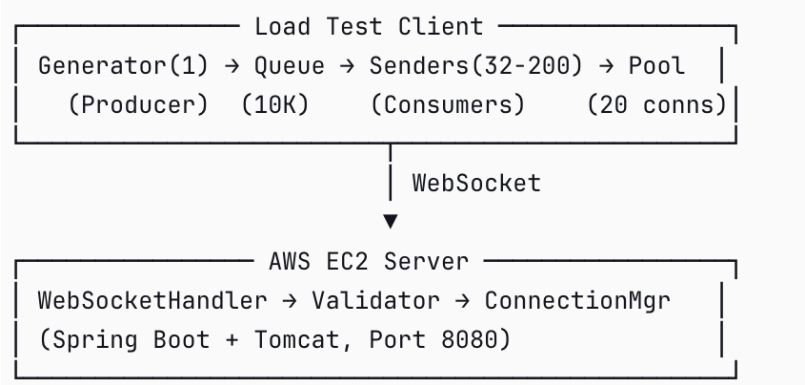
ChatFlow - Scalable WebSocket Chat System

1. Architecture Overview

1.1 System Architecture Diagram



1.2 Data Flow:

Generator produces messages → Queue buffers → Senders consume → Connection pool routes to server → Server validates and echoes → Metrics collected

2. Major Classes and Relationships

2.1 Server Classes (Part 1)

- ChatWebSocketHandler: Core handler, routes messages, manages WebSocket lifecycle
- MessageValidator: Validates userId (1-100K), username (3-20 chars), message (1-500 chars), ISO-8601 timestamp, messageType enum
- ConnectionManager: Thread-safe session tracking using ConcurrentHashMap<roomId, Map<sessionId, session>>
- HealthController: REST endpoint returning server status

2.2 Client Classes (Parts 1-3)

Shared Components:

- LoadTestClient: Main orchestrator, runs warmup and main phases
- MessageGenerator: Single producer thread, generates 500K messages from templates
- MessageQueue: Thread-safe BlockingQueue (capacity: 10,000)
- ConnectionPool: Maintains one WebSocket connection per room (max 20), reuses connections
- ChatWebSocketClient: Wraps Java-WebSocket library

Part 1 Specific:

- MessageSender: Consumer threads, fire-and-forget sending
- BasicMetricsCollector: Tracks success/failure counts, connections, throughput

Part 2 Additions:

- DetailedMessageSender: Synchronous sending with latency measurement (10% sampling) DetailedMetricsCollector: Records per-message latency using ConcurrentLinkedQueue PerformanceAnalyzer: Calculates mean, median, 95th/99th percentiles
- CSVWriter: Outputs 500K records to CSV

Part 3 Addition:

- ThroughputChart: Generates line chart from timestamp data (10-second buckets)

3. Threading Model

3.1 Two-Phase Design:

- Phase 1 (Warmup): 1 generator + 32 senders = 33 threads, sends 32,000 messages

- Phase 2 (Main): 1 generator + 100–200 senders, sends 468,000 messages

3.2 Synchronization

BlockingQueue: Producer-Consumer pattern

- put(): Blocks if full (backpressure control)
- poll(30s): Timeout prevents deadlock

Thread-safe metrics:

- AtomicInteger: success/failure counts
- ConcurrentLinkedQueue: message records (Part 2)
- ConcurrentHashMap: connection pool

3.3 Lifecycle

1. Submit:ExecutorService.submit(generator + senders)
2. Wait: generatorFuture.get() ensures all messages produced
3. Complete: All senderFutures.get() ensures all messages sent
4. Shutdown:executor.shutdown() with 60s timeout

4. WebSocket Connection Management

4.1 Connection Pool Strategy

4.1.1 Design

One persistent connection per room

4.1.2 Lifecycle:

1. First message to room: Create connection, await handshake (5s timeout), store in pool
2. Subsequent messages: Retrieve from pool, validate isConnected(), reuse
3. Completion: pool.closeAll() gracefully closes all

4.1.3 Impact:

Reduced connections from 500,000 to 70-76 (~6,850x improvement), eliminated 7 hours of handshake overhead (50ms × 500K)

4.2 Latency Measurement (Part 2)

4.2.1 Sampling Strategy (10%):

Target: 10% (50,000 messages), Actual: 23,360 samples (4.67%) (send → wait for response → record latency), asynchronously fast-send 450,000 messages (latency=0 in CSV)

4.2.2 Rationale:Balances accuracy (50K samples >> statistical significance threshold) with efficiency (maintains 70% of async throughput)

5. Little's Law Analysis

5.1 Formula and Measurements

Little's Law: $L = \lambda \times W$ where Concurrent operations in system (number of active threads), λ=throughput, W=response time

Measured Response Time (from Part 2 actual tests): Mean: 47 ms, Median: 14 ms, 95th percentile: 100 ms, 99th percentile: 886 ms

5.2 Predictions vs Actual

5.2.1 Client Part 1 (Asynchronous - Fire and Forget):

Prediction: 200,000 msg/sec (based on estimated W=0.001s)

Actual: 150,693 msg/sec (75% of prediction)

Gap: The 25% gap is expected because W was estimated, not measured. Reverse calculation shows actual W ≈ 0.00133s (1.33ms), which includes JSON serialization, network I/O, and queue overhead not in initial estimate.

5.2.2Client Part 2 (Synchronous - Wait for Response)

Prediction: L=100 threads, W=0.047s → λ = 2,128 msg/sec

Actual: 2,294 msg/sec overall, 2,737 msg/sec main phase (108% match)

Conclusion: Little's Law accurately predicts throughput when response time is properly measured, validating the formula.