Tema II
Algoritmi Paraleli C#
SISTEM DE RATE LIMITING DISTRIBUIT
Deadline HARD: 15.01.2026

Obiectiv

Veți implementa un **sistem de rate limiting distribuit** în C# care simulează un API Gateway cu multiple noduri.

Rate limiting-ul este o tehnică esențială în sistemele distribuite pentru a preveni abuzul resurselor. Acesta permite controlul numărului de cereri permise într-un interval de timp limitat.

Soluția dezvoltată trebuie să fie thread-safe și să simuleze un mediu distribuit folosind structuri de date concurente, pentru a stoca starea partajată între "noduri" (simulate prin threads sau task-uri).

Cerințe

- O interfață comună ***IRateLimiter*** pentru toți algoritmii implementați, care include o metodă *bool* ***TryAcquire(string NodeId)***, care returnează ***true*** dacă cererea este permisă, sau ***false*** dacă cererea nu poate fi satisfăcută.
- Implementări pentru cel puțin 2 algorimti din cei trei:
  - Token Bucket,
  - Fixed Window,
  - Sliding Window.
- O suită de teste (***RateLimiterTestHarness***) care demonstrează proprietățiile algoritmilor sub încărcare concurentă variată. Validarea experimentală a performanței celor 2 algoritmi implementați se realizează cu datele de control descrise mai jos.

Detaliere algormiti

Token Bucket

Acest algoritm simulează o "*găleată*" (bucket) care se umple constant cu token-uri la o rată fixă (ex: 10 token-uri pe secundă). Fiecare cerere consumă un token. Dacă nu mai sunt token-uri disponibile, cererea este respinsă. Acest algoritm permite "burst-uri" de cereri (când bucket-ul este plin), dar previne abuzul pe termen lung.

Parametrii algoritmului:

```
public TokenBucketRateLimiter(double capacity, double refillRate)
```

Cerințe minume de implementare:

- **Thread-Safety**: Folosirea unor primitive de sinconizare (de ex. ***SemaphoreSlim*** sau ***ReaderWriterLockSlim)*** pentru sincronizarea accesului la starea bucket-ului;
- **Token Refill**: Refill-ul este calculat dinamic la fiecare cerere, bazat pe timpul trecut de la ultimul refill (ATENȚIE: nu folosim un timer continuu, pentru simplitate și eficiență în medii distribuite);
- **Overflow Protection**: Token-urile nu depășesc capacitatea maximă a bucket-ului.
- **Persistent State**: Stocăm starea (token-uri curente, ultimul timestamp) folosind colecții thread-safe (de ex. ***ConcurrentDictionary***, unde cheia ar putea fi ***NodeId***.

Fixed Window algorithm

Acest algoritm clasifică timpul în intervale fixe cunoscute (de ex. ***1s***) sub numele de ferestre fixe și restricționează cererile acest interval. Adică, dacă limita este depășită, cererile sunt respinse și nu pot fi efectuate din nou, cel puțin până la următoarea alocație sau fereastră.

Parametrii algoritmului:

```
public FixedWindowRateLimiter(int maxRequests, int windowSizeSeconds)
```

Cerințe de Implementare:

- **Window Calculation**: Fereastra curentă se calculează ca

```
currentWindow = timestamp / windowSize (unde timestamp este în secunde, windowSize în secunde).
```

- **Atomic Operations**: Folosim primitive de tip ***Interlocked*** pentru a gestiunea contorul atomic, asigurând thread-safety.
- Window **Reset**: Când fereastra se schimbă, contorul se resetează automat.
- **Space Efficiency**: Per nod, stocăm doar contorul curent și start-ul ferestrei (2 valori), folosind colecții thread-safe (de ex.) ***ConcurrentDictionary***.

Sliding Window algorithm

O variantă mai rafinată a Fixed Window, care utilizează o fereastră glisantă (ex: ultimele 10 de secunde) pentru controlul cerințelor. Acest algoritm stochează câte un contor pentru fereastra curentă și pentru cea precedentă, calculând un "***weighted count***" bazat pe timpul deja scurs din fereastra curentă.

Parametrii algoritmului:

```
public SlidingWindowRateLimiter(int maxRequests, int windowSizeSeconds)
```

Cerințe de Implementare:

- **Two Windows**: Contor pentru fereastra precedentă și curentă. Gestionarea atomică a acestora folosind primitive specifice C#;
- **Weight Calculation**:

```
WeightedCount = (contor precedent * (1 - procent trecut din fereastra curentă)) + contor_curent.
```

- **Window Transition**: Când fereastra se schimbă, contorul curent devine precedent, și se resetează cel curent.

Testare

Pentru validarea implementării algoritmilor implementați veți implementa o baterie de **teste asincrone**

Specificații suită de teste

- **Test 1 (Race Condition)**: ***XXXX*** (cu XXXX = 5000) threads încearcă simultan cereri pentru același nod; verifică că exact numărul maxim este aprobat (fără depășiri din race conditions).
- **Test 2 (Accuracy Under Load)**: Generează un număr ***YYYY*** (cu YYYY = 2000) cereri rapide, verifică limita inițială, așteaptă reset-ul ferestrei și verifică din nou.
- **Test 3 (bonus 5p) (Distributed Simulation)**: Simulează multiple noduri (***ZZZZ*** = 20) cu task-uri concurente și 200 de cereri / nod verificând consistența.

Pentru toate testele numărul de cereri (maxRequests) variază de la 100, 500, 1000.

```
1. public class RateLimiterTestHarness
2. {
3.     // Test 1: Race Condition - Multiple threads pentru același node
4.     public async Task TestConcurrentAccessSameNode(
5.         IRateLimiter limiter,
6.         int threadCount = 5000);
```

```
 7.
 8.     // Test 2: Exactitate sub concurență
 9.     public async Task TestAccuracyUnderLoad(
10.         IRateLimiter limiter,
11.         int maxRequests = 2000);
12.
13.     // Test 3: Distributed simulation
14.     public async Task TestDistributedSimulation(
15.         IRateLimiter limiter,
16.         int nodeCount = 20)
17. }
```

Rezultatul obținut trebuie să fie de forma

Acesta este un exemplu de output valid[1].

```
 1. Algorithm: Fixed Window
 2. ----------------------------------------
 3.
 4. MaxRequests = 100
 5. RaceTest: Approved <= 100 | Time=7ms
 6. AccuracyTest: Approved <= 100 | Requests=2000 | Time=3ms
 7. DistributedTest: Nodes=20, Req/Node=200 | Time=9ms
 8.
 9. MaxRequests = 500
10. RaceTest: Approved <= 500 | Time=6ms
11. AccuracyTest: Approved <= 500 | Requests=2000 | Time=5ms
12. DistributedTest: Nodes=20, Req/Node=200 | Time=3ms
13.
14. MaxRequests = 1000
15. RaceTest: Approved <= 1000 | Time=5ms
16. AccuracyTest: Approved <= 1000 | Requests=2000 | Time=1ms
17. DistributedTest: Nodes=20, Req/Node=200 | Time=5ms
```

Explicație

Implementare corectă

- Cererile aprobate nu depășesc niciodată valoarea MaxRequests;
- Comportamentul este consistent pentru toate valorile MaxRequests.

Modelul timpului de execuție

- Testul de concurență (Race test) durează cel mai mult (~5–7 ms) din cauza contendenței asupra lock-ului pentru un singur nod;
- Testul de acuratețe (Accuracy test) este mai scurt (~1–5 ms) pentru că cererile care depășesc limita sunt respinse rapid;
- Testul distribuit (Distributed test) are timp intermediar (~3–9 ms) datorită nodurilor multiple.

Criterii de Evaluare

Criteriu	Puncte	Descriere
Algoritm 1	40p	Implementare corectă + thread-safety
Algoritm 2	40p	Precizie + memory management
Baterie de teste	20p (25p bonus)	Race conditions + accuracy tests + BONUS
TOTAL	100p + 5p(bonus)	

Format predare: Arhivă ***Nume_Prenume_Grupa.zip*** conținând:

- Soluția .NET completă.

1 Testat pe MacBook Pro M1 CPU Cores: 8, RAM: 16.00 GB