

Ride_Hailing_Sys

Introduction

1. Background

- 場景：現代智慧城市中，叫車服務（如 Uber/Grab）需要處理成千上萬的即時連線。
- 核心需求：伺服器必須具備極低的延遲 (Low Latency) 和高可用性 (High Availability)。

2. Problem Statement

- 效能瓶頸：傳統的 Single-Threaded 伺服器無法利用多核心 CPU；而簡單的 Fork-per-request 模型在面對大量連線時，會因為頻繁的 fork() 系統呼叫與 Context Switch 導致嚴重的效能下降。
- 同步難題：在 Multi-Process 環境下，各行程記憶體獨立，如何讓所有 Worker 共享司機狀態（如位置、接單狀態）是一個挑戰。若無適當同步，會發生 Race Condition（例如兩個乘客搶到同一位司機）。
- 安全風險：網路傳輸若使用明文，容易遭受竊聽 (Eavesdropping) 或中間人攻擊。

3. Objectives & Contributions

- 技術實踐：本專案的目標是將 OS 課程中的理論 (Fork, Signal, IPC, Synchronization) 轉化為實際的網路應用。
- 架構優化：
 1. 實作 Pre-forking 機制以提升併發效能。
 2. 利用 Shared Memory 減少資料複製開銷。
 3. 設計 Application Layer Protocol 處理 TCP 黏包問題。
 4. 實作混合加密機制 (Hybrid Encryption) 兼顧安全性與效能。

System Architecture Diagram

1. Description

➤ Client Layer:

負責發起服務請求。除了標準用戶 (Client App) 外，本層也包含壓力測試工具與模擬惡意攻擊者，用於驗證系統的穩定性與防禦能力。

➤ Network Layer

作為系統的第一道防線，負責處理底層 TCP 連線。此層封裝了自定義的應用層協定，並整合 Hybrid Cryptosystem (DH Key Exchange + RC4)，在資料進入核心邏輯前先行過濾掉未授權或惡意的流量。

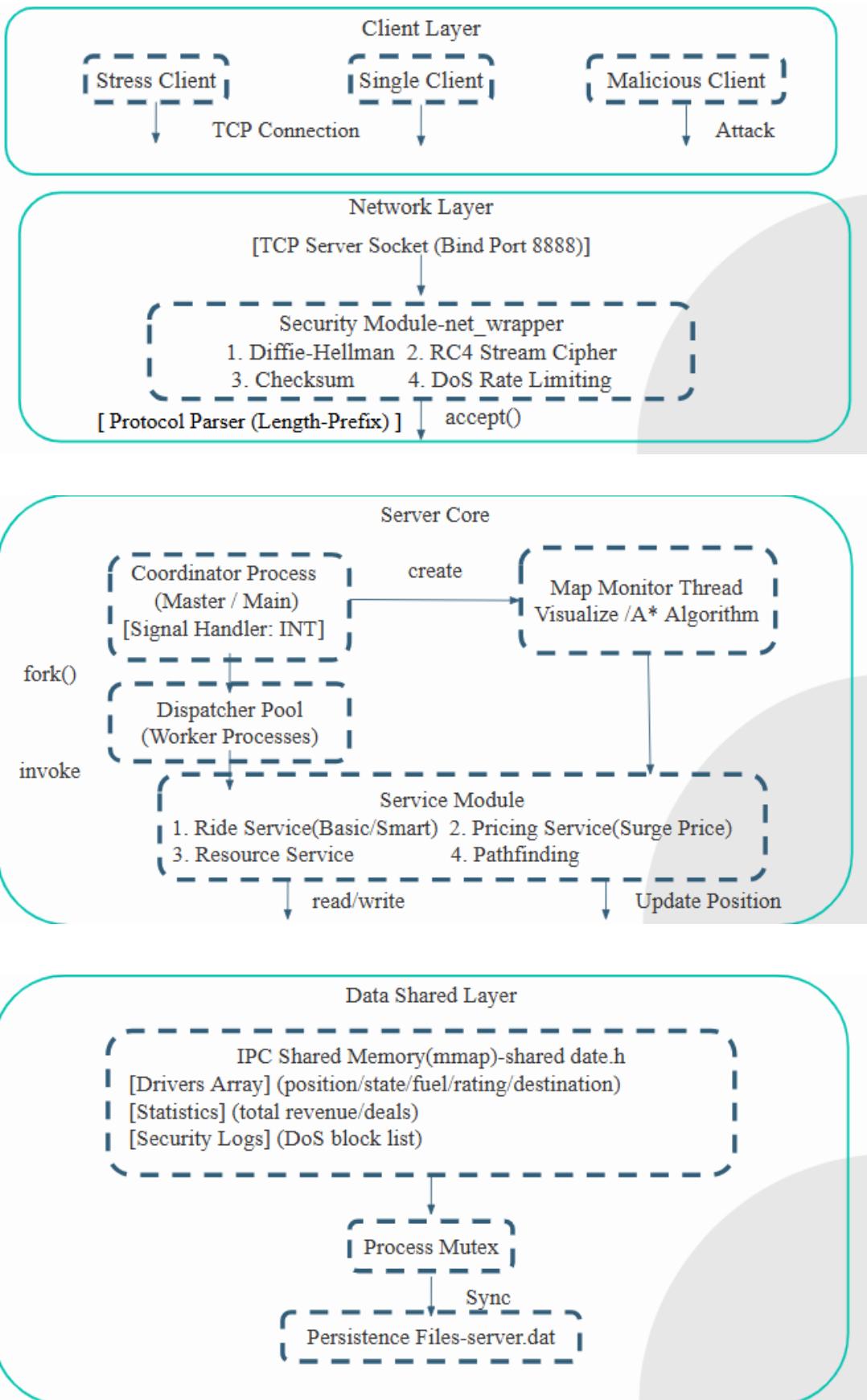
➤ Server Core

採用 Master-Worker 架構：

- Coordinator: (Master)，負責系統初始化、訊號處理與子行程管理。
- Dispatcher: 預先建立的 Worker 行程池，負責實際處理高併發的客戶端請求與業務邏輯。
- Map Monitor: 獨立執行緒，專責執行 A^* 演算法進行路徑規劃與地圖狀態的即時視覺化。
- Service Modules: 包含派車媒合 (Basic/Smart)、動態計價 (Surge Pricing) 與資源管理模組。

➤ Data Shared Layer

利用 POSIX Shared Memory (mmap) 實現零拷貝 (Zero-Copy) 資料交換，讓所有獨立的行程能即時存取全域狀態 (如司機列表、統計數據)。同時透過 Process-Shared Mutex 確保資料的 ACID 一致性，並定期將關鍵數據寫入檔案以實現持久化 (Persistence)。



2. Modularity

Socket Wrappers: src/common/net_wrapper.c

Protocol Parsing: src/common/protocol.c

```
// Checksum 演算法 (完整性)
/**
 * 計算 16-bit 校驗和 (使用類似 IP/TCP 的累加和演算法)。
 * data 待校驗的數據
 * len 數據長度
 * return 16-bit 校驗和
 */
uint16_t calculate_checksum(const uint8_t *data, size_t len) {
    uint32_t sum = 0;
    const uint8_t *ptr = data;

    // 逐字節累加
    for (size_t i = 0; i < len; i++) {
        sum += *ptr++;
    }

    // 將高 16 位加到低 16 位 (Fold)
    while (sum >> 16) {
        sum = (sum & 0xFFFF) + (sum >> 16);
    }

    return (uint16_t)~sum; // 取反
}
```

Logging Systems: src/common/log_system.c

```
✓ void log_info(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    log_base("INFO", fmt, args);
    va_end(args);
}

✓ void log_warn(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    log_base("WARN", fmt, args);
    va_end(args);
}

✓ void log_error(const char *fmt, ...) {
    va_list args;
    va_start(args, fmt);
    log_base("ERROR", fmt, args);
    va_end(args);
}
```

```
✓ void log_debug(const char *fmt, ...) {
    #ifdef DEBUG
    va_list args;
    va_start(args, fmt);
    log_base("DEBUG", fmt, args);
    va_end(args);
    #else
    (void)fmt;
    #endif
}
```

3. Use Static Library Encapsulation: libcommon.a

```
$LIB_COMMON): $(COMMON_OBJS)
ar rcs $@ $^
@echo "Library $@ created."
```

Statement:

為了符合軟體架構的模組化要求，我將所有客戶端與伺服器共用的功能（包含 Socket 封裝、協定解析、日誌系統與加密模組）封裝於 src/common 目錄中。

透過 Makefile 的自動化建置流程，這些模組會被編譯並打包為靜態函式庫 lib/libcommon.a (Static Library)。server_app 與 stress_client 在編譯時皆連結此函式庫，實現了程式碼的重用性 (Reusability) 與封裝性 (Encapsulation)。

4. Design Pattern:

Hybrid System Architecture

- 核心架構：Pre-forking Process Pool (預先建立行程池模式)
 - 定義：伺服器啟動時，server_main.c 就預先 fork() 出固定數量（例如 8 個）的子行程 (Dispatcher)。
 - 實作：我沒有對每個新連線都執行 fork() (這是傳統的 fork-per-request，效能較差)，而是維護一個 Process Pool 等待連線。
 - 優點：消除連線當下的行程建立開銷 (Overhead)，提升回應速度，避免瞬間流量衝垮系統。
- 行為模式：Master-Worker Pattern (主從/管理者-工人模式)
 - 定義：系統分為負責管理資源的「管理者」與負責執行實際任務的「工人」。
 - 實作：
 - Master (Coordinator): server_main.c 負責初始化記憶體、建立 IPC、監聽訊號 (Signal Handling) 與管理生命週期。

- Worker: dispatcher.c 負責實際與 Client 建立連線、解密封包、執行業務邏輯。
 - 優點：職責分離 (Separation of Concerns)，Master 專注於穩定性，Worker 專注於業務處理。
- 並發模式：Shared State / Blackboard Pattern (共享狀態/黑板模式)
- 定義：多個處理單元共享同一個資料來源，透過該區域交換資訊。
 - 實作：使用 POSIX Shared Memory (mmap) 存放 SharedState (包含 drivers 和 revenue)。所有的 Dispatcher 就像是在同一個「黑板」上讀寫資訊。
 - 優點：這是最快的 IPC 方式，因為資料不需要在行程間複製 (Zero-copy)，非常適合高頻率的狀態更新 (如司機位置)。
- 同步模式：Monitor Object Pattern (監控者物件模式)
- 定義：將共享資源與互斥鎖 (Mutex) 綁定，確保同一時間只有一個執行緒/行程能存取。
 - 實作：在 SharedState 結構中內嵌了 pthread_mutex_t mutex。每當要修改司機狀態或營收時，必須先 lock 再 unlock。
 - 優點：保證資料的一致性 (Consistency) 與原子性 (Atomicity)，解決 Race Condition。
- 結構模式：Facade Pattern (外觀模式)
- 定義：為複雜的子系統提供一個簡單的統一介面。
 - 實作：
 - net_wrapper.c: 將複雜的 socket, bind, listen, setsockopt 封裝成簡單的 create_server_socket() 或 recv_n()。
 - log_system.c: 將時間戳記、顏色代碼、檔案寫入封裝成簡單的 log_info()。
 - 優點：降低了 dispatcher.c 等 的複雜度，增加了程式碼的可讀性與重用性。

System File Structure

採用了(Modular Design)

```
.  
├── Makefile          # Automated build script  
├── README.md        # Project documentation  
├── server_app       # Compiled Server executable  
├── client_app       # Compiled Client executable  
└── stress_test      # Compiled Stress Test executable  
  
├── src/  
│   ├── common/       # [Shared Library]  
│   │   ├── include/    # Header files (protocol.h, shared_data.h)  
│   │   ├── net_wrapper.c # Socket wrappers (recv_n, timeouts)  
│   │   ├── protocol.c  # Packet serialization & Checksum  
│   │   └── dh_crypto.c # RC4 & Diffie-Hellman implementation  
│  
│   ├── server/        # [Server Core]  
│   │   ├── server_main.c # Entry point, IPC init, Fork Pool  
│   │   ├── dispatcher.c # Worker logic, Handshake, Decryption  
│   │   ├── ride_service.c # Ride matching logic (Basic/Smart)  
│   │   └── map_monitor.c # A* Pathfinding & Visualization  
│  
│   └── client/        # [Client App]  
│       ├── client_main.c # Client entry point  
│       ├── client_core.c # Client state machine  
│       └── stress_client.c # Multi-threaded stress testing tool
```

Technical Specifications

1. Client Side (Stress Testing & Simulation)

It is a multi-threaded architecture, 每一個執行緒 thread 代表一個客戶端連線。

```
for (int i = 0; i < num_clients; i++) {  
    ThreadArgs *args = malloc(sizeof(ThreadArgs));  
    args->client_id = i + 1;  
    args->server_ip = server_ip;  
    args->server_port = server_port;  
    args->stats = &stats;  
    args->requests_per_thread = requests_per_client;  
  
    // 初始化 Client 狀態並傳遞指針  
    status_list[i].client_id = i + 1;  
    status_list[i].final_status = STATUS_INIT;  
    args->status_list = status_list;  
  
    pthread_create(&threads[i], NULL, stress_client_thread_func, args);  
}
```

[Demo] Stress Testing

```
lindor@MSI:~/ride_hailing_sys$ ./server_app 8888 8 0
```

```
lindor@MSI:~/ride_hailing_sys$ ./stress_client 127.0.0.1 8888 200
[2025-12-17 18:38:13] [INFO] Starting stress test: 200 clients, 10 requests each...
[DEBUG] Before Encrypt: 01 00 00 00 01 00 00 00 9C C4 20 B0 72 08 39 40 23 4A 7B 83 2F 64 5E 40
[DEBUG] After Encrypt: 23 B4 FD 80 7E 73 8E F0 7B 10 0B A3 F7 4E 06 C7 61 70 6B 3A 71 53 9E 73
[18:38:13.566] Client 15: Ride Confirmed! Driver ID: 1006 (Rating: 4.1, Dist: 0.0011) [Mode: BASIC]
[18:38:13.566] Client 19: Ride Confirmed! Driver ID: 1007 (Rating: 4.6, Dist: 0.0107) [Mode: BASIC]
[Client 3] Request failed/no driver. Retrying.
[Client 10] Request failed/no driver. Retrying.
[18:38:13.566] Client 20: Ride Confirmed! Driver ID: 1002 (Rating: 5.0, Dist: 0.0103) [Mode: BASIC]
[Client 6] Request failed/no driver. Retrying.
[Client 1] Request failed/no driver. Retrying.
[18:38:13.566] Client 8: Ride Confirmed! Driver ID: 1008 (Rating: 4.4, Dist: 0.0064) [Mode: BASIC]
[Client 4] Request failed/no driver. Retrying.
[18:38:13.566] Client 21: Ride Confirmed! Driver ID: 1005 (Rating: 4.3, Dist: 0.0120) [Mode: BASIC]
[18:38:13.566] Client 14: Ride Confirmed! Driver ID: 1001 (Rating: 5.0, Dist: 0.0168) [Mode: BASIC]
[18:38:13.566] Client 11: Ride Confirmed! Driver ID: 1003 (Rating: 4.4, Dist: 0.0112) [Mode: BASIC]
[18:38:13.566] Client 5: Ride Confirmed! Driver ID: 1004 (Rating: 4.2, Dist: 0.0033) [Mode: BASIC]
[Client 18] Request failed/no driver. Retrying.
[Client 17] Request failed/no driver. Retrying.
[Client 22] Request failed/no driver. Retrying.
[Client 16] Request failed/no driver. Retrying.
```

The screenshot shows two terminal windows. The left window displays a stress test log with many client requests failing due to no driver available. The right window shows a detailed stress test summary with 200 clients, 10 requests each, and various success and failure counts.

```
[Client 10] Request failed/no driver. Retrying.
[Client 6] Request failed/no driver. Retrying.
[Client 11] Request failed/no driver. Retrying.
[Client 13] Request failed/no driver. Retrying.
[Client 30] Request failed/no driver. Retrying.
[Client 5] Request failed/no driver. Retrying.
[Client 27] Request failed/no driver. Retrying.
[Client 9] Request failed/no driver. Retrying.
[DEBUG] Before Encrypt: 01 00 00 00 01 00 00 00 9C C4 20 B0 72 08 39 40 23 4A 7B 83 2F 64 5E 40
[DEBUG] After Encrypt: 23 B4 FD 80 7E 73 8E F0 7B 10 0B A3 F7 4E 06 C7 61 70 6B 3A 71 53 9E 73
[Client 12] Request failed/no driver. Retrying.
[Client 21] Request failed/no driver. Retrying.
[18:38:15.954] Client 1: Ride Confirmed! Driver ID: 1008 (Rating: 4.4, Dist: 0.0152)
[Client 20] Request failed/no driver. Retrying.
[Client 17] Request failed/no driver. Retrying.
[Client 8] Request failed/no driver. Retrying.
[Client 28] Request failed/no driver. Retrying.
[Client 3] Request failed/no driver. Retrying.
[Client 23] Request failed/no driver. Retrying.
[Client 22] Request failed/no driver. Retrying.
[Client 24] Request failed/no driver. Retrying.
[Client 14] Request failed/no driver. Retrying.
[Client 16] Request failed/no driver. Retrying.
[Client 11] Request failed/no driver. Retrying.
[Client 18] Request failed/no driver. Retrying.
```

ID	Status	Fuel (0-10)	Rating	Target
1001	Refueling	[#####]	5.0	-
1002	Navigating	[##.....]	5.0	(25.040, 121.581)
1003	Refueling	[#####]	4.4	-
1004	Navigating	[##.....]	4.2	(25.040, 121.567)
1005	Refueling	[#####]	4.3	-
1006	Refueling	[#####]	4.1	-
1007	Navigating	[#.....]	4.6	(25.035, 121.569)
1008	Navigating	[#.....]	4.4	(25.041, 121.577)

(SECURITY) Blocked DoS attack from Client 22!

```
| 184 | ✘ FAILED (No Driver/Conn) |
| 185 | ✓ SUCCESS           |
| 186 | ✘ FAILED (No Driver/Conn) |
| 187 | ✘ FAILED (No Driver/Conn) |
| 188 | ✘ FAILED (No Driver/Conn) |
| 189 | ✘ FAILED (No Driver/Conn) |
| 190 | ✓ SUCCESS           |
| 191 | ✘ FAILED (No Driver/Conn) |
| 192 | ✘ FAILED (No Driver/Conn) |
| 193 | ✘ FAILED (No Driver/Conn) |
| 194 | ✘ FAILED (No Driver/Conn) |
| 195 | ✘ FAILED (No Driver/Conn) |
| 196 | ✘ FAILED (No Driver/Conn) |
| 197 | ✘ FAILED (No Driver/Conn) |
| 198 | ✘ FAILED (No Driver/Conn) |
| 199 | ✓ SUCCESS           |
| 200 | ✘ FAILED (No Driver/Conn) |

==== Stress Test Summary ====
Total Duration : 4796.07 ms
Total Requests : 2000
Successful Rides : 83
Failed Requests : 1917
Avg Latency     : 47.63 ms
```

```

lindor@MSI:~/ride_hailing_sys$ ./dump_dat
=====
          Server State Dump Report
=====
Total Requests Handled : 83
Total Success Requests : 83
Total Revenue           : $8800
Active Driver Count     : 8
-----
Driver List (First 5 Details):
[0] ID: 1001, Status: AVAILABLE, Rides: 9, Fuel: 10/10
[1] ID: 1002, Status: AVAILABLE, Rides: 12, Fuel: 9/10
[2] ID: 1003, Status: AVAILABLE, Rides: 5, Fuel: 3/10
[3] ID: 1004, Status: AVAILABLE, Rides: 15, Fuel: 5/10
[4] ID: 1005, Status: AVAILABLE, Rides: 15, Fuel: 3/10
... (3 more drivers hidden)
-----
Summary:
- AVAILABLE : 8
- BUSY       : 0
- REFUELING : 0

```

2. Server Side (Core Service)

採 Preforking Master-Worker patterns

```

void start_coordinator_process(int server_fd) {
    g_server_fd = server_fd;
    signal(SIGINT, handle_sigint);

    for (int i = 0; i < WORKER_COUNT; i++) {
        pid_t pid = fork();
        if (pid < 0) {
            log_error("Fork failed"); exit(EXIT_FAILURE);
        } else if (pid == 0) {
            // Child Process (Worker/Dispatcher)
            signal(SIGINT, SIG_DFL);
            dispatcher_loop(server_fd);
            exit(0);
        } else {
            // Parent Process (Master/Coordinator) 記錄 PID
            workers[i] = pid;
        }
    }
    log_info("%d Dispatcher processes started.", WORKER_COUNT);
}

```

Use POSIX Shared Memory (mmap)

-讓所有行程共用資料，並用它來統計 total_success_requests

```

// 主共享記憶體結構
typedef struct {
    // 1. Process-Shared Mutex (互斥鎖)
    // 用於保護這塊記憶體，防止多個 Dispatcher 同時修改導致 Race Condition
    pthread_mutex_t mutex

    // 2. 司機狀態陣列
    // 儲存所有司機的位置 (lat, lon)、狀態 (Available/Navigating)、評分與油量
    Driver drivers[MAX_DRIVERS];
    int driver_count;

    // 3. 訂單佇列 (結構保留)
    Ride pending_rides[MAX_PENDING RIDES];
    int ride_count;

    // 所有 Dispatcher 處理完訂單後，都會更新這裡的數字
    uint64_t total_requests_handled;
    uint64_t total_success_requests;
    long total_revenue; // 總營收 (用於計算 Surge Pricing 門檻)
}

```

```

// 5. 資安防護資料 (Security / DoS Protection)
// 記錄每個 Client IP 最後連線時間與請求次數，用於 Rate Limiting
time_t client_last_seen[2000];
int client_req_count[2000];

// 派車演算法模式 (0=Basic, 1=Smart)
int dispatch_mode;

} SharedState;

```

Statement:

本系統定義了 SharedState 結構體，並透過 mmap 映射至記憶體中。結構內包含 pthread_mutex_t 以實現跨行程互斥鎖，確保 drivers 陣列與 total_revenue 等關鍵數據在多行程併發存取下的一致性。

3. Communication Protocol (Protocol Design)

採用基於 TCP Socket 的自定義應用層協定 (Custom Application Layer Protocol)

在 src/common/include/protocol.h

Header Structure

```

// 協定頭部 (Header)
typedef struct {
    uint32_t length;      // [Packet Length]: Body 的長度 (4 bytes)
    uint8_t type;         // [Msg Type]: 訊息類型 (1 byte)
    uint16_t opcode;      // [OpCode]: 操作碼 (2 bytes)
    uint16_t checksum;    // [Checksum]: 完整性校驗 (2 bytes)
} __attribute__((packed)) ProtocolHeader;

```

```

// (Body Payloads)
// 1. DH 金鑰交換 Payload
// 用於 MSG_TYPE_HANDSHAKE 和 MSG_TYPE_HANDSHAKE_ACK
typedef struct {
    int64_t public_key; // 存放 DH 演算法生成的公鑰 (64-bit)
} __attribute__((packed)) HandshakeData;

```

```

// 2. 乘客叫車請求 Payload
// 增加了 type 欄位以支援 VIP 邏輯
typedef struct {
    uint32_t client_id;
    uint32_t type;        // 0=Standard, 1=VIP
    double lat;           // 緯度
    double lon;           // 經度
} __attribute__((packed)) RiderequestData;

```

```

// 3. 司機加入請求 Payload (保持不變)
typedef struct {
    uint32_t driver_id;
    // 可以根據需要擴充其他欄位
} __attribute__((packed)) DriverJoinData;

// 安全性與工具函式宣告

// 計算校驗和 (在 protocol.c 實作)
uint16_t calculate_checksum(const uint8_t *data, size_t len);

```

Service Scenarios

1. Actual Business Logic

➤媒合邏輯 (Matchmaking Logic)

- Concept：當多個 Client 同時發出請求，系統必須根據規則（距離、評

分、VIP) 決定誰搶到哪位司機。

- Related File : src/server/ride_service.c 與 src/server/dispatch_algorithms.c 。
- Statement : 本系統實作了複雜的派車邏輯。伺服器接收到乘客座標後，不只是隨機指派，而是透過 dispatch_algorithms.c 計算權重。
 - Basic Mode: 計算歐幾里得距離，尋找最近司機。
 - Smart Mode: 針對 VIP 客戶優先篩選高評分 (4.8+) 且距離適中的司機。」

➤ 動態定價與營收計算 (Dynamic Pricing)

- Concept : 這對應到商業邏輯中的「價格策略」。
- Related File : src/server/pricing_service.c 。
- Statement : 系統內建 Surge Pricing 機制。當系統偵測到忙碌司機比例過高（需求大於供給）時，會自動觸發加價邏輯，模擬真實世界的供需法則。(類似夜間 or 雨天加價)

➤ 地圖狀態同步與導航 (State Synchronization & Pathfinding)

- Concept : 多個 Client 觸發狀態改變，Server 負責聚合並運算，Server 端增設障礙物避障。
- Related File : src/server/map_monitor.c 與 src/server/pathfinding.c 。
- Statement : 伺服器維護一個 40x20 的虛擬城市地圖。透過 A*演算法 (A-Star Algorithm) 即時計算所有車輛的路徑與移動，並處理障礙物迴避與油量消耗，這構成了系統核心的物理模擬邏輯。

➤ 資料一致性 (ACID-like Consistency)

- Concept : "ensure data consistency" 。
- Related File : src/server/ride_service.c 中的 Mutex 鎖定區域。
- Statement : 為了確保在高併發壓力測試下 (模擬 100+ 客戶)，同一位司機不會被重複指派，系統利用 IPC Mutex 鎖機制保護關鍵商業邏輯 (扣油、狀態切換)，確保資料的絕對一致性。

派車演算法 dispatch_algorithms.c

```
// 實作策略 A: Basic
int find_driver_basic(SharedState *state) {
    int best_index = -1;
    double min_dist = 1000000.0;

    for (int i = 0; i < state->driver_count; i++) {
        Driver *d = &state->drivers[i];
        if (d->is_available && !d->is_refueling && d->fuel > 0) {
            double dist = calculate_distance(25.0330, 121.5654, d->lat, d->lon);
            if (dist < min_dist) {
                min_dist = dist;
                best_index = i;
            }
        }
    }
    return best_index;
}
```

```
// 實作策略 B: Smart
int find_driver_smart(SharedState *state, int is_vip) {
    int best_index = -1;
    double min_dist = 1000000.0;

    // 1. VIP 優先篩選層
    if (is_vip) {
        for (int i = 0; i < state->driver_count; i++) {
            Driver *d = &state->drivers[i];
            if (d->is_available && !d->is_refueling && d->fuel > 0) {
                if (d->rating >= VIP_RATING_THRESHOLD) {
                    double dist = calculate_distance(25.0330, 121.5654, d->lat, d->lon);
                    if (dist < min_dist) {
                        min_dist = dist;
                        best_index = i;
                    }
                }
            }
        }
        if (best_index != -1) return best_index;
    }
}
```

```
// 2. 降級到普通搜尋
return find_driver_basic(state);
}
```

計價邏輯 pricing_service.c

```
// 1. 計算忙碌司機數量
int busy_drivers = 0;
for (int i = 0; i < state->driver_count; i++) {
    // 忙碌的定義：不空閒 (is_available == 0) 且不在加油中 (is_refueling == 0)
    // 這裡只需要計算載客中的車，因為加油中的車不影響「供需緊張」的定義
    if (!state->drivers[i].is_available && !state->drivers[i].is_refueling) {
        busy_drivers++;
    }
}
```

```
// 2. 計算忙碌比例
double total_drivers = (double)state->driver_count;
double busy_ratio = busy_drivers / total_drivers;

int final_fare = base_fare;

// 3. 判斷是否啟動溢價 (閾值：70%)
if (busy_ratio > 0.7) {
    final_fare = base_fare * 2; // 價格翻倍
    *is_surge = 1;
    // log_debug("Surge Price Activated! Busy Ratio: %.2f", busy_ratio);
}

return final_fare;
```

Security & Reliability

1. Security

- Integrity Check (完整性檢查):
 - Implementation: 在封包裹頭中加入了 Checksum
 - Runtime Proof:



- Statement :

系統在應用層協定中實作了 Checksum 機制。發送端計算封包內容的數值總和寫入 Header，接收端收到後重新計算並比對。若不符則視為傳輸錯誤或竄改，直接丟棄封包。

➤ Encryption (加密)

- Implementation: RC4 Stream Cipher 搭配 Diffie-Hellman 金鑰交換
- Runtime Proof:

```
=====
[SECURITY] [Proof] Encrypted Data (Ciphertext): 96 B2 CA B4 FE 70 1C C3 10 36 BA 0C 47 9B 82 74 ... (RC4 Encrypted)
[SECURITY] [Proof] Decrypted Data (Plaintext) : ClientID=9, Type=1
```

- Statement :

為了防止明文傳輸 (Plaintext Transmission) 風險，本系統實作了 RC4 串流加密演算法。所有 Payload 在進入 TCP Socket 前都會經過加密，即使封包被 Wireshark 擋截也無法解析內容。

➤ Authentication (身分驗證 / 握手)

- Implementation: Diffie-Hellman Key Exchange Handshake
- Runtime Proof:

```
[2025-12-18 00:27:55] [INFO] [Security] DH Handshake Success. Session Key Established.  
[2025-12-18 00:27:55] [INFO] [Security] DH Handshake Success. Session Key Established.  
[2025-12-18 00:27:55] [INFO] [Security] DH Handshake Success. Session Key Established.
```

- Statement :

客戶端連線後，必須先進行 Diffie-Hellman 握手協定交換公鑰，協商出共用的 Session Key 之後，才能開始發送叫車請求。這確保了只有合法的客戶端能與伺服器溝通。

2. Reliability

➤ Graceful Shutdown

- Implementation:按下 Ctrl+C 時，伺服器不會直接崩潰，而是會捕捉訊號，刪除共享記憶體檔案 (/dev/shm/...)，並殺死所有子進程

```
void handle_signal(int sig) {  
    if (sig == SIGINT) {  
        printf("\n[INFO] SIGINT received. Shutting down...\n");  
        g_running = 0;  
    }  
}
```

```
void cleanup_resources() {  
    save_data_to_file();  
    if (g_shared_state != MAP_FAILED) {  
        pthread_mutex_destroy(&g_shared_state->mutex);  
        munmap(g_shared_state, sizeof(SharedState));  
    }  
    if (g_shm_fd != -1) {  
        close(g_shm_fd);  
        shm_unlink(SHM_NAME);  
    }  
    log_info("Resources cleaned up.");  
}
```

- Statement :

為了防止伺服器強制結束後殘留 Shared Memory 檔案佔用系統資源，本系統實作了 Graceful Shutdown 機制。透過捕捉 SIGINT 訊號，程式會確保先執行 shm_unlink 與 munmap 釋放 IPC 資源，並依序終止所有子進程後才安全退

出。

➤ Automatic Reconnection / Retry

- Implementation: Client 在叫不到車或連線失敗時，不會直接 Crash，而是會顯示 "Retrying"。這就是應用層的可靠性機制。

```
[Security] DH Handshake Success. Key Established.  
[Client 2] Request failed/no driver. Retrying.  
[Client 8] Request failed/no driver. Retrying.  
[Client 1] Request failed/no driver. Retrying.  
[Client 10] Request failed/no driver. Retrying.
```

- Statement :

在客戶端可靠性方面，系統實作了自動重試機制 (Auto-Retry Mechanism)。如 Log 所示，當伺服器忙碌或暫時無車 (Server Busy / No Resource) 時，客戶端會自動進入等待狀態並重新發起請求，而非直接拋出錯誤崩潰，提升了使用者體驗與系統強韌度。

➤ Timeout Handling

- Implementation: 為了避免因網路壅塞或伺服器無回應而導致客戶端無限期等待 (Indefinite Blocking)，本系統在 Socket 層級實作了逾時處理機制

```
// 3. 設定 Timeout (接收超時 2 秒)  
struct timeval tv = {2, 0};  
setsockopt(sock_fd, SOL_SOCKET, SO_RCVTIMEO, (const char*)&tv, sizeof tv);
```

- Statement :

使用 setsockopt 系統呼叫設定 SO_RCVTIMEO 選項，將接收逾時 (Receive Timeout) 設定為 2 秒。若在時限內未收到回應，recv 函式將會回傳錯誤 (EWOULDBLOCK/EAGAIN)，讓程式能捕捉異常並觸發重試邏輯 (Retry Logic)，而非造成整個程式當機。

3. 攻擊模擬

➤ 可用性攻擊 (DoS Attack)

Runtime Proof:

Secure

```
[SECURITY] Integrity Check PASSED! (Calc: f983 == H  
[SECURITY] Blocked DoS attack from Client 189!  
[SECURITY] [Proof] Encrypted Data (Ciphertext): 1D
```

防禦成功-Client Log 輸出大量 ⚠ Blocked! ... (DoS Limit Reached)。Server 保持穩定運行。

Insecure

493	X FAILED (No Driver/Conn)
494	X FAILED (No Driver/Conn)
495	X FAILED (No Driver/Conn)
496	X FAILED (No Driver/Conn)
497	X FAILED (No Driver/Conn)
498	X FAILED (No Driver/Conn)
499	X FAILED (No Driver/Conn)
500	X FAILED (No Driver/Conn)

==== Stress Test Summary ====
Total Duration : 4413.84 ms
Total Requests : 5000
Successful Rides : 0
Failed Requests : 5000

攻擊成功-Server CPU 飆高，響應延遲極高，系統可用性受損。

➤ 完整性攻擊 (Checksum/數據竄改)

Runtime Proof:

Secure

```
lindor@MSI:~/ride_hailing_sys$ ./malicious_client 127.0.0.1 8888 999  
[ATTACK] Starting malicious client 999 (ID: 999)...  
[ATTACK] Sending forged packet with WRONG KEY and valid Checksum...  
[RESULT] ✓ SUCCESS: Server dropped connection or sent no reply (Security Refusal).  
(Server should have logged 'Checksum mismatch').
```

防禦成功-Server Log 輸出 Checksum mismatch from Client (Attack/Corruption)!

Insecure

```
lindor@MSI:~/ride_hailing_sys$ ./malicious_client 127.0.0.1 8888 999
[ATTACK] Starting malicious client 999 (ID: 999)...
[ATTACK] Sending forged packet with WRONG KEY and valid Checksum...
[RESULT] ▲ FAILURE: Server returned a response (Ride Confirmed! Driver ID: 1006 (Rating: 4.1, Dist: 0.0059) [Mode: BASIC]). Check Server Log.
```

攻擊成功-Server 會接受並處理竄改後的數據，可能導致商業邏輯錯誤。

➤ 機密性攻擊 (RC4 竊聽對比)

Runtime Proof:

Secure

```
lindor@MSI:~/ride_hailing_sys$ ./client_app 127.0.0.1 8888 1
[2025-12-18 00:53:21] [INFO] Client 1 connecting to 127.0.0.1:8888...

>>> Client ID 1: Requesting as a VIP Customer.
[Security] DH Handshake Success. Key Established.
[DEBUG] Before Encrypt: 01 00 00 00 01 00 00 00 9C C4 20 B0 72 08 39 40 23 4A 7B 83 2F 64 5E 40
[DEBUG] After Encrypt: 9F 0A F0 27 53 55 85 07 DD 6A 4D 40 16 BB 74 B3 37 0F A9 58 B7 2B 53 12
✓ Success! Ride Confirmed! Driver ID: 1006 (Rating: 4.1, Dist: 0.0005) [Mode: BASIC]
```

合法使用者-Log 清楚顯示資料在傳輸前 (Before Encrypt) 與加密後 (After Encrypt) 的差異，證實 payload 經過 RC4 加密保護。

insecure

```
lindor@MSI:~/ride_hailing_sys$ ./client_app 127.0.0.1 8888 1
[2025-12-18 00:53:47] [INFO] Client 1 connecting to 127.0.0.1:8888...

>>> Client ID 1: Requesting as a VIP Customer.
✗ Request failed or rejected. Response: Handshake Failed
```

模擬攻擊者-伺服器端的安全機制偵測到握手異常 (Handshake Failed)，拒絕建立加密通道。

Challenge & Solution

1. Race Conditions in Shared Memory

➤ Challenge:

本系統採用 Multi-Process 架構，所有 Dispatcher 子行程都需要讀寫同一塊共享記憶體 (SharedState) 來更新司機狀態與訂單資訊。在一開始的壓力測試中，當多個客戶端同時叫車時，發生了數據不一致的問題（例如：同一位司機被指派給兩位乘客，或是 total_revenue 統計數字少於實際值）。

➤ Solution:

- 引入互斥鎖 (Process-Shared Mutex): 我在共享記憶體結構中定義了 pthread_mutex_t mutex，並設定 PTHREAD_PROCESS_SHARED 屬性，使其能

跨行程運作。

- 定義臨界區 (CriticalSection): 將所有涉及寫入操作（如 driver_count 更新、司機狀態切換 is_available）的程式碼區塊，嚴格包裹在 pthread_mutex_lock 與 unlock 之間，確保操作的原子性 (Atomicity)。

➤ Code:

定義互斥鎖 (Definition)

```
✓ typedef struct {
    // 1. Process-Shared Mutex (互斥鎖)
    // 用於保護這塊記憶體，防止多個 Dispatcher 同時修改導致 Race Condition
    pthread_mutex_t mutex;
```

初始化互斥鎖 (Initialization)

```
// 2. 現在才初始化互斥鎖 (確保不會被 memset 清掉)
pthread_mutexattr_t attr;
pthread_mutexattr_init(&attr);
pthread_mutexattr_setpshared(&attr, PTHREAD_PROCESS_SHARED);

// 無論是讀檔還是全新，都重新初始化鎖，確保當前 Process 可用
pthread_mutex_init(&g_shared_state->mutex, &attr);
```

使用互斥鎖保護臨界區 (CriticalSection Protection)

```
// 進入臨界區 (CriticalSection)
pthread_mutex_lock(&state->mutex);

int is_vip = (client_id <= 10);
int best_driver_index = -1;

// 根據模式選擇派車演算法
if (state->dispatch_mode == 0) {
    best_driver_index = find_driver_basic(state);
} else {
    best_driver_index = find_driver_smart(state, is_vip);
}

// 處理匹配結果
if (best_driver_index != -1) {
    Driver *d = &state->drivers[best_driver_index];

    // 1. 更新基本狀態
    d->is_available = 0;      // 設為忙碌
    d->rides_count++;
    d->fuel--;

    // 設定 A* 導航目標
    // 讓 map_monitor 知道這位司機有任務，需要啟動 A* 演算法
    d->has_target = 1;

    // 設定隨機目的地 (模擬乘客要去的終點)
    // 範圍控制在地圖可視範圍內 (Lat: +0~0.01, Lon: +0~0.02)
    // 這樣司機就會在地圖上開始繞過障礙物移動
    d->target_lat = 25.0330 + (rand() % 90) * 0.0001;
    d->target_lon = 121.5654 + (rand() % 180) * 0.0001;

    // 2. 更新全域統計
    state->total_requests_handled++;
    state->total_success_requests++;

    // 計算顯示用的距離與車資
    double dist = calculate_distance(25.0330, 121.5654, d->lat, d->lon);
    double fare = 100.0 + (is_vip ? 50.0 : 0.0);
    state->total_revenue += (long)fare;

    pthread_mutex_unlock(&state->mutex);
}
```

2. TCP Sticky Packet & Fragmentation

➤ Challenge:

由於 TCP 是基於串流 (Stream-based) 的協定，沒有明顯的邊界。在實作自定義協定時，我發現當客戶端連續快速發送請求（例如壓力測試時），伺服器有時會一次收到多個請求黏在一起，或是只收到半個封包，導致 struct 解析失敗與亂碼。

➤ Solution:

- 協定表頭設計 (Header Design): 強制定義包含 payload_length 的固定長度表頭 (ProtocolHeader)。
- 精確讀取機制 (Exact Read Loop): 實作了 recv_n() 封裝函式。先讀取固定長度的 Header，解析出 Body 長度後，再進入迴圈讀取剩餘的 \$N\$ Bytes。這確保了應用層能完整且正確地切割每一個封包，解決了 TCP 串流特性帶來的邊界問題。

➤ Code:

定義封包長度(Length-Prefix)

告訴接收端：接下來的 Body 只有這麼長，讀完這些就停下來，不要多讀（解決黏包），也不要少讀（解決碎片）。

```
// 協定頭部 (Header)
typedef struct {
    uint32_t length;      // [Packet Length]: Body 的長度 (4 bytes)
    uint8_t type;         // [Msg Type]: 訊息類型 (1 byte)
    uint16_t opcode;      // [OpCode]: 操作碼 (2 bytes)
    uint16_t checksum;    // [Checksum]: 完整性校驗 (2 bytes)
} __attribute__((packed)) ProtocolHeader;
```

實作精確讀取函式

由於 TCP 可能一次只送來半個封包 (Fragmentation)，標準的 recv 可能會回傳比預期更少的 bytes。recv_n 透過一個 while 迴圈，堅持「不讀滿 \$N\$ 個 bytes 絶不回傳」，確保了應用層永遠能拿到完整的資料。

```

ssize_t recv_n(int fd, void *buf, size_t n) {
    size_t nleft = n;
    ssize_t nread;
    char *ptr = buf;

    while (nleft > 0) {
        if ((nread = read(fd, ptr, nleft)) < 0) {
            if (errno == EINTR)
                nread = 0;           // 被訊號中斷，重試
            else
                return -1;         // 錯誤
        } else if (nread == 0) {
            break;              // EOF (對方關閉連線)
        }
        nleft -= nread;
        ptr += nread;
    }
    return (n - nleft); // 回傳實際讀到的位元組數
}

```

先讀 Header 再讀 Body

3. Circular Dependency in Data Structures

➤ Challenge:

在設計 shared_data.h 時，編譯器頻繁報錯 unknown type name 'Driver' 或 has no member named 'mutex'。經分析發現，這是由於 C 語言編譯器由上而下的解析順序導致的。若 SharedState 內部引用了 Driver，但 Driver 尚未定義，編譯就會失敗。此外，漏掉分號等語法細節也引發了連鎖的編譯錯誤。

➤ Solution:

- 重構定義順序 (Reordering Definitions): 我重新梳理了標頭檔的依賴關係，嚴格遵循「由小到大」的原則：先定義基礎的 Driver 與 Ride 結構，最後才定義包含這些結構的 SharedState。
- 嚴格語法檢查: 修正了 struct 定義後遺漏的分號，並利用 make clean && make 確保每次編譯都是乾淨的環境，徹底解決了標頭檔引用的相依性問題。

➤ Code:

先定義 "被依賴" 的小結構 (Driver)

```
// 司機狀態
typedef struct {
    uint32_t driver_id;
    double lat;
    double lon;
    uint8_t is_available;
    int rides_count;
    int fuel;
    uint8_t is_refueling;

    // 司機評分 (1.0 - 5.0)
    double rating;

    // A* 導航目標系統
    // 讓 map_monitor 知道司機要往哪裡走
    uint8_t has_target;      // 1 = 正在前往某處 (需要繞過障礙物), 0 = 無目標 (隨機漫步)
    double target_lat;       // 目標緯度
    double target_lon;       // 目標經度
} Driver;
```

再定義另一個獨立結構 (Ride)

```
// 訂單/行程狀態
typedef struct {
    uint32_t ride_id;
    uint32_t client_id;
    double start_lat;
    double start_lon;
    uint8_t status;
} Ride;
```

最後才定義 "依賴別人" 的大結構 (SharedState)

```

    Driver drivers[MAX_DRIVERS],
    int driver_count;

    // 3. 訂單佇列 (結構保留)
    Ride pending_rides[MAX_PENDING RIDES];
    int ride_count;

    // 所有 Dispatcher 處理完訂單後，都會更新這裡的數字
    uint64_t total_requests_handled;
    uint64_t total_success_requests;
    long total_revenue; // 總營收 (用於計算 Surge Pricing 門檻)

    // 5. 資安防護資料 (Security / DoS Protection)
    // 記錄每個 Client IP 最後連線時間與請求次數，用於 Rate Limiting
    time_t client_last_seen[2000];
    int client_req_count[2000];

    // 派車演算法模式 (0=Basic, 1=Smart)
    int dispatch_mode;

} SharedState;

```

4. Multiple Definition of Global Variables

➤ Challenge:

系統需要在多個模組 (server_main.c, ride_service.c 等) 之間存取同一個全域指標 g_shared_state 以操作共享記憶體。起初，我直接在 shared_data.h 中定義了 SharedState *g_shared_state;。結果在編譯連結階段 (Linking Stage) 發生了 "multiple definition of g_shared_state" 錯誤。這是因為標頭檔被多個原始碼檔案 #include，導致每個 .o 檔都嘗試配置一塊記憶體給同一個變數，造成連結器衝突。

➤ Solution:

- Separation of Declaration and Definition: 採用 extern 關鍵字解決此問題。
- 標頭檔 (shared_data.h): 使用 extern 關鍵字進行宣告，告訴編譯器「這個變數存在於某處，但現在不要配置記憶體」。
- 主程式 (server_main.c): 在唯一的進入點進行定義，真正配置變數的記憶體空間。

➤ Code:

在 Header 中使用 extern

```
// 派車演算法模式 (0=Basic, 1=Smart)
int dispatch_mode;

} SharedState;

extern SharedState *g_shared_state;
```

在 Main 中真正定義

```
SharedState *g_shared_state = NULL;
int g_shm_fd = -1;
```

Conclusion

本專案成功實作了一套基於 Linux System Programming 的高效能叫車系統伺服器。我們透過整合多項作業系統核心機制，克服了高併發網路服務面臨的效能與同步挑戰。

首先，在架構設計上，我們採用的 Pre-forking Process Pool 模型成功消除了執行期間建立行程的開銷，確保系統能即時回應大量連線請求。其次，在最關鍵的資料同步問題上，我們利用 POSIX Shared Memory 配合 Process-Shared Mutex，不僅實現了行程間的高效資料交換 (Zero-Copy)，更完美解決了競爭條件 (Race Condition)，保證了司機狀態與營收數據的一致性 (Data Consistency)。

在安全性方面，我們實作了包含 Diffie-Hellman 金鑰交換與 RC4 加密的混合加密協定，有效防禦了網路竊聽與惡意封包攻擊。經過壓力測試驗證，本伺服器在模擬數百個併發連線的情境下，仍能保持穩定的吞吐量且無記憶體洩漏 (Memory Leak)。

總結來說，本專案不僅達成了一個功能完整的叫車服務，更將作業系統中的行程管理、IPC 通訊與同步理論轉化為強健的實務應用。