UNIVERSIDADE da MADEIRA

Sistema de Streaming de Eventos em Tempo Real com Kafka

Relatório de Projeto - Sistemas Distribuídos

Pedro Jaques - 2046919

Tomás Freitas - 2118322

Resumo

Este projeto implementa um sistema de streaming de eventos em tempo real para monitorização de corridas, utilizando uma arquitetura de microserviços baseada em RabbitMQ. O sistema é composto por um simulador que lê ficheiros GPX e publica eventos em exchanges fanout, um backend FastAPI que subscreve a essas exchanges e expõe uma interface WebSocket, e um frontend que consome os eventos e apresenta os atletas num mapa. A solução está containerizada com Docker e orquestrada via Docker Compose (ambiente local) e Helm/Kubernetes (ambiente de cluster). A pipeline GitHub Actions constrói e publica imagens multi-arch e atualiza o chart Helm, assegurando repetibilidade dos deployments.

Introdução

Os sistemas distribuídos modernos enfrentam desafios significativos no processamento e disseminação de grandes volumes de dados em tempo real. Este projeto foi desenvolvido no âmbito da Unidade Curricular de Sistemas Distribuídos, com o objetivo de desenvolver uma solução prática que demonstre conceitos fundamentais de arquitetura distribuída, comunicação entre componentes e orquestração de serviços.

O contexto escolhido é um sistema de monitorização de eventos de corridas, especificamente de corridas trail, onde múltiplos participantes geram eventos de localização e telemetria em tempo real. A arquitetura implementada segue padrões de microserviços, utilizando RabbitMQ como broker de mensagens para desacoplar produtor e consumidor.

Os objetivos específicos deste projeto incluem:

- Implementar um sistema de streaming de dados escalável e resiliente;
- Explorar padrões de comunicação em sistemas distribuídos;
- Demonstrar práticas de containerização e orquestração;
- Estabelecer pipelines CI/CD automatizadas para integração contínua e entrega contínua;
- Implementar monitorização dos serviços com Prometheus.

Descrição da Pipeline CI/CD

A pipeline CI/CD está montada com GitHub Actions e é despoletada quando existe um push para o master branch do repositório de trabalho. O foco atual é o build e publicação das imagens Docker e atualização automática do chart Helm para disponibilização da versão mais recente das alterações à aplicação.

Estágios da Pipeline

Build e Push:

- Build multi-arch (linux/amd64, linux/arm64) para frontend, backend e simulator;
- Push para o Docker Hub com tag v<run>-<sha>.

Atualização de Helm Chart:

- Atualiza chart/values.yaml com a nova tag das imagens;
- Commit automatizado com [skip ci] para evitar loop do workflow.

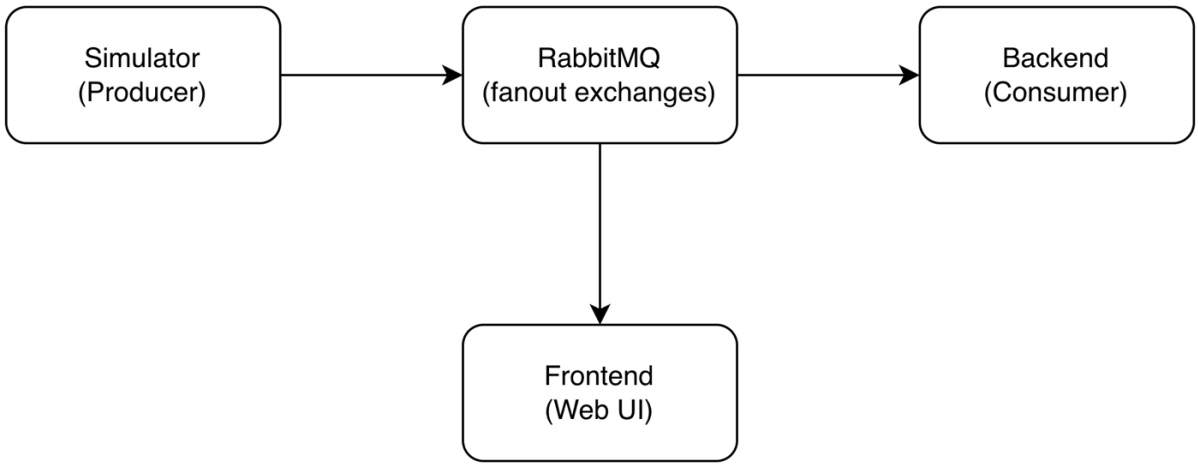
Fluxo de Deployment

1. Push para master
2. Build multi-arch e push das imagens
3. Atualização automática do Helm chart com a nova tag
4. (Manual) Deploy em cluster usando o chart atualizado

Descrição do Sistema Implementado

Visão Geral da Arquitetura

O sistema implementado segue uma arquitetura de microserviços desacoplados, com os seguintes componentes principais:



Componentes:

1. **Simulator (Produtor):** lê ficheiros GPX e publica eventos JSON em exchanges RabbitMQ do tipo fanout com nome race-<gpx>. Cada atleta corre num thread próprio com variação de velocidade.
2. **RabbitMQ:** Broker de mensagens. O simulador declara exchanges fanout por corrida, o backend cria uma queue durável e faz bind automático a todas as exchanges de utilizador (descobertas via API de gestão a cada 10s).
3. **Backend (Consumidor/API):** FastAPI que consome da queue e difunde eventos via WebSocket . Expõe métricas Prometheus e conta mensagens, conexões e envios.
4. **Frontend (Interface Web):** Página HTML/JS com Leaflet que carrega GPX locais, mantém múltiplas WebSocket (uma por corrida ou all) e mostra marcadores e leaderboard em tempo real.

Opções de Implementação

Linguagens e Frameworks

Backend:

- Linguagem: Python 3.9+
- Framework: FastAPI (async + WebSockets)
- Bibliotecas principais: aio_pika (RabbitMQ assíncrono), aiohttp (descoberta de exchanges), prometheus_fastapi_instrumentator, prometheus_client (métricas)

Simulator:

- Linguagem: Python 3.9+
- Bibliotecas principais: pika (RabbitMQ), gpxpy, threading/time, random

Frontend:

- Linguagem: HTML5, CSS3, JavaScript (vanilla)
- Bibliotecas:
    - Leaflet.js: Visualização de mapas interativos
    - OpenStreetMap: Tiles de mapa

Mecanismos de Comunicação

Critério	**RabbitMQ**	**Kafka**	**HTTP/REST**
Latência	Muito baixa	Baixa	Média/alta
Throughput	Alto (fanout leve)	Muito alto	Médio
Persistência/Replay	Limitada (fila por consumidor)	Completa (offsets)	Não
Complexidade Operacional	Baixa	Alta	Baixa
Suporte a broadcast	Fanout direto	Por partição	Não

Opção Selecionada:

RabbitMQ (exchanges fanout)

Justificação desta escolha:

- Broadcast simples: fanout entrega a todos os consumidores sem coordenação de consumer groups.
- Operação leve: deployment simples via container único, removendo a necessidade de ZooKeeper.
- Integração rápida: bibliotecas pika e aio_pika disponíveis e simples.

Topologia de Mensagens:

- Exchanges fanout race-<nome_gpx> criadas pelo simulador.
- Queue durável backend-queue no consumidor, ligada a todas as exchanges de utilizador descobertas via API de gestão.
- Mensagens são JSON, reenviadas tal-qual pelo backend para WebSockets.

Contrato de Mensagens (JSON):

```
{
    "athlete": "string",
    "gender": "string",
    "race": "string",
    "location": {
        "latitude": "float",
        "longitude": "float"
    },
    "elevation": "float",
    "time": "ISO8601 timestamp",
    "event": "string"
}
```

Reflexão sobre Mecanismos de Comunicação

Pontos Fortes da Implementação:

1. Desacoplamento: Produtor não conhece consumidores e vice-versa, facilitando evolução independente. Consome como subscriber anônimo. Apenas eventos futuros.
2. Descoberta dinâmica: backend reconfigura bindings automaticamente quando novas exchanges são criadas.
3. Simplicidade operacional: stack leve em desenvolvimento local (RabbitMQ + três serviços).
4. Métricas expostas: contadores Prometheus para mensagens consumidas, enviadas e conexões WebSocket.

Limitações e Trade-offs:

1. Sem replay histórico: uma vez consumida a mensagem da fila, clientes novos não recebem eventos passados.
2. Ordenação limitada: ordenação apenas dentro da fila do backend, não há ordenação global entre corridas.
3. Escalabilidade horizontal: arquitetura atual usa um único backend; não há shard por corrida nem múltiplas filas.
4. Segurança: uso de credenciais default e ausência de TLS/autorização.

Decisão Crítica - Ligação às Exchanges:

- Backend cria a queue durável backend-queue e faz bind a todas as exchanges não amq.* via API de gestão, re-scan a cada 10s.
- Clientes WebSocket escolhem race ou all, o backend replica a mensagem tal como recebida.

Métricas Recolhidas e Testes Efetuados

Métricas de Sistema

O backend expõe métricas Prometheus, e há recolha automatizada de indicadores e apresentação em dashboards no Grafana. Observações manuais indicam atualização em tempo real (latência sub-segundo) em ambiente Docker Compose. Próximos passos: definir scraping (Prometheus/Grafana) e medir latência end-to-end, consumo de CPU/RAM e taxa de eventos.

Backend Total Resource Consumption

Mostra uma visão agregada do consumo de recursos do serviço backend.

- **backend_cpu_usage %:** percentagem de CPU usada pelo backend. O valor baixo (~2.3%) indica que o serviço está longe de saturar o CPU.
- **backend_memory_usage %:** percentagem de memória utilizada. Um valor perto de 48% sugere uso moderado e estável, sem pressão imediata de memória.

Backend CPU Resource Consumption

Evolução temporal do uso de CPU.

- Permite observar tendências, picos e quedas. O gráfico mostra flutuações suaves e uma ligeira descida ao longo do tempo, o que indica carga previsível e ausência de picos anómalos.

Backend Total Memory Consumption

Evolução do consumo absoluto de memória.

- A subida progressiva indica que o backend vai alocando memória ao longo do tempo (possivelmente caches, buffers ou estruturas internas). Desde que estabilize ou seja libertada, não é necessariamente um problema; se crescer indefinidamente, pode indicar memory leak.

Backend per Pod Resource Consumption

Consumo de recursos por pod individual.

- **CPU por pod:** confirma que cada instância consome pouco CPU.
- **Memória por pod:** semelhante ao total, mas isolado por pod, útil para decisões de autoscaling e resource limits no Kubernetes.

Container Network

Tráfego de rede do container.

- **receive_bytes_total** e **transmit_bytes_total** mostram bytes recebidos e enviados.
- As linhas estáveis indicam tráfego contínuo e previsível. Quedas bruscas podem corresponder a redução de carga ou interrupções no fluxo de dados.

HTTP Latency

Latência das requisições HTTP (percentil 95).

- Mede o tempo de resposta percebido pela maioria dos clientes.
- O valor baixo e estável indica que o backend responde rapidamente e cumpre facilmente requisitos de latência (bem abaixo dos 500 ms definidos no projeto).

Messages Consumed

Número de mensagens consumidas (ex.: de um broker).

- Mostra o ritmo de processamento por rota ou consumidor.
- A redução gradual até zero sugere fim da simulação, paragem de produtores ou alteração no fluxo de mensagens.

Request Status

Taxa de respostas HTTP por código de estado (2xx, 4xx, 5xx).

- A dominância de respostas bem-sucedidas (2xx) e ausência de erros indica comportamento saudável do serviço.

Connections and Disconnections

Monitoriza eventos de ligação e desligamento.

- Útil para detetar instabilidade, reconexões frequentes ou problemas de rede/cliente.
- O gráfico vazio ou estável sugere ausência de churn significativo.

Testes

Manuais (Docker Compose):

- docker compose up --build
- Validar frontend em http://localhost:8080
- Abrir WebSocket ws://localhost:8000/ws?race=race-trail_route_1 (ou all) e confirmar eventos no mapa
- Trocar corrida no seletor e verificar reconexão a múltiplos sockets

Lacunas: adicionar testes unitários para parsing de mensagens, integração com RabbitMQ em container e ciclo WebSocket (conectar, enviar, desconectar). Integrar estes testes na pipeline CI antes do build.

Resultados

Funcionalidade Implementada

- Leitura e parsing de ficheiros GPX (4 trails);
- Publicação de eventos em exchanges RabbitMQ fanout;
- Consumo assíncrono no backend e broadcast via WebSocket;
- Visualização em tempo real no mapa + leaderboard;
- Suporte a múltiplas corridas (race ou all);
- Containerização (Docker) e orquestração local (Docker Compose);
- Helm chart e pipeline GitHub Actions para build/push de imagens;
- Exposição de métricas Prometheus no backend;
- Instrumentação Prometheus com dashboards em Grafana;
- Autoscaling configurado em Kubernetes.

Performance Observada

Medições formais ainda não foram realizadas. Em testes manuais via Docker Compose, a atualização no mapa é percebida em tempo real (<1s).

Aprendizagens e Desafios Superados

Desafios Identificados e Resolvidos:

1. **Bootstrap do broker:**
   - Problema: Serviços poderiam subir antes de RabbitMQ
   - Solução: wait_for_rabbitmq() no simulador e backend com retries
2. **Descoberta dinâmica de rotas:**
   - Problema: Novas corridas (exchanges) não chegavam ao backend
   - Solução: Varredura periódica (10s) da API de gestão para bind automático
3. **Broadcast WebSocket:**
   - Problema: Diferentes corridas e modo all

- Solução: Uma WebSocket por corrida, mais uma para all; frontend gere várias ligações

4. **Extração de rota GPX:**
   - Problema: Algumas bibliotecas não expunham pontos diretamente
   - Solução: Estratégia híbrida (toGeoJSON, getLatLngs, acesso a _layers)

Validação de Requisitos

Requisito	**Status**	**Evidência**
Arquitetura Distribuída	Validado	RabbitMQ + serviços desacoplados
Streaming em Tempo Real	Validado	WebSocket /ws?race= funcional
Escalabilidade	Validado	Autoscaling
Containerização	Validado	Dockerfiles e compose funcionais
Orquestração	Validado	Docker Compose + Kubernetes/Helm
CI/CD	Validado	Build/push automatizado

Conclusão e Trabalho Futuro

Conclusões

Este projeto demonstrou a implementação de um sistema distribuído completo, do produtor ao consumidor WebSocket, suportado por RabbitMQ. A escolha de exchanges fanout simplificou o broadcast em tempo real e o desacoplamento entre produtor e clientes.

A pipeline automatiza o build/push das imagens e mantém o chart Helm atualizado, reforçando reprodutibilidade dos deployments. A containerização com Docker e a orquestração via Docker Compose/Helm facilitam a execução local e em cluster.

Testes formais ainda são um ponto em aberto; a verificação atual foi manual. A latência percebida é sub-segundo em ambiente local, mas precisa ser medida e registada.

Reflexão Técnica

O projeto contribuiu para o entendimento prático de:

- Padrões de Comunicação Assíncrona: Vantagens e limitações de message brokers
- Operacionalização de Sistemas: Importância de logging, monitorização e resilência
- Infraestrutura como Código: Vantagens de manifestos reproduzíveis
- Qualidade de Software: Benefícios de automatização e testes contínuos

Limitações Atuais

1. Segurança: Sem autenticação/autorização ou criptografia entre componentes
2. Estado Persistente: Sem base de dados para análise histórica
3. Tratamento de Falhas: Sem circuit breakers ou timeout granular
4. Documentação API: OpenAPI/Swagger não implementado

Trabalho Futuro

Curto Prazo (1-2 sprints):

Testes Automatizado:

- Testes unitários para parsing de mensagens RabbitMQ;
- Testes de integração com RabbitMQ em container;
- Testes de ciclo WebSocket (connect, send, disconnect);
- Integrar testes na pipeline CI/CD antes do build.

Testes Abrangentes:

- Testes de carga (JMeter, Locust);
- Testes de caos (chaos engineering);
- Cobertura de código a 90%+.

Autenticação WebSocket:

- Token JWT para conexões;
- Rate limiting por cliente;
- Auditoria de conexões.

Médio Prazo (1-2 meses):

Persistência de Dados:

- PostgreSQL para armazenamento de eventos;
- API REST para consulta de histórico;
- Análise pós-evento (velocidade média, elevação, etc.).

Escalabilidade e Alta Disponibilidade:

- HPA (Horizontal Pod Autoscaler) para backend baseado em métricas de conexões WebSocket;
- Cluster/replicação de RabbitMQ para HA;

- Load balancing e service mesh (Istio) para distribuição de tráfego;
- Múltiplas filas por race para paralelismo.

Resiliência Aprimorada:

- Circuit breaker para falhas de RabbitMQ;
- Retry policies com exponential backoff;
- Graceful degradation quando backend cai;
- Health checks e readiness probes.

Enriquecimento de Métricas:

- Alertas Prometheus para SLO (latência p99 < 500ms, uptime > 99%);
- Correlação com eventos (número de atletas vs. Latência);
- Análise de padrões de carga por corrida.

Longo Prazo:

Machine Learning:

- Predição de performance de atletas
- Detecção de anomalias em telemetria
- Otimização de rotas

Expansão de Funcionalidades:

- Suporte a múltiplas corridas simultâneas
- Sistema de notificações push
- Análise comparativa entre corredores
- Integração com smartwatches e dispositivos IoT

Nota sobre Autoria e Assistência de IA

Este relatório foi desenvolvido com assistência do modelo Claude Haiku 4.5. O assistente de IA foi utilizado para:

- Estruturação e organização do documento de acordo com os requisitos especificados;
- Rascunho de secções técnicas baseadas na análise do código-fonte do projeto;
- Revisão e atualização de conteúdo para refletir a arquitetura real implementada (RabbitMQ vs. especulação inicial sobre Kafka);
- Formulação de recomendações técnicas e roadmap futuro;

Responsabilidade do autor: Todas as decisões arquiteturais, escolhas de implementação e validação técnica do projeto são da responsabilidade do autor. O conteúdo técnico foi revisto e validado contra o código-fonte e documentação do projeto.