The 25 Million TPS Challenge

Architecture of a Massive Scale Distributed Counter System

System Design Masterclass

Abstract

This document serves as a comprehensive technical guide to designing a distributed system capable of handling 25 million write operations per second (concurrent). It moves beyond high-level abstractions to explore the physical limitations of hardware, the physics of network packets, kernel-level interrupt handling (NAPI), and advanced aggregation patterns required to survive "Web Scale" traffic.

Contents

1 The Problem Statement

Imagine you are the Lead Architect for a streaming platform hosting the **World Cup Final**.

- **Traffic:** 25 Million concurrent users.
- **Event:** Every user clicks a ”Like” button or sends a heartbeat simultaneously.
- **Throughput:** 25 Million Writes Per Second (WPS).
- **Latency Requirement:** Updates must be reflected globally within 3 seconds.
- **Constraint:** High Availability (Active-Active). The system cannot go down.

Educator’s Note

Why this breaks standard architectures: A standard database (Postgres/MySQL) handles 10k TPS. A standard Kafka broker handles 100k TPS. To handle 25M TPS directly, you would need clusters of thousands of servers, which is cost-prohibitive and operationally impossible to manage. We must change the *physics* of the data.

2 Phase 1: The Network Entry (The Front Door)

Before a packet even hits our servers, it must traverse the global internet.

2.1 Geo-DNS and Route Optimization

When a user in Mumbai clicks ”Like”, the request does not go to a server in Virginia. We use **Geo-DNS**.

1. The client resolves `api.live.com`.
2. The Name Server checks the client’s Source IP.
3. If IP is Asia-based, it returns the VIP (Virtual IP) of the Mumbai Data Center.

2.2 Anycast Routing (BGP)

For high availability, we use **Anycast**.

- Multiple Data Centers (Mumbai, Singapore, Tokyo) announce the *same* IP address range via BGP (Border Gateway Protocol).
- Internet routers (Core/Backbone) calculate the shortest topological path (lowest hops).
- **Failover:** If the Mumbai DC goes dark (stops announcing BGP), routers automatically shift the next packet to Singapore. This happens at the network hardware layer, transparent to the user.

3 Phase 2: The Physics of the Edge (Deep Dive)

This section explains why we cannot simply ”scale up” bandwidth. We run into the limitations of the CPU and OS Kernel.

3.1 The Network Packet Tax

The user sends a payload of just 4 bytes: `{id: 999}`. However, the internet requires wrapping.

Layer	Size (Bytes)	Purpose
Ethernet Header	14	MAC Addresses (Physical Routing)
IP Header	20	IP Addresses (Logical Routing)
TCP Header	20	Sequence Numbers, Flags, Ports
HTTP Headers	100+	Host, User-Agent, Cookies, Content-Length
Payload	**4**	**The actual data**
Total	**158**	

$$\text{Efficiency} = \frac{4}{158} \approx 2.5\%$$

We are wasting 97.5% of our bandwidth on headers.

3.2 The Interrupt Storm (Why CPUs Die)

If we allow 25 million packets per second (PPS) to hit our servers, we trigger an **Interrupt Storm**.

Deep Dive: Under the Hood

The Life of a Packet (Kernel Level):

1. **NIC (Network Interface Card):** Receives electrical signal. Uses DMA to copy packet to a Ring Buffer in RAM.
2. **IRQ (Hard Interrupt):** NIC signals the CPU: "Stop! Data is here."
3. **Context Switch 1:** CPU freezes the Application (User Mode), saves registers, switches to Kernel Mode.
4. **ISR (Interrupt Service Routine):** CPU acknowledges the interrupt.
5. **SoftIRQ:** The Kernel processes the packet (Checksums, Firewall rules).
6. **Context Switch 2:** CPU copies data to the Socket Buffer and switches back to User Mode.

The Math: A context switch takes $\approx 1\mu s$. At 1M PPS, the CPU spends 1 second per second just switching context. Utilization hits 100% with 0% application work done.

3.3 The Solution: Edge Aggregation (Micro-Batching)

To solve this, we place **Edge Servers** (Golang/Rust/Nginx) at the network edge.

The Logic:

1. Accept the TCP connection.
2. Read the HTTP Request.
3. **DO NOT** call the database. **DO NOT** call Kafka yet.
4. Update a local variable in RAM: `local_counters[match_id]++`.

5. Return `200 OK` immediately.

The Flush (NAPI Logic applied to App Layer): Every 1 second, a background timer wakes up. It takes the value (e.g., 5,000) and sends **ONE** message to the backend.

Impact:

- Input: 25,000 requests/sec (per server).
- Output: 1 message/sec (per server).
- **Reduction Factor:** 25,000x.

3.4 The Financial Impact: To Batch or Not to Batch?

A simple architectural decision—whether to write directly to Kafka or aggregate at the edge—has massive financial and operational implications.

Metric	Without Edge Aggregation	With Edge Aggregation
Write Volume	25,000,000 writes/sec	1,000 writes/sec (from 1k Edge Nodes)
Kafka Cluster Size	250+ High-End Brokers	3 Brokers (Minimum Cluster)
Network Efficiency	3% Data / 97% Overhead	99% Data / 1% Overhead
Consumer Nodes	1000+ CPUs to process stream	2-4 CPUs to process stream
Est. Monthly Cost	**$150,000+**	**$5,000**

Table 1: Infrastructure comparison: Direct Writes vs. Edge Aggregation

Educator's Note

Key Takeaway: In the "Without Aggregation" model, you are paying primarily for processing TCP headers and context switches, not for processing business value. The "Edge Aggregation" model shifts the complexity to the cheapest resource (RAM on the Edge) to save the most expensive resource (Network IO and Downstream Compute).

4 Phase 3: The Data Pipeline (Ingestion)

Now that we have compressed the traffic, we send it to the backend.

4.1 Kafka Architecture

Even with aggregation, we have thousands of Edge Servers sending data.

Educator's Note

The Hot Key Trap: A Junior Engineer partitions Kafka by `MatchID`.

- `hash(Match_999) % 1000 Partitions = Partition 42`.
- All traffic for the World Cup hits Partition 42.
- Partition 42 crashes. The system fails.

The Senior Solution: Random Partitioning We configure the Kafka Producer on the Edge Server to use a **Round-Robin** or **Random** strategy.

- Message 1 for Match 999 $\rightarrow$ Partition 0.
- Message 2 for Match 999 $\rightarrow$ Partition 1.
- Message 3 for Match 999 $\rightarrow$ Partition 2.

This spreads the load evenly across the entire Kafka cluster.

5 Phase 4: Stream Processing (Map-Reduce)

Since data for Match 999 is now scattered across all partitions, we need to sum it up. We use a **Stream Processor** (Flink, Spark, or custom Go consumers).

5.1 The Map Phase

We have 100 Consumers reading from 100 Kafka partitions.

- Consumer A reads Partition 0. It sees: `{match:999, val:500}`.
- Consumer B reads Partition 1. It sees: `{match:999, val:200}`.

Each consumer maintains a local sum in its own memory.

5.2 The Reduce Phase

Every few seconds, the consumers flush their local sums to the **Global Store**.

6 Phase 5: Global Consistency (Active-Active)

We have data centers in India and the US. How do we keep them in sync?

6.1 The Architecture

- **Region A (India):** Has its own Edge, Kafka, and Consumers. Calculates a "Region A Total".
- **Region B (US):** Has its own Edge, Kafka, and Consumers. Calculates a "Region B Total".

6.2 Global Aggregation

We do not replicate the raw stream (too expensive). We replicate the **result**.

- A Global Service reads `Total_A` and `Total_B`.
- `Global_Total = Total_A + Total_B`.
- This value is written to a Global Cache (e.g., Redis backed by CRDTs or simply aggregated on read).

7 Phase 6: Probabilistic Counting

For "View Counts", storing 25 Million User IDs to ensure uniqueness requires ≈ 200 MB of RAM per match.

$$25,000,000 \times 8 \text{ bytes (Long)} = 200 \text{ MB}$$

7.1 HyperLogLog (HLL) - For Unique Users

We use HLL when we need to answer "How many *unique* people are watching?".

- **Mechanism:** Hashes the User ID and looks at the number of leading zeros in binary.
- **Memory Usage:** Fixed ≈ 12 KB.
- **Accuracy:** 99.19% accurate (Standard Error 0.81%).
- **Benefit:** We can count billions of unique users with negligible memory.

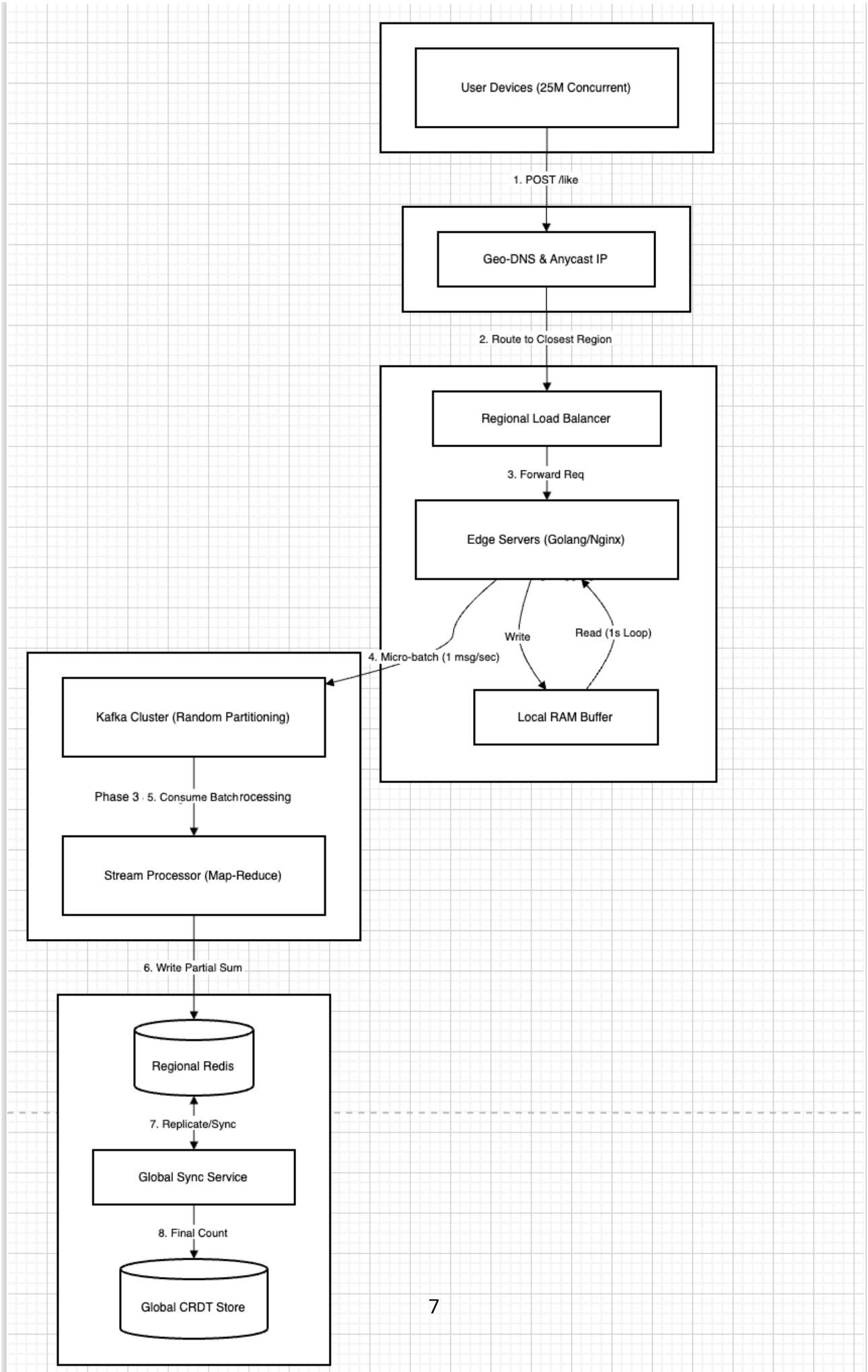
7.2 Count-Min Sketch (CMS) - For Frequency/Top-K

If we need to track "Which emojis are being used the most?", we cannot use a simple counter because there are thousands of possible emojis (and infinite combinations if we consider text).

- **Problem:** Tracking frequency of infinite items in limited memory.
- **Structure:** A 2D array of counters (width w, depth d) and d hash functions.
- **Write Operation:** When an emoji "Fire" arrives:
  - Hash it d times.
  - Increment the counter at the hashed index for each row.
- **Read Operation:** Hash "Fire" d times again. Check the counters. Take the **minimum** value among them.
- **Why Minimum?** Hash collisions cause over-counting. The minimum value is the one with the least collisions, thus the closest to the truth.

8 Appendix: Architecture Diagram

The following diagram illustrates the complete data flow, highlighting the compression at the edge and the map-reduce pattern in the backend.


User Devices (25M Concurrent)
1. POST /like
Geo-DNS & Anycast IP
2. Route to Closest Region
Regional Load Balancer
3. Forward Req
Edge Servers (Golang/Nginx)
Write
Read (1s Loop)
Local RAM Buffer
4. Micro-batch (1 msg/sec)
Kafka Cluster (Random Partitioning)
Phase 3 · 5. Consume Batch rocessing
Stream Processor (Map-Reduce)
6. Write Partial Sum
Regional Redis
7. Replicate/Sync
Global Sync Service
8. Final Count
Global CRDT Store