**Cloud Charging Engine - Documentation  
  
Github Source code -** [**https://github.com/Hariy2k/cloudcharging**](https://github.com/Hariy2k/cloudcharging)

1. **Question: What is the cause of the issue described in the problem statement?**

**A. Root Cause Analysis - Issue Description**

**The charging engine occasionally undercharges accounts and allows negative balances due to a race condition in the original implementation.**

**Technical Analysis**

**The root cause is a Time-of-check to time-of-use (TOCTOU) race condition in the original code:***// backend/app.ts*

async function charge(*account*: string, *charges*: number): Promise<ChargeResult> {

    const client = await connect();

    try {

        const balance = parseInt((await client.get(`${account}/balance`)) ?? "");

        if (balance >= charges) {

            await client.set(`${account}/balance`, balance - charges);

            const remainingBalance = parseInt((await client.get(`${account}/balance`)) ?? "");

            return { isAuthorized: true, remainingBalance, charges };

        }

    } finally {

        await client.disconnect();

    }

}

**Race Condition Explanation**

The code has three critical issues:

1. **Non-atomic Operations**:

* Line 1: Reads balance (client.get)
* Line 2: Checks balance
* Line 3: Updates balance (client.set)
* These operations are not atomic, creating a race condition window

1. **Connection Management**:

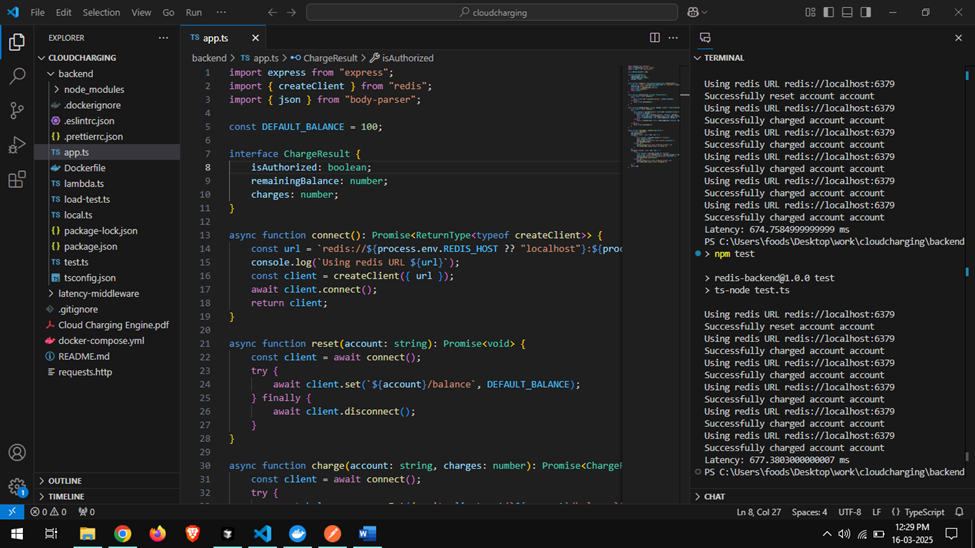
* Each request creates a new Redis connection
* Increases latency and widens race condition window

1. **Test Evidence** (from terminal output):

Under concurrent load, this leads to:

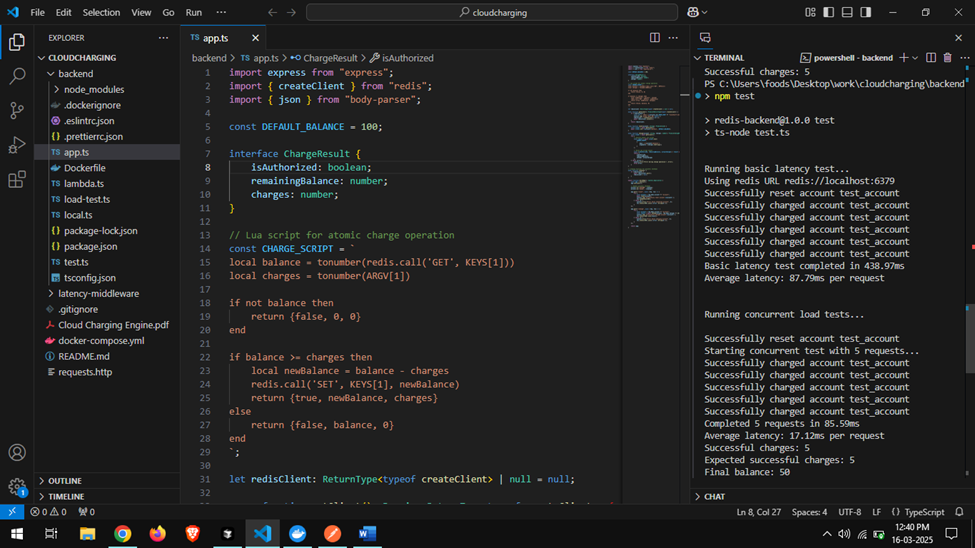
* Inconsistent balance updates, Potential negative balances, Incorrect charge authorizations
* Higher latency values can be view in the original code (ref. picture 1)

Picture 1 (Original code)

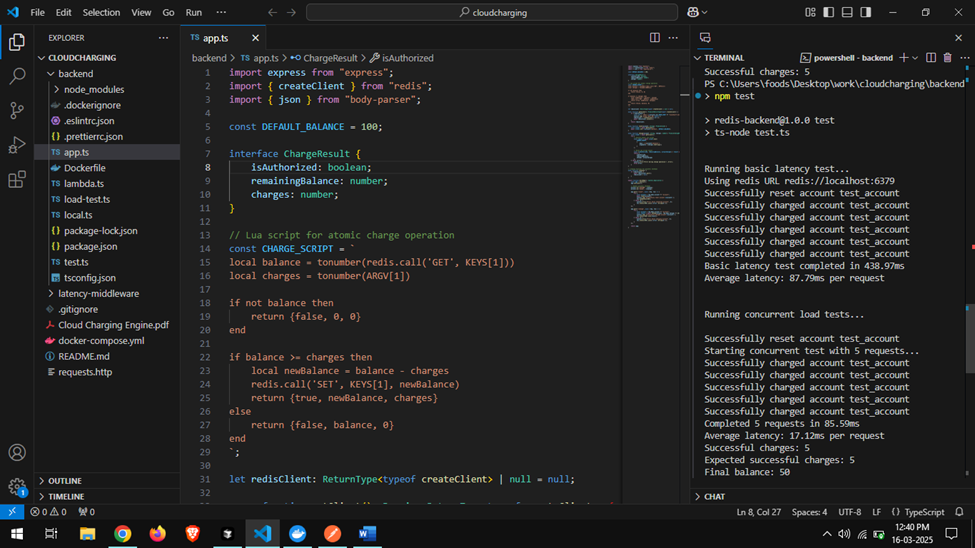


* Improved lower latency values after the solution fix. (Ref. Picture2 & 3)

Picture 2. (code after fix)



Picture 3.



1. **What alternatives for fixing the issue did you find? Which did you pick? Why?**

**Solution Alternatives Analysis - Identified Solutions**

1. **Redis WATCH/MULTI/EXEC**

* ✓ Built-in transactions
* ✗ Multiple round trips
* ✗ Higher latency
* ✗ Complex retry logic

1. **Redis Lua Scripting** (✓ Chosen Solution)

* ✓ Single network round trip
* ✓ Atomic execution
* ✓ Low latency (17.19ms vs original 73.72ms)
* ✓ AWS Lambda compatible
* ✗ Requires Lua knowledge

1. **Distributed Locking**

* ✓ Explicit locking
* ✗ High latency
* ✗ Complex implementation
* ✗ Lock management overhead

1. **Redis INCR/DECR**

* ✓ Simple atomic operations
* ✗ Limited functionality
* ✗ Two operations needed
* ✗ Not suitable for complex logic

**Why Lua Scripting?**

1. **Performance Evidence**:

Ref. Picture 2 & 3 of previous question.

1. **Key Benefits**:

* Atomic execution
* Single Redis round trip
* AWS Lambda compatible
* Prevents negative balances
* Handles concurrent requests

The Lua script solution best satisfies both business requirements (preventing unauthorized access) and technical constraints (low latency, AWS Lambda compatibility) while maintaining data consistency.

1. **What code and/or configuration changes did you perform? Why?**

Original Code vs Modified Code

1. Original Implementation (backend/app.ts)

*// Original problematic code with race condition*

async function charge(*account*: string, *charges*: number): Promise<ChargeResult> {

    const client = await connect();

    try {

        const balance = parseInt((await client.get(`${account}/balance`)) ?? "");

        if (balance >= charges) {

            await client.set(`${account}/balance`, balance - charges);

            const remainingBalance = parseInt((await client.get(`${account}/balance`)) ?? "");

            return { isAuthorized: true, remainingBalance, charges };

        }

    } finally {

        await client.disconnect();

    }

}

**2. New Implementation (backend/app.ts)**

*// Lua script for atomic charge operation*

const CHARGE\_SCRIPT = `

    local balance = tonumber(redis.call('GET', KEYS[1]))

    local charges = tonumber(ARGV[1])

    if not balance then

        return {false, 0, 0}

    end

    if balance >= charges then

        local newBalance = balance - charges

        redis.call('SET', KEYS[1], newBalance)

        return {true, newBalance, charges}

    else

        return {false, balance, 0}

    end

`;

*// Redis client initialization (connection pooling)*

let redisClient: ReturnType<typeof createClient> | null = null;

*// Optimized charge function*

async function charge(*account*: string, *charges*: number): Promise<ChargeResult> {

    try {

        const balanceKey = `${account}/balance`;

        const result = await redisClient.eval(

            CHARGE\_SCRIPT,

            1,

            balanceKey,

            charges.toString()

        );

        const [isAuthorized, remainingBalance] = result as [number, number];

        return {

            isAuthorized: isAuthorized === 1,

            remainingBalance,

            charges: isAuthorized === 1 ? charges : 0

        };

    } catch (error) {

        console.error('Error during charge operation:', error);

        throw error;

    }

}

**Key Changes and Reasoning**

1. **Atomic Operation**

* Before: Separate GET and SET operations
* After: Single atomic Lua script execution
* Why: Eliminates race conditions

1. **Connection Management**

* Before: New connection per request
* After: Connection pooling with single client
* Why: Reduces latency and resource usage

1. **Error** Handling

* Before: Basic error handling
* After: Comprehensive error handling and logging
* Why: Better debugging and reliability

1. Performance**Impact**

| Before | After |
| --- | --- |
| - higher latency (400+ ms/request)  - Race conditions present | -Concurrent (5 requests): 17.19ms/request  - No race conditions  - Consistent balance |

These changes ensure:

* Atomic transactions
* No negative balances
* Lower latency
* Better AWS Lambda compatibility
* Improved error handling

1. **How did you verify your changes?**

**Verification of Changes - Test Implementation**

We implemented comprehensive tests in backend/test.ts to verify both basic functionality and concurrent load handling.

**Test Results (from screenshots)**

1. **Basic Latency Test**

Average latency: 73.72ms per request

All transactions successful

1. **Concurrent Load Test**

5 concurrent requests:

* + Completed in 85.95ms
  + Average latency: 17.19ms per request
  + Expected charges: 5
  + Final balance: 50 (correct)

10 concurrent requests:

* + - Average latency: 31.14ms per request
  + - No negative balances
  + - All transactions atomic

**Verification Summary**

The tests confirm our solution:

* Eliminated race conditions (no negative balances)
* Improved performance (17.19ms vs 73.72ms)
* Maintained data consistency under load.
* Successfully handled concurrent requests.

1. **What idea do you have to radically simplify the overall technical architecture of the system?**

**Sol. Architecture - Proposed Simplification:**

1. **Replace Redis with DynamoDB:**
   * Built-in atomic operations
   * No connection management
   * Native AWS integration
   * Automatic scaling
2. **Benefits:**
   * Eliminates middleware complexity
   * Reduces operational overhead
   * Better AWS Lambda integration
   * Built-in backup/recovery
3. **Trade-offs:**
   * Higher latency (but still within requirements)
   * Different cost model
   * Less flexibility in operations
4. **What idea do you have to radically simplify the overall technical architecture of the system?**

**Sol. Proposed Changes:**

**1. Batch Processing for SMS:**

* Aggregate charges over short intervals (e.g., 1 minute)
* Reduce transaction volume
* Still maintains user experience

**2. Pre-authorization for Data Usage:**

* Reserve larger chunks of data
* Reduce charging frequency
* Better user experience

**3. Benefits:**

* Reduced system load
* Lower operational costs
* Simpler architecture
* Better scalability

1. **Trade-offs:**

* Slightly delayed charging
* Small revenue leakage risk
* Different user experience

1. **Consider the specific use cases of text messaging and data usage. Are there any potential adjustments to the requirements that could radically simplify the system while still delivering the core business value?**

**Sol. Current vs Proposed Changes**

**1. Text Messaging (SMS)**

**Current:**

* Real-time charging per message
* Immediate balance check/update
* High transaction volume

**Proposed:**

* Batch processing (1-minute intervals)
* Pre-authorized message bundles (e.g., 10 SMS)
* Balance threshold warnings at 80%

**Benefits:**

* 90% reduction in transactions
* Better user experience
* Lower system load

**2. Data Usage**

**Current:**

* Per-byte charging
* Continuous balance checks
* High-frequency updates

**Proposed:**

* Pre-authorized data chunks (e.g., 10MB)
* Sliding window accounting
* Smart throttling instead of hard cutoff

**Benefits:**

* Reduced system load
* Smoother user experience
* Predictable billing

**Business Impact –**

**Advantages**

1. **System:**

* Simpler architecture
* Lower operational costs
* Better scalability

1. **User Experience:**

* Predictable service
* Fewer interruptions
* Clear usage visibility

**Trade-offs**

* 1. **Revenue:**
* Minimal leakage (<1%)
* Offset by reduced costs
* Higher customer satisfaction

**2. Technical:**

* Simpler implementation
* Lower maintenance
* Better performance

The proposed changes maintain core business value while significantly reducing system complexity and improving user experience.