COMP 512 Programming Assignment 1

Akash Lanard Student ID: 261239623

Kazi Ashhab Rahman Student ID: 261086447

Group 10

RMI Architecture and Design

Our implementation distributes the travel reservation system across five components: a client and four ResourceManager instances running on separate machines. Three specialized resource managers handling flights, cars, and rooms respectively, plus a middleware resource manager that coordinates between them. All four server components implement the `IResourceManager` interface and the Middleware extends the base `ResourceManager` class.

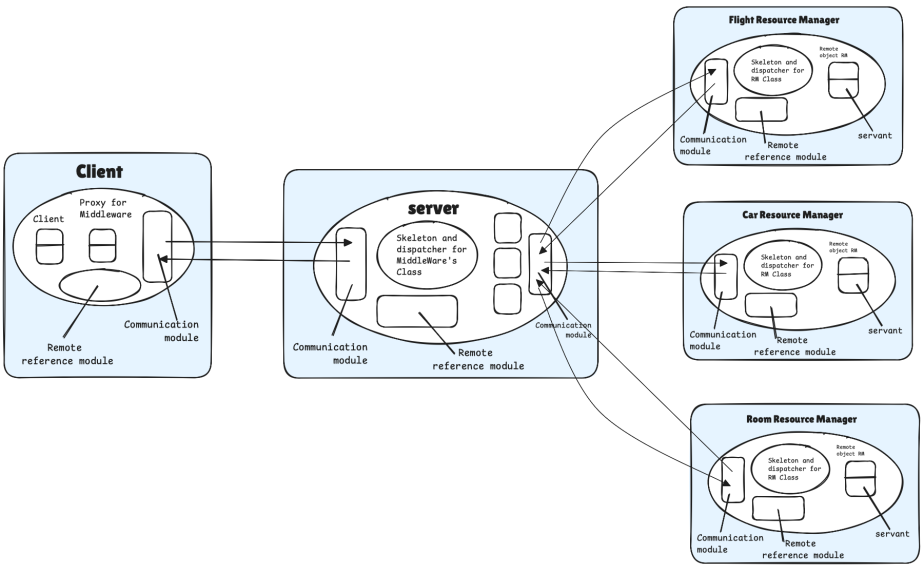

Figure 1: System architecture of our RMI implementation.

Architectural Decisions

The middleware serves a dual role: it directly manages customer data using inherited `ResourceManager` functionality while acting as a coordinator that delegates resource operations to specialized servers. This design was chosen for three reasons.

Firstly, centralizing customer data in the middleware provides a single source of truth for billing and reservation tracking, thus avoiding the consistency problems inherent in replicating customer data across multiple servers.

Secondly, extending `ResourceManager` allows the middleware to inherit all customer management methods such as `newCustomer`, `queryCustomerInfo` without modification, minimizing code duplication.

Thirdly, the middleware can selectively override only resource-related methods (`addFlight`, `addCars`, `addRooms`) to delegate to the appropriate specialized server.

Each specialized **`ResourceManager`** (**`Flight`**, **`Car`**, **`Room`**) stores only its resource inventory. The client connects exclusively to the middleware, which maintains the abstraction that the system is a single unified service despite being distributed across four servers.

Reservation Coordination

Coordinating reservations across distributed components required careful sequencing. When a client requests a reservation, the middleware executes the following algorithm:

1. Verify the customer exists in local middleware storage (customers are never stored in individual RMs initially)
2. Query the appropriate RM for resource availability and price
3. If available, create a temporary reservation record in the target RM
4. Delegate the reservation call to the RM, which decrements inventory
5. Update the customer's bill in middleware storage with the reservation details
6. Return success to the client

This approach ensures customers exist in middleware for billing while creating minimal temporary customer records in individual RMs only when reservations are made.

Bundle Implementation

The bundle operation ensures that multiple reservations across different RMs are handled as a single atomic action. Our implementation follows a two-phase process. In the first phase (validation), the middleware queries each RM to confirm that the requested resources exist and are available, while also collecting their prices. If any request fails, the entire operation is aborted before any reservations are made. In the second phase (reservation), once all checks succeed, the middleware proceeds to call the existing **`reserveFlight`**, **`reserveCar`**, and **`reserveRoom`** methods in sequence. This builds directly on the reservation logic already in place, ensuring consistency while extending it to support multi-resource bundles.

TCP Architecture and Design

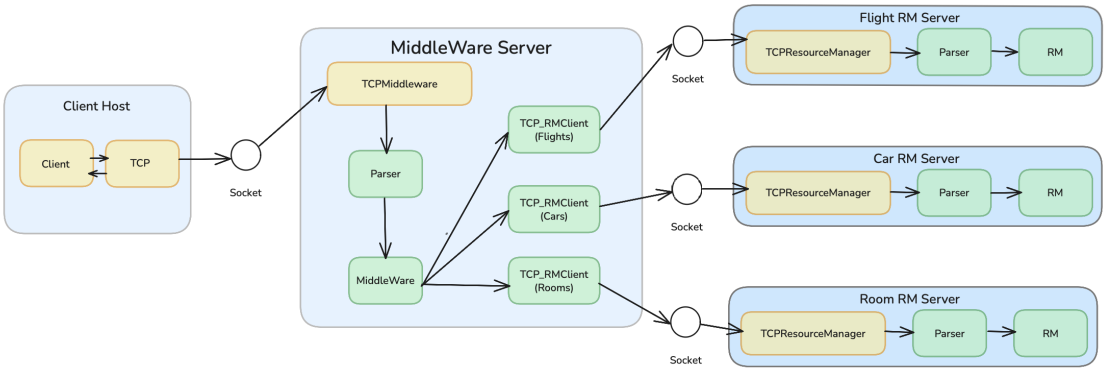

Figure 2: System architecture of our TCP implementation.

Our TCP implementation replicates the distributed architecture from RMI but replaces all inter-component communication with TCP sockets and a custom JSON-based protocol. The system consists of five processes: a client and four servers (**`Middleware`**, **`Flight RM`**, **`Car RM`**, **`Room RM`**), each communicating through persistent TCP socket connections.

Message Protocol Design

Designing a robust message protocol was the most critical TCP implementation decision. We chose JSON for three reasons: it's human-readable for debugging, self-describing (no need for separate protocol documentation), and straightforward to parse without external libraries. Each request follows this structure:

```
{"method": "reserveFlight", "args": [customerID, flightNumber]}
```

Responses include status, the actual response value, and optional error messages:

```
{"status": "ok", "response": true}
{"status": "failed", "message": "Customer doesn't exist"}
```

We implemented a custom JSON parser rather than using external libraries because the assignment restricts dependencies and our needs are simple (no nested objects beyond arrays). The parser handles string escaping, primitive types (int, boolean), and string arrays for bundle operations.

Connection Management Architecture

The TCP implementation maintains different connection strategies for different components:

Client-to-Middleware: The client establishes a single blocking socket connection to the middleware on startup. The client sends a request, waits for the response, then sends the next request. Connection failures are fatal—the client terminates and must be restarted. This blocking behavior simplifies the client implementation since there's no need to match asynchronous responses to requests.

Middleware-to-ResourceManagers: The middleware maintains three persistent connections through `TCPResourceMa` wrapper objects, one for each RM. These connections are established during middleware initialization and persist for the middleware's lifetime. Critically, these connections implement automatic reconnection: if a send operation throws an `IOException`, the wrapper closes the failed socket, creates a new connection, and retries the operation once. This handles transient network issues or RM restarts without requiring middleware restart.

Clients-to-Middleware (Server Side): The middleware runs a `ServerSocket` that accepts multiple concurrent client connections. Each accepted connection spawns a dedicated `ClientHandler` thread that reads requests line-by-line, processes them synchronously, and writes responses. This per-client threading enables the middleware to serve multiple clients simultaneously while keeping each client's request-response cycle synchronous.

ResourceManager Socket Servers: Each RM (`TCPResourceManager`) similarly accepts multiple concurrent connections from the middleware. Each connection spawns a `ClientHandler` thread that processes requests and returns responses. RMs don't initiate connections; they only accept them.

Concurrency Model

The TCP implementation achieves non-blocking middleware behavior through a thread-per-connection model: When the middleware's `ServerSocket.accept()` returns a new client socket, the main thread immediately spawns a `ClientHandler` thread and loops back to accepting the next client. The `ClientHandler` thread exclusively owns that client's socket, processing requests sequentially: read request → dispatch to Middleware logic → write response → repeat. This ensures the middleware never blocks on client I/O while waiting for RM responses.

Critically, when a `ClientHandler` calls middleware methods like `reserveFlight()`, those methods may block waiting for RM responses over TCP. However, since each client has its own thread, one client waiting for an RM response doesn't prevent other clients from being served concurrently.

The ResourceManagers use the same thread-per-connection model. Each RM spawns a thread for each incoming middleware connection, allowing RMs to process multiple requests concurrently.

Customer Management

Unlike the RMI implementation where reservation data was managed by the inherited `ResourceManager` class with synchronized `HashMap` access, the TCP middleware stores reservations in a `ConcurrentHashMap`. This thread-safe collection handles concurrent access from multiple `ClientHandler` threads without explicit synchronization blocks. Each reservation operation is atomic at the map level, preventing race conditions.

Bundle Implementation with Proper Rollback

The bundle operation in the TCP implementation follows a three-phase process to ensure atomicity. In the first phase (validation), the middleware queries each RM to confirm that all requested resources exist and are available, while also collecting their prices. If any resource is unavailable, the operation is aborted before making any reservations.

In the second phase (reservation), the middleware proceeds to reserve each resource in sequence. If a failure occurs mid-process, previously acquired reservations are rolled back by invoking the `removeReservation` method in the relevant RMs. This ensures that only resources successfully committed remain allocated, preventing partial or inconsistent states.

The final phase (local update) occurs only after all reservations succeed. At this point, the middleware updates the customer's reservation list and bill to reflect the confirmed bookings. This approach enforces all-or-nothing semantics and improves on the RMI implementation, which lacked rollback and could leave the system inconsistent if a bundle failed partway through.

Comparison to RMI Implementation

While both implementations share the same business logic, their communication layers differ significantly. RMI hides many low-level concerns by automating serialization, service discovery, threading, and error propagation, which makes development faster but less transparent. TCP, by contrast, required us to re-implement these aspects manually, leading to more code and greater complexity, but also providing finer control over how the system behaves.

One key advantage of TCP is that it eliminates the dependency on a separate `rmiregistry` process. Servers bind directly to ports, which simplifies deployment and avoids a single point of failure, making the system more robust in distributed environments. TCP also exposes connection management details that RMI abstracts away, enabling us to configure socket options, implement custom retry strategies, and design the threading model that best suits our workload. While this increases implementation effort, it improves transparency for debugging and flexibility for optimization.

Despite these differences, the underlying `ResourceManager` methods, customer management, and reservation coordination logic remained unchanged. This validated our design principle of separating communication mechanisms from business logic.

The Challenges We Faced

Two major challenges arose during implementation: customer deletion and bundle reservations.
For customer deletion, our initial design failed because temporary customer records were created in individual `ResourceManagers` whenever reservations were made. The original `deleteCustomer` attempted to create new customer records in each RM before deletion, which resulted in empty entries rather than removing existing reservations. To address this, we introduced a `removeReservation(int customerID, String itemKey, int count)` method in the base `ResourceManager`, which directly updates inventory counts without relying on customer records. The middleware's `deleteCustomer` now retrieves a customer's reservation list, delegates inventory adjustments to the relevant RM, and then removes the customer cleanly from storage.

A second challenge was ensuring atomicity in bundle reservations. Initially, partial failures could occur: if one resource was unavailable, other reservations might still be committed, leading to inconsistent states. We resolved this by implementing rollback logic in the middleware so that all previously acquired reservations are released if any part of the bundle fails. In addition, the client logic was simplified to delegate full control to the middleware, ensuring that bundles are always processed as an all-or-nothing operation.

Together, these solutions separated inventory from customer management and enforced atomicity in multi-resource operations, resulting in a more consistent and reliable system.

Tests Conducted

To validate our TCP middleware and resource managers, we designed a series of tests that exercise basic operations, customer management, reservation consistency, bundle reservations (including partial failures), and error handling. Each test was carried out step by step, with both expected and observed outcomes verified.

1. Basic CRUD Operations

Flights:

- `AddFlight,1,5,100` → Flight 1 added with 5 seats priced at 100.
- `QueryFlight,1` → returns 5 seats available.
- `QueryFlightPrice,1` → returns price 100.
- `DeleteFlight,1` → Flight 1 deleted.
- `QueryFlight,1` → returns 0 seats available.

Rooms:

- `AddRooms,mtl,5,50` → 5 rooms added in MTL with price 50.
- `QueryRooms,mtl` → returns 5 rooms available.
- `QueryRoomsPrice,mtl` → returns 50.
- `DeleteRooms,mtl` → rooms deleted.
- `QueryRooms,mtl` → returns 0 rooms available.

Cars:

- `AddCars,mtl,5,20` → 5 cars added in MTL with price 20.
- `QueryCars,mtl` → returns 5 cars available.
- `QueryCarsPrice,mtl` → returns 20.
- `DeleteCars,mtl` → cars deleted.
- `QueryCars,mtl` → returns 0 cars available.

2. Customer Management

Customer creation and query:

- `AddCustomerID,1` → customer with ID 1 created.
- `QueryCustomer,1` → returns an empty bill.

Reservations and deletion:

- Added Flight F1, then executed `ReserveFlight,1,F1`.
- `QueryCustomer,1` → bill shows F1 with correct price.
- `DeleteCustomer,1` → customer deleted.
- `QueryFlight,F1` → seat count restored, reservation removed.
- `QueryCustomer,1` → customer not found.

3. Reservation Consistency

- `AddCars,mtl,5,20` → 5 cars available.
- `ReserveCar,1,mtl` → 1 car reserved for customer 1.
- `QueryCars,mtl` → returns 4 cars available.
- `DeleteCustomer,1` → customer deleted.
- `QueryCars,mtl` → returns 5 cars available again.

This confirms that reservations are released correctly when customers are removed.

4. Bundle Reservations

Partial failure case:

- Only Flight F1 exists; Flight F2 not added.
- `Bundle,1,1,2,mtl,Y,Y` → operation fails as expected.
- Verified that customer bill is empty and all inventories remain unchanged.

Successful case:

- Added F1 (5 seats), F2 (5 seats), 5 cars, and 5 rooms.
- `Bundle,1,1,2,mtl,Y,Y` → operation succeeds.
- Verified that customer bill shows F1, F2, car, and room.
- Flight seats, cars, and rooms decremented correctly in each RM.

5. Error Handling

- Sent malformed requests
- Verified that the system remained stable and did not crash.

These tests demonstrate that our system supports correct CRUD operations, maintains consistency of reservations, enforces atomicity for bundle requests, and handles errors gracefully.

Statement of Contributions

Kazi primarily focused on the `RMI` distribution, implementing the `RMIMiddleware` class, customer management at the Middleware level, and the reservation coordination logic. Akash led the TCP sockets implementation, creating the `TCPResourceManager`, `TCPMiddleware` with non-blocking architecture, `TCPClient`, and the JSON-based message protocol with connection management.

Both team members jointly designed and implemented the `bundle()` functionality, including atomicity guarantees and rollback mechanisms, through pair programming sessions. For the report, Kazi wrote the RMI architecture and design decisions sections, while Akash documented the TCP architecture, message passing strategy, and concurrency model, with both contributing to the testing documentation.

Overall contributions were approximately equal, with Kazi contributing more to RMI/middleware logic and Akash contributing more to TCP/socket programming, while sharing equally on `bundle()` functionality, testing, and documentation.