Project 4: Part I
Reddit-like Engine Simulation

Martin Kent and Alex Vargas

November 1, 2025

Abstract

This report presents a Reddit-like social media engine simulator implemented in Gleam using the actor model. The system simulates concurrent user interactions including posting, commenting, voting, and direct messaging across multiple subreddits. The implementation leverages the actor model to handle thousands of concurrent client processes communicating with a centralized engine, with performance metrics collected through ping-pong latency measurements.

Contents

1 Overview

The Reddit-like engine simulation is built using Gleam's actor model, providing message-passing concurrency based on Erlang's OTP framework. The system consists of three main components: a centralized engine process that manages all data structures and handles state updates, independent client actor processes representing individual users that maintain local state and perform simulated activities, and a simulator coordinator process responsible for initialization and performance metric collection.

To simulate realistic behavior, the system uses two Zipf distributions: one for subreddit popularity with parameter $s = 1.0$ modeling how a few subreddits attract many subscribers while most have fewer members, and another (soft distribution) for user activity levels with $s = 0.5$ creating varying engagement patterns with some highly active users and others less engaged. The Zipf probability mass function $P(k) = \frac{1/k^s}{\sum_{n=1}^{N} 1/n^s}$ determines these distributions, where k is the rank, s is the distribution parameter, and N is the total number of items.

To run the code run:

```
gleam build
gleam run num_of_clients
```

Listing 1: Running the Project

2 Engine Implementation

The engine maintains a centralized `EngineState` containing dictionaries for users, posts, comments, subreddits, user inboxes, direct messages, and a comment counter. All data structures use dictionary lookups for O(1) average-case access time, and the engine is implemented as a single actor process to ensure consistency. The engine processes incoming messages the `handle_message_engine` function.

User registration creates a new `User` record with karma initialized to zero, stores the user's subject for future communication, initializes an empty inbox, and updates both the users and inboxes dictionaries. Subreddit management includes creation which checks for existence before creating a new subreddit with empty members and posts, joining which adds the user to the subreddit's members list and adds the subreddit to the user's subscribed list, and leaving which removes these bidirectional associations using list filtering.

For content creation, posts are handled by generating a unique post ID using the current dictionary size, creating a `Post` record with metadata including user ID, subreddit ID, and content text, initializing empty comments list and zero vote counts, and adding the post ID to the subreddit's posts list. Comments support hierarchical structure by determining if the parent is a post (prefix "post") or comment (prefix "comment"), then either appending the comment ID to the post's comments list or recursively updating the parent comment's comments list while maintaining flat dictionary storage that preserves hierarchical relationships.

The voting system implements karma calculation where upvotes increment the target's upvote count and author's karma by one, while downvotes increment the target's downvote count and decrement the author's karma by one, with both content and user dictionaries updated atomically. Feed generation retrieves the user's subscribed subreddit list, gathers all post IDs from those subreddits, fetches full post records including nested

comment IDs, limits results to the most recent 100 posts if the feed exceeds this size, and sends a `ReceiveFeed` message to the requesting client.

The messaging system maintains a two-level structure where `DirectMessage` records store actual message content and `UserInbox` maps sender IDs to lists of message IDs. When sending a message, the system creates a message record, updates the recipient's inbox with a new entry if the sender hasn't messaged them before or appends to the existing list, and updates the direct messages dictionary. When requesting an inbox, the system retrieves all message IDs for the user, fetches the corresponding message records, and sends them to the client. The engine also implements a `Pong` message handler for latency measurement which receives pings with iteration numbers, prints current state statistics for posts, comments, and direct messages, and sends pong responses with timestamps for round-trip time calculation.

3 Client Implementation

Each client actor maintains local state including references to the simulator and engine subjects, its own subject for receiving messages, a unique user ID string, an activity timeout value that varies based on Zipf distribution, a list of subscribed subreddit IDs, a cached feed of posts, current karma value, and an inbox dictionary of direct messages. The `start_client` function creates a new actor with OTP initialization, sends a `RegisterAccount` message to the engine, initializes state with empty feed, zero karma, and empty inbox, sets up the message handler, and starts the actor process.

Upon receiving a `Connect` message, the client starts a periodic ticker with its assigned activity timeout interval that sends `ActivitySim` messages at regular intervals, with each activity message triggering one randomly selected action. The `ActivitySim` handler uniformly selects from six activities: direct messaging which either selects a random post author to message if the inbox is empty or replies to a random sender from the inbox if messages exist, posting which randomly selects one subscribed subreddit and sends a `PostInSubReddit` message with generated content, post interaction which randomly selects a post from the feed and randomly performs a comment, upvote, or downvote with equal probability, comment interaction which selects a random post and random comment from that post, requests the full comment data, then either recurses into nested comments with one-third probability or acts on the comment with two-thirds probability enabling deep thread exploration, feed refresh which sends a `RequestFeed` message to update the local cache, and inbox check which sends a `RequestInbox` message to update the local inbox cache.

The client handles several incoming message types including `ReceiveFeed` which updates the local feed cache, `ReceiveKarma` which updates the local karma value, `DirectMessageInbox` which updates the local inbox dictionary, `ClientJoinSubreddit` which sends `JoinSubreddit` requests for multiple subreddits, and `ActOnComment` which provides received comment data for interaction. The uniform random selection of activities creates a balanced mix of content creation at 16.7%, content interaction through commenting and voting at 33.3%, communication through messaging at 16.7%, and information retrieval through feeds and inbox at 33.3%, simulating realistic social media usage patterns where users spend significant time consuming and interacting with existing content.

4 Simulator and Tick Implementation

The simulator acts as a central coordinator responsible for initializing the engine and spawning client actors, creating subreddits and assigning memberships according to Zipf distribution, measuring system performance through ping-pong latency tests, and collecting and reporting performance metrics. The initialization sequence begins with generating two Zipf distributions: one for subreddit popularity with $s = 1.0$ and $n =$ num_clients/2, and another for client activity levels with $s = 0.5$ and $n =$ num_clients. The `zipf_distribution` function generates probability weights by calculating $1/\text{rank}^s$ for each rank, summing all weights, and normalizing by dividing each weight by the total.

Client spawning calculates an activity timeout for each client from the Zipf distribution using the formula wait $= \min\left(\frac{0.05}{\text{scale} \times \text{zipf\_weight}}, 30.0\right)$ seconds where the scale factor equals $\sqrt{\text{num\_clients}}$, then creates client actors with unique IDs and stores their subjects in a dictionary. This creates highly active clients with low timeouts and less active clients with high timeouts, simulating realistic engagement patterns. Subreddit creation and assignment creates $n/2$ subreddits named "subreddit1", "subreddit2", etc., then uses the `assign_subreddits` function to distribute memberships where each user joins an average of 5 subreddits, the number each user joins follows the client Zipf distribution, and users weighted-randomly select subreddits based on the subreddit Zipf distribution ensuring popular subreddits get more members. After setup completes, the simulator sends `Connect` messages to all clients which start their individual ticker loops, and begins the performance monitoring ticker.

The tick implementation consists of two components: per-client tickers provided by `tick.gleam` that run in separate spawned processes with infinite loops that sleep for the client's interval, send an `ActivitySim` message, and recurse, creating asynchronous independent behavior with intervals varying from 50ms to 30,000ms based on Zipf distribution; and the simulator's performance ticker that sends 10 ping messages at 1-second intervals where each ping records a timestamp before sending, pong responses trigger response time calculations, and after 10 iterations the simulation ends and reports results.

Performance metrics collection maintains two dictionaries mapping iteration numbers to send timestamps and round-trip times in seconds. Round-trip calculation occurs when receiving a pong by getting the current timestamp, retrieving the send timestamp for that iteration, calculating the difference in seconds, and storing the result. The engine's pong handler prints system state statistics including total posts created, total comments created, and total direct messages sent, providing insight into system activity level and growth rate. Key simulation parameters include the number of clients specified via command-line argument, number of subreddits set to num_clients/2, average of 5 subscriptions per user, zipf_s_subreddits of 1.0 for high concentration in popular subreddits, zipf_s_clients of 0.5 for moderate concentration in active users, and simulation duration of 10 seconds.

5 Results and Performance Analysis

5.1 Experimental Setup

The simulation was run with a 100-second duration, collecting performance metrics at 1-second intervals. The system tracked engine response delay (ping-pong round-trip time),

total counts of posts, comments, and direct messages, as well as delta values representing content generation rates per second. The test scenario initialized clients with Zipf-distributed activity levels and subreddit memberships, allowing the system to reach steady-state behavior and then observe performance degradation under sustained load.

5.2 Response Time and Throughput

The engine delay measurements reveal a clear performance degradation pattern as the simulation progresses. Initially, response times remain extremely low with delays under 1ms for the first 27 seconds, indicating the system handles early load efficiently. The first significant delay spike occurs at iteration 28 with 45.8ms, followed by fluctuations in the 10-60ms range through iteration 64. At iteration 65, response times begin exponential growth, starting at 589ms and increasing to 17.7 seconds by iteration 98. This sharp degradation indicates the engine reaches a critical threshold around 21,000 posts, 5,000 comments, and 12,000 messages where the single-process architecture becomes overwhelmed by message queue buildup.

Content generation rates demonstrate consistent throughput despite increasing response times. Post creation maintains steady rates between 250-450 posts per second for the first 64 iterations, with a notable pattern of approximately 300-380 posts/second in the latter stable period. Comment generation starts slowly with only 8 comments in the first 8 seconds due to the lack of initial posts to comment on, then stabilizes around 90-130 comments per second once sufficient content exists. Direct message rates follow a similar cold-start pattern, beginning at 1-5 messages per second and reaching steady-state at 200-300 messages per second after iteration 30. The consistency of these delta values even as engine delay increases suggests clients continue generating activity independently while the engine struggles to keep pace with processing.

Table 1: System Performance at Key Intervals

Time	**Delay (s)**	**Posts**	**Comments**	**Messages**	**ΔPosts**	**ΔComments**	**ΔMsgs**
0	0.0002	18	0	1	18	0	1
10	0.0000	2,940	160	470	249	26	83
20	0.0002	6,293	686	1,787	300	59	135
30	0.0348	9,531	1,440	3,575	263	78	176
40	0.0045	12,815	2,314	5,706	298	80	233
50	0.0077	16,127	3,286	7,984	312	99	210
60	0.0463	19,454	4,351	10,402	339	104	246
64	0.1526	20,786	4,764	11,406	368	102	264
70	3.6551	22,773	5,452	12,987	336	126	249
80	6.5500	26,099	6,539	15,635	334	107	275
90	10.9723	29,489	7,674	18,362	326	115	296
98	17.7323	32,169	8,622	20,596	359	122	269
99	18.6550	32,476	8,707	20,859	307	85	263

5.3 Distribution Analysis

The Zipf distributions successfully created realistic user behavior patterns as evidenced by the content generation characteristics. The cold-start period for comments and messages

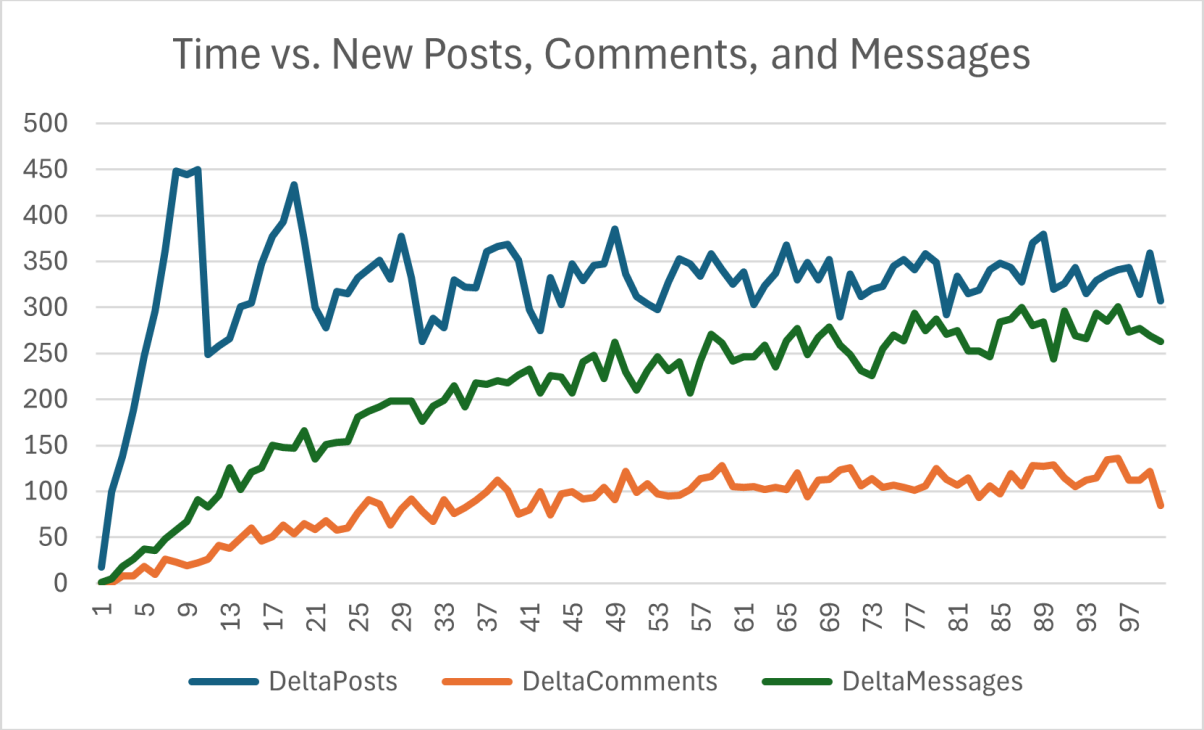

Figure 1: Zipf Activity Deltas

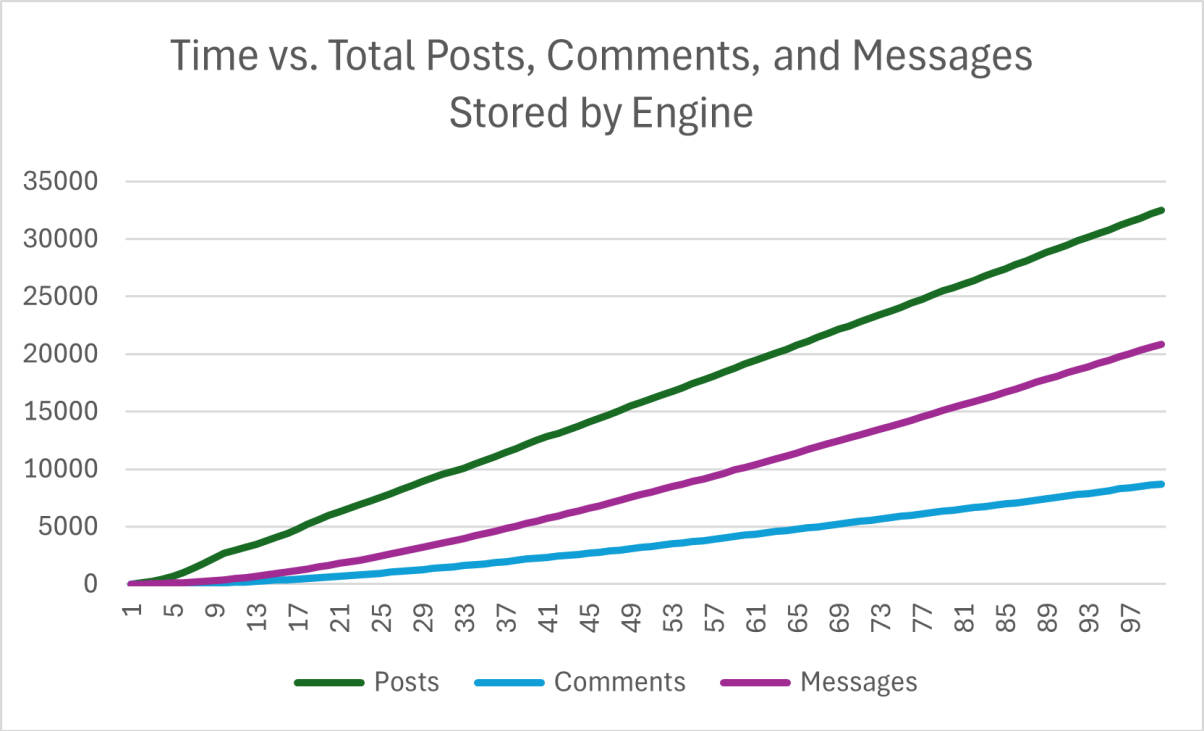

Figure 2: Activity Load Distribution

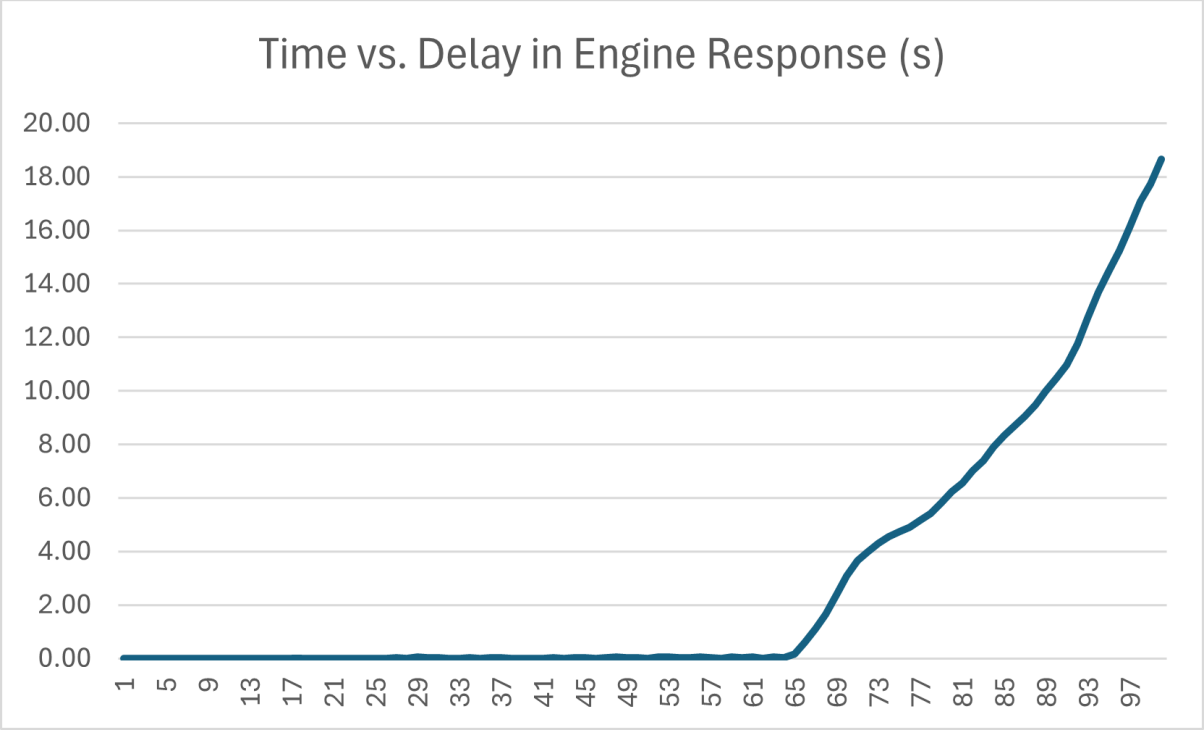

Figure 3: Engine Response Delay

demonstrates that less active users (with longer timeouts) took time to begin participating, while highly active users (short timeouts) immediately began posting. The steady-state post generation rate of approximately 320 posts per second across 100 clients implies an average posting interval of roughly 0.3 seconds per client, but the Zipf distribution with $s = 0.5$ means this is heavily skewed.