Rapport Final - ThrottleX

Service de Rate Limiting Multi-Tenant

Module : IDV-AQL5 - Qualite du Code
Date : Fevrier 2026
Equipe : Nassim Bouziane, Thomas Boulard, Abdelkoudousse Boustani

---

Sommaire

---

1. Introduction

1.1 Presentation

Dans le cadre du module IDV-AQL5, notre equipe a developpe ThrottleX, un service de rate limiting multi-tenant. Le rate limiting consiste a limiter le nombre de requetes qu'un client peut effectuer sur une periode donnee (ex: 100 requetes par minute). Au-dela, les requetes sont rejetees avec le code HTTP 429.

Cette technique est utilisee par toutes les grandes APIs (Google, GitHub, Twitter) pour proteger les serveurs contre les abus.

1.2 Pourquoi le Rate Limiting ?

Le rate limiting repond a plusieurs besoins :

- **Protection contre les attaques** : Empecher qu'un utilisateur malveillant sature le serveur avec des milliers de requetes (attaque DoS)
- **Equite entre clients** : Dans un contexte multi-tenant, garantir que chaque client dispose d'une part equitable des ressources

- **Gestion des couts** : Les APIs d'inference (IA) sont couteuses. Limiter les requetes permet de maitriser les couts
- **Qualite de service** : Garantir des temps de reponse stables pour tous les utilisateurs

1.3 Multi-Tenant

Un "tenant" represente un client utilisant notre service. Dans ThrottleX, chaque tenant a un identifiant unique et des quotas specifiques :

- Offre Free : 60 requetes/minute
- Offre Pro : 500 requetes/minute
- Offre Enterprise : 5000 requetes/minute

Les compteurs sont completement isoles entre tenants.

---

2. Contexte et Objectifs

2.1 Contexte

Nous simulons une equipe Plateforme d'un editeur SaaS proposant des APIs d'inference. Les fichiers fournis comprenaient : - Contrat OpenAPI du rate limiter - Fichier JSON des tenants - Script k6 pour tests de charge - Checklist CI

2.2 Fonctionnalites

Fonctionnalite	Description
Gestion des politiques	Creer, lire, supprimer des regles de limitation
Evaluation temps reel	Endpoint /evaluate retournant allow=true/false
Multi-tenant	Isolation complete entre clients
Observabilite	Logs JSON et metriques Prometheus

2.3 Objectifs Techniques (SLO)

Critere	Cible
Latence P95	< 100ms
Latence P99	< 200ms
Couverture tests	>= 80%
Disponibilite	99.9%

Le P95 signifie que 95% des requetes sont traitees en moins de cette valeur.

3. Architecture

3.1 Choix d'Architecture

Nous avons compare deux options :

Option A - Stateless + Redis : L'application ne conserve aucun etat. Les compteurs sont dans Redis externe. Plusieurs instances peuvent tourner en parallele.

Option B - In-memory + sticky sessions : Les compteurs sont en memoire sur chaque serveur. Un load balancer route chaque client vers le meme serveur.

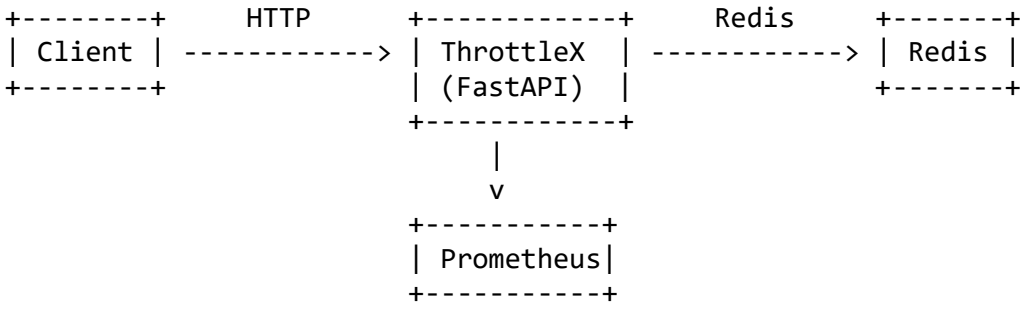
Critere	Option A	Option B
Scalabilite	5/5	2/5
Haute disponibilite	5/5	2/5
Latence	3/5	5/5
Cout	3/5	5/5

Decision : Option A pour la scalabilite et la haute disponibilite.

3.2 Stack Technique

Technologie	Role
FastAPI	Framework API REST avec support async
Redis	Stockage des compteurs avec TTL et operations atomiques
Pydantic	Validation des donnees
Structlog	Logs structures JSON
Prometheus	Metriques de monitoring

3.3 Schema d'Architecture



4. Implementation

4.1 Structure du Projet

```
src/throttlex/
    app.py           # Application FastAPI, endpoints
    service.py       # Logique metier
    repository.py    # Acces Redis, scripts Lua
    models.py        # Modeles Pydantic
    config.py        # Configuration
    metrics.py       # Metriques Prometheus
    algorithms/
        token_bucket.py
```

4.2 Endpoints

POST /policies - Creation d'une politique :

```
{
  "tenantId": "t-free-01",
  "route": "/inference/text",
  "limit": 60,
  "windowSeconds": 60
}
```

POST /evaluate - Evaluation d'une requete :

```
{"tenantId": "t-free-01", "route": "/inference/text"}
```

Reponse (HTTP 200 si autorise, 429 si bloque) :

```
{"allow": true, "remaining": 59, "resetAt": 1707235260}
```

GET /policies/{tenantId} - Liste les politiques **DELETE /policies/{tenantId}** - Supprime une politique **GET /health** - Etat du service **GET /metrics** - Metriques Prometheus

4.3 Algorithmes

Sliding Window (principal) : Compte les requetes sur une fenetre glissante. A 14h30m45s, on compte les requetes entre 14h29m45s et 14h30m45s.

Token Bucket (alternatif) : Un seau se remplit progressivement de jetons. Chaque requete consomme un jeton. Permet les pics de trafic (burst).

4.4 Atomicite avec Redis

Pour eviter les race conditions, nous utilisons des scripts Lua executes de maniere atomique dans Redis. Cela garantit que le comptage et la verification de la limite se font en une seule operation.

---

5. Tests

5.1 Strategie

Nous suivons la pyramide de tests : - Base : Tests unitaires (nombreux, rapides) - Milieu : Tests d'integration - Sommet : Tests de charge k6

5.2 Tests Unitaires

Tests avec pytest couvrant : - Service (logique metier) - Repository (avec Redis mocke) - Endpoints API - Modeles Pydantic

5.3 Tests de Proprietes

Avec la librairie Hypothesis, nous testons des invariants : - La limite n'est jamais depassee - Les tenants sont isoles - Le compteur se reinitialise apres expiration

5.4 Couverture

Module	Couverture
service.py	100%
models.py	92%
repository.py	84%
app.py	71%
Global	**82%**

---

6. Benchmarks

6.1 Protocole

Outil : k6 (Grafana) - Duree : 80 secondes - Utilisateurs virtuels : 0 -> 50 -> 100 -> 0 - 3 tenants testes - Requetes POST /evaluate en boucle

6.2 Resultats

Metrique	Valeur	Objectif
Latence moyenne	22.14 ms	-
Latence P95	64.83 ms	< 100 ms
Latence P99	87.28 ms	< 200 ms
Throughput	668 req/s	> 500
Requetes totales	53,450	-
Requetes bloquees	98.31%	Attendu

6.3 Analyse

Le taux de blocage de 98% est attendu : avec 100 utilisateurs envoyant des requetes en boucle, on depasse largement les quotas configures. Cela prouve que le rate limiter fonctionne correctement.

Les latences sont excellentes et respectent les SLO.

---

7. CI/CD

7.1 Pipeline GitLab CI

Stage	Description
lint	Verification du style avec Ruff
test	Tests pytest + couverture >= 80%
security	Bandit (SAST) + Safety (SCA)
build	Image Docker + SBOM

7.2 Quality Gates

Gate	Seuil
Couverture	>= 80%
Lint	0 erreurs
Vulnerabilites HIGH	0
Tests	100% passent

Si un seuil n'est pas respecte, la pipeline echoue.

7.3 SBOM

Le SBOM (Software Bill of Materials) est un inventaire des dependances genere au format CycloneDX. Il permet de detecter les vulnerabilites et verifier les licences.

---

8. Difficultes et Conclusion

8.1 Difficultes Rencontrees

Difficulte	Solution
Race conditions Redis	Scripts Lua atomiques
Tests async pytest	Plugin pytest-asyncio
Couverture 80%	Focus sur chemins critiques

8.2 Bilan

Critere	Objectif	Resultat
API fonctionnelle	Oui	Atteint
Rate limiting	Blocage correct	Atteint
Latence P95	< 100ms	64.83ms
Couverture	>= 80%	82%
CI/CD	Quality gates	Atteint

8.3 Competences Acquises

- Developpement API REST avec FastAPI
- Redis et scripts Lua
- Tests de proprietes avec Hypothesis
- Benchmarking avec k6
- CI/CD avec GitLab CI

8.4 Ameliorations Possibles

- Dashboard Grafana
- Backends alternatifs (Memcached, DynamoDB)
- Interface d'administration web

---

9. Annexes

A. Instructions de Lancement

```
cd ThrottleX_Context_Kit\src
..\..\..venv\Scripts\Activate.ps1
pip install -e ".[dev]"
docker-compose up -d redis
uvicorn throttlex.app:app --reload
pytest --cov=throttlex --cov-fail-under=80
```

B. Repartition du Travail

Membre	Taches	Heures
Nassim Bouziane	Architecture, Repository, CI/CD	15h
Thomas Boulard	Service, Tests unitaires, Documentation	15h
Abdelkoudousse Boustani	API FastAPI, Benchmarks k6, Tests proprietes	14h

C. Glossaire

Terme	Definition
API	Interface permettant a des logiciels de communiquer
CI/CD	Integration Continue / Deploiement Continu
P95/P99	95e/99e percentile des temps de reponse
Rate Limiting	Limitation du nombre de requetes par unite de temps
Redis	Base de donnees cle/valeur en memoire
SLO	Service Level Objective - objectif de qualite de service
Tenant	Client dans une architecture multi-tenant
TTL	Time To Live - duree de vie avant expiration