**Chapter 4**

Design a Rate Limiter

Rate limiter – controls the rate of traffic sent by a client or service

* E.g. Instagram users cannot post more than 2 stories per second
* Advantages
  + Prevent resource starvation from DOS
  + Reduced cost – less server requirements and limit third-party API usage
  + Prevent servers from overloading

**Step 1 – Understand the problem and establish design scope**

* Can be implemented using different algorithms
* General requirements
  + Accurately limit excessive requests
  + Low latency – should not slow down HTTP responses
  + Use as little memory as possible
  + Distributed rate limiting – can be shared across multiple servers
  + Exception handling – show clear exceptions to users where they are throttled
  + Fault tolerance – problems with the limiter shouldn’t affect the entire system

**Step 2 – Propose high-level design and get buy-in**

*Where to put the rate limiter?*

* Client side – generally unreliable
  + Client requests can easily be forged by malicious actors
  + Server-side or middleware is more suitable
* Third-party
  + Many cloud microservice providers offers API gateway
  + API gateway – a fully managed service that supports rate limiting, SSL termination, authentication, IP whitelisting, servicing static content, etc.
  + Less control on the algorithm
* Server-side
  + Gives full control of the algorithm
  + Evaluate the technology stack before starting
  + Identify the best algorithm for the business needs
  + Requires more time to implement

*Algorithms*

* Token bucket
* Leaking bucket
* Fixed window counter
* Sliding window log
* Sliding window counter

*Token bucket algorithm*

* Simple and widely used
* Concept
  + Token bucket – a container with fixed capacity
  + Tokens generate in the bucket at a preset rate periodically
    - No additional tokens are added if bucket is full
  + Each request entering consumes one token
    - Requests go through if there are enough tokens available
    - Requests are dropped when the number of tokens are not enough
* Implementation
  + Takes two parameters
    - Bucket size – token capacity of the bucket
    - Refill rate – tokens generated every unit time
  + Different buckets for different API endpoints
  + If throttle request is IP based, each IP address requires a bucket
  + Consider a system-wide bucket
* Cons
  + Two parameters makes tuning challenging

*Leaking bucket algorithm*

* Similar to token bucket but requests are processed at a fixed rate
* Concept
  + Check if queue is full when request arrives
    - If not full, request is added to queue
    - If full, request is dropped
  + Requests are pulled from the queue and processed at a fixed interval
* Implementations
  + Takes two parameters
    - Bucket size and Outflow rate – rate of requests processed
* Cons
  + Poor handlings for bursts of traffic, as queue will be filled with old requests

*Fixed window counter algorithm*

* Concept
  + Intervals of time are divided with assigned counters to each window
  + Each request increments the counter by one
  + Once the counter reaches a fixed threshold, new requests are dropped until a new time window starts
* Cons
  + A burst of traffic at the edges of time windows can allow double the requests to flow through

*Sliding window log algorithm*

* Concept
  + Keeps track of a requested timestamps
  + When new requests enter, removed all outdated timestamps – older than the start of the current time window
  + Add timestamp of new request to the log
  + If the log size is greater than allowed count, request is rejected
* Cons
  + Consumes lots of memory

*Sliding window counter algorithm*

* Hybrid approach of fixed window counter and sliding window log
* Implementation
  + Requests in current window + requests in previous window \* overlap percentage of the rolling window and previous window
* Pros
  + Smooths out spikes in traffic because rate is based on the average rate of the previous window
  + Memory efficient

*Where shall we store the counters?*

* Database is not good idea due to slowness of disk access
* In memory cache is fast and supports time-based expiration strategies

**Step 3 – Design deep dive**

*High level architecture*

* Rules are stored on disk
  + Workers frequently pull rules from the disk and store them in the cache
* Use middleware to fetch counters and last request timestamps from Redis
* When a request is rate limited, return a HTTP response 429 and a X-Ratelimit-Retry-After header to the client
* Information return to the client
  + X-Ratelimit-Remaining - remaining allowed requests within the window
  + X-Ratelimit-Limit – amount of calls the client can make per time window
  + X-Ratelimit-Retry-After – time left until you can make new requests again

*Distributed environments*

* Two challenges supporting multiple servers and concurrent threads

Race condition – two requests concurrently read the counter value before either of them write the value back, each will increment the counter by one and write it back without checking the other thread

* Solutions
  + Locks
    - most obvious solution but slows down the system significantly
  + Lua scripts
  + Sorted sets data structure in Redis

Synchronization issues – synchronization of data when using multiple rate limiter servers

* Solutions
  + Sticky sessions – allows clients to send traffic to the same rate limiter
    - Not advisable due to lack of scalability and flexibility
  + Centralized data stores (like Redis)

*Performance Optimizations*

* Multiple data centers setup is critical
  + Latency is high for users located far away from the data center
* Synchronize data with an eventual consistency model
  + Eventual consistency – a guarantee that when an update is made in a distributed database, the update will eventually be reflected in all nodes that store the data, resulting in the same response every time the data is queried

*Monitoring*

* Goals
  + The rate limiting algorithms and rules are effective
* Test for:
  + If the rate limiting rules are too strict – too many valid requests are dropped

*Other considerations*

* Soft vs. hard rate limiting
  + Hard – number of requests cannot exceed threshold
  + Soft – request can conditionally exceed the threshold
* Rate limiting at different levels
  + Most of the strategies in the chapter is in the application layer
* Avoid being rate limited
  + Use client caches to avoid making frequent calls
  + Understand the limit and decide on an appropriate threshold
  + Add backup to retry logic and exception handling