ParkZone Multi-Threaded Client-Server Architecture

Updated: October 24, 2025
Phase 2 Revision: Multi-Threading Implementation

Executive Summary

ParkZone implements a **multi-threaded TCP/IP client-server architecture** that supports unlimited simultaneous client connections. The server maintains a singleton `ParkingSystem` instance shared across all client threads, enabling real-time synchronization and broadcast updates to all connected clients.

Architecture Overview

High-Level Design

```
┌─────────────┐          TCP/IP          ┌───────────────────────────────┐
│ Client GUI  │◄─────────────────────────►│  ParkingSystemServer          │
│ (Customer)  │       Port 8080          │  - Main acceptLoop() thread   │
└─────────────┘                          │  - Spawns ClientHandler       │
                                         │    threads for each client    │
┌─────────────┐          TCP/IP          └───────────────┬───────────────┘
│ Client GUI  │◄─────────────────────────────────────►   │
│ (Customer)  │       Port 8080                 Thread 1 │
└─────────────┘                          ┌───────────────▼────────────┐
                                         │ ClientHandler (Thread 1)   │
┌─────────────┐          TCP/IP          │ - Reads client messages    │
│ Admin GUI   │◄─────────────────────────┤ - Handles authentication   │
│ (Operator)  │       Port 8080          │ - Processes requests       │
└─────────────┘                 Thread 2 │ - Sends responses          │
                                         └─────────────┬──────────────┘
                                                       │
                              ┌────────────────────────┼─────────────────────┐
                              │            ALL THREADS ACCESS                │
                              │                                              │
                         ┌────▼───────────────────────────────────────┐      │
                         │   ParkingSystem (SINGLETON)                │      │
                         │   - users: List<User>                      │      │
                         │   - slots: List<ParkingSlot>               │◄─────┘
                         │   - tickets: List<Ticket>                  │
                         │   - Thread-safe operations                 │
                         │   - Synchronized methods                   │
                         └────────────────────────────────────────────┘
```

Component Responsibilities

1. ParkingSystemServer (Main Thread)

Responsibility: Accept incoming TCP/IP connections and spawn client handler threads.

Key Methods:

```java
public class ParkingSystemServer {
    private ServerSocket serverSocket;
    private ExecutorService clientPool;  // Thread pool
    private ParkingSystem parkingSystem;  // Singleton reference
    private Map<String, ClientHandler> clientsById;

    public void start() {
        serverSocket = new ServerSocket(8080);
        acceptLoop();
    }

    private void acceptLoop() {
        while (running) {
            Socket clientSocket = serverSocket.accept();
            ClientHandler handler = new ClientHandler(clientSocket, parkingSystem);
            clientPool.execute(handler);  // Spawn new thread
            clientsById.put(handler.getClientId(), handler);
        }
    }

    public void broadcast(Message msg) {
        for (ClientHandler handler : clientsById.values()) {
            handler.send(msg);
        }
    }
}
```

Threading Model:

- Main thread runs `acceptLoop()` continuously
- Each accepted connection spawns a **new ClientHandler thread**
- Thread pool (ExecutorService) manages thread lifecycle
- Server can handle **unlimited concurrent connections**

---

2. ClientHandler (Per-Client Thread)

Responsibility: Handle all communication with ONE specific client.

Key Methods:

```java
public class ClientHandler implements Runnable {
    private Socket socket;
    private ParkingSystem parkingSystem;  // Shared singleton
    private AuthSession session;
    private String clientId;

    @Override
    public void run() {
        readLoop();  // Blocks waiting for client messages
    }

    private void readLoop() {
        while (socket.isConnected()) {
            Message msg = readMessage();
            handleMessage(msg);
        }
    }

    private void handleMessage(Message msg) {
        switch(msg.getType()) {
            case LOGIN_REQUEST:
                handleLogin(msg);
                break;
            case PARK_VEHICLE:
                handleParkVehicle(msg);
                break;
            case END_PARKING:
                handleEndParking(msg);
                break;
            // ... other cases
        }
    }

    private void handleParkVehicle(Message msg) {
        // 1. Call shared ParkingSystem (thread-safe)
        Ticket ticket = parkingSystem.issueTicket(vehicle, slot);

        // 2. Send response to THIS client
        send(new Response(SUCCESS, ticket));

        // 3. Broadcast update to ALL other clients
        server.broadcast(new SpaceUpdate(slot.getSlotID(), true));
    }
}
```

Key Characteristics:

- Each ClientHandler runs in its **own thread**
- Maintains session state for ONE client
- Calls **shared ParkingSystem methods** (thread-safe)
- Can send messages to its client OR broadcast to all clients via server

---

3. ParkingSystem (Singleton, Thread-Safe)

Responsibility: Central business logic shared across all client threads.

Why Singleton?

```
public class ParkingSystem {
    private static ParkingSystem instance;  // Only ONE instance
    private List<User> users;
    private List<ParkingSlot> slots;
    private List<Ticket> tickets;

    // Private constructor prevents multiple instances
    private ParkingSystem() {
        users = Collections.synchronizedList(new ArrayList<>());
        slots = Collections.synchronizedList(new ArrayList<>());
        tickets = Collections.synchronizedList(new ArrayList<>());
    }

    // Thread-safe singleton access
    public static synchronized ParkingSystem getInstance() {
        if (instance == null) {
            instance = new ParkingSystem();
        }
        return instance;
    }

    // Thread-safe operations
    public synchronized Ticket issueTicket(Vehicle vehicle, ParkingSlot slot) {
        // Only ONE thread can execute this at a time
        slot.assignVehicle(vehicle);
        Ticket ticket = new Ticket(vehicle, slot, LocalDateTime.now());
        tickets.add(ticket);
        return ticket;
    }

    public synchronized List<ParkingSlot> getAvailableSlots() {
        return slots.stream()
            .filter(slot -> !slot.isOccupied())
            .collect(Collectors.toList());
    }
}
```

Thread Safety Mechanisms:

1. **Singleton pattern** - only one instance exists
2. **Synchronized methods** - only one thread executes at a time
3. **Synchronized collections** - thread-safe data structures
4. **Atomic operations** - prevent race conditions

Why This Matters:

- 30 clients connected = 30 ClientHandler threads
- All 30 threads call the SAME ParkingSystem instance
- Without synchronization → data corruption, double-booking
- With synchronization → safe concurrent access

Communication Flow

Message Format (over TCP/IP)

```java
public class Message implements Serializable {
    private MessageType type;
    private String sessionToken;
    private Map<String, Object> payload;

    enum MessageType {
        // Client → Server
        LOGIN_REQUEST,
        CREATE_ACCOUNT_REQUEST,
        REGISTER_VEHICLE,
        GET_AVAILABLE_SLOTS,
        PARK_VEHICLE,
        END_PARKING,

        // Server → Client
        LOGIN_RESPONSE,
        CREATE_ACCOUNT_RESPONSE,
        AVAILABLE_SLOTS_RESPONSE,
        PARK_VEHICLE_RESPONSE,
        END_PARKING_RESPONSE,

        // Server → All Clients (broadcast)
        SPACE_UPDATE,
        NEW_SLOT_ADDED,
        OVERSTAY_ALERT
    }
}
```

Request-Response Pattern

Client sends request:

```java
// Client GUI
Message request = new Message(PARK_VEHICLE);
request.setPayload("slotID", 7);
request.setPayload("plateNumber", "ABC123");
outputStream.writeObject(request);
```

Server processes (in ClientHandler thread):

```java
// ClientHandler.handleMessage()
case PARK_VEHICLE:
    int slotID = (int) msg.getPayload("slotID");
    String plate = (String) msg.getPayload("plateNumber");

    // Call shared ParkingSystem
    Ticket ticket = parkingSystem.issueTicket(vehicle, slot);

    // Send response to THIS client
    Message response = new Message(PARK_VEHICLE_RESPONSE);
    response.setPayload("success", true);
    response.setPayload("ticketID", ticket.getTicketID());
    send(response);
```

Server broadcasts to ALL clients:

```java
// After parking, notify everyone
Message broadcast = new Message(SPACE_UPDATE);
broadcast.setPayload("slotID", 7);
broadcast.setPayload("occupied", true);
server.broadcast(broadcast);  // Goes to all ClientHandler threads
```

Multi-Threading Implementation

Thread Pool Management

```
public class ParkingSystemServer {
    private ExecutorService clientPool;

    public void start() {
        // Create thread pool for handling clients
        clientPool = Executors.newCachedThreadPool();

        // Or use fixed pool if you want to limit connections
        // clientPool = Executors.newFixedThreadPool(100);

        acceptLoop();
    }

    private void acceptLoop() {
        while (running) {
            Socket clientSocket = serverSocket.accept();

            // Spawn new thread from pool
            ClientHandler handler = new ClientHandler(clientSocket);
            clientPool.execute(handler);  // Runs handler.run() in new thread
        }
    }
}
```

Concurrent Client Handling

Scenario: 3 clients connect simultaneously

```
 Time ──────────────────────────────────────────────▶

Client 1: ──────┬─ connect() ──▶ Thread 1 spawned ──▶ Parking flow
                │
Client 2: ───────┴─ connect() ──▶ Thread 2 spawned ──▶ Login flow
                   │
Client 3: ─────────┴─ connect() ──▶ Thread 3 spawned ──▶ Query slots

All 3 threads run SIMULTANEOUSLY
All 3 access the SAME ParkingSystem singleton
Synchronization prevents conflicts
```

Broadcast Communication

When Client 1 parks in slot 7:

```
┌─────────────┐
│ Client 1    │ parks in slot 7
└──────┬──────┘
       │
    ┌──▼──────────────┐
    │ ClientHandler1 │ calls parkingSystem.issueTicket()
    └──┬───────┬──────┘
       │       │
       │       └──────► Broadcast SPACE_UPDATE to server
       │
    ┌──▼─────────────┐
    │ ParkingSystem │ updates slot 7 → occupied
    │  (Singleton)  │
    └────────────────┘


                 ┌──────────────────┐
                 │ Server           │ receives broadcast request
                 └─────┬───┬───┬────┘
                       │   │   │
        ┌──────────────┘   │   └───────────┐
        │                  │               │
     ┌──▼───────┐       ┌──▼───────┐     ┌──▼───────┐
     │Handler 1 │       │Handler 2 │     │Handler 3 │
     │(Client 1)│       │(Client 2)│     │(Admin)   │
     └────┬─────┘       └────┬─────┘     └────┬─────┘
          │                  │                │
     ┌────▼─────┐       ┌────▼─────┐     ┌────▼─────┐
     │Client 1  │       │Client 2  │     │Admin GUI │
     │GUI       │       │GUI       │     │          │
     │updates   │       │updates   │     │updates   │
     │display   │       │display   │     │display   │
     └──────────┘       └──────────┘     └──────────┘
```

All connected clients see the update in real-time!

Professor's Requirements Addressed

□ 1. Multiple Clients Simultaneously

Implementation:

- `ExecutorService` thread pool spawns unlimited ClientHandler threads
- Each client gets dedicated thread that runs concurrently
- Main server thread continues accepting new connections

Code Evidence:

```
clientPool.execute(new ClientHandler(socket));  // Spawns new thread
```

□ 2. TCP/IP Network Connection

Implementation:

- ServerSocket listens on port 8080
- Socket connections for each client
- ObjectInputStream/ObjectOutputStream for serialized messages

Code Evidence:

```
ServerSocket serverSocket = new ServerSocket(8080);
Socket clientSocket = serverSocket.accept();
```

□ 3. Multi-Threaded Solution (from examples)

Implementation:

- Follows the multi-threaded server pattern from class examples
- One thread per client connection
- Thread-safe shared resources

Code Evidence:

```
public class ClientHandler implements Runnable {
    @Override
    public void run() {
        readLoop();  // Blocks in dedicated thread
    }
}
```

✅ 4. Singleton Pattern

Implementation:

- ParkingSystem is singleton
- Only ONE instance shared across all threads
- Thread-safe getInstance() method

Code Evidence:

```
public static synchronized ParkingSystem getInstance() {
    if (instance == null) {
        instance = new ParkingSystem();
    }
    return instance;
}
```

✅ 5. Two-Way Communication with ALL Clients Simultaneously

Implementation:

- Server can broadcast to all ClientHandler threads
- Each ClientHandler can send individual responses
- Clients receive both direct responses AND broadcast updates

Professor's Analogy:

> "Your application has one server, like me, but it's able to have a two-way communication directly with every single client simultaneously."

Our Implementation:

```
// Individual communication
handler.send(response);  // To one client

// Broadcast communication
server.broadcast(update);  // To all clients
```

Sequence Diagram Updates

What Changed from Previous Version:

OLD (Single-Threaded):

```
Client → System → ParkingSlot → Ticket
```

NEW (Multi-Threaded):

```
ClientGUI → ClientHandler (Thread) → ParkingSystem (Singleton) → Domain Objects
                                          ▲
                                          |
                                   All threads access here
```

Key Additions:

1. **ParkingSystemServer** - main server thread

2. **ClientHandler** - per-client thread
3. **Broadcast operations** - notify all clients
4. **Session management** - track authenticated users
5. **Thread annotations** - show which thread executes what

Thread Safety Guarantees

Race Condition Prevention

Problem: Two clients try to book the same slot simultaneously

```
Thread 1: isAvailable(slot7) → true ─┐
                                     ├─► CONFLICT!
Thread 2: isAvailable(slot7) → true ─┘

Both think slot 7 is available!
```

Solution: Synchronized method

```
public synchronized Ticket issueTicket(Vehicle v, ParkingSlot slot) {
    if (slot.isOccupied()) {
        throw new SlotOccupiedException();
    }

    slot.assignVehicle(v);  // Atomic operation
    Ticket ticket = new Ticket(v, slot, LocalDateTime.now());
    tickets.add(ticket);
    return ticket;
}
```

Result:

```
Thread 1: acquires lock → books slot 7 → releases lock ⍰
Thread 2: waits for lock → tries to book → gets exception ⍰

No double-booking!
```

Data Consistency

Thread-Safe Collections:

```
private List<User> users = Collections.synchronizedList(new ArrayList<>());
private List<ParkingSlot> slots = Collections.synchronizedList(new ArrayList<>());
private List<Ticket> tickets = Collections.synchronizedList(new ArrayList<>());
```

Synchronized Methods:

```
public synchronized void addUser(User user) { ... }
public synchronized List<ParkingSlot> getAvailableSlots() { ... }
public synchronized Ticket findTicket(int ticketID) { ... }
```

Testing Multi-Threading

Test Scenario: 10 Concurrent Clients

```java
@Test
public void testConcurrentParking() {
    ParkingSystemServer server = new ParkingSystemServer();
    server.start();

    // Spawn 10 client threads
    ExecutorService clients = Executors.newFixedThreadPool(10);

    for (int i = 0; i < 10; i++) {
        clients.execute(() -> {
            Socket socket = new Socket("localhost", 8080);
            // Each thread tries to park
            Message parkRequest = new Message(PARK_VEHICLE);
            // ...
        });
    }

    clients.shutdown();
    clients.awaitTermination(10, TimeUnit.SECONDS);

    // Verify: Only 10 tickets issued, no duplicates
    assertEquals(10, parkingSystem.getActiveTickets().size());
}
```

Performance Characteristics

Scalability

- **Clients Supported:** Unlimited (limited by system resources)
- **Thread Pool:** Dynamically scales with CachedThreadPool
- **Response Time:** O(1) for synchronized operations
- **Memory:** ~1MB per ClientHandler thread

Bottlenecks

1. **Synchronized methods** - only one thread at a time
   - **Mitigation:** Keep critical sections small
2. **Broadcast operations** - O(n) with number of clients
   - **Mitigation:** Async send, don't block on client socket writes

---

Deployment Architecture

```
Production Server (AWS EC2 / DigitalOcean)
├── ParkingSystemServer.jar
│   └── Runs on port 8080
│   └── Persistent data in /data/*.dat files
│
└── Clients connect from anywhere
    ├── Customer laptops/desktops
    ├── Operator workstations
    └── Admin terminals
```

Firewall Rules:

- Open TCP port 8080 for incoming connections
- Restrict admin operations to internal network

Summary

ParkZone implements a **production-ready multi-threaded client-server architecture** that:

Supports **unlimited concurrent clients**
Uses **TCP/IP networking** (port 8080)
Spawns **one thread per client** (ClientHandler)
Shares **singleton ParkingSystem** across all threads
Provides **thread-safe operations** (synchronized methods)
Enables **real-time broadcast updates** to all clients
Maintains **individual two-way communication** with each client

This satisfies all of Professor Smith's requirements for Phase 2.

Document Version: 2.0
Last Updated: October 24, 2025
Author: Mario Salinas