CS6650 Assignment 1 Report (1C Baseline)

Basic Info
- Course: CS6650
- Assignment: Assignment 1 - WebSocket Chat Server and Client
- Name/NEU ID: `Lihan Zhou/002339887`
- Date: `2026-02-13`
- Git Repository URL: `https://github.com/Eternity1824/chatflow.git`

---

1. Repository Deliverables

This repository includes all required directories:

- `/server`: Netty-based WebSocket server (`com.chatflow.server.ChatServer`)
- `/client-part1`: basic multithreaded load test client
- `/client-part2`: detailed metrics client (latency/statistics/CSV)
- `/results`: generated CSV and throughput chart assets
- README/instructions:
  - `AGENTS.md` and script/config usage are documented
  - runtime config is under `config/client.yml`
  - helper scripts are in `scripts/`

Run commands used in this project:

```bash
./gradlew build
./gradlew :server:run
./gradlew :client-part1:test
./gradlew :client-part2:test
scripts/run-client.sh
python3 scripts/plot_throughput.py
```

---

2. Design Document (<=2 pages)

2.1 Architecture Overview

```mermaid
flowchart LR
    MG[Message Generator Thread] --> Q[Blocking Queue]
    Q --> ST1[Sender Thread 1]
    Q --> STN[Sender Thread N]
    ST1 --> CP[Connection Pool]
    STN --> CP
    CP --> WS[WebSocket Server /chat/{roomId}]
    WS --> CP
    CP --> M[Metrics Collector]
    M --> CSV1[results/metrics.csv]
    M --> CSV2[results/summary.csv]
    M --> CSV3[results/throughput_10s.csv]
    CSV3 --> PLOT[scripts/plot_throughput.py]
```

Server-side request path:

1. `RoomIdExtractorHandler`: parse roomId and handle `/health`
2. `WebSocketServerProtocolHandler`: HTTP->WebSocket handshake
3. `BackpressureHandler`: toggle `AUTO_READ` by writability
4. `WebSocketChatHandler`: parse JSON, validate, enforce JOIN/TEXT rule, echo response

Client-side (Part 2) data path:

1. `MessageGenerator` generates templates and pushes to queue
2. `SenderThread` pulls templates, serializes JSON, sends with retry/backoff and batching
3. `ConnectionPool` manages persistent per-room channels and handshake limits
4. `WebSocketClientHandler` receives ACK, computes latency, records per-message metrics
5. `DetailedMetricsCollector` outputs summary + CSV + throughput buckets

2.2 Major Classes and Responsibilities

Server:

- `server/src/main/java/com/chatflow/server/ChatServer.java`
  - bootstrap Netty pipeline, event loop, and socket options
- `server/src/main/java/com/chatflow/server/RoomIdExtractorHandler.java`
  - roomId extraction from `/chat/{roomId}` or query; `/health` response
- `server/src/main/java/com/chatflow/server/WebSocketChatHandler.java`
  - JSON parse (streaming), schema validation, status response writing
- `server/src/main/java/com/chatflow/server/BackpressureHandler.java`
  - pause/resume channel reads when write buffer crosses watermark
- `common/src/main/java/com/chatflow/protocol/MessageValidator.java`
  - protocol validation (`userId`, `username`, `message`, timestamp, type)

Client Part 1:

- `client-part1/src/main/java/com/chatflow/client/ChatClient.java`
  - warmup + main phase orchestration
- `client-part1/src/main/java/com/chatflow/client/SenderThread.java`
  - basic send loop with retries and flush behavior
- `client-part1/src/main/java/com/chatflow/client/ConnectionPool.java`
  - shared WebSocket connection management
- `client-part1/src/main/java/com/chatflow/client/MetricsCollector.java`
  - aggregate throughput/connections stats

Client Part 2:

- `client-part2/src/main/java/com/chatflow/client/ChatClient.java`
  - full experiment orchestration + CSV export
- `client-part2/src/main/java/com/chatflow/client/MessageGenerator.java`
  - randomized data generation with room and type distribution
- `client-part2/src/main/java/com/chatflow/client/SenderThread.java`
  - per-room batching, retry (max 5), exponential backoff + jitter
- `client-part2/src/main/java/com/chatflow/client/ConnectionPool.java`
  - connection reuse, reconnect tracking, handshake concurrency control
- `client-part2/src/main/java/com/chatflow/client/WebSocketClientHandler.java`
  - ACK parse and per-message latency accounting
- `client-part2/src/main/java/com/chatflow/client/DetailedMetricsCollector.java`
  - latency percentiles, room throughput, type distribution, CSV writers

2.3 Threading Model

Warmup phase (required format):

- 32 sender threads
- each sends 1000 messages
- total warmup messages = 32,000

Main phase:

- configurable thread count (`mainThreads`; auto fallback: `max(32, CPU*4)`)
- remaining messages sent after warmup
- one dedicated message-generator thread always feeds queue

Concurrency design:

- producer-consumer with `BlockingQueue`
- sender threads do not generate messages; they only send
- channels are reused via connection pool (persistent WebSocket preferred)
- retries use exponential backoff to avoid synchronized retry storms

2.4 WebSocket Connection Management Strategy

Implemented strategy:

- key by `(roomId, index)` for bounded per-room connection reuse
- health of channel checked before reuse; broken channels removed
- in-flight connect deduplication via `inFlightConnections`
- handshake timeout and bounded concurrent handshakes (`Semaphore`)
- reconnect counter increments when stale channel is replaced
- on write failure:
  - remove connection
  - retry up to 5 times
  - apply exponential backoff with jitter

Backpressure handling:

- server: `BackpressureHandler` toggles `AUTO_READ`
- client sender: waits briefly when channel non-writable (`parkNanos`)
- batching reduces flush overhead and syscall frequency

2.5 Little's Law Calculation and Prediction

Little's Law: `L = lambda * W`

- `L`: average number of in-flight messages
- `lambda`: throughput (msg/s)
- `W`: average response time (s)

Pre-implementation conservative estimate (one outstanding request per connection):

1. Assume active connections `C = 20` (from single-core run summary)
2. Measured single-message RTT `W_single = 19.01 ms` (using measured mean latency as proxy)
3. Predicted throughput `lambda_pred = C / (W_single / 1000) = 20 / 0.01901 = 1,052.08 msg/s`

Pipeline-aware estimate (used by this implementation):

1. Let `k` be avg in-flight messages per connection (batch + async write effect)
2. Effective `L = C * k`

3. Predicted throughput `lambda_pred_pipe = (C * k) / W`

Observed-to-predicted comparison:

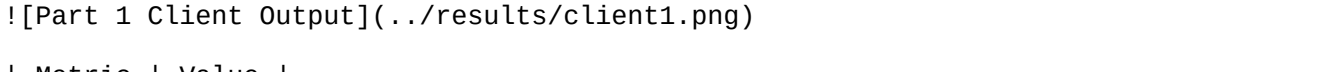
- observed throughput (single-core run): `68,989.30 msg/s` (2026-02-07)
- predicted throughput:
  - conservative: `1,052.08 msg/s`
  - pipeline-aware: `68,989.30 msg/s` (with effective `k=65.57` in-flight msgs/connection)
- gap explanation:
  - asynchronous pipelining allows multiple in-flight messages per connection
  - batching and non-blocking flush reduce per-message overhead
  - server/client event loops process frames concurrently

---

3. Test Results and Evidence

3.1 Part 1 Output (Basic Metrics)

Part 1 screenshot:
- `results/client1.png`



Metric	Value
successful messages	`4,983,925`
failed messages	`0`
total runtime	`72,242 ms`
throughput	`68,989.30 msg/s`
total connections	`20`
reconnections	`0`

3.2 Part 2 Output (Detailed Metrics)

Part 2 screenshot:
- `results/console.png`

Metric	Value
mean latency	`19.01 ms`
median latency	`19 ms`
p95 latency	`24 ms`
p99 latency	`27 ms`
min latency	`9 ms`
max latency	`152 ms`
status 200	`4,983,925 (99.68%)`
status 400	`16,075 (0.32%)`
JOIN	`475,986 (9.52%)`
LEAVE	`238,850 (4.78%)`
TEXT	`4,269,089 (85.38%)`
UNKNOWN	`16,075 (0.32%)`

3.3 Performance Charts

Throughput-over-time chart (10s buckets):

- input: `results/archive/single-core-2026-02-07/throughput_10s.csv`
- script: `python3 scripts/plot_throughput.py`
- output image: `results/archive/single-core-2026-02-07/throughput_10s.png`

3.4 Single-Core Metrics

Finalized single-core run data (archive: `single-core-2026-02-07`):

Metric	Value
successful_messages	`4,983,925`
error_responses	`16,075`
failed_messages	`0`
total_runtime_ms	`72,242`
throughput_msg_per_sec	`68,989.30`
total_connections	`20`
reconnections	`0`
mean_latency_ms	`19.01`
median_latency_ms	`19`
p95_latency_ms	`24`
p99_latency_ms	`27`

Data sources:

- `results/archive/single-core-2026-02-07/summary.csv`
- `results/archive/single-core-2026-02-07/metrics.csv`
- `results/archive/single-core-2026-02-07/throughput_10s.csv`
- `docs/README.md` (latency summary and distributions)

3.5 EC2 Deployment Evidence

Screenshots:
- EC2/console evidence: `results/console.png`
- health endpoint proof: `results/health.png`

Example verification commands (for appendix):

```bash
curl http://<ec2-public-ip>:8080/health
```

---

4. Requirement Checklist

- [x] WebSocket server endpoint + roomId handling
- [x] `/health` endpoint
- [x] protocol validation and error responses
- [x] multithreaded client with warmup + main phases
- [x] dedicated message-generation thread + queue
- [x] retry/reconnect strategy

- [x] detailed latency and CSV metrics (Part 2)
- [x] throughput-over-time visualization pipeline
- [x] final screenshots inserted into PDF
- [x] final single-core numeric table filled
- [x] EC2 evidence inserted

---

5. Appendix (Optional)