Error())
        return
    }

    // FETCH PROGRESS
    progReq := &pb.GetProgressRequest{
        UserId:  currentUser,
        MangaId: m.Id,
    }

    progResp, _ := client.GetProgress(ctx, progReq)

    clearScreen()
    printHeader("MANGA INFO")

    fmt.Println("ID:", m.Id)
    fmt.Println("Title:", m.Title)
    fmt.Println("Author:", m.Author)
    fmt.Println("Genres:", m.Genres)
    fmt.Println("Status:", m.Status)
    fmt.Println("Total Chapters:", m.TotalChapters)

    fmt.Println("\nDescription:")
    fmt.Println(m.Description)

    if progResp.Exists {
        fmt.Println("\nYour Progress: Chapter",
progResp.CurrentChapter)
    }

    fmt.Println("\nOptions:")
```

```
    fmt.Println("1) UPDATE PROGRESS")
    fmt.Println("2) MAIN MENU")

    choice := input("> ")

    switch choice {
    case "1", "progress":
        updateProgressHTTP()
        lastMangaID = "" // reset after update
    default:
        lastMangaID = "" // reset when returning to menu
        return
    }

}
```

```go
func updateProgressHTTP() {
    clearScreen()
    printHeader("UPDATE PROGRESS")

    if lastMangaID == "" {
        lastMangaID = input("Enter manga ID: ")
    }

    chapterStr := input("Enter current chapter: ")
    ch, _ := strconv.Atoi(chapterStr)

    payload := map[string]interface{}{
        "manga_id": lastMangaID,
        "chapter":  ch,
    }

    data, _ := json.Marshal(payload)

    req, _ := http.NewRequest(
        "POST",
        HTTP_API+"/users/progress",
        bytes.NewBuffer(data),
    )

    resp, err := doAuthRequest(req)
    if err != nil {
        fmt.Println("Request failed:", err)
        time.Sleep(time.Second)
        return
    }
    defer resp.Body.Close()

    var res map[string]string
    json.NewDecoder(resp.Body).Decode(&res)

    fmt.Println(res["message"])
    time.Sleep(time.Second)
}
```

internal/tcp/progress_server.go:

```
        case "PROGRESS":
            var update ProgressUpdate
            if err := json.Unmarshal(raw, &update); err != nil {
                continue
            }
            update.Timestamp = time.Now().Unix()
            s.Broadcast <- update
        }
    }
```

```
func NewProgressSyncServer(port string) *ProgressSyncServer {
    return &ProgressSyncServer{
        Port:      port,
        Clients:   make(map[string]*ClientConn),
        Broadcast: make(chan ProgressUpdate, 100),

        Buffer:    make([]ProgressUpdate, 0, 100),
        MaxBuffer: 100,
    }
}
```

```
func (s *ProgressSyncServer) broadcastLoop() {
    for update := range s.Broadcast {
        data, _ := json.Marshal(update)

        s.mu.Lock()
        // buffer always
        if len(s.Buffer) >= s.MaxBuffer {
            s.Buffer = s.Buffer[1:] // drop oldest
        }
        s.Buffer = append(s.Buffer, update)
        for _, client := range s.Clients {
            fmt.Fprintln(client.conn, string(data))
        }
        s.mu.Unlock()
    }
}
```

```
func (s *ProgressSyncServer) handleClient(conn net.Conn) {
    addr := conn.RemoteAddr().String()

    client := &ClientConn{
        conn:     conn,
        lastPing: time.Now(),
    }
```

```
    s.mu.Lock()
    s.Clients[addr] = client
    for _, evt := range s.Buffer {
        data, _ := json.Marshal(evt)
        fmt.Fprintln(conn, string(data))
    }
    s.mu.Unlock()

    fmt.Println("Client connected:", addr)

    scanner := bufio.NewScanner(conn)

    for scanner.Scan() {
        raw := scanner.Bytes()

        var base struct {
            Type string `json:"type"`
        }
        if err := json.Unmarshal(raw, &base); err != nil {
            continue
        }

        switch base.Type {
        case "PING":
            client.lastPing = time.Now()
            conn.Write([]byte(`{"type":"PONG"}` + "\n"))

        case "PROGRESS":
            var update ProgressUpdate
            if err := json.Unmarshal(raw, &update); err != nil {
                continue
            }
            update.Timestamp = time.Now().Unix()
            s.Broadcast <- update
        }
    }

    // disconnect
    s.mu.Lock()
    delete(s.Clients, addr)
    s.mu.Unlock()
```

```
    conn.Close()
    fmt.Println("Client disconnected:", addr)
}
```

4. Notification System

UDP is used to broadcast notifications about new chapters. This mechanism prioritizes speed and tolerates packet loss.
internal/udp/notification_server.go:

```go
func (s *NotificationServer) Start() error {
    addr, err := net.ResolveUDPAddr("udp", s.Port)
    if err != nil {
        return err
    }

    conn, err := net.ListenUDP("udp", addr)
    if err != nil {
        return err
    }

    s.conn = conn // store the connection for broadcasting

    fmt.Println("UDP Notification Server running on", s.Port)

    buf := make([]byte, 2048)

    for {
        n, clientAddr, err := conn.ReadFromUDP(buf)
        if err != nil {
            fmt.Println("UDP read error:", err)
            continue
        }

        var msg struct {
            Type string `json:"type"`
            ID   string `json:"id"`
        }

        if err := json.Unmarshal(buf[:n], &msg); err != nil {
            continue
        }

        switch msg.Type {
        case "REGISTER":
            s.addClient(*clientAddr)
            conn.WriteToUDP([]byte(`{"type":"REGISTER_ACK"}`), clientAddr)

        case "ACK":
            fmt.Println("✅ ACK received from", clientAddr)
        case "NOTIFY":

        default:
            fmt.Println("Unknown UDP message:", msg.Type)
        }
    }
}
```

5. gRPC Services

Internal services use gRPC and Protocol Buffers to ensure efficient and type-safe communication between components.

cmd/grpc-server/main.go:

```go
func main() {

    addr := flag.String("addr", ":9092", "gRPC listen address")
    flag.Parse()

    // init DB (reuse your package)
    dbPath, _ := filepath.Abs("mangahub.db")
    db := database.InitDB(dbPath)

    defer db.Close()

    lis, err := net.Listen("tcp", *addr)
    if err != nil {
        log.Fatalf("failed to listen %v", err)
    }

    log.Println("grpc DB path:", dbPath)
    grpcServer := grpc.NewServer()
    svc := &grpcinternal.GRPCMangaServer{DB: db}
    pb.RegisterMangaServiceServer(grpcServer, svc)

    // Health check service
    hs := health.NewServer()
    hs.SetServingStatus("", healthpb.HealthCheckResponse_SERVING)
    healthpb.RegisterHealthServer(grpcServer, hs)

    quit := make(chan os.Signal, 1)
    signal.Notify(quit, os.Interrupt, syscall.SIGTERM)

    go func() {
        <-quit
        log.Println("🛑 Shutting down gRPC server...")
        grpcServer.GracefulStop()
        db.Close()
        os.Exit(0)
    }()

    log.Printf("gRPC server listening on %s", *addr)
    if err := grpcServer.Serve(lis); err != nil {
        log.Fatalf("gRPC serve error: %v", err)
    }

}
```

IV. DISCUSSION AND CONCLUSION

Through the MangaHub project, the development team gained practical experience in building a distributed system using multiple communication protocols. The project demonstrates how REST, TCP, UDP, WebSocket, and gRPC can coexist within a single application.

Future improvements may include:

- Deployment on cloud infrastructure
- Integration with real manga APIs
- Recommendation systems
- Mobile client support

Overall, MangaHub successfully meets its educational objectives and provides a strong foundation for advanced backend and distributed system development.

V. REFERENCES

- - github.com/gin-gonic/gin - HTTP web framework
- - github.com/golang-jwt/jwt/v4 - JWT authentication
- - github.com/gorilla/websocket - WebSocket support
- - github.com/mattn/go-sqlite3 - SQLite database driver
- - google.golang.org/grpc - gRPC framework
- - google.golang.org/protobuf - Protocol Buffers
- - github.com/stretchr/testify - Testing utilities