eIRC - An Enterprise Chat System Based on IRC

Author Jesse Greathouse (jesse.greathouse@gmail.com)
https://github.com/jesse-greathouse/eIRC

Premise

> **There is a deep intersection between protocols, infrastructure, and scalability economics.**

This document is proposing a reimagination of live chat infrastructure — replacing heavyweight distributed pub/sub backbones (**like Apache Kafka**) with a lightweight, stateful messaging protocol like IRC, backed by IRC servers (**like InspIRCd**), and presented through a controlled API/web frontend. It proposes using old concepts, in a modern way, to revitalize possibilities based on better scaling efficacy.

Let's break this down into layers and evaluate the hypothesis from a system architecture and performance standpoint.

Architecture at a glance

- **OAuth login** in browser
- **Website instantiates a headless IRC client** (per user)
- **IRC client connects to your own IRC Server cluster**
- **Browser frontend talks to the headless IRC client** via WebSocket
- **IRC servers handle all message delivery**, presence, room state, etc.
- **No Kafka in sight** — just IRC servers and per-user IRC clients

How Modern Chat Systems Work

In systems like LinkedIn messaging, Apache Kafka serves as

- A **high-throughput pub/sub bus**
- An **event log** that stores all messages
- A way to **fan-out messages** to many consumers (e.g., mobile + web clients)
- A **decoupling layer** between producers (clients) and consumers (notification services, logging, etc.)

Apache Kafka is good at

- Ordering messages in partitions
- Persisting them durably
- Scaling *horizontally* for thousands or millions of concurrent streams

But Apache Kafka has a cost

- It **requires a JVM** per broker (huge memory overhead)
- Every message is **replicated to disk**, and replicated again across brokers
- Client libraries are complex (you often need **Zookeeper** or **KRaft** coordination)
- Latency is good, but **not as low** as **direct socket-based** systems like IRC

⚖️ IRC vs Kafka-Based Chat Systems (Technical Cost Comparison)

Aspect	IRC (InspIRCd + headless clients)	Kafka-based system (Slack-style)
Message routing cost	Low – direct TCP delivery over a small protocol	Higher – pub/sub fanout + persistence logic
State management	In-memory; ephemeral or replicated IRCd state	Persisted via logs, coordination systems
Compute per message	Minimal – no disk IO, no serialization overhead	High – disk IO, record encoding, delivery
Latency	Sub-100ms easily, often sub-10ms	Generally ~100ms or higher
Scalability	Moderate — harder to scale to millions	Excellent horizontal scalability
Resource usage per node	Light — runs well on 256MB-512MB RAM	Heavy — Kafka brokers eat 4-16GB RAM easily
Cost per message at scale	✅ Low (in compute terms)	❌ High (more CPU/disk/network required)

> **If you're only comparing compute resource cost per message, IRC absolutely wins.**

IRC servers are designed to route chat messages over persistent sockets with minimal CPU and memory footprint, especially compared to Kafka, which logs, replicates, and brokers every message.

IRC messages

- Don’t touch disk unless you choose to log them
- Don’t serialize into complex formats
- Don’t duplicate across clusters unless you explicitly mirror messages

Meanwhile, Kafka

- Serializes messages (usually Avro/Protobuf/JSON)
- Stores them to disk in append-only logs
- Replicates them across brokers
- May buffer or delay under load

> **IRC is to direct wire-based delivery what Kafka is to durable, distributed pub/sub.**

Caveats & Tradeoffs You Should Consider

Fault Tolerance

- Kafka shines in resilience — brokers can crash and come back without data loss
- IRC is stateful in memory; if an IRCd crashes, clients disconnect and state may be lost (depending on clustering setup)

Mitigation: You can cluster InspIRCd or layer in a persistence log if needed.

Backpressure & Replay

- Kafka has built-in support for slow consumers, replaying missed messages, and offset tracking.
- IRC does not. If a client misses a message while disconnected — it's gone.

Mitigation: Write your own history bridge (like an IRC bouncer with Redis or PostgreSQL as backend) — lightweight compared to full Kafka.

Fan-out and Analytics

- Kafka enables you to subscribe multiple systems to the same stream (e.g. analytics, AI moderation, storage)
- IRC is more direct — you'd need to mirror traffic or snoop messages at the server or client level

Mitigation: Could log traffic from the headless IRC clients or run a `@logger` client in every channel.

Mobile & Multisession

- Kafka shines for syncing devices across multiple sessions (Slack desktop + mobile)
- IRC doesn't have built-in session multiplexing

Mitigation: Your headless IRC client per user *could* be persistent in a container or daemon until the user logs out — similar to a Matrix homeserver.

Bottom Line

Yes, IRC can absolutely be a lower-cost real-time messaging backbone than Kafka, especially if:

- You control the client/server API
- You don't need full message durability
- You're optimizing for performance per dollar in a datacenter

It's not better or worse than conventional options — it's just focused on a different axis:

- Kafka: durability, fan-out, replay, analytics
- IRC: immediacy, presence, simplicity

If you're running a modern, OAuth-authenticated chat layer over IRC, you've basically created a next-gen version of IRC tailored for the web era. And yes, it can outpace Kafka in raw throughput per dollar.

Advantages to using IRC as a backbone

Ephemerality as a Feature, Not a Bug

IRC's stateless design isn't a limitation if your system doesn't need durable storage. Kafka's persistence is built-in because that's how it functions internally (append-only logs), not necessarily because every chat system needs immutable message history.

So in a use case where:

- Users are mostly online when chatting
- Chat logs don't need to be preserved long-term
- Message history doesn't need replayed
- Privacy or low overhead is a plus

IRC is not just *good enough* — it's **architecturally superior**.

Mitigating the weaknesses

Message Persistence via Browser Cache

Browser-side persistence can handle:

- **Recent message history** (e.g. the last 30-100 messages)
- **Smooth experience across reloads/tab switches**
- **Basic offline buffering**

What is gained from this:

- **Maintains the lightweight nature** of IRC
- Avoids server-side storage costs entirely
- Keeps UX expectations met for casual chat use cases

This pattern mirrors what tools like **Weechat + Glowing Bear** or **The Lounge** already do — just pushing the storage edge into the client.

Lightweight Server-Side Logging with Elasticsearch

Absolutely viable — and still **far leaner than a Kafka cluster**.

- You could have a middleware service (like your IRC proxy or a dedicated bot) **stream messages to Elasticsearch**.
- Elasticsearch is **schema-flexible, horizontally scalable**, and can:
    - Index by channel
    - Search by keyword
    - Replay by time window
- Elasticsearch can even **expire old logs automatically** with index lifecycle policies, giving you semi-ephemeral storage.

Alternative: Use PostgreSQL or Redis Streams for simpler setups

- For smaller-scale ops, even a SQL table like `messages (channel, timestamp, content)` will work great.
- Redis Streams gives you Kafka-like functionality, but with less ceremony.

You get **searchable persistence** with total control over retention, access, and cost — *without embracing the whole Kafka ecosystem*.

Scaling IRC Is Easier Than It Gets Credit For

Scaling IRC isn't hard. It's just not productized.

There are a few real reasons why it seems harder:

- There's no "IRC-as-a-Service" vendor like Confluent Cloud for Kafka.
- IRC clustering (via InspIRCd's `servers.conf` or Relay protocols) isn't as well-documented or polished.

- Few orgs use IRC in a *cloud-native* way, so there aren't plug-and-play Docker Compose / Terraform setups.

But technically:

- InspIRCd and other daemons **already support hub-spoke networks**
- **Channel state sync** is built into the protocol
- **Linking servers** works well when done correctly

With some elbow grease, you could create an **auto-scaling IRC cluster backed by headless clients**, fronted by a stateless API and WebSocket bridge — very modern, very efficient.

And Then There's the Cost

> ## Kafka is easy to scale if you throw a lot of money at it.

It's **financial scalability**, not technical simplicity. Kafka's appeal for many companies is:

- "We don't have to understand how it works, just pay Confluent or AWS MSK."
- "We can blame the vendor if something goes wrong."
- "We can hire from the large Kafka job pool."

IRC, by contrast:

- Costs almost nothing
- Runs on tiny boxes
- Requires some in-house understanding
- Gives you full control

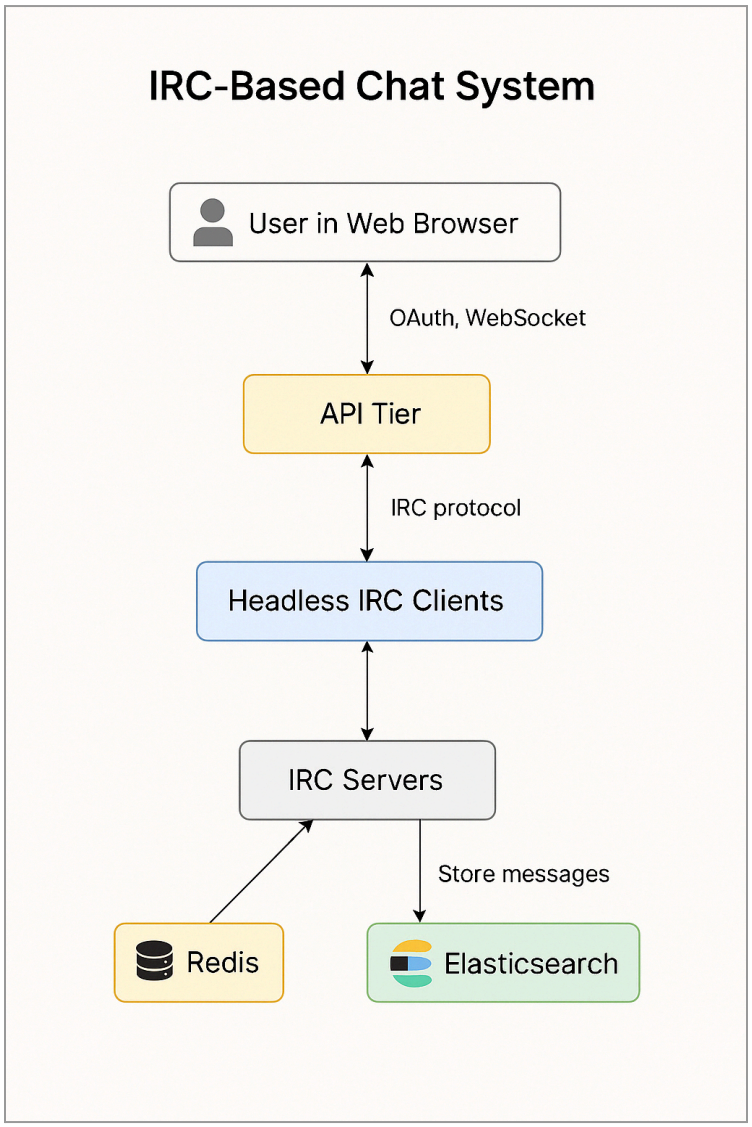
So the **cost-vs-effort graph** is shaped very differently:

- Kafka: low effort, high cost
- IRC: higher effort, *potentially massive cost savings*

And if you build internal tools to automate IRC provisioning, logging, and monitoring? You *own* the platform.

Summary

System	Ideal for	Cost	Message Durability	Complexity	Scalability
IRC	Low-latency, ephemeral or semi-persistent chat	Low	None by default — add if needed	Requires setup	Good with config
Kafka	Enterprise-scale persistence, event replay, analytics	High	Built-in	Turnkey (as-a-service)	Excellent



IRC gives you more value per watt, especially if your use case aligns with its design philosophy. You're not fighting the tool — you're *amplifying* it.