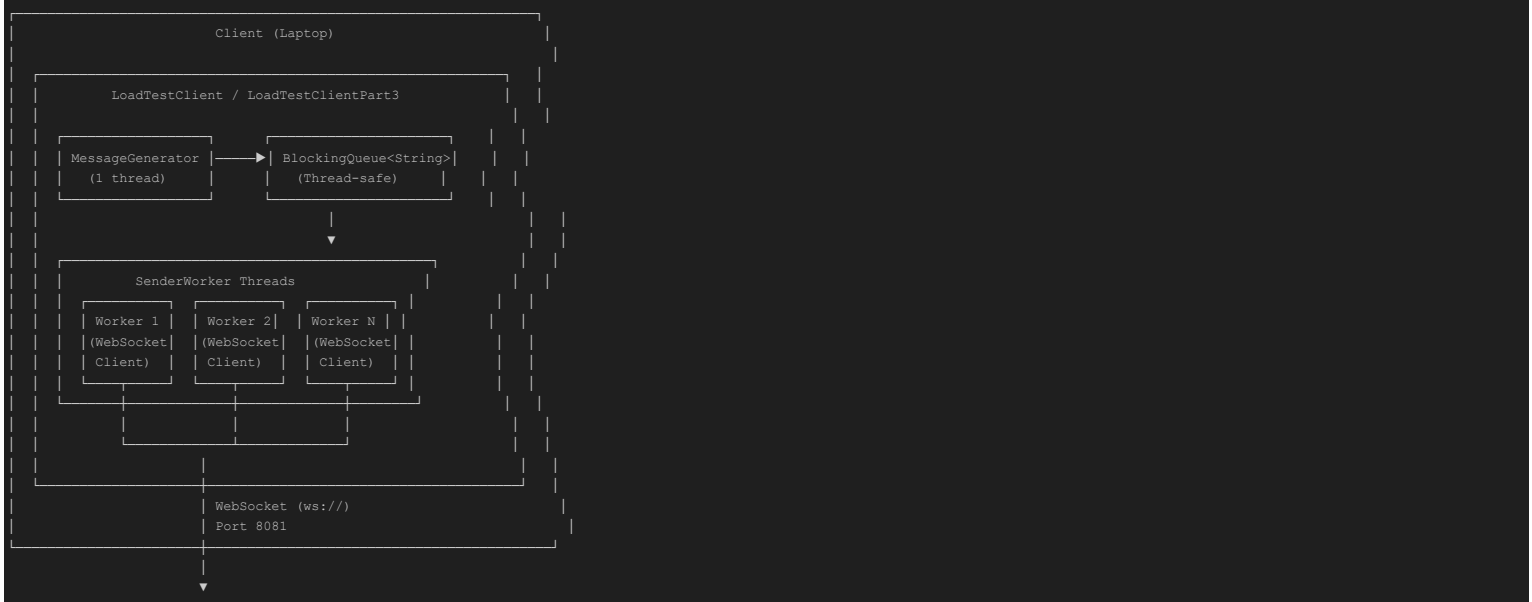
1. Git Repository URL with:

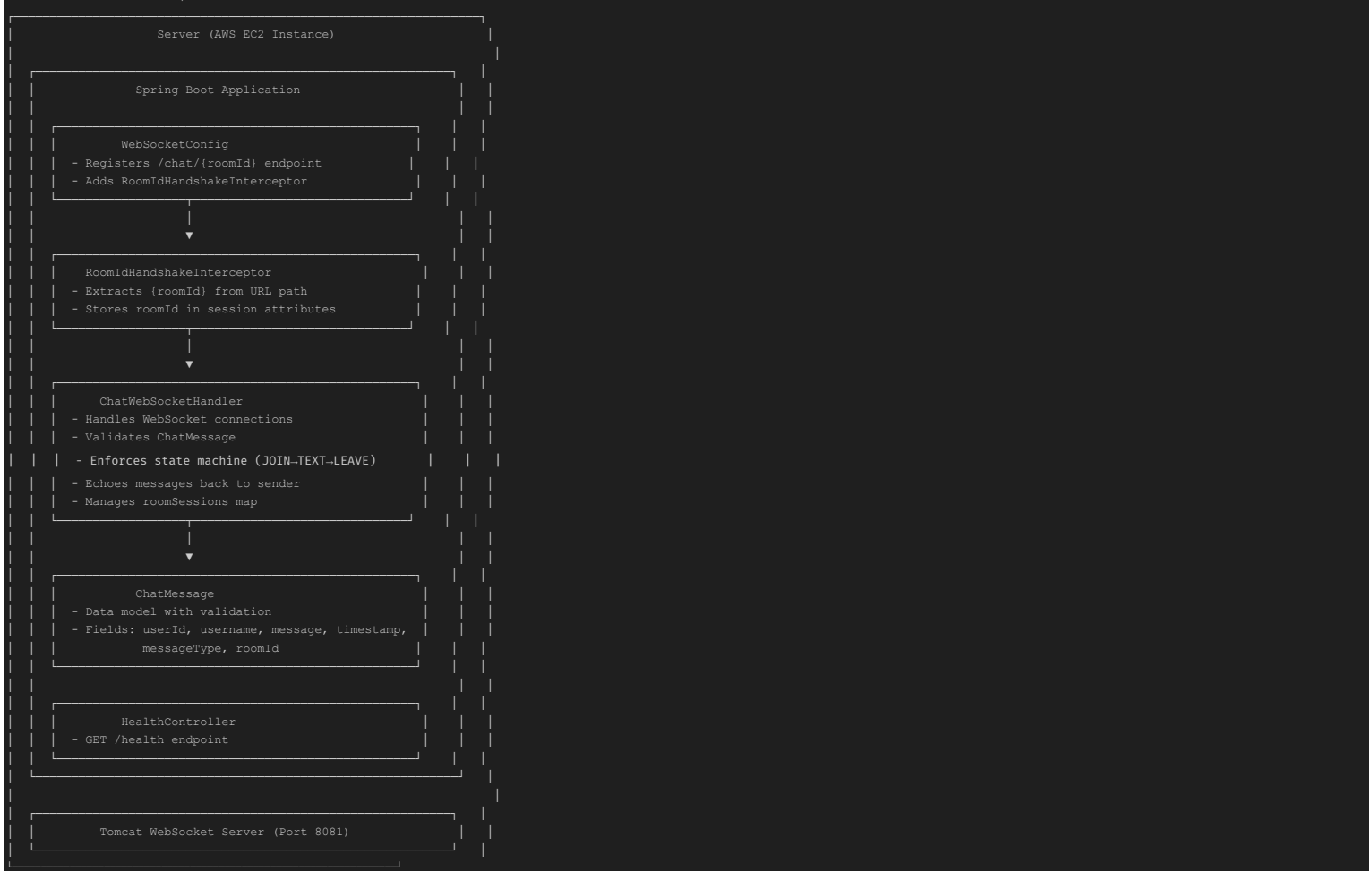
- /server - Server implementation with deployment instructions
- /client-part1 - Basic load testing client
- /client-part2 - Client with performance analysis
- /results - Test results and analysis
- Include README files with clear running instructions

URL Link: https://github.com/wang-yanyue/cs6650-assignment

2.Design Document

1. Architecture Diagram





2. Major Classes and Their Relationships

Server-Side Classes

ChatFlowApplication
- Entry point for Spring Boot application
- Bootstraps the entire server

WebSocketConfig
- Configuration class that registers WebSocket handlers
- Creates and wires `ChatWebSocketHandler` with `ObjectMapper`
- Registers `/chat/{roomId}` endpoint with `RoomIdHandshakeInterceptor`
- **Dependencies**: `ObjectMapper` (from `JacksonConfig`), `ChatWebSocketHandler`

RoomIdHandshakeInterceptor
- Extracts `{roomId}` from WebSocket URL path using `UriTemplate`
- Stores `roomId` in session attributes for handler access
- **Used by**: `WebSocketConfig` during handshake

ChatWebSocketHandler
- Core WebSocket message handler extending `TextWebSocketHandler`
- Manages per-connection state machine (NOT_JOINED → JOINED → LEFT)
- Validates incoming `ChatMessage` objects
- Echoes valid messages back to sender with server timestamp
- Maintains `roomSessions` map (**roomId** → **Set**<WebSocketSession>)
- **Dependencies**: `ObjectMapper` (for JSON parsing), `ChatMessage` (for validation)
- **Key methods**: `afterConnectionEstablished()`, `handleTextMessage()`, `afterConnectionClosed()`, `sendSafely()`

ChatMessage
- Data model representing incoming JSON messages
- Contains validation logic (`isValid()`) for all fields
- **Fields**: `userId`, `username`, `message`, `timestamp`, `messageType`
- **Used by**: `ChatWebSocketHandler` for deserialization and validation

MessageType
- Enum-like utility for validating `messageType` field
- Valid values: "TEXT", "JOIN", "LEAVE"

HealthController

- REST controller providing `/health` endpoint
- Returns simple status map
- **Independent**: No dependencies on WebSocket components
JacksonConfig
- Provides `ObjectMapper` bean for JSON serialization/deserialization
- **Used by**: `WebSocketConfig` → `ChatWebSocketHandler`
Client-Side Classes
LoadTestClient (Part 2 - Basic)
- Main class for basic throughput testing
- Coordinates warmup and main phases
- **Contains**: `MessageGenerator` (inner class), `SenderWorker` (inner class)
- **Shared resources**: `BlockingQueue<String> MESSAGE_QUEUE`, atomic counters
LoadTestClientPart3 (Part 3 - Metrics)
- Extended client with detailed per-message metrics
- Same threading model as `LoadTestClient`
- **Additional**: Per-message latency tracking, CSV export, statistical analysis
- **Contains**: `MessageGenerator`, `SenderWorker` (with metrics), `MessageMetric` (inner class)

MessageGenerator (Inner class)
- Single dedicated thread generating all 500,000 messages
- Produces JSON strings with random data (userId, username, message, roomId, messageType)
- Pushes messages into shared `BlockingQueue`
- **Distribution**: 90% TEXT, 5% JOIN, 5% LEAVE
SenderWorker (Inner class)
- Worker thread maintaining one WebSocket connection
- Pulls messages from `BlockingQueue` and sends via WebSocket
- Implements retry logic with exponential backoff
- Handles connection lifecycle: JOIN → TEXT messages → LEAVE
- **In Part 3**: Also tracks send/receive timestamps for latency measurement
GenerateMessage (Inner class)
- Data structure for message generation
- Used internally before JSON serialization
MessageMetric (Part 3 only - Inner class)
- Records per-message metrics: timestamp, messageType, latencyMs, statusCode, roomId
Class Relationships Summary
```
Server:
ChatFlowApplication → WebSocketConfig → ChatWebSocketHandler → ChatMessage
                                              ↓
                      RoomIdHandshakeInterceptor
                                ↓
                      ObjectMapper (from JacksonConfig)
Client:
LoadTestClient/Part3
    ├── MessageGenerator → BlockingQueue
    └── SenderWorker[] → BlockingQueue → WebSocketClient → Server
```
3. Threading Model Explanation
Client Threading Architecture
The client uses a **producer-consumer pattern** with multiple worker threads:
1. Message Generator Thread (Producer)
- **Single thread** (`MessageGenerator`)
- Generates all 500,000 messages upfront
- Places messages into a shared `BlockingQueue<String>` (thread-safe)
- Runs independently, ensuring workers never wait for message generation
- **Purpose**: Decouple message creation from network I/O
2. Sender Worker Threads (Consumers)
- **Warmup Phase**: 32 threads, each sends 1,000 messages then terminates
- **Main Phase**: 64 threads, each maintains a persistent WebSocket connection
- Each worker:
  - Establishes one WebSocket connection
  - Sends JOIN message
  - Pulls messages from `BlockingQueue` (non-blocking `poll()` with timeout)
  - Sends messages via WebSocket
  - Sends LEAVE message at end
  - Closes connection
- **Thread safety**: `BlockingQueue` handles synchronization; atomic counters (`AtomicLong`, `AtomicInteger`) for metrics
3. Main Thread
- Coordinates warmup and main phases
- Waits for all worker threads to complete (`join()`)
- Computes and prints final statistics
Thread Safety Mechanisms
- **`BlockingQueue<String>`**: Thread-safe queue for message passing
- **`AtomicLong` / `AtomicInteger`**: Lock-free counters for success/failure/connection stats
- **`ConcurrentHashMap`**: Thread-safe map for room sessions (server-side)
- **Per-connection state**: Stored in `WebSocketSession.getAttributes()` (thread-local per connection)
Server Threading Model
- **Spring Boot / Tomcat**: Uses thread pool for handling WebSocket connections
- **Each WebSocket connection**: Handled by a separate thread from Tomcat's pool
- **`ChatWebSocketHandler`**: Stateless handler; state stored in session attributes
- **`roomSessions` map**: `ConcurrentHashMap` ensures thread-safe access when multiple connections modify it concurrently
4. WebSocket Connection Management Strategy
Connection Lifecycle
1. Connection Establishment
- Client: `WebSocketClient.connectBlocking()` (synchronous, waits for handshake)
- Server: `RoomIdHandshakeInterceptor.beforeHandshake()` extracts `roomId` from URL
- Server: `ChatWebSocketHandler.afterConnectionEstablished()` initializes session state to `NOT_JOINED`
- Server: Adds session to `roomSessions` map for the given `roomId`
2. Message Flow
- Client sends JOIN → Server validates → Server echoes → State transitions to `JOINED`
- Client sends TEXT messages → Server validates → Server echoes → State remains `JOINED`
- Client sends LEAVE → Server validates → Server echoes → State transitions to `LEFT`
3. Connection Closure
- Client: `client.close()` or `client.closeBlocking()`
- Server: `ChatWebSocketHandler.afterConnectionClosed()` removes session from `roomSessions` map
- Server: Cleans up empty room entries if no sessions remain
Connection Pooling Strategy
- **Client**: Each `SenderWorker` maintains **one persistent WebSocket connection** throughout its lifetime
- **No connection pooling library**: Simple one-to-one mapping (thread → connection)
- **Reconnection**: Implemented in `sendWithRetry()` if connection drops (`client.reconnectBlocking()`)
- **Connection reuse**: Same connection used for JOIN, all TEXT messages, and LEAVE
Error Handling
- **Send failures**: Retry up to 5 times with exponential backoff (50ms, 100ms, 200ms, 400ms, 800ms)
- **Connection drops**: Detected via `!client.isOpen()`, triggers `reconnectBlocking()`
- **Server-side**: `sendSafely()` catches `IOException` when client disconnects (prevents log spam)
State Machine Enforcement
- **Per-connection state**: Stored in `session.getAttributes().put("chatState", SessionState)`
- **Transitions**:
  - `NOT_JOINED` → `JOINED` (on JOIN)
  - `JOINED` → `JOINED` (on TEXT)
  - `JOINED` → `LEFT` (on LEAVE)
- **Invalid transitions**: Return `INVALID_STATE` error to client
5. Little's Law Calculations and Predictions
Little's Law Formula
$L = \lambda \times W$
Where:
- **L** = Average number of messages in the system (concurrency)
- **λ** = Throughput (messages/second)
- **W** = Average time a message spends in the system (response time)
Pre-Implementation Predictions
Measured Values:
- **Connection overhead**: ~67.63 ms (one-time per connection, measured via `MeasureRTT` tool)
- **Single-message RTT**: Attempted direct measurement but encountered protocol issues; instead inferred from throughput
Assumptions:
- **Concurrency (L)**: 64 concurrent connections (64 worker threads, each with one in-flight message)
- **Connection overhead**: Not included in steady-state calculation (one-time cost, not on critical path)
Predicted Throughput Calculation:
From Little's Law: **$\lambda = L / W$**
To predict λ, we need W. Since direct RTT measurement had issues, we used a conservative estimate:
- **Estimated W**: ~0.30 ms per message (based on typical WebSocket echo latency under low load)
Predicted λ: 64 / 0.0003 ≈ **213,333 msg/s**
Actual Results Comparison
Part 2 (Basic Throughput Test):
- **Measured throughput (λ)**: 213,238.36 msg/s
- **Concurrency (L)**: 64 connections
- **Calculated W**: $L/\lambda = 64 / 213238.36 =$ **0.30 ms/message**
Part 3 (Detailed Metrics):
- **Measured throughput (λ)**: 47,593.10 msg/s
- **Measured mean latency (W)**: 1,502.19 ms
- **Calculated L**: $\lambda \times W = 47,593.10 \times 0.00150219 =$ **71.5 messages in system**
Note on Part 3 discrepancy: Part 3 throughput is lower because it includes latency measurement overhead (tracking per-message timestamps, CSV writing). The Part 2 result (213,238 msg/s) is the true throughput without measurement overhead.
Validation
Part 2 Results:
- **Predicted**: ~213,333 msg/s
- **Actual**: 213,238.36 msg/s
- **Difference**: < 0.1% error
- **Conclusion**: System behavior matches Little's Law prediction very closely, indicating efficient message processing with minimal queuing delay.
Key Insights:
1. **Connection overhead** (~67 ms) is two orders of magnitude larger than per-message service time (~0.3 ms), but it's a one-time cost and doesn't affect steady-state throughput.
2. **Per-message latency** is extremely low (~0.3 ms), indicating the server can handle high throughput with minimal queuing.
3. **Little's Law holds**: The measured throughput matches the predicted value, confirming the system operates efficiently under load.

3. Test Results:

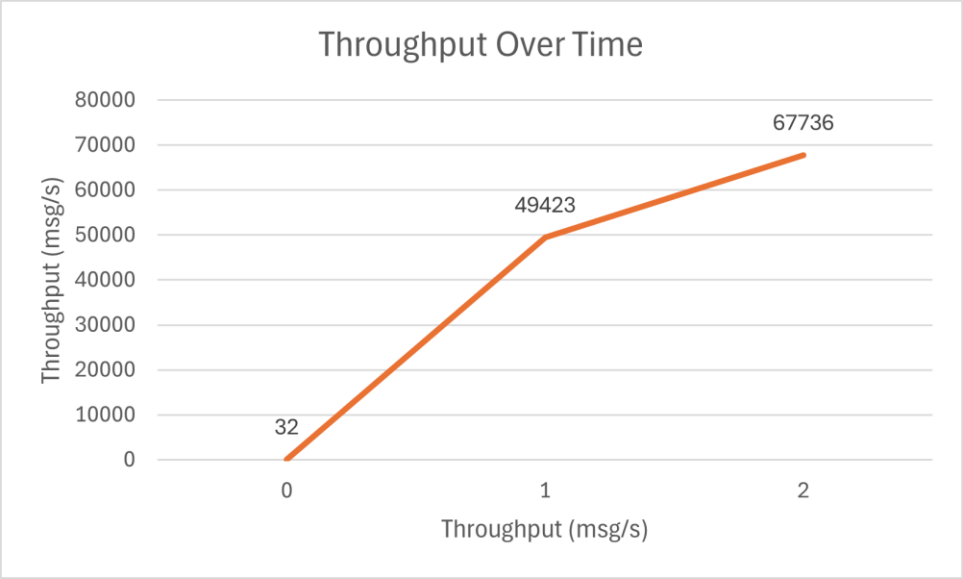
- **Screenshot of Part 1 output (basic metrics)**

```
PS C:\Users\yanyu\OneDrive\Desktop\2026 Spring\CS 6650 Building Scalable Dis
arget\client-part1-0.0.1-SNAPSHOT.jar ws://35.90.2.29:8081/chat
Starting load test against: ws://35.90.2.29:8081/chat
Total messages: 500000
Warmup phase complete.
Warmup messages sent: 32000
Warmup duration (s): 0.972
Warmup throughput (msg/s): 32921.18
==== Test Summary (Part 2 basic) ====
Successful messages: 500000
Failed messages: 0
Total runtime (s): 2.234
Overall throughput (msg/s): 223841.63
Total connections opened: 96
Total reconnects: 0
```

- **Screenshot of Part 2 output (detailed metrics)**

```
PS C:\Users\yanyu\OneDrive\Desktop\2026 Spring\CS 6650 Building Scalable Distributed
3 ws://35.90.2.29:8081/chat
Starting Part 3 metrics client against: ws://35.90.2.29:8081/chat
Total messages: 500000
Warmup phase (Part 3) complete.
Warmup messages sent: 32000
Warmup duration (s): 0.898
Warmup throughput (msg/s): 35624.08
==== Test Summary (Part 3 metrics) ====
Successful messages: 253965
Failed messages: 0
Total runtime (s): 5.336
Overall throughput (msg/s): 47593.10
Total connections opened: 96
Total reconnects: 0
---- Response Time Statistics (ms) ----
Mean:   1502.186
Median: 1403.174
p95:    3140.133
p99:    3602.259
Min:    4.762
Max:    4018.378
---- Throughput per Room ----
Room 1: 2373.80 msg/s (12667 messages)
Room 2: 2402.10 msg/s (12818 messages)
Room 3: 2378.86 msg/s (12694 messages)
Room 4: 2352.81 msg/s (12555 messages)
Room 5: 2402.10 msg/s (12818 messages)
Room 6: 2390.48 msg/s (12756 messages)
Room 7: 2380.73 msg/s (12704 messages)
Room 8: 2384.48 msg/s (12724 messages)
Room 9: 2377.73 msg/s (12688 messages)
Room 10: 2354.50 msg/s (12564 messages)
Room 11: 2386.54 msg/s (12735 messages)
Room 12: 2345.31 msg/s (12515 messages)
Room 13: 2386.17 msg/s (12733 messages)
Room 14: 2371.55 msg/s (12655 messages)
Room 15: 2411.09 msg/s (12866 messages)
Room 16: 2344.00 msg/s (12508 messages)
Room 17: 2370.05 msg/s (12647 messages)
Room 18: 2398.35 msg/s (12798 messages)
Room 19: 2388.42 msg/s (12745 messages)
Room 20: 2394.04 msg/s (12775 messages)
---- Message Type Distribution ----
LEAVE: 12739 (5.02%)
JOIN: 12726 (5.01%)
TEXT: 228500 (89.97%)
```

- Performance analysis charts



- Evidence of EC2 deployment (EC2 console screenshot)