Minor Project Synopsis Report

Distributed-Rate-Limiter-System

Project Category: Deep-Tech

Projexa Team Id: 26E3179

Submitted in partial fulfilment of the requirement of the degree of

BACHELOR OF SCIENCE

in

COMPUTER SCIENCE

to

K.R. Mangalam University

by

Sidharth Krishna S	**(2401720003)**
Ankit Kumar Yadav	**(2401720012)**
Diwakar	**(2401720013)**
Palak Kashyap	**(2401840007)**
Devasya Dahiya	**(2401720001)**

Under the supervision of

Dr. Ravinder Beniwal

Department of Computer Science and Engineering

School of Engineering and Technology

K.R Mangalam University, Gurugram- 122001, India

January 2026

INDEX

ABSTRACT

Modern software systems extensively rely on Application Programming Interfaces (APIs) to deliver services such as authentication, data access, payments, and content delivery. With the rapid growth of cloud-native and microservices-based applications, APIs often experience unpredictable and high request volumes caused by traffic spikes, automated bots, or malicious attacks. Uncontrolled API traffic can lead to server overload, performance degradation, service downtime, and increased infrastructure costs.

The **Distributed Rate Limiter System** is designed to address these challenges by providing a scalable and reliable mechanism to control API request traffic in distributed environments. The system enforces configurable request limits per user, IP address, or API key within defined time windows, ensuring fair resource usage and protecting backend services from abuse. Unlike traditional single-server rate limiting approaches, this system maintains consistency across multiple application servers.

The proposed solution uses **Redis** as a centralized, high-performance in-memory datastore to manage request counters and tokens with low latency and atomic operations. Multiple rate-limiting algorithms, including **Token Bucket, Fixed Window, and Sliding Window**, are implemented to support diverse application requirements. Rate limit configurations are stored in a persistent database, allowing dynamic updates without redeployment.

Additionally, the system provides an **admin dashboard** for real-time monitoring, analytics, and configuration of rate limits, enabling administrators to visualize traffic patterns and detect violations. Overall, the Distributed Rate Limiter System enhances API reliability, security, scalability, and performance, making it a practical and industry-relevant solution for modern backend and distributed systems.

KEYWORDS: Distributed Systems, Rate Limiting, APIs, Redis, Backend Engineering, Traffic Control

1. INTRODUCTION

In recent years, the rapid growth of web and cloud-based applications has led to an increased reliance on Application Programming Interfaces (APIs) for communication between services. APIs play a critical role in enabling core functionalities such as user authentication, data exchange, payment processing, and content delivery. As applications scale and attract large numbers of users, they often face unpredictable traffic patterns caused by peak usage periods, flash sales, automated bots, or malicious attacks.

One of the major challenges in modern backend systems is controlling excessive API requests while maintaining system performance and availability. Without effective traffic control mechanisms, servers can become overloaded, leading to slow response times, service outages, and increased operational costs. Traditional rate-limiting techniques applied at a single-server level are insufficient in distributed environments, where multiple application instances handle requests concurrently.

To overcome these limitations, a **Distributed Rate Limiter System** is required. Such a system ensures consistent enforcement of request limits across all servers in a distributed architecture. By centralizing rate-limiting decisions and maintaining shared state, the system can prevent abuse, ensure fair resource usage, and improve overall system stability.

This project focuses on the design and implementation of a Distributed Rate Limiter System using modern backend technologies. The system leverages **Redis** as a centralized in-memory datastore for low-latency request tracking and supports multiple rate-limiting algorithms to accommodate different application requirements. Additionally, it provides real-time monitoring and configuration capabilities, making it suitable for real-world, industry-level backend applications.

2. MOTIVATION

With the rapid expansion of cloud-based and API-driven applications, managing incoming request traffic has become a critical challenge for modern backend systems. Applications today are expected to remain available, responsive, and secure even under heavy and unpredictable workloads. However, uncontrolled API traffic caused by sudden user spikes, automated scripts, or malicious attacks can easily overwhelm backend services, leading to downtime and degraded user experience.

Traditional rate-limiting techniques often operate at a single-server level and fail to scale effectively in distributed environments. As applications adopt microservices and horizontal scaling, enforcing consistent request limits across multiple servers becomes complex. This gap between theoretical scalability and practical traffic control motivated the development of a distributed rate-limiting solution.

The motivation behind this project is to design a system that not only prevents API abuse and server overload but also ensures fair resource utilization among users. By leveraging distributed system concepts and high-performance in-memory storage, the project aims to bridge the gap between academic concepts and real-world backend engineering practices.

Furthermore, this project serves as a practical exploration of deep-tech system design, demonstrating how backend infrastructure components such as Redis, rate-limiting algorithms, and monitoring tools work together to build reliable and scalable systems used in industry today.

3. LITERATURE REVIEW

- **Designing Data-Intensive Applications – Martin Kleppmann (2017)**

Martin Kleppmann's book *Designing Data-Intensive Applications* provides a detailed explanation of the challenges involved in building scalable and reliable distributed systems. The author emphasizes that in distributed environments, maintaining consistency across multiple application instances is a major concern. Systems that rely on local state at individual servers often fail to enforce global constraints accurately.

The book highlights the importance of shared state management and coordination mechanisms to ensure correct behaviour across distributed nodes. This concept is directly applicable to rate limiting, where request counters must remain synchronized across multiple servers. The literature suggests that without centralized or coordinated state storage, enforcing consistent request limits becomes unreliable. These principles form the theoretical foundation for implementing a centralized distributed rate limiter.

- **Redis Documentation – Redis In-Memory Data Store**

The official Redis documentation describes Redis as a high-performance, in-memory data structure store commonly used for caching, session management, and real-time analytics. Redis supports atomic operations, key expiration, and extremely low-latency read/write access, making it suitable for time-sensitive applications.

According to the documentation, Redis can efficiently manage counters and tokens using atomic commands, ensuring accuracy even under high concurrency. These features make Redis an ideal choice for implementing distributed rate limiting, where request counts must be updated consistently across multiple application servers. The documentation also highlights Redis's ability to handle time-based expiration, which is essential for implementing rate-limiting algorithms such as Fixed Window and Token Bucket.

- **AWS API Gateway – Rate Limiting and Throttling**

Amazon Web Services (AWS) API Gateway documentation explains how rate limiting and throttling are used to protect backend services from excessive traffic and denial-of-service attacks. AWS enforces request limits at different levels, such as per API, per user, or per client, to ensure fair usage and service availability.

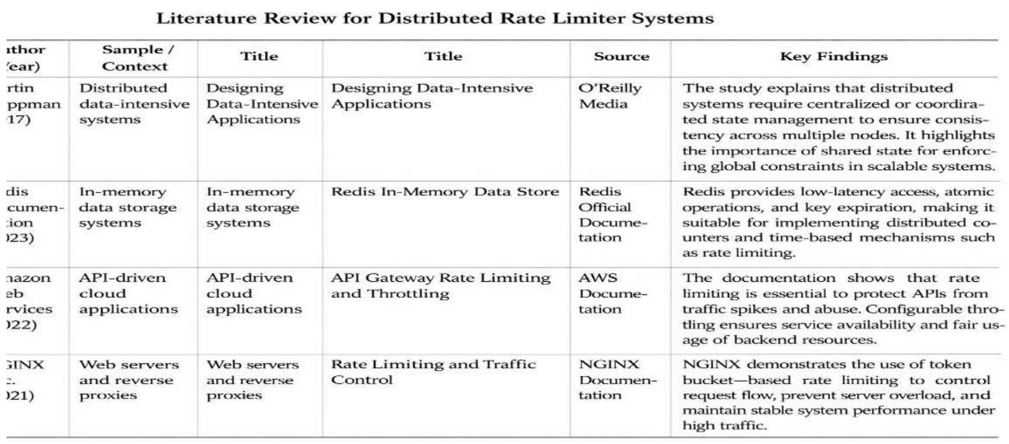
The documentation demonstrates that rate limiting is a critical component of modern API infrastructure. It also discusses how configurable thresholds and time windows help manage sudden traffic spikes. These industry practices reinforce the importance of implementing rate limiting as a core backend feature rather than an optional add-on. The concepts presented by AWS validate the real-world relevance of building a distributed rate limiter system.

- **NGINX Documentation – Rate Limiting and Traffic Control**

The NGINX documentation explains rate limiting as a mechanism to control the rate of incoming requests and protect servers from overload. NGINX implements rate limiting using algorithms such as the Token Bucket method, allowing controlled bursts of traffic while maintaining an average request rate. According to the documentation, rate limiting improves system stability and prevents resource exhaustion caused by abusive clients.

The use of shared memory zones in NGINX for maintaining request state highlights the necessity of centralized tracking in distributed environments. These concepts directly influence the design of the Distributed Rate Limiter System by demonstrating practical algorithm usage and traffic control strategies in production systems.

Table: Sample of Literature Review Table



Literature Review for Distributed Rate Limiter Systems

ıthor 'ear)	Sample / Context	Title	Title	Source	Key Findings
rtin ppman 17)	Distributed data-intensive systems	Designing Data-Intensive Applications	Designing Data-Intensive Applications	O'Reilly Media	The study explains that distributed systems require centralized or coordinated state management to ensure consistency across multiple nodes. It highlights the importance of shared state for enforcing global constraints in scalable systems.
dis cumen- ion)23)	In-memory data storage systems	In-memory data storage systems	Redis In-Memory Data Store	Redis Official Docume-tation	Redis provides low-latency access, atomic operations, and key expiration, making it suitable for implementing distributed counters and time-based mechanisms such as rate limiting.
ıazon eb rvices)22)	API-driven cloud applications	API-driven cloud applications	API Gateway Rate Limiting and Throttling	AWS Docume-tation	The documentation shows that rate limiting is essential to protect APIs from traffic spikes and abuse. Configurable throtling ensures service availability and fair usage of backend resources.
ӀINX :.)21)	Web servers and reverse proxies	Web servers and reverse proxies	Rate Limiting and Traffic Control	NGINX Documentation	NGINX demonstrates the use of token bucket—based rate limiting to control request flow, prevent server overload, and maintain stable system performance under high traffic.

4. GAP ANALYSIS

Early research and studies on large-scale systems primarily focused on understanding **distributed data management and consistency issues**. Works such as *Designing Data-Intensive Applications* emphasized the importance of shared state and coordination in distributed environments but did not provide concrete, application-level implementations for controlling API request traffic. These studies were more theoretical in nature and focused on system design principles rather than practical traffic control mechanisms.

Subsequent technological advancements introduced **in-memory data stores** like Redis, which enabled fast and atomic operations suitable for real-time applications. While Redis documentation highlights its efficiency and suitability for distributed counters, it does not present a complete, end-to-end system design for API rate limiting. The responsibility of integrating Redis into a full backend solution is left to developers.

Industry solutions such as **AWS API Gateway** and **NGINX** demonstrate practical implementations of rate limiting and traffic control. However, these solutions are often **platform-dependent**, tightly coupled with specific infrastructures, and provide limited flexibility for customization at the application level. Additionally, their internal working mechanisms are abstracted, reducing transparency for learning and experimentation.

From the literature reviewed, it is evident that there exists a gap between **theoretical distributed system concepts** and **customizable, application-level rate limiting implementations**. Most existing solutions either focus on theory without practical deployment or provide ready-made tools without offering flexibility or insight into internal design.

This project addresses the identified gap by designing and implementing a **custom Distributed Rate Limiter System** that combines theoretical principles with practical backend engineering. The system uses Redis for centralized coordination, supports multiple rate-limiting algorithms, allows dynamic configuration, and provides real-time monitoring—thereby bridging the gap between academic research and real-world system-level applications.

5. PROBLEM STATEMENT

Modern backend and cloud-based applications depend heavily on APIs to deliver essential services such as authentication, data exchange, and communication between distributed components. As these systems scale to handle large numbers of users, they often experience **unpredictable and high volumes of API requests** due to traffic spikes, automated bots, or malicious activities. Uncontrolled API traffic can result in **server overload, degraded performance, unfair resource usage, and service downtime**.

Existing rate-limiting techniques are typically applied at a single-server level and fail to function effectively in **distributed environments**, were multiple application servers process requests concurrently. This leads to **inconsistent enforcement of request limits** and increases the risk of API abuse.

Hence, the problem is to design and implement a **Distributed Rate Limiter System** that can **consistently regulate API request traffic across multiple servers**, enforce fair usage policies, prevent abuse, and maintain system stability with **low latency and high scalability**.

6. OBJECTIVES

- To design and develop a **distributed rate limiting mechanism** capable of handling high-volume API requests.
- To enforce **fair usage policies** by limiting requests per user, IP address, or API key.
- To implement **multiple rate-limiting algorithms** such as Token Bucket, Fixed Window, and Sliding Window.
- To ensure **consistent rate limit enforcement** across multiple application servers in a distributed architecture using Redis.
- To prevent **API abuse, request flooding, and brute-force attacks**, thereby improving system security.
- To improve **application performance and reliability** by reducing server overload while ensuring scalability and ease of integration.

7. Tools/Technologies Used

For this project, we have used modern backend and distributed system technologies that are suitable for building scalable, reliable, and high-performance applications. Each technology was selected based on its efficiency, industry relevance, and suitability for implementing a distributed rate limiting system.

PROGRAMMING LANGUAGE: JAVA

Java is used as the core programming language for developing the Distributed Rate Limiter System due to its robustness, platform independence, and strong support for concurrent and distributed applications. Java is a general-purpose, object-oriented programming language that follows the principle of *Write Once, Run Anywhere*, making it suitable for enterprise-level backend systems. Its built-in support for multithreading, networking, and memory management makes it ideal for handling high volumes of API requests in distributed environments.

Reasons for Selecting Java:

1. High performance and scalability
2. Strong support for concurrency and multithreading
3. Platform independent
4. Rich ecosystem and libraries
5. Widely used in enterprise backend systems

BACKEND FRAMEWORK: SPRING BOOT

Spring Boot is used to develop RESTful services for the rate limiter. It simplifies application configuration, supports rapid development, and provides production-ready features required for scalable backend systems.

DISTRIBUTED DATA STORE: REDIS

Redis is used as a centralized in-memory data store to maintain request counters and tokens. Its low latency, atomic operations, and key expiration features ensure consistent rate limiting across distributed servers.

DATABASE: POSTGRESQL

MONITORING & DEPLOYMENT: **PROMETHEUS & GRAFANA**

8.METHODOLOGY

The methodology of the Distributed Rate Limiter System is based on a **distributed client–server architecture** designed to efficiently control API request traffic in high-load environments. The system ensures fairness, consistency, and scalability while maintaining low latency.

Initially, incoming client requests are received by the **Application Server**, which acts as the entry point for all API calls. Before executing any business logic, the application server forwards essential request details such as user identifier, IP address, or API key to the **Rate Limiter Service**. This pre-processing step ensures that unauthorized or excessive requests are blocked early, reducing unnecessary load on backend services.

The **Rate Limiter Service** is responsible for evaluating each request against predefined rate limit rules. These rules specify the maximum number of requests allowed within a particular time window and are configurable based on application requirements. To support different traffic patterns, the system implements multiple rates limiting algorithms such as **Token Bucket, Fixed Window, and Sliding Window**.

For maintaining consistency across distributed servers, **Redis** is used as a centralized in-memory data store. Redis stores counters and tokens associated with each client and performs atomic operations to prevent race conditions. Its key expiration feature helps manage time-based limits efficiently. By using Redis as a shared state, the system ensures that rate limits are enforced uniformly across all application instances.

Rate limit configurations and metadata are stored in **PostgreSQL**, allowing administrators to update rules dynamically without redeploying the application. Based on the evaluation result, the Rate Limiter Service either allows the request to proceed or denies it by returning an **HTTP 429 (Too Many Requests)** response.

All rate limiting decisions are logged and monitored using an **admin dashboard**, enabling real-time analysis of API traffic, violations, and system performance. This methodology ensures a scalable, reliable, and industry-ready solution for managing API request traffic in distributed systems.

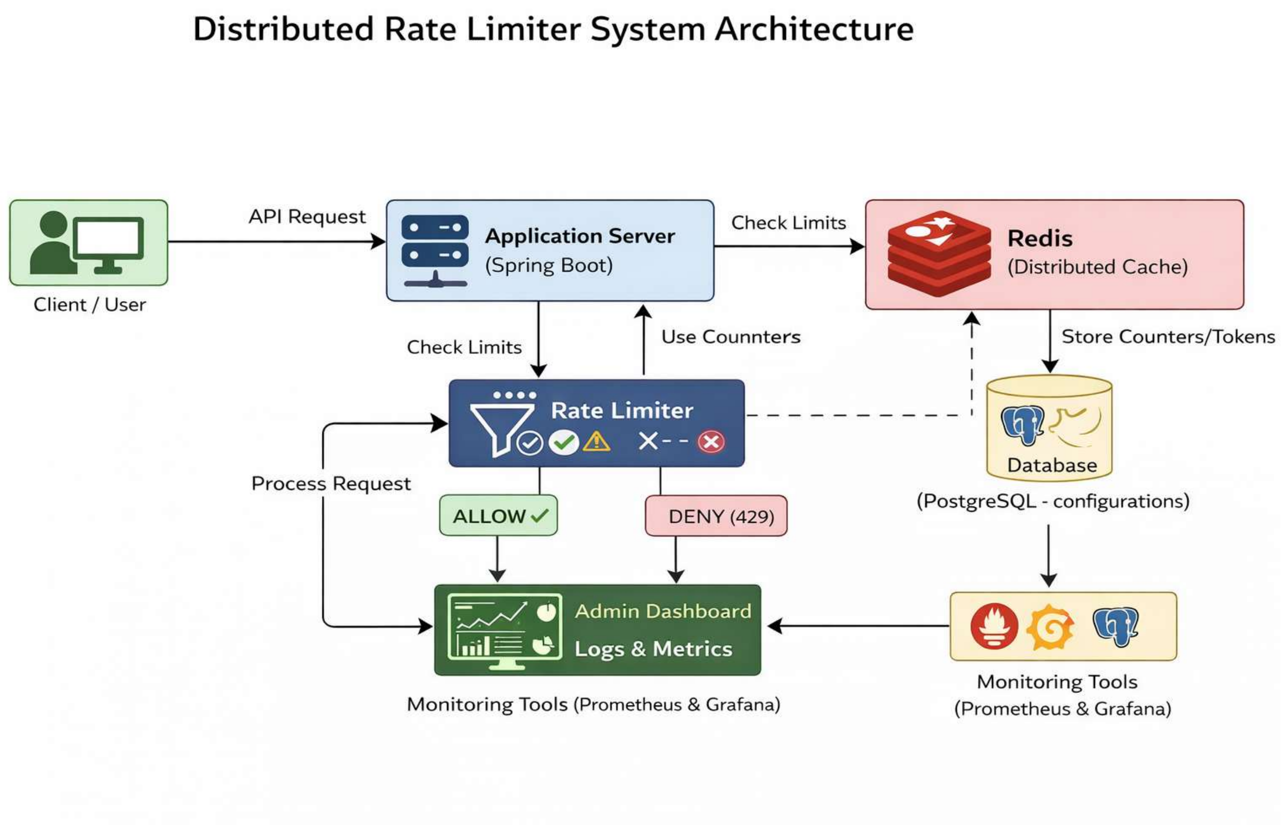


Figure 2: Architectural Diagram

REFERENCES

- **Kleppmann M. (2017)**
  Designing Data-Intensive Applications. O'Reilly Media.
  Reference for distributed systems, scalability, and consistency principles.

- **Redis Documentation**
  Redis: In-Memory Data Structure Store.
  https://redis.io/docs/
  Reference for centralized counters, atomic operations, and low-latency data access.

- **Amazon Web Services (AWS)**
  API Gateway – Request Throttling and Rate Limiting.
  https://docs.aws.amazon.com/apigateway/
  Industry reference for real-world API rate limiting and traffic control.

- **NGINX Documentation.**
  Rate Limiting and Traffic Control.
  https://docs.nginx.com/nginx/admin-guide/security-controls/controlling-access/
  Practical implementation reference for rate limiting algorithms and request control.