Solution

Step 1: Gather the requirements
Functional Requirements:

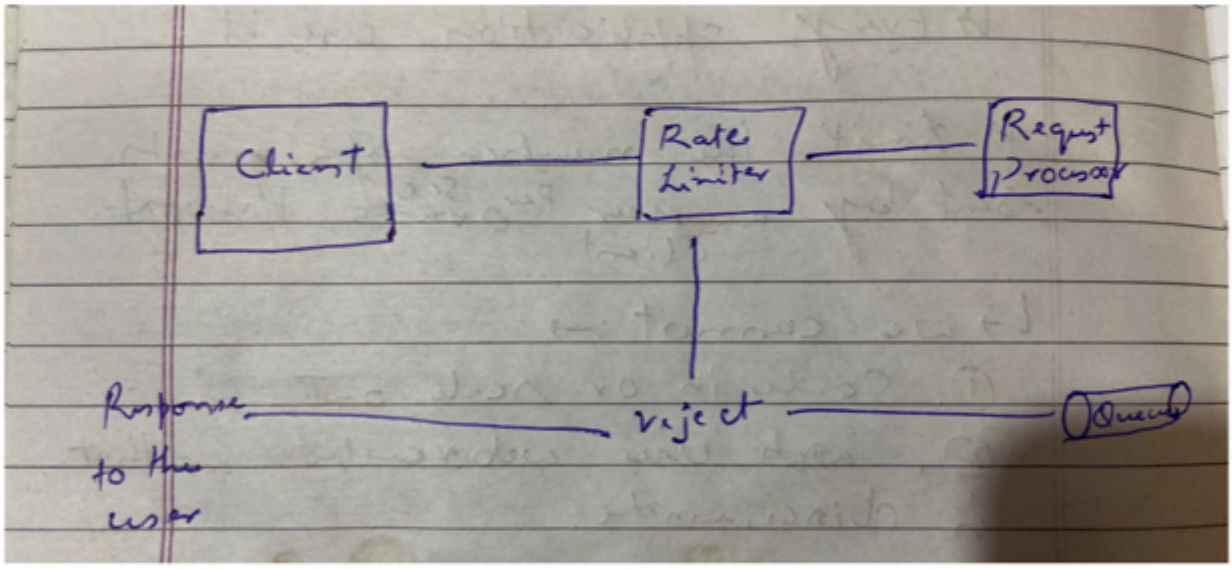
1. Limit the number of requests received from a client to an API to 15 requests per second
2. The user should get an error message whenever the threshold is exceeded within a single server or a cluster of servers

Non-functional Requirements:

1. Low latency - Make fast decisions
2. Accuracy - Be as accurate as possible
3. Scalability - Supports large number of hosts
4. Availability - Should be able to protect the service from external attacks

Step 2: Define microservices
The system will have only one service - Rate Limiting Service

Step 3: Draw the logical architecture



Step 4: Deep dive into the microservice
Let us see how we design our rate limiter service. We can use different algorithms for limiting rate limiting. Let us look at them one by one:

1. **Fixed Window Algorithm:** In this algorithm, the time window is considered from the start of the time-unit to the end of the time-unit. For example, a period would be considered 0-60 seconds for a minute irrespective of the time frame at which the API request has been made.

Let's take the example where we want to limit the number of requests per user. Under this scenario, for each unique user, we would keep a count representing how many requests the user has made and a timestamp when we started counting the requests. We can keep it in a hashtable, where the 'key' would be the 'UserID' and 'value' would be a structure containing an integer for the 'Count' and an integer for the Epoch time:

Key: Value
UserID: {count, startTime}

Our rate limiter is allowing 15 requests per minute per user, so whenever a new request comes in, our rate limiter will perform the following steps:

1. If the 'UserID' is not present in the hash-table, insert it, set the 'Count' to 1, set 'StartTime' to the current time (normalized to a minute), and allow the request.
2. Otherwise, find the record of the 'UserID' and if CurrentTime – StartTime >= 1 min, set the 'StartTime' to the current time, 'Count' to 1, and allow the request.
3. If CurrentTime - StartTime <= 1 min and
    a. If 'Count < 15', increment the Count and allow the request.
    b. If 'Count >= 15', reject the request.

Problems with this approach: This approach has the following drawbacks:

a. In the algorithm since we are resetting the 'StartTime', it can potentially allow twice the number of requests per minute in case a client sends more requests at the end of the minute.
b. **Atomicity**: In a distributed environment, the "read-and-then-write" behavior can create a race condition. Imagine if a client's current 'Count' is "14" and that it issues two more requests. If two separate processes served each of these requests and concurrently read the Count before either of them updated it, each process would think that the client could have one more request and that it had not hit the rate limit.

2. **Sliding Window Algorithm:** In this algorithm, the time window is considered from the fraction of the time at which the request is made plus the time window length. For example, if there are two messages sent at the 300th millisecond and 400th millisecond of a second, we'll count them as two messages from the 300th millisecond of that second up to the 300th millisecond of next second.
We can maintain a sliding window if we can keep track of each request per user. We can store the timestamp of each request in a Redis Sorted Set in our 'value' field of hash-table.

Key: Value
UserID: {Sorted Set <UnixTime>}

Whenever a new request comes in, the Rate Limiter will perform the following steps:

1. Remove all the timestamps from the Sorted Set that are older than “CurrentTime - 1 minute”.
2. Count the total number of elements in the sorted set. Reject the request if this count is greater than our throttling limit of “15”.
3. Insert the current time in the sorted set and accept the request.

3. **Sliding Window with counter Algorithm:** What if we keep track of request counts for each user using multiple fixed time windows, e.g., 1/60th the size of our rate limit’s time window. For example, if we have an hourly rate limit we can keep a count for each minute and calculate the sum of all counters in the past hour when we receive a new request to calculate the throttling limit. This would reduce our memory footprint. Let’s take an example where we rate-limit at 500 requests per hour with an additional limit of 10 requests per minute. This means that when the sum of the counters with timestamps in the past hour exceeds the request threshold (500), a client has exceeded the rate limit. In addition to that, it can’t send more than ten requests per minute. This would be a reasonable and practical consideration, as none of the real users would send frequent requests. Even if they do, they will see success with retries since their limits get reset every minute.
   We can store our counters in a Redis Hash since it offers incredibly efficient storage for fewer than 100 keys. When each request increments a counter in the hash, it also sets the hash to expire an hour later. We will normalize each ‘time’ to a minute.

Step 5: Identify the need for scale
Let us see how much memory we would need to use for various approaches we discussed above:

1. Fixed Window: Let’s assume the simple solution where we are keeping all of the data in a hash-table.
   Let’s assume ‘UserID’ takes 8 bytes. Let’s also assume a 2 byte ‘Count’, which can count up to 65k, is sufficient for our use case. Although epoch time will need 4 bytes, we can choose to store only the minute and second part, which can fit into 2 bytes. Hence, we need a total of 12 bytes to store a user’s data:
   $$8 + 2 + 2 = 12 \text{ bytes}$$
   Let’s assume our hash-table has an overhead of 20 bytes for each record. If we need to track one million users at any time, the total memory we would need would be 32MB:
   $$(12 + 20) \text{ bytes} * 1 \text{ million} \Rightarrow 32\text{MB}$$
   If we assume that we would need a 4-byte number to lock each user’s record to resolve our atomicity problems, we would require a total 36MB memory.

Also, if we assume a rate limit of 15 requests per second and 1 million users at a time, this would translate into 15 million QPS for our rate limiter

2. Sliding Window: Let's assume 'UserID' takes 8 bytes. Each epoch time will require 4 bytes. Let's suppose we need a rate limiting of 500 requests per hour. Let's assume 20 bytes overhead for hash-table and 20 bytes overhead for the Sorted Set. At max, we would need a total of 12KB to store one user's data:

   8 + (4 + 20 (sorted set overhead)) * 500 + 20 (hash-table overhead) = 12KB

   Here we are reserving 20 bytes overhead per element. In a sorted set, we can assume that we need at least two pointers to maintain order among elements — one pointer to the previous element and one to the next element. On a 64bit machine, each pointer will cost 8 bytes. So we will need 16 bytes for pointers. We added an extra word (4 bytes) for storing other overhead. If we need to track one million users at any time, total memory we would need would be 12GB:

   12KB * 1 million ~= 12GB

3. Sliding Windows with counter: Let's assume 'UserID' takes 8 bytes. Each epoch time will need 4 bytes, and the Counter would need 2 bytes. Let's suppose we need a rate limiting of 500 requests per hour. Assume 20 bytes overhead for hash-table and 20 bytes for Redis hash. Since we'll keep a count for each minute, at max, we would need 60 entries for each user. We would need a total of 1.6KB to store one user's data:

   8 + (4 + 2 + 20 (Redis hash overhead)) * 60 + 20 (hash-table overhead) = 1.6KB

   If we need to track one million users at any time, total memory we would need would be 1.6GB:

   1.6KB * 1 million ~= 1.6GB

   So, our 'Sliding Window with Counters' algorithm uses 86% less memory than the simple sliding window algorithm.

Step 6: Propose the distributed architecture

Data Sharding

We can shard based on the 'UserID' to distribute the user's data. For fault tolerance and replication we should use Consistent Hashing. If we want to have different throttling limits for different APIs, we can choose to shard per user per API. Take the example of URL Shortener; we can have different rate limiter for createURL() and deleteURL() APIs for each user or IP.

If our APIs are partitioned, a practical consideration could be to have a separate (somewhat smaller) rate limiter for each API shard as well. Let's take the example of our URL Shortener where we want to limit each user not to create more than 100 short URLs per hour. Assuming we are using Hash-Based Partitioning for our createURL() API, we can rate limit each partition to allow a user to create not more than three short URLs per minute in addition to 100 short URLs per hour.

Data Caching

Our system can get huge benefits from caching recent active users. Application servers can quickly check if the cache has the desired record before hitting backend servers. Our rate limiter can significantly benefit from the Write-back cache by updating all counters and timestamps in cache only. The write to the permanent storage can be done at fixed intervals. This way we can ensure minimum latency added to the user's requests by the rate limiter. The reads can always hit the cache first; which will be extremely useful once the user has hit their maximum limit and the rate limiter will only be reading data without any updates.

Least Recently Used (LRU) can be a reasonable cache eviction policy for our system.

Token Bucket Rate Limiter

```
#include <iostream>
#include <bits/stdc++.h>
#include <chrono>
#include <ctime>
#include <thread>

class RateLimiter {
public:
    RateLimiter(uint64_t capacity,
                uint64_t tokens_per_period,
                uint64_t interval_ms):
                    capacity_(capacity),
                    tokens_per_period_(tokens_per_period),
                    interval_ms_(interval_ms),
                    stop_(false),
                    token_adder_(&RateLimiter::token_refill_thread, this)
    {
        last_refill_period_ = get_current_period();
        next_refill_time_ms_ = (last_refill_period_ + 1) * interval_ms_;
        available_ = capacity;
    }

    virtual ~RateLimiter() {
        std::unique_lock<std::mutex> lock(mutex_);
        stop_ = true;
        lock.unlock();
        token_adder_.join();
    }
```

```
    virtual void RateLimit() {
        std::unique_lock<std::mutex> lock(mutex_);
        while (available_ == 0) {
            req_can_proceed_.wait(lock);
        }
        assert(available_ > 0);
        available_--;

        return;
    }

private:
    void token_refill_thread() {
        std::unique_lock<std::mutex> lock(mutex_);
        while (!stop_) {
            lock.unlock();
            sleep_until_next_refill_time();
            lock.lock();
            refill_locked();
            req_can_proceed_.notify_all();
        }
    }

    void sleep_until_next_refill_time() {
        auto now = std::chrono::steady_clock::now().time_since_epoch();
        auto now_in_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(now).count();

        if  (next_refill_time_ms_ <= now_in_ms)
            return;

        std::chrono::milliseconds next_refill_ms(next_refill_time_ms_);
        std::chrono::time_point<std::chrono::steady_clock>
next_refill_time_point(next_refill_ms);
        std::this_thread::sleep_until(next_refill_time_point);
    }

    uint64_t get_current_period() const {
        auto now = std::chrono::steady_clock::now().time_since_epoch();
        auto now_in_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(now).count();
        return (now_in_ms/interval_ms_);
```

```
    }

    void refill_locked() {
        uint64_t cur_period = get_current_period();
        assert(available_ <= capacity_);
        if ((cur_period == last_refill_period_) || (available_ ==
capacity_)) {
            return;
        }

        int64_t num_tokens_missed = (cur_period - last_refill_period_) *
tokens_per_period_;
        assert(num_tokens_missed >= 0);
        available_ = std::min(available_ + num_tokens_missed, capacity_);
        last_refill_period_ = cur_period;
        next_refill_time_ms_ = (last_refill_period_ + 1) * interval_ms_;
        return;
    }

    uint64_t capacity_;
    const uint64_t tokens_per_period_;
    const long interval_ms_;
    int64_t last_refill_period_;
    int64_t  next_refill_time_ms_;
    uint64_t available_;
    std::mutex mutex_;
    std::condition_variable req_can_proceed_;
    std::thread token_adder_;
    bool stop_;
};

int main() {
    RateLimiter limit(10000, 10000, 1000);
    std::chrono::time_point<std::chrono::system_clock> time_point;

    for (int i=0; i < 500000; ++i) {
        limit.RateLimit();
        time_point = std::chrono::system_clock::now();
        std::time_t ttp = std::chrono::system_clock::to_time_t(time_point);

        std::cout << i << std::endl;

        // if (!(i % 443))
```

```
        //    std::this_thread::sleep_for(std::chrono::milliseconds(2000)
);
    }
}
```