Rate Limiter (LLD Project - I)

Problem Statement

Design and implement a Rate Limiter system that restricts the number of requests a user can make within a given time window. The system should support different rate limiting algorithms, per-user limits, and be thread-safe for concurrent request handling.

Requirements

- **Per-User Limits:** Rate limiting applied individually for each user (or API key).

- **Configurable Limits:** Support configurable request limits (e.g., 10 requests per minute).

- **Time Window:** Enforce limits based on a fixed or sliding time window.

- **Multiple Algorithms:** Support for token bucket, fixed window, or sliding window algorithms.

- **Thread-Safe:** Correctly handle concurrent requests in a multithreaded environment.

Core Entities

- **RateLimiter:** Main class responsible for managing rate limiting logic and user buckets.
- **UserBucket:** Represents the state of a single user’s rate limit (tokens, timestamps, counters).
- **RateLimitAlgorithm:** Interface or abstract class defining contract for rate limiting algorithms (e.g., token bucket, fixed window).
- **TokenBucket:** Implementation of RateLimitAlgorithm applying token bucket logic.

Class Design

1. RateLimiter

Methods:

- `bool isRequestAllowed(userId)` – Check and update user’s rate limit status.
- `void setRateLimit(int maxRequests, int refillRate)` – Configure limits.

Fields:

- Map of userId → UserBucket
- Configurable rate limit parameters

---

2. UserBucket

Methods:

- `bool allowRequest()` – Check token availability and update state atomically.
- `void refillTokens()` – Refill tokens based on elapsed time.

Fields:

- Capacity (max tokens)
- Current tokens
- Last refill timestamp
- Mutex/lock for thread safety

---

3. RateLimitAlgorithm (Interface)

Methods:

- `bool allowRequest()`
- `void refill()`

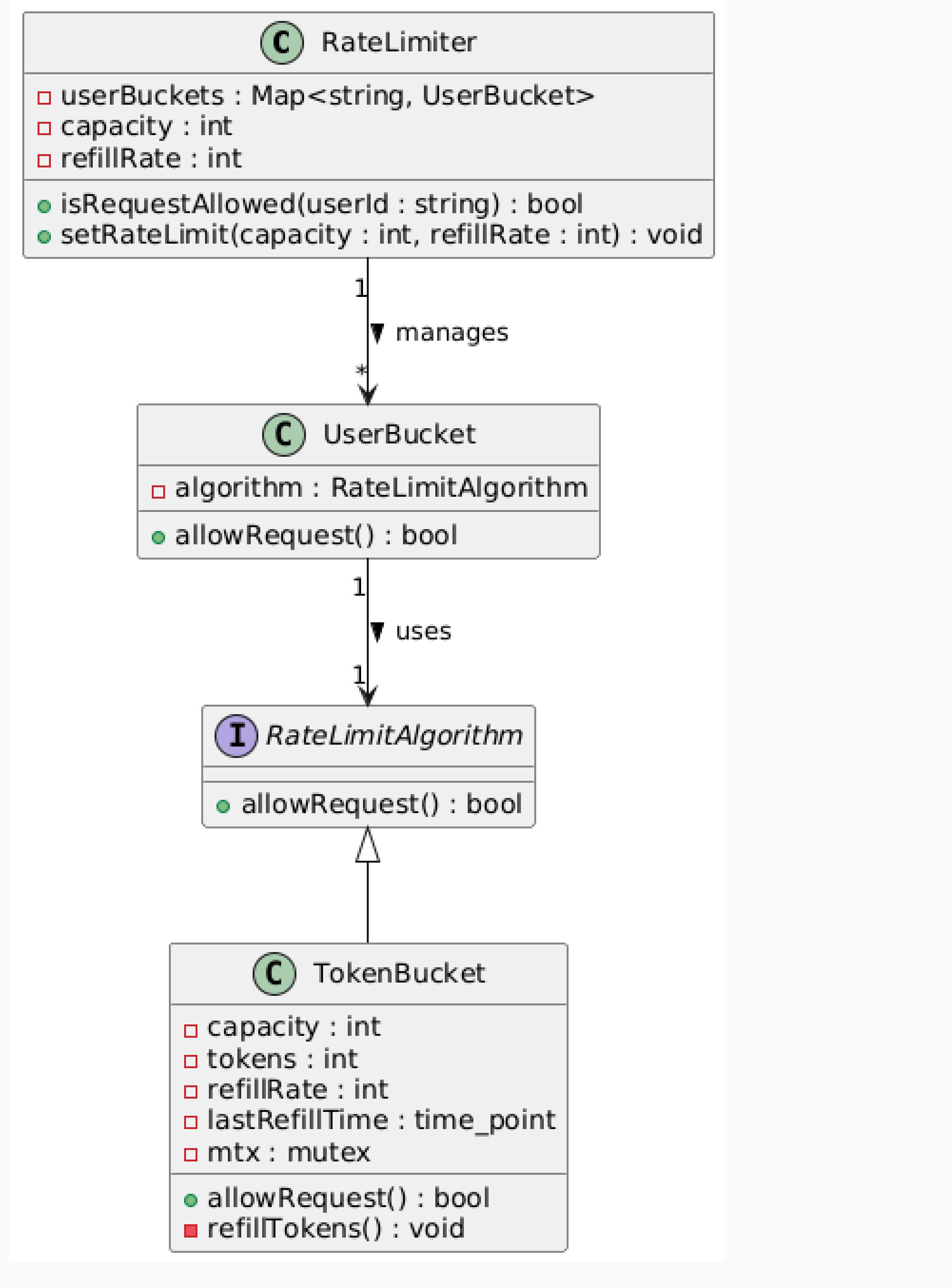
---

4. TokenBucket (implements RateLimitAlgorithm)

Fields:

- Capacity
- Tokens
- Refill rate
- Last refill time.

Methods: Implements `allowRequest` and `refill` logic according to token bucket algorithm.



Example Usage

RateLimiter limiter(10, 1); // 10 requests max, refill 1 token per second

if (limiter.isRequestAllowed("user123")) {

```
    // Process request
} else {
    // Reject request (HTTP 429 Too Many Requests)
}
```

Coding:

```
#include <iostream>
#include <unordered_map>
#include <chrono>
#include <mutex>
#include <memory>
#include <string>
#include<thread>

using namespace std;
using namespace std::chrono;

// Interface for rate limiting algorithms (optional for extension)
class RateLimitAlgorithm {
public:
    virtual bool allowRequest() = 0;
    virtual ~RateLimitAlgorithm() = default;
};

// TokenBucket implements RateLimitAlgorithm
class TokenBucket : public RateLimitAlgorithm {
private:
    int capacity;
    int tokens;
    int refillRatePerSec;
    time_point<steady_clock> lastRefillTime;
    mutex mtx;

public:
    TokenBucket(int capacity, int refillRatePerSec)
```

```
        : capacity(capacity), tokens(capacity), refillRatePerSec(refillRatePerSec),
          lastRefillTime(steady_clock::now()) {}

    bool allowRequest() override {
        lock_guard<mutex> lock(mtx);
        refillTokens();

        if (tokens > 0) {
            tokens--;
            return true;
        }
        return false;
    }

private:
    void refillTokens() {
        auto now = steady_clock::now();
        auto secondsPassed = duration_cast<seconds>(now - lastRefillTime).count();

        if (secondsPassed > 0) {
            int tokensToAdd = secondsPassed * refillRatePerSec;
            tokens = min(capacity, tokens + tokensToAdd);
            lastRefillTime = now;
        }
    }
};

// Represents per-user bucket wrapping RateLimitAlgorithm
class UserBucket {
private:
    unique_ptr<RateLimitAlgorithm> algorithm;

public:
    UserBucket(int capacity, int refillRatePerSec) {
        algorithm = make_unique<TokenBucket>(capacity, refillRatePerSec);
    }

    bool allowRequest() {
        return algorithm->allowRequest();
    }
};

// RateLimiter manages all user buckets
class RateLimiter {
private:
    unordered_map<string, shared_ptr<UserBucket>> userBuckets;
    int capacity;
    int refillRatePerSec;
    mutex globalMutex;
```

```
public:
    RateLimiter(int capacity, int refillRatePerSec)
        : capacity(capacity), refillRatePerSec(refillRatePerSec) {}

    bool isRequestAllowed(const string& userId) {
        lock_guard<mutex> lock(globalMutex);

        if (userBuckets.find(userId) == userBuckets.end()) {
            userBuckets[userId] = make_shared<UserBucket>(capacity, refillRatePerSec);
        }

        return userBuckets[userId]->allowRequest();
    }
};

// Simple test
int main() {
    RateLimiter limiter(5, 1);  // max 5 requests, refill 1 token/sec

    string user = "user123";

    cout << "Sending 7 rapid requests:\n";
    for (int i = 1; i <= 7; ++i) {
        bool allowed = limiter.isRequestAllowed(user);
        cout << "Request " << i << ": " << (allowed ? "Allowed" : "Blocked") << "\n";
        this_thread::sleep_for(chrono::milliseconds(300));
    }

    cout << "\nWaiting 3 seconds to refill tokens...\n";
    this_thread::sleep_for(chrono::seconds(3));

    cout << "Sending 3 more requests:\n";
    for (int i = 1; i <= 3; ++i) {
        bool allowed = limiter.isRequestAllowed(user);
        cout << "Request " << i << ": " << (allowed ? "Allowed" : "Blocked") << "\n";
    }

    return 0;
}
```