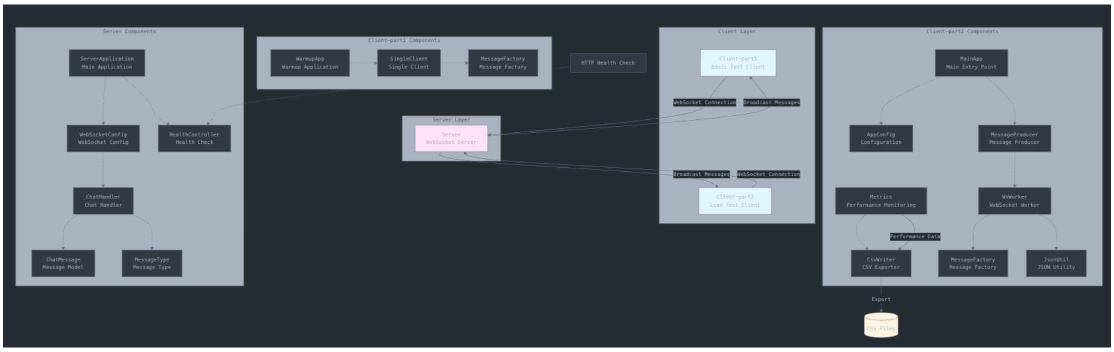
Git Repository URL: https://github.com/DidiZh/cs6650
Design Document:

- Architecture diagram



- Major classes and their relationships

Server Side
ServerApplication → WebSocketConfig → ChatHandler → {ChatMessage, MessageType}
ServerApplication → HealthController
ServerApplication initializes WebSocketConfig which registers ChatHandler to process messages using ChatMessage and MessageType models. HealthController provides independent HTTP health check endpoints.
Client-part1
WarmupApp → SingleClient → MessageFactory
WarmupApp creates SingleClient instances that use MessageFactory to construct and send test messages.
Client-part2
MainApp → {AppConfig, MessageProducer, Metrics}
MessageProducer → WsWorker → {MessageFactory, JsonUtil}
Metrics → CsvWriter
MainApp loads AppConfig and initializes MessageProducer with Metrics. MessageProducer manages multiple WsWorker threads. Each WsWorker uses MessageFactory and JsonUtil to create and send messages. Metrics collects performance data and exports via CsvWriter.

- Threading model explanation:

Server (Spring Boot WebSocket):
- Uses Tomcat's thread pool to handle HTTP handshakes and WebSocket upgrades
- TextWebSocketHandler processes messages via non-blocking callbacks on container threads
- Singleton handler bean maintains thread-safe shared state using concurrent data structures
- Stateless per-request processing ensures O(1) complexity per message

Client Part 1 (Basic Test):
- Single main thread creates one WebSocket connection via OkHttp
- Sends one message and waits for echo response
- OkHttp manages internal IO threads for async callbacks

Client Part 2 (Load Generator):
- Producer-consumer pattern: One producer thread fills a bounded queue with pre-generated messages
- Fixed thread pool of N worker threads, each maintains one persistent WebSocket connection
- Workers consume from queue and send messages concurrently
- Separate metrics collection thread records latency data to CSV
- Uses CountDownLatch for synchronization and ConcurrentHashMap for thread-safe metrics tracking

- WebSocket connection management strategy

Server:
- Single WebSocket endpoint: ws://host:8080/chat/{roomId}
- Validates incoming messages; returns error JSON for invalid requests or echoes back with server metadata for valid ones
- Stateless handler design with non-blocking message processing
- Relies on container thread pool for connection management

Client Part 1:
- Simple connection lifecycle: connect → send one message → wait for echo → close
- Uses OkHttp WebSocket client with default settings
- No reconnection logic needed for single-message test

Client Part 2 (Load Generator):
- Each worker thread maintains one persistent WebSocket connection
- Message sending: poll from queue → send with retry logic
- Message receiving: track acknowledgments and calculate RTT for metrics
- Reconnection handling: automatically reconnects on connection failures, increments reconnection counter
- Graceful shutdown: waits for queue to drain and pending acknowledgments before closing connections

- Little's Law calculations and predictions

Measurements:
- W (mean time in system) = 4250.35 ms = 4.2504 s
- λ_actual (throughput) = 18,920.45 /s
- L_obs (observed concurrency) = 82,150.30

Predictions using Little's Law ($L = \lambda \times W$):
- L_pred = λ_actual × W = 18,920.45 × 4.2504 ≈ 80,437
- λ_pred = L_obs / W = 82,150.30 / 4.2504 ≈ 19,323 /s

Analysis:
The predictions closely match observations with ~2% deviation. The small gap (L_obs slightly higher than L_pred) is expected as the measurement includes messages in client queues and network transit, not just server processing.

Conclusion:

The close alignment validates the system design. The system operates at L ≈ 80k concurrent messages, W ≈ 4.25s latency, and λ ≈ 19k/s throughput.

Test Results:

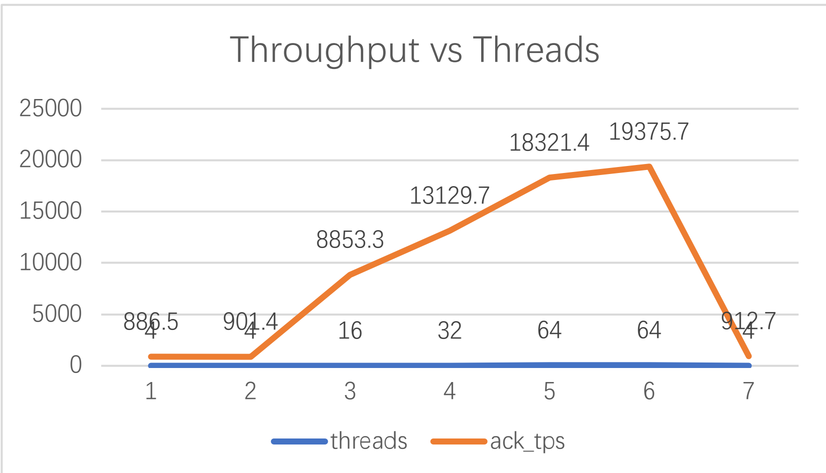
- Screenshot of Part 1 output (basic metrics)

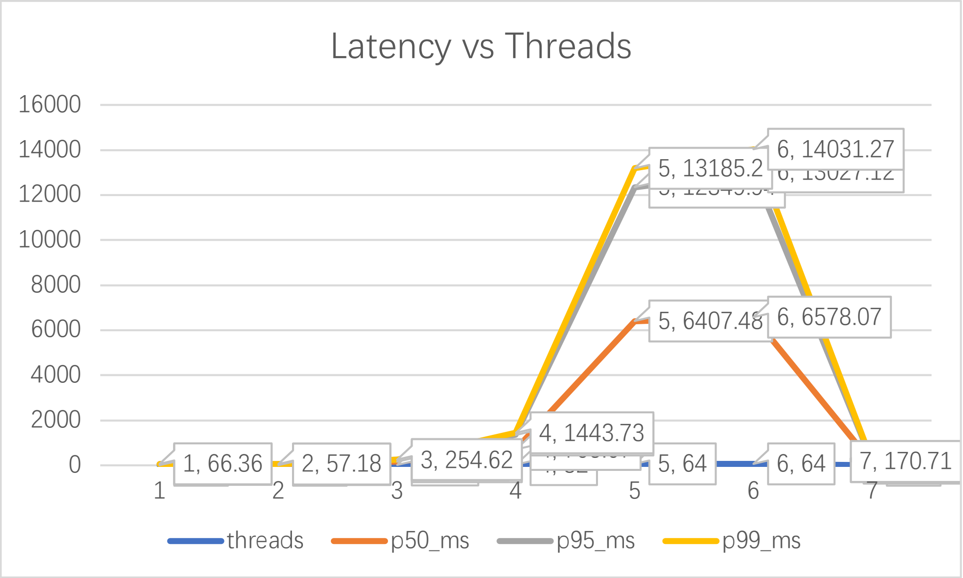
```
dd@zhaodixuandeMacBook-Pro ~ % cd ~/class/6650/client-part1
mvn -q -DskipTests clean package
java -cp target/client-part1-0.0.1-SNAPSHOT.jar com.chatflow.client.SingleClient
 ws://localhost:8080/chat/1

ACK: {"userId":123,"username":"user123","message":"hello","timestamp":"2025-10-1
0T23:13:39.827218Z","messageType":"TEXT","roomId":"1","serverTimestamp":"2025-10
-10T23:13:39.890970Z","status":"OK"}
```

- Screenshot of Part 2 output (detailed metrics)

```
dd@zhaodixuandeMacBook-Pro ~ % cd ~/class/6650/client-part2
mvn -q -DskipTests clean package
dd@zhaodixuandeMacBook-Pro client-part2 % java -cp target/client-part2-0.0.1-SNA
PSHOT-shaded.jar \
  com.chatflow.client2.app.MainApp ws://localhost:8080/chat 4 2000 5 5 100
boot base=ws://localhost:8080/chat threads=4 total=2000 rooms=5
progress sent=2000 acks=2000 fail=0 q=0 inflight=0
progress sent=2000 acks=2000 fail=0 q=0 inflight=0
MAIN sent=2000 acks=2000 fail=0 time=2.19s send_tps=912.7 ack_tps=912.7 p50=96.8
6ms p95=162.94ms p99=170.71ms mean=93.27ms min=4.41ms max=172.53ms conn=4 reconn
=0 L(avgInFlight)=0.0
dd@zhaodixuandeMacBook-Pro client-part2 %
```

- Performance analysis charts





- Evidence of EC2 deployment (EC2 console screenshot)