### **System Design: The Rate Limiter - Grokking Modern System Design Interview for Engineers & Managers**

The document focuses on designing a rate limiter, which is a critical component in modern system design, particularly for APIs and services that need to handle high traffic efficiently and securely. The document is part of a comprehensive course aimed at preparing engineers and managers for system design interviews. Below is a detailed summary of the document's contents:

#### **Introduction**

* **Definition**: A rate limiter restricts the number of requests a service processes within a specified timeframe. It blocks any requests exceeding the predefined limit, ensuring service stability and preventing abuse.
* **Purpose**: Rate limiters serve as a defensive mechanism to protect services from overuse, whether intentional or accidental. They are essential in mitigating denial-of-service (DOS) attacks, brute-force attempts, and other abusive behaviors.

#### **Why Rate Limiters are Necessary**

1. **Preventing Resource Starvation**:
   * Rate limiters help avoid situations where resources are exhausted due to excessive requests, which can occur during DOS attacks or due to software/configuration errors.
2. **Managing Policies and Quotas**:
   * Rate limiters ensure fair resource usage by applying limits on the time duration or quantity allocated to users. This prevents any single user from monopolizing shared resources.
3. **Controlling Data Flow**:
   * In systems processing large amounts of data, rate limiters distribute the load evenly across machines, preventing any single machine from becoming a bottleneck.
4. **Avoiding Excess Costs**:
   * Rate limiters control operational costs by preventing uncontrolled experiments and limiting free-tier usage to prevent unexpectedly high bills.
5. **Throttling API Requests**:
   * By limiting the number of requests to API servers, rate limiters ensure that the servers can handle the load without degradation in performance.

#### **Designing a Rate Limiter**

The document outlines a structured approach to designing a rate limiter, covering the following key aspects:

1. **Requirements**:
   * **Functional Requirements**: Define the types of throttling (e.g., fixed window, sliding window, token bucket) and the locations where the rate limiter can be placed (e.g., client-side, server-side).
   * **Non-Functional Requirements**: Ensure the rate limiter is scalable, reliable, and has low latency.
2. **High-Level Design**:
   * Provides an overview of the system architecture, highlighting the major components and their interactions. It focuses on scalability, fault tolerance, and performance optimization.
3. **Detailed Design**:
   * Delves into the specifics of the rate limiter’s implementation, discussing various building blocks such as databases, caches, and load balancers. It includes considerations for data storage, synchronization, and request handling.
4. **Rate Limiter Algorithms**:
   * **Fixed Window Algorithm**: Limits requests in fixed time intervals but can lead to burstiness at window boundaries.
   * **Sliding Window Algorithm**: Provides smoother rate limiting by accounting for requests in a sliding time window.
   * **Token Bucket Algorithm**: Allows for bursty traffic by using tokens that accumulate over time.
   * **Leaky Bucket Algorithm**: Ensures a constant request rate by processing requests at a fixed rate.
5. **Quiz**:
   * Includes a quiz to assess the reader's understanding of rate limiters, reinforcing the concepts covered in the document.

### **Use Cases and Scenarios**

* **APIs and Services**: Commonly used in APIs to prevent overuse and protect backend services.
* **Cloud Services**: Helps manage free-tier limits and control costs.
* **Data Processing Systems**: Ensures even distribution of data processing tasks across multiple machines.

### **Conclusion**

Rate limiters are essential for maintaining the reliability and stability of modern services. They prevent resource starvation, manage policies and quotas, control data flow, and avoid excess costs. The document provides a comprehensive guide to understanding and designing effective rate limiters, making it a valuable resource for system design interviews and practical implementation.

2. Requirements of a Rate Limiter’s Design.pdf

PDF

As an expert in system design, summarize the following document in great detail.

ChatGPT

### **Requirements of a Rate Limiter’s Design - Grokking Modern System Design Interview for Engineers & Managers**

This document focuses on the essential requirements and design considerations for implementing a rate limiter, an important component in system architecture for managing and controlling the flow of requests to ensure system reliability and performance. Below is a detailed summary of the document's contents:

#### **Introduction**

* **Purpose**: The document aims to outline the functional and non-functional requirements of a rate limiter, different types of throttling, appropriate placement strategies, and models for implementing a rate limiter.

#### **Functional Requirements**

1. **Request Limiting**: The rate limiter must restrict the number of requests a client can send to an API within a specified time window.
2. **Configurability**: The limit of requests per time window should be configurable to accommodate different use cases and requirements.
3. **Notification**: The system should notify clients when they exceed the defined threshold, either through an error message or another form of notification.

#### **Non-functional Requirements**

1. **Availability**: The rate limiter must be highly available to ensure it can consistently protect the system from excessive requests.
2. **Low Latency**: Since all API requests pass through the rate limiter, it should introduce minimal latency to maintain a good user experience.
3. **Scalability**: The design should be scalable to handle an increasing number of client requests over time without degradation in performance.

#### **Types of Throttling**

1. **Hard Throttling**: Enforces a strict limit on the number of API requests. Any requests exceeding this limit are discarded.
2. **Soft Throttling**: Allows a slight exceedance of the predefined limit by a certain percentage (e.g., 5%), providing some flexibility.
3. **Elastic or Dynamic Throttling**: The number of allowable requests can exceed the limit if the system has available resources. There is no specific upper limit percentage.

#### **Placement of the Rate Limiter**

1. **Client-side**: Placing the rate limiter on the client side is straightforward but insecure, as it can be easily tampered with.
2. **Server-side**: Placing the rate limiter on the server side is more secure and reliable. The rate limiter intercepts requests before they reach the API servers.
3. **Middleware**: The rate limiter acts as middleware, throttling requests before they reach the API servers. This approach can be effective and flexible, depending on the technology stack and organizational priorities.

#### **Models for Implementing a Rate Limiter**

1. **Centralized Database Model**:
   * **Description**: Rate limiters interact with a centralized database (e.g., Redis or Cassandra) to store, retrieve, and update counters.
   * **Advantages**: Ensures a single source of truth for rate limits, preventing clients from exceeding the limit.
   * **Drawbacks**: Can cause increased latency and potential race conditions due to high concurrency.
2. **Distributed Database Model**:
   * **Description**: Uses a cluster of independent nodes with a distributed database to track rate limits.
   * **Advantages**: Better fault tolerance and scalability.
   * **Drawbacks**: Potential for clients to momentarily exceed rate limits if state synchronization is delayed. Requires sticky sessions to ensure clients consistently interact with the same node.

#### **Building Blocks**

The rate limiter design utilizes several key building blocks:

1. **Databases**: Store rules defined by service providers and metadata of users.
2. **Caches**: Cache rules and user data for frequent access to reduce latency.
3. **Queues**: Hold incoming requests that are allowed by the rate limiter, ensuring orderly processing.

### **Conclusion**

The document provides a comprehensive guide to understanding the requirements and designing a rate limiter. It covers the critical functional and non-functional requirements, types of throttling, appropriate placement strategies, and implementation models. By following these guidelines, one can design a robust rate limiter that ensures system reliability, scalability, and performance.

3. Design of a Rate Limiter.pdf

PDF

As an expert in system design, summarize the following document in great detail.

ChatGPT

### **Design of a Rate Limiter - Grokking Modern System Design Interview for Engineers & Managers**

This document provides a comprehensive guide to the design of a rate limiter, an essential component in distributed systems for controlling the rate of requests. Below is a detailed summary of its contents:

#### **Overview**

* **Objective**: To design rate limiters that effectively gauge and throttle the resources being used across a system.
* **Topics Covered**:
  + High-level design
  + Detailed design
  + Request processing
  + Handling race conditions
  + Ensuring the rate limiter is not on the client’s critical path
  + Conclusion

#### **High-Level Design**

* **Deployment**: The rate limiter can be deployed as a separate service that interacts with web servers to decide if a request should be forwarded based on predefined rules.
* **Example**: Lyft’s open-sourced rate limiting component, which defines rules such as limiting marketing messages to five per day.

#### **Detailed Design**

The high-level design expands into several components, each serving a specific purpose:

1. **Rule Database**:
   * **Function**: Stores the rules defined by the service owner.
   * **Content**: Specifies the number of requests allowed per client per unit time.
2. **Rules Retriever**:
   * **Function**: Periodically checks and updates rules from the database to the cache.
   * **Benefit**: Ensures the rate limiter uses the most current rules without querying the database every time.
3. **Throttle Rules Cache**:
   * **Function**: Caches the rules to serve requests faster than accessing persistent storage.
   * **Benefit**: Increases system performance by reducing latency.
4. **Decision Maker**:
   * **Function**: Makes decisions based on the rules in the cache using rate-limiting algorithms.
   * **Process**: Checks if a request exceeds the allowed limit and either forwards it or rejects it.
5. **Client Identifier Builder**:
   * **Function**: Generates a unique ID for each request, used as a key in the key-value database.
   * **Identifiers**: Could be a combination of remote IP address, login ID, etc.
6. **Handling Exceeded Requests**:
   * **Strategies**:
     + Return HTTP 429 (Too Many Requests) response.
     + Queue requests for later processing if system overload occurs.

#### **Request Processing**

* **Process Flow**:
  1. **Identification**: Client identifier builder identifies the request.
  2. **Decision**: Decision maker checks the request count against the rules.
  3. **Forwarding**: If the request count is within the limit, it is forwarded to the request processor.
  4. **Throttling**: Based on throttling type (hard, soft, or elastic), requests are either served, queued, or rejected.

#### **Race Condition**

* **Problem**: High concurrency can cause race conditions using the "get-then-set" approach.
* **Solution**:
  + **Locking Mechanism**: Ensures only one process updates the counter at a time (causes potential bottleneck).
  + **Set-then-get Approach**: Incrementing values in a performant manner without locking.
  + **Sharded Counters**: Distributes load and reduces contention by using multiple shards.

#### **Critical Path Consideration**

* **Scenario**: High request volume can cause latency if rate limiting is on the critical path.
* **Solution**: Split the work into online and offline parts.
  + **Online**: Check and respond to the request quickly if it is within the limit.
  + **Offline**: Update the counters and cache asynchronously to reduce immediate computation load.

#### **Example**

* **Request Handling**:
  + A request with ID: 101 has a count of 3, a maximum limit of 5.
  + **Process**:
    1. Rate limiter checks the count (3) against the limit (5).
    2. Responds with "Allowed" if within limit.
    3. Updates the count offline to reduce latency.

#### **Conclusion**

* **Non-functional Requirements**:
  + **Availability**: Multiple rate limiters eliminate single points of failure.
  + **Low Latency**: Caching reduces the need for database access, improving response times.
  + **Scalability**: The number of rate limiters can be adjusted based on traffic volume.

By following these design principles, the rate limiter ensures high availability, low latency, and scalability, meeting the critical non-functional requirements for robust system performance.

4. Rate Limiter Algorithms.pdf

PDF

As an expert in system design, summarize the following document in great detail.

ChatGPT

### **Rate Limiter Algorithms - Grokking Modern System Design Interview for Engineers & Managers**

This document provides a detailed exploration of various rate limiter algorithms, which are crucial for controlling the rate of requests in distributed systems. Below is a comprehensive summary of its contents:

#### **Overview**

* **Objective**: To understand the working of different rate limiter algorithms, their parameters, advantages, and disadvantages.
* **Algorithms Covered**:
  + Token Bucket
  + Leaking Bucket
  + Fixed Window Counter
  + Sliding Window Log
  + Sliding Window Counter
  + Comparison of rate-limiting algorithms

#### **Token Bucket Algorithm**

* **Concept**: Utilizes a bucket filled with tokens at a constant rate. Each request consumes a token.
* **Flow**:
  + Adds a token to the bucket periodically.
  + Discards new tokens if the bucket is full.
  + Processes requests if there are enough tokens.
  + Accepts requests based on available tokens.
* **Essential Parameters**:
  + Bucket Capacity (C)
  + Rate Limit (R)
  + Refill Rate (R1)
  + Requests Count (N)
* **Advantages**:
  + Allows burst traffic as long as tokens are available.
  + Space-efficient.
* **Disadvantages**:
  + Locking mechanisms can increase processing delay.
  + Choosing optimal parameters is challenging.

#### **Leaking Bucket Algorithm**

* **Concept**: Similar to the token bucket but processes requests at a constant rate, like water leaking from a bucket.
* **Flow**:
  + Requests are added to a bucket.
  + Processed at a constant rate.
* **Essential Parameters**:
  + Bucket Capacity (C)
  + Inflow Rate (Rin)
  + Outflow Rate (Rout)
* **Advantages**:
  + Avoids burst traffic.
  + Space-efficient.
  + Suitable for stable outflow applications.
* **Disadvantages**:
  + Burst requests can fill the bucket and cause delays.
  + Determining optimal bucket size and outflow rate is difficult.

#### **Fixed Window Counter Algorithm**

* **Concept**: Divides time into fixed intervals (windows) and assigns a counter to each.
* **Flow**:
  + Increments the counter for each request in a window.
  + Discards requests if the counter exceeds the limit.
* **Essential Parameters**:
  + Window Size (W)
  + Rate Limit (R)
  + Requests Count (N)
* **Advantages**:
  + Space-efficient.
  + Services new requests within the window.
* **Disadvantages**:
  + Burst traffic at window edges can exceed limits.

#### **Sliding Window Log Algorithm**

* **Concept**: Tracks each incoming request's arrival time in a log (hash map).
* **Flow**:
  + Stores arrival times of requests.
  + Allows requests based on log size and arrival time.
* **Essential Parameters**:
  + Log Size (L)
  + Arrival Time (T)
  + Time Range (Tr)
* **Advantages**:
  + Avoids edge conditions.
* **Disadvantages**:
  + Consumes extra memory for storing timestamps.

#### **Sliding Window Counter Algorithm**

* **Concept**: Combines fixed window counter and sliding window log algorithms.
* **Flow**:
  + Maintains counts for previous and current windows.
  + Smooths out request bursts using a rolling window.
* **Essential Parameters**:
  + Rate Limit (R)
  + Window Size (W)
  + Requests in Previous Window (Rp)
  + Requests in Current Window (Rc)
  + Overlap Time (Ot)
* **Advantages**:
  + Space-efficient.
  + Smooths request bursts.
* **Disadvantages**:
  + Assumes even distribution of requests, which may not always be accurate.

#### **Comparison of Rate-Limiting Algorithms**

* **Memory Efficiency**: Refers to the number of states required for normal operation.
* **Burst Handling**: Ability to manage increased traffic within a time unit.
* **Summary Table**:
  + **Token Bucket**: Space-efficient, allows burst.
  + **Leaking Bucket**: Space-efficient, no burst.
  + **Fixed Window Counter**: Space-efficient, burst at window edges.
  + **Sliding Window Log**: Not space-efficient, no burst.
  + **Sliding Window Counter**: Space-efficient, smooths bursts.

#### **Conclusion**

* Each algorithm has unique advantages and disadvantages.
* The choice of algorithm depends on specific use cases and requirements.
* Factors like memory efficiency and burst handling are crucial in selecting the appropriate rate limiter algorithm.

By understanding these algorithms, one can design effective rate limiters that balance performance, resource utilization, and system stability.

ChatGPT can make mistakes. Check important info.

ChatGPT