Lidando com rate limiting

Contexto

Sua equipe foi encarregada de criar um **serviço de proxy interno** para consumir a API pública de uma empresa parceira disponível em https://score.hsborges.dev/docs.

Esse serviço tem inspiração no padrão de projeto Proxy e visa lidar com as limitações impostas pela API original por meio da gestão do `Client ID` disponível da empresa.

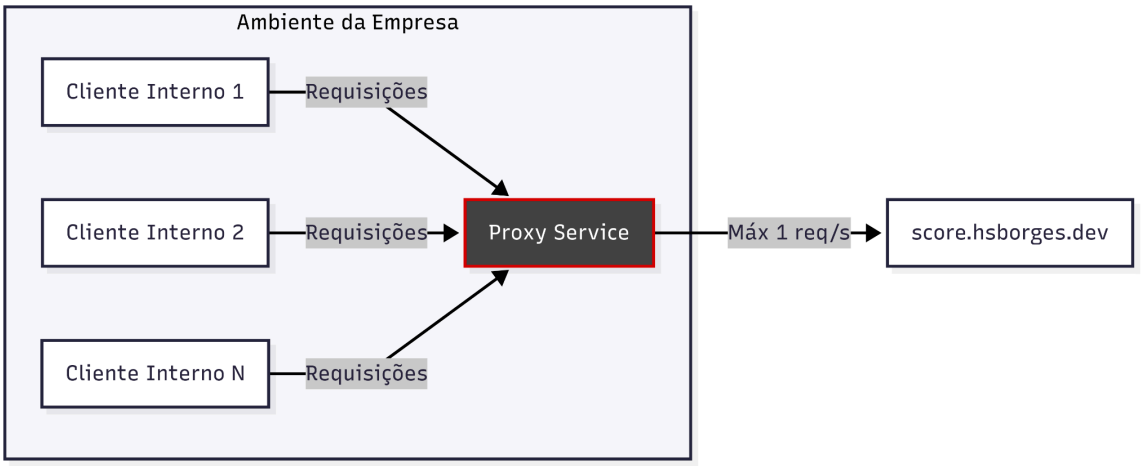
A API impõe **limite de 1 requisição por segundo**. Se o limite for excedido, o provedor aplica uma **penalidade**: a próxima resposta só é liberada **após 2 segundos adicionais** (além da latência normal), impactando todos os serviços internos.

O objetivo é **projetar e implementar** um **proxy resiliente** que:

1. **Acomode picos de requisições internas** sem violar o *rate limit* externo;
2. **Minimize latência média e penalidades**;
3. **Monitore e exponha métricas** para tomada de decisão.

Seguindo os princípios do padrão Proxy, o serviço proxy deverá manter a interface e comportamento original dos serviços disponíveis na API do fornecedor, somente se abstraindo de receber o `Client ID`, que deverá ser fornecido para o servidor proxy funcionar. Você pode configurar esse ID via variável de ambiente ou qualquer outra forma que achar mais adequado.

A seguir está um diagrama de alto nível exemplificando o desafio:



Regras do Jogo (restrições)

- ***Rate limit* externo fixo:** 1 req/s.
- **Penalidade:** ao violar o limite, o provedor impõe **+2s** de atraso.
- **Ambiente interno ruidoso:** múltiplos clientes internos podem disparar múltiplas requisições simultâneas.
- **Sem "comprar" mais cota:** o time **não pode** negociar limites maiores — apenas **projetar** para respeitar o limite existente.
- **Implementação livre:** linguagem, libs e tecnologias à escolha do grupo (documentar).

Missão

Projetar e entregar um **serviço proxy** com:

[Obrigatórios]

1. **Fila/Buffer interno** para lidar com picos (backpressure);
2. ***Scheduler*** para emitir no **máximo 1 chamada/segundo** ao upstream;
3. **Caching** para memorizar resultados recentes evitando repetidas requisições;

[Opcionais]

4. **Política de enfileiramento** configurável (ex.: FIFO, prioridade por tipo de operação, deadline/TTL);
5. **Estratégia de degradação** (fallback) quando a fila cresce demais (ex.: shed load, respostas cacheadas, etc);
6. **Observabilidade:** logs estruturados, métricas (contadores, histograma de latência, taxa de erro, tamanho da fila), e dashboard simples;
7. **Configurações** (via arquivo/env): limites, tamanhos de fila, timeouts, política de retry;

Requisitos Funcionais

- **RF1**: Expor `GET /proxy/score` que aceita parâmetros da chamada para o upstream.
- **RF2**: Expor `GET /metrics` com métricas de uso
- **RF3**: Expor `GET /health` (liveness/readiness).

Requisitos Não Funcionais

- **RNF1**: Tolerar **rajadas** internas (e.g., 20 req em 1s) sem violar 1 req/s externo.
- **RNF2**: **Evitar** penalidades recorrentes. Caso ocorram, **registrar** e **ajustar** a cadência automaticamente (controle adaptativo).
- **RNF3**: **Throughput** sustentado: próximo a 1 req/s para o upstream (estável).
- **RNF4**: **Tempo de espera** previsível (fila controlada com limites e políticas de descarte definidas).
- **RNF5**: **Observabilidade**: logs/metrics suficientes para auditar decisões do rate limiter e do circuito.

Cenários de Teste (aceitação)

Implemente um **test harness** (script ou testes automatizados) que dispare requisições internas e verifique:

1. **Rajada controlada**
    - Enviar 20 requisições em 1 segundo para o proxy.
    - **Esperado**: upstream recebe ~1 req/s; zero ou mínimas penalidades; fila cresce, mas estabiliza; nenhuma violação do limite externo.
2. **Penalidade proposital**
    - Simular ou forçar 2+ chamadas paralelas diretas ao upstream (sem o proxy) para observar a penalidade.

- **Esperado**: com o proxy ativo, a **mesma** carga **não** dispara penalidade.

3. **Timeout e Circuit Breaker**
    - Simular que a API remota começa a responder lentamente (> 3s) ou com erros 5xx.
    - **Esperado**: o **circuit breaker** abre após limiar, evita novas tentativas por janela configurada, fornece fallback (mensagem amigável/cached), e fecha posteriormente.
4. **Política de Fila**
    - Configurar prioridades por tipo de requisição (p.ex., adicionando código no header).
    - **Esperado**: pedidos prioritários passam à frente quando a fila está cheia; pedidos com TTL vencido são descartados (DROPPED) com motivo.
5. **Observabilidade**
    - Métricas refletem: taxa de enfileiramento, tamanho máximo da fila, latência percentil, retries, status do circuito, contagem de quedas por política.

Entregáveis

1. **Código-fonte** (repo público) com README contendo:
    - Decisões de design e padrões utilizados;
    - Como rodar (Docker Compose ou equivalente), variáveis de ambiente, seed de testes;
    - Endpoints internos e exemplos de uso (Swagger/curl/HTTPie/Postman/Insomnia).
2. **Relato técnico curto** (no README ou documento) contendo:
    - **Padrões** adotados e rejeitados (com justificativa);
    - **Experimentos** (resultados dos testes);
    - **Análise crítica**: trade-offs.

Critérios de Avaliação (rubrica)

- **Aderência ao Problema (50%)**: respeita 1 req/s e implementa um cache? (requisitos obrigatórios)
- **Qualidade de Arquitetura (50%)**: uso consistente de, pelo menos dois, padrões e qualidade dos documentos entregues.
- **(BÔNUS)** Até 20% na nota da P1 caso requisitos bônus sejam entregues e sigam bons princípios (DP)

Segurança & Ética de Uso

- Evitar causar carga desnecessária ao provedor: priorize **simulação** e **limites** claros durante os testes.
- Nunca expor credenciais em repositórios públicos.

Roteiro

1. **Aula 1 — Descoberta & Design**: entender API/contratos, definir métricas, desenhar arquitetura e padrões (esboço).
2. **Aula 2 — MVP**: fila, rate limiter, emissor a 1 req/s.
3. **Aula 3 — Finalização:** finalização da aplicação e submissão.