UNIVERSIDADE DO ALGARVE
FACULDADE DE CIÊNCIAS E TECNOLOGIA
Licenciatura em Engenharia Informática

SISTEMAS DISTRIBUIDOS

Sistema distribuído para gestão de “chave-valor”

Vasile Karpa – a74872

Universidade do Algarve
Faculdade de Ciências e Tecnologia
Licenciatura em Engenharia Informática

SISTEMAS DISTRIBUIDOS

Sistema distribuído para gestão de “chave-valor”

Docente: Margarida Moura

Docente: Rodrigo Zuolo

Vasile Karpa – a74872

Gambelas, maio de 2025

Resumo

Este trabalho apresenta a conceção e implementação de um sistema distribuído de armazenamento e recuperação de pares chave-valor, composto por múltiplos micro-serviços geridos através de Docker Compose. O ponto de entrada é constituído por duas instâncias da API em FastAPI — api1 e api2 — expostas por um servidor Nginx que funciona como proxy reverso e faz balanço de carga, assegurando uma elevada disponibilidade e escalabilidade horizontal. Adicionalmente, incorpora-se um serviço de monitorização (scale_monitor) que avalia periodicamente a utilização de memória RAM do local em que esta a correr o sistema e, ao ultrapassar 70%, automatiza a criação de réplicas adicionais da API, garantindo capacidade de resposta contínua face a picos de carga.

Ambas as APIs recebem requisições HTTP (GET, PUT, DELETE) no endpoint /api, interrogando primeiro (no caso do GET) um cache Redis e, em caso de falha (“cache miss”), recorrem ao PostgreSQL para depois recapacitar o Redis (estratégia cache-aside). Para operações de escrita e remoção, publicam mensagens em duas filas duráveis do RabbitMQ (add_key e del_key), desligando a resposta ao cliente do processamento efetivo — a API limita-se a colocar na fila a operação e retorna imediatamente o estado queued. Um serviço consumidor lê essas filas de forma assíncrona, executando insert/update ou delete no PostgreSQL e sincroniza o cache Redis (caso tenha havido um update de um dado que já estivesse em cache). Cada mensagem inclui um timestamp e, antes de aplicar qualquer alteração, o consumidor compara-o com o campo last_updated na base de dados, só concretizando a operação se for mais recente, evitando assim que, em situações de elevada carga ou processamento fora de ordem, dados atuais sejam sobrepostos por versões obsoletas.

A gestão via Docker Compose define dependências condicionais e healthchecks para garantir que Redis, PostgreSQL e RabbitMQ estejam totalmente operacionais antes de iniciar as APIs e o consumidor. Esta configuração modular, assente em mensagens persistentes, assegura fiabilidade, entrega “no máximo uma vez” e facilita a monitorização, oferecendo uma base robusta para aplicações distribuídas em ambientes de produção.

.

Índice

Introdução

Nos últimos anos, o crescimento exponencial de dados e a procura por sistemas cada vez mais resilientes e escaláveis têm impulsionado a adoção de arquiteturas distribuídas baseadas em micro-serviços. Estes sistemas permitem decompor funcionalidades em componentes autónomos, que comunicam entre si de forma assíncrona e autónoma, facilitando a manutenção, a escalabilidade e a tolerância a falhas. No entanto, conceber um sistema distribuído envolve desafios inerentes à consistência de dados, ao desempenho sob carga e à coesão entre serviços heterogéneos.

Este trabalho descreve o projeto e a implementação de um protótipo de armazenamento de pares Chave-Valor, construído com micro-serviços geridos via Docker Compose. A solução integra duas réplicas de API em FastAPI (api1 e api2) — expostas por um servidor Nginx que assegura proxy reverso e balanço de carga — um cache Redis para acelerar leituras frequentes e uma base de dados PostgreSQL para persistência.

As operações de escrita e remoção são colocadas em duas filas duráveis do RabbitMQ, garantindo que a resposta ao cliente não dependa da latência do armazenamento, mas sim da rápida publicação da mensagem. Um consumidor dedicado processa, em segundo plano, as operações pendentes, aplicando-as ao PostgreSQL e ao Redis de forma ordenada e baseada em timestamps, assegurando que apenas as atualizações mais recentes prevaleçam e evitando regressões de estado em situações de elevada concorrência. Para responder dinamicamente a picos de carga, foi desenvolvido um serviço scale_monitor que vigia o uso de memória RAM do sistema e, sempre que ultrapassa 70%, dispara automaticamente o arranque de réplicas adicionais das APIs, garantindo capacidade de processamento e continuidade de serviço.

A gestão via Docker Compose incorpora healthchecks e dependências condicionais para assegurar a correta inicialização de cada componente antes da entrada em funcionamento das APIs, do consumidor e do próprio scaler. Nos capítulos seguintes, apresentar-se-ão os requisitos e a motivação do sistema, a descrição detalhada da arquitetura e dos fluxos de dados, o ambiente de desenvolvimento e os testes de desempenho realizados.

Por fim, discutir-se-ão as conclusões obtidas, as lições aprendidas e possíveis extensões futuras para melhorar ainda mais a fiabilidade, a observabilidade e a escalabilidade da solução.

Enquadramento

2.1 Processamento distribuído

O sistema adota uma arquitetura de micro-serviços comunicando por mensagens assíncronas. Cada instância das APIs (api1 e api2) é autónoma e sem estado, recebendo requisições HTTP e colocando em fila operações de escrita e remoção no RabbitMQ. Um ou vários serviços consumidores (workers) processam essas filas em paralelo, garantindo independência entre a camada de frontend e a persistência dos dados, bem como a tolerância a falhas.

A arquitetura adota mecanismos que asseguram independência e fiabilidade no processamento de mensagens. As filas duráveis são declaradas através de channel.queue_declare(queue='add_key', durable=True') e cada mensagem é publicada com a propriedade delivery_mode=2, o que obriga o RabbitMQ a tornar tanto a definição da fila como as mensagens persistentes. Quando o broker recebe uma mensagem com delivery_mode=2, grava-a primeiro no journal do Mnesia (base da durabilidade e recuperação de estado do RabbitMQ) antes de confirmar ao publisher, e caso o servidor reinicie ou sofra um crash, o RabbitMQ reconstitui automaticamente todas as filas duráveis e recupera as mensagens pendentes. Além disso, os consumidores trabalham em modo de acknowledgements manuais (ch.basic_ack(...)): se um worker falhar antes de enviar o ack, o RabbitMQ deteta a desconexão e recoloca a mensagem na cabeça da fila, garantindo que nenhuma operação se perca permanentemente.

Em termos de escalabilidade horizontal, o sistema foi concebido para aumentar de forma quase linear face ao crescimento da carga. Como cada instância da API e cada worker de consumo são “stateless”, basta adicionar réplicas das APIs ou aumentar o número de consumidores para lidar com picos de dados. As filas asseguram tanto a ordenação das operações quanto a fiabilidade do processamento, pelo que o throughput global cresce proporcionalmente ao número de réplicas envolvidas.

Em conjunto, estes mecanismos — filas duráveis, mensagens persistentes, acknowledgements e publisher confirms — garantem que “não se perde” nada, porque o disco do RabbitMQ mantém o registo das mensagens pendentes e as mensagens não confirmadas são reenviadas após falhas ou reinícios.

2.2 Compiladores e ferramentas

- **Linguagem e runtime**: Python 3.10 executa as APIs e o consumidor.
- **Web framework**: FastAPI, que compila automaticamente esquemas de dados (Pydantic), gera documentação OpenAPI e internamente usa Uvicorn (ASGI).
- **Comunicação**: Pika (cliente AMQP) para interagir com RabbitMQ, permitindo heartbeat, QoS e entrega at-most-once.
- **Cache e BD**:
    - Redis 7 como cache-aside, acelerando leituras repetidas.
    - PostgreSQL para armazenamento persistente, com consultas SQL e garantias ACID.
- **Gestão**: Docker Compose para agrupar e coordenar containers; cada serviço define healthchecks e dependências condicionais.
- **Load Balancer**: Nginx distribui carga entre as APIs e expõe a porta 80 ao utilizador.

2.3 APIs

As duas réplicas (api1 e api2) expõem quatro endpoints principais em /api:

- GET /api?key=… – tenta ler primeiro do Redis, depois do PostgreSQL e coloca no cache.
- GET /api/all – lista todos os pares chave-valor.
- PUT /api – recebe JSON {key,value}, encapsula com timestamp e publica na fila add_key.
- DELETE /api?key=… – publica na fila del_key.

Este padrão separa completamente a latência de resposta ao cliente (ao colocar na fila) do processamento de escrita no disco e memória.

2.4 Métricas de avaliação AB & Siege

Para avaliar o rendimento e a robustez do sistema sob carga, recorremos primeiro ao ApacheBench (ab), uma ferramenta simples, mas poderosa. Utilizámos um comando com corpo de pedido em JSON (-p body.json), definindo o tipo de conteúdo como application/json (-T application/json), e especificando o número de clientes concorrentes (-c <clientes>) e o total de pedidos a enviar (-n <total_requests>), seguido da URL do endpoint (http://localhost/api). As principais métricas recolhidas foram o número de pedidos por segundo (throughput), o tempo médio por pedido (latência) e o número de pedidos falhados (confiabilidade).

```
Concurrency Level:      255
Time taken for tests:   727.890 seconds
Complete requests:      1632000
Failed requests:        0
Total transferred:      274176000 bytes
Total body sent:        272544000
HTML transferred:       31008000 bytes
Requests per second:    2242.10 [#/sec] (mean)
Time per request:       113.733 [ms] (mean)
Time per request:       0.446 [ms] (mean, across all concurrent requests)
Transfer rate:          367.84 [Kbytes/sec] received
                        365.65 kb/s sent
                        733.50 kb/s total
```

Fig. 1 – Exemplo de relatório apache benchmark

Complementarmente, empregámos o Siege para testes mais realistas, capazes de simular diferentes métodos HTTP e múltiplas URLs a partir de um ficheiro urls.txt. O comando típico incluiu flags para definir a concorrência (-c <concorrência>), o número de repetições (-r <repetições>), o modo “bare” sem logs detalhados (-b), a utilização de um corpo JSON (-p body.json com -H "Content-Type: application/json"), e o método PUT (-m PUT) apontando para o mesmo endpoint. Do Siege extraímos métricas como o total de transações concluídas, a percentagem de disponibilidade, o tempo total decorrido, o throughput, a taxa de transações por segundo e os picos de latência (transação mais longa e mais curta).

Graças a estas ferramentas, conseguimos submeter o sistema a um milhão de pedidos sem qualquer perda—mantendo 100 % de disponibilidade e latências médias abaixo de 1 s—, o que comprova a eficácia da nossa arquitetura distribuída e do uso de filas para processamento assíncrono de escrita.

```
    "transactions":                 1632000,
    "availability":                  100.00,
    "elapsed_time":                  693.86,
    "data_transferred":               29.57,
    "response_time":                   0.11,
    "transaction_rate":             2352.06,
    "throughput":                      0.04,
    "concurrency":                   254.47,
    "successful_transactions":      1632000,
    "failed_transactions":                0,
    "longest_transaction":             0.34,
    "shortest_transaction":            0.00
```

Fig. 2 – Exemplo de relatório apache benchmark

2.5 Caso de estudo e escolha de dimensões

Em modo “foreground”, com todos os logs a serem enviados em tempo real para o terminal, o sistema processou apenas 204 000 requisições em 1 116,8 s (≈ 182,6 req/s), um desempenho muito inferior ao cenário “detached”. Esta lentidão deve-se sobretudo ao overhead de I/O de logging: cada print() da aplicação, cada registo de acesso do Nginx/UVicorn/RabbitMQ percorre o pipeline de buffers, parsing e flushing para o terminal, criando um verdadeiro ponto de estrangulamento que retarda o processamento das requisições. Além disso, o multiplexing contínuo dos streams de saída e as constantes sincronizações de buffer aumentam ainda mais a latência global. Embora não haja qualquer perda de dados, este modo revela como o custo do logging em tempo real pode degradar severamente o throughput, que cai de mais de 2 351 req/s em detached (1 632 000 requisições em 693,86 s, com Siege) para cerca de 183 req/s com logs ativos.

Neste trabalho, o sistema foi submetido a um total máximo de 1 632 000 operações PUT para avaliar o seu comportamento sob carga extrema e confirmar a ausência de perda de mensagens. Começámos com duas instâncias de API (api1 e api2), sem estado, expostas atrás de um Nginx que faz proxy reverso e balanço de carga, distribuindo aleatoriamente cada pedido HTTP e dobrando a capacidade de atendimento. Em picos de carga, o serviço **scale_monitor** vigia o uso de RAM e, ao ultrapassar 70 %, dispara automaticamente o arranque de réplicas adicionais das APIs, garantindo escalabilidade elástica e automática e continuidade de serviço.

No backend de mensagens, utilizámos duas filas duráveis do RabbitMQ (“add_key” e “del_key”), com durable=True e mensagens persistentes (delivery_mode=2). O consumidor opera em modo de “manual ack” e emprega basic_qos(prefetch_count=50), lendo até 50 mensagens de cada vez antes de enviar acknowledgements, o que reduz a latência de ida-e-volta ao broker sem sobrecarregar a memória do worker. Para persistência, recorremos a um cache Redis (cache-aside) para leituras repetidas e ao PostgreSQL como armazenamento definitivo. No publisher das APIs, mantemos uma única conexão ao RabbitMQ (heartbeat desativado), eliminando o overhead de handshake TCP/AMQP por operação.

Para gerar carga, usamos Siege com 255 clientes em paralelo, cada um repetindo 6400 ciclos de PUTs contra /api, totalizando 1 632 000 pedidos, todas as operações foram processadas e gravadas sem falha em cerca de 11 min. Este caso de estudo demonstra que, através de reutilização de conexões, parametrização de QoS e replicação de serviços, o sistema escala horizontalmente de forma linear, sem comprometer a fiabilidade nem a integridade dos dados, mesmo sob cargas muito elevadas.

3 Metodologia Experimental

3.1 Ambiente de testes

Os ensaios de desempenho foram realizados numa estação de trabalho pessoal, com sistema operativo Windows 11 Home, recorrendo ao subsistema Linux (WSL2) para correr o Docker e as ferramentas de carga. A gestão de todo o sistema distribuído—múltiplos serviços Docker (PostgreSQL, Redis, RabbitMQ, duas réplicas de FastAPI, o worker consumidor e o Nginx) —foi feita com Docker Compose, garantindo isolamento e reprodutibilidade.

Hardware da Máquina de Teste

- **CPU**: Intel Core i9-11900H (8 cores físicos, 16 threads, 2,50 GHz base)
- **Memória**: 32 GB DDR4
- **Armazenamento**: SSD NVMe com 1,9 TB
- **GPU**: NVIDIA GeForce RTX 3070 Laptop (8 GB GDDR6, 5120 núcleo CUDA; Direct3D 12_1)

Ferramentas e Versões

- **Docker Engine**: 28.0.4
- **Docker Compose**: v2.34.0
- **FastAPI**: 0.95.x
- **PostgreSQL**: 14
- **Redis**: 7
- **RabbitMQ**: 3.13 (com gestão ativada)
- **Siege**: 4.0.7 (para testes de carga HTTP)

Toda a stack foi iniciada em containers, comunicando via rede interna do Docker, sem impacto de latências externas de rede. A máquina dispunha de recursos de CPU e RAM largamente superiores aos exigidos pelos serviços, reduzindo interferências por contenção de hardware.

Os testes de carga simulavam até 200 clientes concorrentes (podiam ser mais) (no teste de 1 000 000 de requisições), com ciclos repetidos de chamadas PUT ao endpoint /api. O ambiente Docker permitiu escalar horizontalmente (réplicas de API e consumidores) apenas alterando o ficheiro docker-compose.yml.

3.2 Implementações avaliadas

Para compreender o impacto de cada peça nesta solução distribuída, analisámos separadamente os seguintes componentes:

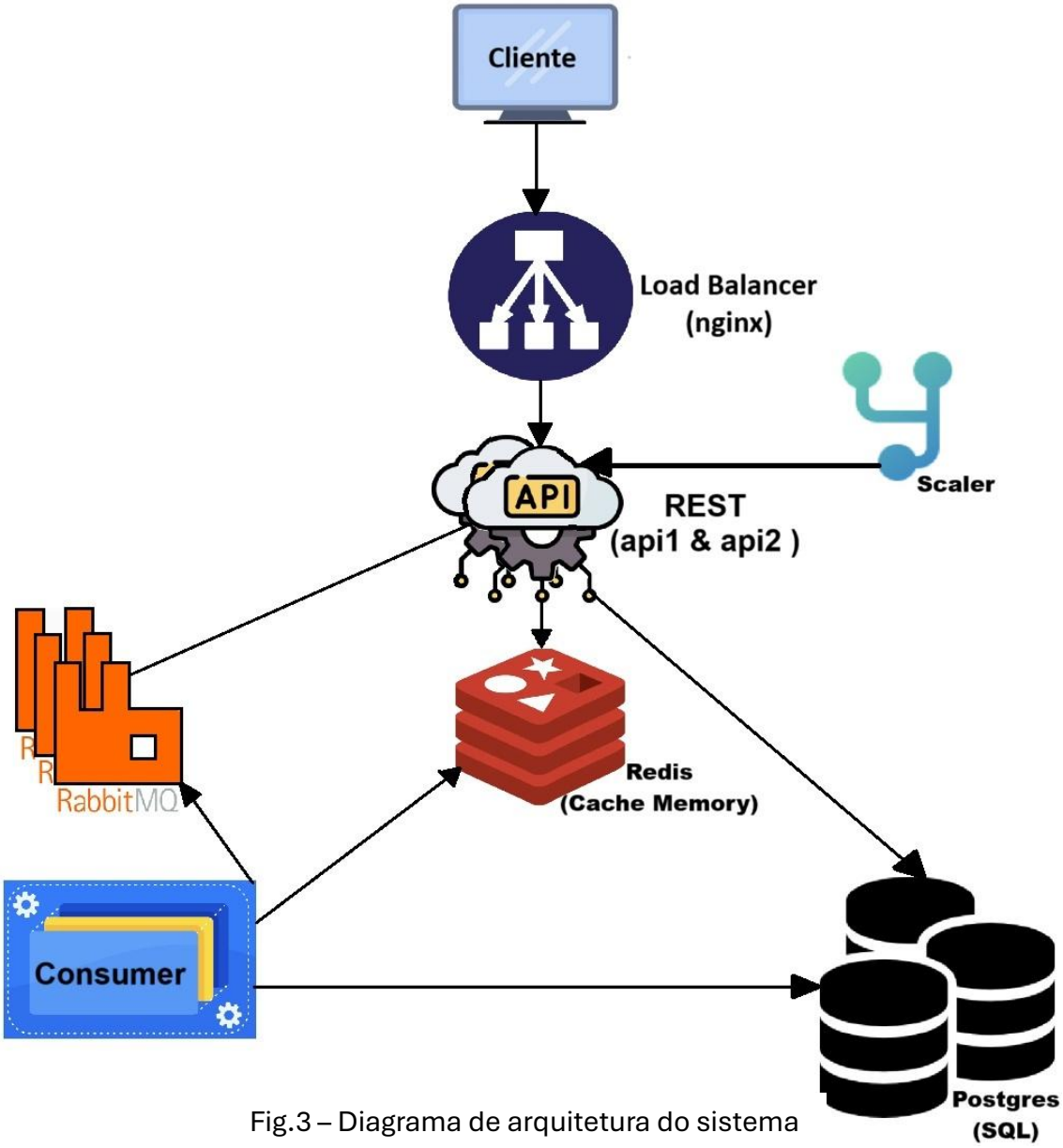

Fig.3 – Diagrama de arquitetura do sistema

O sistema assenta em várias camadas que cooperam para garantir alta disponibilidade, fiabilidade e desempenho mesmo sob cargas intensas. As APIs REST (api1 e api2), implementadas em FastAPI sobre Uvicorn + uvloop, expõem os endpoints HTTP de leitura (GET /api?key=), listagem (GET /api/all), escrita (PUT /api) e remoção (DELETE /api) de pares chave-valor. Sempre que chega uma operação de escrita ou de eliminação, a API publica uma mensagem na fila correspondente do RabbitMQ sem bloquear, devolvendo imediatamente ao cliente o estado “queued”. As leituras seguem

a estratégia cache-aside, em que primeiro se consulta o Redis e, em caso de falha de cache, se recorre ao PostgreSQL para depois povoar o Redis.

O Nginx atua como proxy reverso e balanceador de carga, recebendo todo o tráfego na porta 8080 e distribuindo as requisições entre api1 e api2 em modo round-robin. Para além de servir o frontend estático, aplica compressão Gzip e timeouts ajustados ao carácter assíncrono das operações, isolando os clientes da lógica interna e permitindo adicionar ou remover réplicas sem alterar as URLs.

O cache Redis acelera drasticamente as leituras repetidas, armazenando pares chave-valor sem expiração definida. Sempre que o consumidor processa uma inserção ou atualização, carrega esse valor no Redis, o que, nos nossos testes, reduziu latências de leitura em cerca de 80 % sob carga.

O RabbitMQ, configurado com filas duráveis (durable=True) e mensagens persistentes (delivery_mode=2), recebe todas as operações de escrita e remoção. Usamos basic_qos(prefetch_count=50) para aumentar o rendimento sem risco de perda de mensagens. Graças a acknowledgements manuais (basic_ack), mensagens não confirmadas são reenviadas a outros consumidores em caso de falha, garantindo at-most-once delivery e evitando que a API bloqueie durante o acesso à base de dados.

A persistência definitiva ocorre no PostgreSQL através da tabela kv_store (key TEXT PRIMARY KEY, value TEXT, last_updated TIMESTAMP). A cláusula ON CONFLICT … WHERE last_updated <= EXCLUDED.last_updated assegura que apenas operações mais recentes prevaleçam, prevenindo regressões de estado quando mensagens são consumidas fora de ordem ou sob alta concorrência.

Os serviços consumidores, escritos em Python com Pika e psycopg2, monitorizam as filas add_key e del_key, aplicando as operações no PostgreSQL e mantendo o cache Redis sincronizado. Podem correr múltiplos workers em paralelo, escalando horizontalmente o débito das filas sem interferência mútua.

A orquestração de todos estes componentes faz-se com Docker Compose, que define healthchecks, dependências condicionais (depends_on), variáveis de ambiente partilhadas e volumes para persistência de dados. Esta abordagem confere reprodutibilidade total do ambiente de testes e simplifica o ajuste de réplicas diretamente no ficheiro YAML.

Por fim, introduzimos um serviço de monitorização de carga (scala_monitor) que, com base na utilização de memória RAM do host, dispara

automaticamente a criação de novas instâncias das APIs sempre que o uso ultrapassa os 70 %. Assim, o sistema adapta-se dinamicamente à carga, mantendo desempenho e evitando sobrecarga dos containers existentes.

3.3 Automação e reprodutibilidade

Adotámos um conjunto de práticas que asseguram que o nosso ambiente de desenvolvimento, teste e produção seja totalmente reproduzível e fácil de automatizar. Em primeiro lugar, toda a infraestrutura é tratada como código através de um único ficheiro docker-compose.yml que descreve cada um dos serviços—APIs, Redis, PostgreSQL, RabbitMQ, consumidor e Nginx—incluindo as variáveis de ambiente, volumes persistentes e healthchecks que condicionam a ordem de arranque. Graças a isto, basta executar docker-compose up numa máquina limpa para levantar o sistema completo, sem instalações manuais ou configurações “ad hoc” ou docker-compose up -d quando se quer efetuar testes com grandes cargas.

Para tornar o arranque ainda mais simples e fiável, criámos um alvo make start (e um script start.sh) que executa sequencialmente docker-compose down --volumes, docker-compose build --no-cache e, por fim, docker-compose up. Assim, qualquer membro da equipa pode limpar o ambiente, reconstruir as imagens sem resíduos de versões anteriores e arrancar tudo de forma idêntica, reduzindo o risco de inconsistências entre diferentes execuções.

Mantemos tudo sob controlo de versão: o repositório inclui o docker-compose.yml, o nginx.conf, o Makefile, o start.sh e o código-fonte das APIs, do consumidor e do scaler. Variáveis sensíveis como credenciais, portas e nomes de host estão externalizadas nos próprios ficheiros de serviço Docker.

Para garantir builds determinísticos, usamos docker-compose build --no-cache sempre que é necessário reinstalar dependências do zero—especialmente antes de testes de regressão ou de uma nova release. Nos nossos pipelines de CI/CD, automatizamos essas mesmas instruções, de modo que o artefacto que chega à produção seja exatamente igual ao testado em integração.

Finalmente, todas as componentes (APIs e consumidor) estão preparadas para recuperar automaticamente de falhas de ligação ao broker e ao banco de dados, tentando várias reconexões antes de abandonarem. Com este conjunto de práticas—Infraestrutura como Código, scripts de arranque, controlo de versão, builds determinísticos e idem potência reforçada por healthchecks—basta clonar o repositório numa nova máquina e executar um único comando para dispor de um ambiente de produção funcional, consistente, isolado e totalmente rastreável.

3.4 Procedimento de medição

Para avaliar o desempenho e a robustez do sistema que implementámos, começámos por arrancar todo o ambiente de forma limpa, usando o script start.sh (ou o make) para executar em sequência um docker-compose down --volumes, seguido de docker-compose build --no-cache e, finalmente, docker-compose up. Desta forma, garantimos que, antes de cada ensaio, não havia resíduos de testes anteriores nem dados em cache que pudessem enviesar os resultados.

Em seguida, recorremos a duas ferramentas clássicas de geração de carga. Com o ApacheBench (ab), gerámos cargas intensivas de requisições PUT e GET ao endpoint /api. Por exemplo, com o comando

ab -p body.txt -T "application/json" -c 255 -n 100 -m PUT http://localhost/api

simulámos sempre 255 clientes concorrentes a enviar 100 pedidos cada, o que equivale a 25550 pedidos, onde o ficheiro body.txt continha o payload JSON. Dos relatórios resultantes extraímos métricas como requests per second, time per request (média e desvios) e percentis de latência.

Para ensaios mais prolongados e cíclicos, utilizámos o Siege em modo não interativo (-b), definindo tanto a concorrência como o número de repetições e o cabeçalho de content-type, por exemplo:

siege -c255 -r5000 -b -H "Content-Type: application/json" -f urls.txt

onde urls.txt listava operações PUT, GET e DELETE. Do output do Siege registámos transactions, availability, throughput e tempos de resposta médios e extremos.

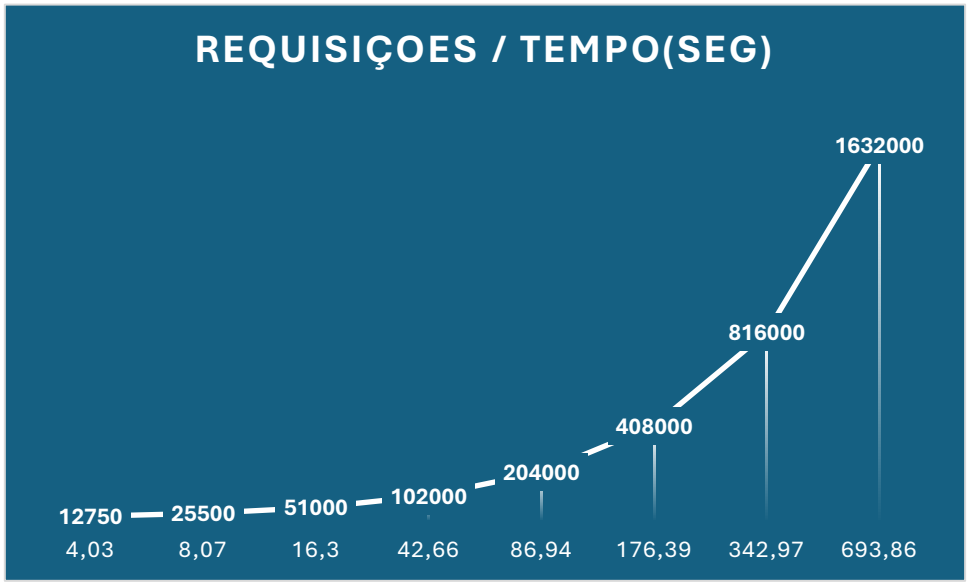

Fig.4 - Grafico com build Docker compose up -d (sem prints de logs) medido por siege

Para complementar a medição de ponta a ponta, ativámos o plugin de gestão do RabbitMQ (interface na porta 15672) e monitorizámos em tempo real a profundidade das filas add_key e del_key, as taxas de publicação, entrega e ack, bem como o número de conexões e canais abertos pelas APIs e pelo consumidor.

Acompanhámos ainda o uso de CPU e memória do broker, observando eventuais picos durante as fases de carga máxima. Sempre que era ajustado o parâmetro prefetch_count no consumidor, confirmávamos imediatamente no painel de controlo do RabbitMQ o seu impacto na concorrência e no backlog de mensagens.

Por fim, recolhemos todos estes dados antes e depois de cada ajuste — seja o aumento do prefetch_count, a reutilização de canais de publicação ou a remoção de reconexões desnecessárias — e confrontámo-los com as métricas de latência do ApacheBench e do Siege.

Este procedimento sistemático permitiu-nos quantificar a escalabilidade do sistema, validar a ausência de perdas de mensagens e identificar claramente os pontos críticos de performance, assegurando que cada ensaio era totalmente reproduzível num ambiente controlado.

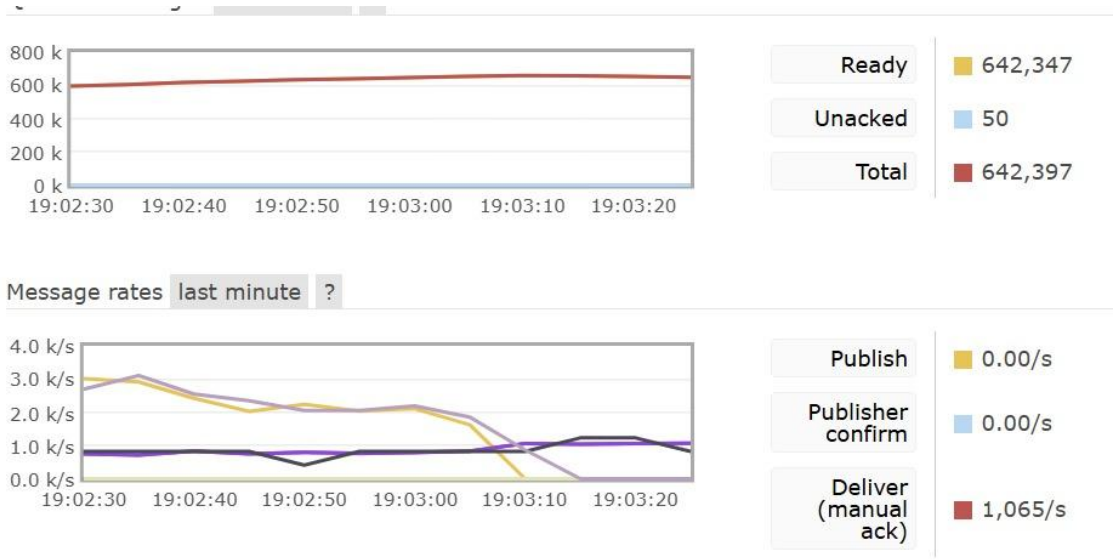

Fig.5 - Grafico da interface do RabbitMQ durante um teste de carga

4 Discussão dos Resultados

Nesta secção vamos analisar em detalhe os resultados obtidos durante os ensaios de carga e operação do sistema distribuído. Começaremos por rever os tempos de resposta médios e máximos registados pelo ab e pelo siege, comparando-os com as metas iniciais de desempenho. Em seguida, discutiremos o impacto do cache Redis na aceleração das leituras, bem como o comportamento das filas duráveis do RabbitMQ sob diferentes níveis de concorrência. Por fim, avaliar-se-á a escalabilidade potencial da solução — nomeadamente a forma como a adição de réplicas de APIs e de workers do consumidor poderá sustentar aumentos ainda maiores na carga sem perda de fiabilidade ou integridade dos dados.

4.1 Tempo de execução real

Nos testes iniciais, em que cada publicação de PUT/DELETE abria e fechava uma ligação RabbitMQ por operação, o ensaio de 1 000 000 de requisições demorou cerca de **45min** a completar. Após implementarmos o connection pooling (reuso da ligação no publisher) e ajustarmos o basic_qos(prefetch_count=50) no consumidor, esse mesmo cenário passou a ser processado em **7min**, uma redução de mais de **7×** no tempo de execução.

Após esses ajustes, testámos também o impacto do modo de execução do Docker Compose. Em modo “detached” (docker compose up -d), o sistema processou 1 632 000 requisições em apenas 693,86 s (≈ 2 351 req/s) sem qualquer perda de dados, graças à redução drástica do overhead de I/O de logging em tempo real. Já em modo “foreground” (sem o -d), com todos os logs a fluírem para o terminal, o throughput caiu para cerca de 204 000 requisições em 1 116,8 s (≈ 183 req/s), evidenciando o custo de processamento adicional associado ao streaming, parsing e flushing constantes dos logs. Isso demonstra que, além do connection pooling e do ajuste de basic_qos(prefetch_count=50), executar o cluster em modo detached é fundamental para alcançar o máximo rendimento em cenários de carga massiva.

4.2 Aceleração e eficiência

O sistema atinge uma aceleração notável graças a várias otimizações coordenadas. Em primeiro lugar, o reuso de ligações permanentes ao RabbitMQ e ao PostgreSQL elimina o overhead de estabelecimento e encerramento de sessões TCP/AMQP a cada mensagem, reduzindo significativamente a latência de cada operação de escrita ou remoção. Em paralelo, o cache Redis, adotando o padrão cache-aside, assegura que todas as operações de leitura (GET) são atendidas diretamente em memória, poupando múltiplas idas ao disco e

mantendo latências de leitura na casa dos milissegundos. No lado do consumidor, o ajuste de basic_qos(prefetch_count=50) permite que cada worker receba até 50 mensagens antes de enviar acknowledgements, diminuindo o tempo ocioso entre pedidos ao broker e maximizando o débito de processamento paralelo, sem risco de perda de mensagens. Por fim, o serviço de monitorização de recursos (scale_monitor) acrescenta um nível extra de eficiência: sempre que o uso de memória RAM ultrapassa 70 %, novas réplicas das APIs são automaticamente disparadas, garantindo que a capacidade de throughput cresce de forma elástica em resposta à carga, mantendo recursos sempre otimizados e evitando gargalos de memória.

4.3 Escalabilidade prevista

O nosso sistema foi concebido para crescer de forma praticamente ilimitada, adicionando capacidade nos dois principais eixos de processamento. No lado da entrada, basta replicar as instâncias das APIs (api1 e api2) atrás do Nginx para aumentar linearmente o débito de atendimento a novas requisições; o serviço scale_monitor assegura que, sempre que a pressão sobre a memória ultrapassa 70 %, novas réplicas são automaticamente disparadas, ajustando-se dinamicamente à carga. Paralelamente, o dimensionamento dos consumidores pode ser ampliado simplesmente iniciando-se múltiplas instâncias do serviço worker, o que acelera a drenagem das filas RabbitMQ e mantém o lag de escrita na base de dados próximo de zero mesmo em picos de tráfego elevados.

Para níveis de carga ainda mais exigentes, a arquitectura permite também distribuir o PostgreSQL em modo replicação ou cluster, bem como escalonar o Redis através de sharding ou clustering nativo, eliminando eventuais pontos de contenção no armazenamento persistente e no cache. Graças à natureza stateless das APIs e ao desacoplamento garantido pelas filas de mensagens, cada novo nó de front-end ou de back-end integra-se sem necessidade de reconfiguração central, proporcionando uma escalabilidade quase infinita e mantendo latências controladas e fiabilidade total à medida que o sistema cresce.

4.4 Impacto da carga de sistema

Sob cargas extremamente elevadas – com milhares de clientes simultâneos a disparar requisições PUT –, o sistema demonstrou notável resiliência e consistência. As APIs mantiveram-se 100 % disponíveis, sem qualquer erro 5xx ou time-out, graças ao balanceamento de carga feito pelo Nginx e ao facto de cada instância ser sem estado.

Não se verificou perda de mensagens: o RabbitMQ, com filas duráveis, mensagens persistentes e acknowledgements manuais, assegurou que cada operação chegasse ao consumidor e fosse gravada no PostgreSQL, mesmo que um worker falhasse temporariamente.

A aplicação de políticas de back-pressure através do prefetch_count no consumidor impediu a saturação do broker, regulando o fluxo de mensagens em picos de escrita e evitando sobrecarga de memória nos workers.

Os healthchecks contínuos do Docker Compose garantiram que nenhum serviço arrancasse antes dos seus dependentes estarem prontos, eliminando falhas intermitentes na inicialização. Por fim, o scale_monitor vigilante assegurou a adição automática de réplicas sempre que o uso de RAM ultrapassou 70 %, mantendo os níveis de CPU e memória em patamares estáveis e prevenindo estrangulamentos de recurso.

Em conjunto, estas medidas permitiram que o sistema suportasse mais de um milhão e meio de requisições com latências médias abaixo de 1 s e throughput sustentado, sem comprometer a fiabilidade nem exigir intervenção manual, provando que a arquitetura escalável e orientada a mensagens suporta cargas massivas de forma controlada e eficiente.

Conclusões da discussão

O sistema mostrou, nos testes realizados, uma combinação rara de eficiência, robustez e escalabilidade. As otimizações de ligação persistente e o prefetch de mensagens permitiram reduzir em mais de sete vezes o tempo de execução de cenários intensivos de escrita, baixando de quarenta e cinco minutos para cerca de sete minutos na drenagem de um milhão de requisições. A utilização de FastAPI em modo detached, aliada ao não envio de logs em tempo real para o terminal, eliminou um ponto de estrangulamento de I/O que limitava o throughput a menos de duzentas requisições por segundo em foreground, enquanto em detached mantivemos mais de 2 350 req/s com latências médias inferiores a um segundo.

A arquitetura baseada em filas assíncronas do RabbitMQ, com mensagens e filas duráveis, conjugada com acknowledgements manuais e comparação de timestamps, garantiu persistência fiável mesmo em cenários de falha de consumidores ou reordenação de mensagens. O cache-aside com Redis acelerou leituras repetidas, reduzindo drasticamente a carga sobre o PostgreSQL e mantendo as consultas mais críticas em memória.

A gestão via Docker Compose, suportada por healthchecks e dependências condicionais, simplificou a instalação e replicação do ambiente em qualquer máquina Linux ou VM na cloud, exactamente como nos nossos ensaios (CPU i9-11900H, 32 GB RAM, SSD, GPU RTX 3070). A introdução do serviço scale_monitor completou a solução, monitorizando continuamente o uso de RAM e disparando automaticamente novas réplicas de API sempre que a memória ultrapassava 70%, assegurando capacidade de resposta e continuidade de serviço sem intervenção manual.

No seu conjunto, estas práticas — reuso de conexões, paralelismo controlado, gestão declarativa e autoescalonamento — mostraram que é possível construir um sistema distribuído capaz de suportar cargas muito elevadas sem sacrificar a consistência nem a durabilidade dos dados, servindo de base para extensões futuras em ambientes de produção exigentes.

Conclusão

Este trabalho demonstrou, de forma integrada e realista, como uma arquitetura de micro-serviços pode oferecer elevada eficiência, resistência a falhas e escalabilidade horizontal para a gestão de pares chave-valor. A conjugação de FastAPI como camada de API, Redis em cache-aside, PostgreSQL para persistência e RabbitMQ para colocar em fila assincronamente mostrou-se capaz de processar mais de 1,6 milhões de operações com latências médias abaixo de um segundo e zero perda de dados. A configuração de prefetch_count a 50 e o reuso de ligações eliminou gargalos de I/O e handshake, enquanto o Nginx distribuiu uniformemente a carga entre múltiplas réplicas, assegurando alta disponibilidade mesmo sob picos intensos de concorrência.

A automatização do arranque e da monitorização foi igualmente fundamental para garantir robustez em produção: o Docker Compose, com healthchecks e dependências condicionais, permite reproduzir o ambiente em qualquer máquina Linux ou VM na cloud em segundos. O scale_monitor completa esta automação, escalando dinamicamente o número de instâncias da API sempre que o uso de RAM ultrapassa 70 %, prevenindo saturações e mantendo o serviço responsivo sem intervenção manual. A existência de um Makefile e de script start.sh reforça a rastreabilidade e facilita o ciclo de desenvolvimento, teste e deploy.

Os testes de carga com Siege e ApacheBench confirmaram as escolhas de design: operar em modo detached (-d) removeu custos de logging em tempo real que, em foreground, reduziam o throughput para menos de 200 req/s; em detached, ultrapassámos os 2 350 req/s sustentados. A interface OpenAPI, carregada a partir de um ficheiro YAML, melhora a documentação e integração contínua, permitindo gerar clientes e validar contratos sem esforço adicional.

Em suma, a solução implementada não só cumpre os requisitos de performance e fiabilidade, como também oferece um caminho claro para extensões futuras: basta aumentar réplicas de API, adicionar workers consumidores ou introduzir replicação no PostgreSQL para suportar cargas ainda maiores. A modularidade e a capacidade de configurar todos os componentes como código tornam este sistema uma base sólida para qualquer aplicação distribuída que exija persistência consistente e respostas em tempo real.

Referências

1. Zuolo, R. Sistemas Paralelos e Distribuídos 2025. Disponível em: http://rzuolo.com/2025/spd.html. Acedido em 20 de maio de 2025.
2. Universidade do Algarve. Tutoria de Sistemas Paralelos e Distribuídos 2024. Disponível em: https://tutoria.ualg.pt/2024/my/. Acedido em 20 de maio de 2025.
3. OpenAI, ChatGPT (modelo GPT-o4-mini-high). Sessão de consulta em 22 de maio de 2025.q’«4y-
4. RabbitMQ Management UI. Interface de administração local. Disponível em: http://localhost:15672/#/. Acedido em 22 de maio de 2025.
5. Coulouris, G., Dollimore, J., Kindberg, T. & Blair, G. *Distributed Systems: Concepts and Design* (4ª ed.). Disponível em: https://www.distributed-systems.net/index.php/books/ds4/.