A **Rate Limiter** is a crucial component in distributed systems, especially in APIs and backend services, to **control the rate at which users or systems can make requests**. It protects your service from abuse, prevents overloading, and ensures **fair usage**.

**🚦 Why Use a Rate Limiter?**

* Prevent **Denial-of-Service (DoS)** attacks.
* Protect backend services from overload.
* Enforce **fair usage** policies (e.g., 1000 requests/day).
* Save **compute and storage costs**.
* Improve overall **stability and performance**.

**⚙️ Key Design Considerations**

* **Who to rate limit?** (user ID, IP address, API key)
* **What to limit?** (requests per second, per minute, per day)
* **What to do when the limit is hit?** (throttle, reject, queue)
* **Scope of limit?** (per endpoint, per user, global)
* **Where to enforce?** (client-side, edge, API gateway, backend)

**🧠 Common Algorithms**

| **Algorithm** | **Description** | **Memory Usage** | **Accuracy** | **Common Use Case** |
| --- | --- | --- | --- | --- |
| **Token Bucket** | Tokens added at fixed rate. Each request consumes 1 token. | Moderate | High | Allows burst, real-time APIs |
| **Leaky Bucket** | Requests leak out at a constant rate. Buffer holds bursts. | Moderate | Medium | Smooth traffic flow |
| **Fixed Window** | Count requests in fixed intervals (e.g., per minute). | Low | Medium | Simple scenarios |
| **Sliding Window** | Similar to Fixed, but with time-weighted intervals (less spiky). | Medium | High | More fairness than fixed |
| **Sliding Log** | Store timestamps of each request and slide the window manually. | High | Very High | Precision-critical apps |

**🧱 Architecture: Rate Limiter in a Scalable System**

text

CopyEdit

┌────────────┐

│ Client │

└─────┬──────┘

│

▼

┌────────────┐

│ API Gateway│ ← Rate Limiter here (e.g., Nginx, Envoy)

└─────┬──────┘

▼

┌─────────────────┐

│ Rate Limiter DB │ ← Redis / Memcached / Local in-memory

└─────────────────┘

▼

┌────────────┐

│ Backend │

└────────────┘

**🛠️ Implementation Example: Token Bucket with Redis**

python

CopyEdit

import time

import redis

r = redis.Redis()

def is\_request\_allowed(user\_id, max\_tokens=10, refill\_rate=1):

key = f"rate\_limit:{user\_id}"

now = time.time()

# Get previous data

last\_refill, tokens = r.hmget(key, "last\_refill", "tokens")

last\_refill = float(last\_refill or now)

tokens = float(tokens or max\_tokens)

# Refill tokens

elapsed = now - last\_refill

new\_tokens = min(max\_tokens, tokens + elapsed \* refill\_rate)

if new\_tokens < 1:

return False # Not enough tokens

# Deduct one token

r.hmset(key, {"tokens": new\_tokens - 1, "last\_refill": now})

r.expire(key, 60)

return True

**🧰 Tools & Libraries**

| **Tool** | **Use Case** |
| --- | --- |
| **Redis** | Fast, atomic counter storage |
| **Nginx** | Built-in rate limiting directives |
| **Envoy** | Modern service mesh / API Gateway |
| **Kong / AWS API Gateway** | Rate limiting plugins / config |
| **Istio** | Policy enforcement in service mesh |
| **Rate-limiter-flexible (Node.js)** | Flexible rate limiter library |
| **Bucket4j (Java)** | Token bucket implementation |

**📦 Real-World Examples**

**1. API Gateway Rate Limiting**

nginx

CopyEdit

limit\_req\_zone $binary\_remote\_addr zone=mylimit:10m rate=5r/s;

server {

location /api/ {

limit\_req zone=mylimit burst=10 nodelay;

proxy\_pass http://backend\_service;

}

}

**2. Global Rate Limiter with Redis**

* Store per-user token bucket in Redis.
* TTL handles inactivity clean-up.
* Use Lua scripts for atomicity and consistency.

**⚖️ Trade-offs**

| **Strategy** | **Pros** | **Cons** |
| --- | --- | --- |
| In-Memory | Fast, simple | Not scalable |
| Redis-based | Scalable, centralized | Adds latency & Redis overhead |
| API Gateway | Transparent, centralized | Limited flexibility |
| DB-based | Long-term tracking | High latency & DB overhead |

**📝 Interview Focus Areas**

✅ Concepts:

* Token vs. Leaky Bucket
* Fixed vs. Sliding Window
* Local vs. Distributed rate limiters
* How to handle bursts

✅ Coding:

* Simple in-memory limiter
* Redis-backed token bucket

✅ Scalability:

* Global vs. per-user limits
* Latency trade-offs
* Using caching/storage for limits

✅ Edge Cases:

* Race conditions
* Time sync issues
* Hot users / bots

Would you like a visual diagram or PDF version of this too?

4o

**You said:**

Is coding in C++ allowed for such questions?

**ChatGPT said:**

Yes, **C++ is absolutely allowed** for system design interviews — **especially for low-level design and implementation-oriented questions**, such as:

* Designing a **Rate Limiter**
* Building a **Thread Pool**
* Implementing **LRU Cache**
* Writing **custom memory managers**
* Building your own **URL shortener**, **queue**, or **storage layer**

However, here's how it typically plays out:

**🔍 System Design Interviews – High-Level Design**

In **high-level system design** interviews (e.g., design a URL shortener, messaging system, Uber backend):

* **Language doesn’t matter much.**
* Most of your answers will be diagrams, component interaction, scaling strategies, trade-offs, etc.
* Code is **not usually required**, though pseudocode may help.

**Verdict:** ✅ C++ is allowed, but code is rarely needed here.

**🧠 Low-Level / Object-Oriented Design (LLD)**

In **LLD interviews** (common at Amazon, Google, etc.), you may be asked to implement class structures or design patterns. Examples:

* Design a Rate Limiter
* Design Elevator System
* Design a Parking Lot
* Implement an in-memory DB/cache

Here, **code is expected**, and **C++ is a great choice** due to:

* Class structure
* STL containers
* Templates
* Memory control

**Verdict:** ✅ C++ is **encouraged** for LLD questions.

**💡 Pro Tip:**

If you’re using C++:

* Use unordered\_map, deque, priority\_queue, etc. to show real-world data structure usage.
* Avoid overcomplicating with manual memory unless asked.
* Show design patterns (e.g., Singleton for rate limiter, Factory for URL shortener storage backends).

Would you like a **Rate Limiter implementation in C++** as an example? I can show one using the **Token Bucket** or **Fixed Window Counter** approach.

4o

**You said:**

Yes please

**ChatGPT said:**

Great! Here's a clean and interview-friendly **C++ implementation of a Token Bucket-based Rate Limiter**, using chrono for time and unordered\_map for user-specific state.

**🧪 Token Bucket Rate Limiter in C++**

cpp

CopyEdit

#include <iostream>

#include <unordered\_map>

#include <chrono>

using namespace std;

using namespace std::chrono;

class TokenBucketRateLimiter {

private:

struct Bucket {

double tokens;

steady\_clock::time\_point lastRefill;

};

int maxTokens;

double refillRatePerSecond; // tokens per second

unordered\_map<string, Bucket> userBuckets;

public:

TokenBucketRateLimiter(int maxTokens, double refillRatePerSecond)

: maxTokens(maxTokens), refillRatePerSecond(refillRatePerSecond) {}

bool allowRequest(const string& userId) {

auto now = steady\_clock::now();

if (userBuckets.find(userId) == userBuckets.end()) {

userBuckets[userId] = {maxTokens - 1, now};

return true;

}

Bucket& bucket = userBuckets[userId];

double secondsPassed = duration\_cast<milliseconds>(now - bucket.lastRefill).count() / 1000.0;

// Refill tokens

bucket.tokens = min((double)maxTokens, bucket.tokens + secondsPassed \* refillRatePerSecond);

bucket.lastRefill = now;

if (bucket.tokens >= 1.0) {

bucket.tokens -= 1.0;

return true;

}

return false;

}

};

**🧪 Usage Example**

cpp

CopyEdit

int main() {

TokenBucketRateLimiter limiter(5, 1.0); // Max 5 tokens, refill 1 token/sec

string user = "user123";

for (int i = 0; i < 10; ++i) {

if (limiter.allowRequest(user)) {

cout << "Request " << i + 1 << " allowed\n";

} else {

cout << "Request " << i + 1 << " denied (rate limited)\n";

}

this\_thread::sleep\_for(chrono::milliseconds(300)); // simulate time between requests

}

return 0;

}

**🧠 Key Points to Explain in Interviews**

* Why steady\_clock (for monotonic time)?
* How tokens refill over time.
* Why unordered\_map per user (per-user rate limiting).
* How to make it distributed (hint: Redis + Lua for atomicity).

Would you like a **Fixed Window** version as well, or want to see how to **scale this across multiple machines using Redis**?

4o

**You said:**

Yes please

**ChatGPT said:**

Awesome! Let’s cover both:

**1️⃣ Fixed Window Counter (C++ Implementation)**

This is simpler than token bucket. It counts requests per fixed window (e.g., per second). It’s easy to implement but suffers from "burstiness at boundary".

**🧪 C++ Code: Fixed Window Rate Limiter**

cpp

CopyEdit

#include <iostream>

#include <unordered\_map>

#include <chrono>

using namespace std;

using namespace std::chrono;

class FixedWindowRateLimiter {

private:

int maxRequests;

int windowSizeSeconds;

struct WindowData {

int count;

time\_t windowStart;

};

unordered\_map<string, WindowData> userWindows;

public:

FixedWindowRateLimiter(int maxRequests, int windowSizeSeconds)

: maxRequests(maxRequests), windowSizeSeconds(windowSizeSeconds) {}

bool allowRequest(const string& userId) {

time\_t currentWindow = time(nullptr) / windowSizeSeconds;

auto& data = userWindows[userId];

if (data.windowStart != currentWindow) {

// New window

data.windowStart = currentWindow;

data.count = 1;

return true;

}

if (data.count < maxRequests) {

data.count++;

return true;

}

return false; // Rate limited

}

};

**✅ Usage:**

cpp

CopyEdit

int main() {

FixedWindowRateLimiter limiter(3, 5); // 3 requests per 5 seconds

string user = "user456";

for (int i = 0; i < 5; ++i) {

if (limiter.allowRequest(user)) {

cout << "Request " << i + 1 << " allowed\n";

} else {

cout << "Request " << i + 1 << " denied\n";

}

this\_thread::sleep\_for(chrono::milliseconds(1000));

}

return 0;

}

**2️⃣ How to Scale a Rate Limiter Using Redis (Distributed System)**

In interviews, **scalability** is key — especially for APIs behind load balancers.

**🧱 Architecture:**

arduino

CopyEdit

Client → Load Balancer → API Instances → Redis (shared state)

Each instance reads/writes rate limit data to **Redis**, ensuring consistency across servers.

**💡 Strategy:**

Use Redis' atomic INCR and EXPIRE to track counts per key.

**🧪 Example Redis Key:**

css

CopyEdit

rate\_limit:{user\_id}:{timestamp}

Set TTL = window size.

**🔥 Sample Lua script (for atomic counter in Redis):**

lua

CopyEdit

local key = KEYS[1]

local limit = tonumber(ARGV[1])

local current = redis.call("INCR", key)

if current == 1 then

redis.call("EXPIRE", key, 60)

end

if current > limit then

return 0

else

return 1

end

You can execute this Lua script using a Redis C++ client (e.g., hiredis or redis-plus-plus).

**✅ In Interview, Be Ready to Discuss:**

| **Topic** | **Explanation** |
| --- | --- |
| **Race conditions** | Use Lua script or transactions |
| **Horizontal scaling** | Centralize limits in Redis |
| **Per-user vs. global limits** | Separate keys |
| **Fail-safes** | Circuit breakers or fallback plans if Redis is down |
| **Burst handling** | Use token bucket instead of fixed window |

Would you like help integrating Redis into a C++ client for this? Or a diagram to visualize the distributed rate limiter?

4o

**You said:**

Yes please

**ChatGPT said:**

Awesome! Here's how you can integrate Redis into a **C++ distributed rate limiter**, using the **redis-plus-plus** C++ client (a modern C++ wrapper over hiredis).

**🔌 Setup: Redis in C++ using redis-plus-plus**

**1. Install dependencies:**

You’ll need:

* redis-plus-plus
* hiredis

For Ubuntu (example):

bash

CopyEdit

sudo apt install libhiredis-dev

git clone https://github.com/sewenew/redis-plus-plus.git

cd redis-plus-plus

mkdir build && cd build

cmake ..

make

sudo make install

Make sure to link with -lredis++ -lhiredis -lpthread.

**🧪 C++ Code: Distributed Rate Limiter using Redis**

cpp

CopyEdit

#include <sw/redis++/redis++.h>

#include <iostream>

#include <chrono>

using namespace sw::redis;

using namespace std;

class DistributedRateLimiter {

private:

Redis redis;

int maxRequests;

int windowSeconds;

public:

DistributedRateLimiter(const string& redisUri, int maxRequests, int windowSeconds)

: redis(Redis(redisUri)), maxRequests(maxRequests), windowSeconds(windowSeconds) {}

bool allowRequest(const string& userId) {

auto now = chrono::system\_clock::to\_time\_t(chrono::system\_clock::now());

string windowKey = "rate\_limit:" + userId + ":" + to\_string(now / windowSeconds);

long long count = redis.incr(windowKey);

if (count == 1) {

redis.expire(windowKey, windowSeconds); // Set TTL

}

return count <= maxRequests;

}

};

**✅ Usage Example**

cpp

CopyEdit

int main() {

DistributedRateLimiter limiter("tcp://127.0.0.1:6379", 5, 60); // 5 reqs/min

string user = "user123";

for (int i = 0; i < 10; ++i) {

bool allowed = limiter.allowRequest(user);

cout << "Request " << (i + 1) << (allowed ? " ✅ allowed" : " ❌ denied") << endl;

this\_thread::sleep\_for(chrono::milliseconds(500));

}

return 0;

}

**🧱 Architecture Diagram: Distributed Rate Limiter**

sql

CopyEdit

+---------+ +------------------+ +--------+

| Client | -----> | Load Balancer | -----> | API |

+---------+ +------------------+ | Server |

+--+---+

|

Redis (central)

[rate\_limit:user:window]

**✅ What to Say in an Interview**

* **“I’m using Redis' atomic INCR for counting requests and setting TTL with EXPIRE to reset the window."**
* **“This design is horizontally scalable because all servers share Redis as source of truth."**
* **“To reduce load on Redis, we can add local warm cache for low-frequency users."**