

Relatório: Sistema Distribuído de Monitoramento de Fábrica

Disciplina: INF1304 - Sistemas Distribuídos
Período: 2025.2
Nome: Miguel Brandt **Matrícula:** 2311799 **Data de Entrega:** 02 de Outubro de 2025

1. Introdução

1.1 Contextualização

Este trabalho apresenta a implementação de um sistema distribuído de monitoramento de sensores para ambientes industriais inteligentes (Smart Factories). O sistema foi desenvolvido utilizando tecnologias modernas de processamento de streams de dados, containerização e arquitetura de microserviços, demonstrando conceitos fundamentais de sistemas distribuídos.

1.2 Objetivos

Os principais objetivos deste projeto são:

- 1. **Implementar um pipeline de processamento de dados em tempo real** utilizando Apache Kafka como backbone de mensageria
- 2. **Demonstrar padrões de sistemas distribuídos** incluindo balanceamento de carga, replicação e tolerância a falhas
- 3. **Desenvolver uma arquitetura escalável** baseada em microserviços containerizados
- 4. **Implementar mecanismos de observabilidade** para monitoramento e debugging do sistema distribuído
- 5. **Validar a resiliência do sistema** através de testes de falhas e recuperação

1.3 Escopo do Projeto

O sistema simula um ambiente de fábrica com múltiplos sensores (temperatura, vibração, energia) que geram leituras periódicas. Estes dados são processados por um cluster Kafka com 3 brokers, consumidos por múltiplos processadores para detecção de anomalias, persistidos em banco de dados PostgreSQL, e visualizados através de um dashboard web em tempo real.

1.4 Requisitos e Instalação

1.4.1 Pré-requisitos

Para executar este sistema, são necessários os seguintes componentes instalados:

```
# Verificar instalações
docker --version      # Docker 20.10+
make --version         # GNU Make 4.0+
```

1.4.2 Instalação e Execução

O sistema pode ser instalado e executado com os seguintes comandos:

```
# Clonar repositório
git clone https://github.com/bathwaterpizza/inf1304-t1.git
cd (...)

# Inicializar e construir sistema completo
make all
```

Após a inicialização, o sistema estará disponível em:

- **Dashboard de Monitoramento:** <http://localhost:5000>
- **Kafka UI:** <http://localhost:8080>

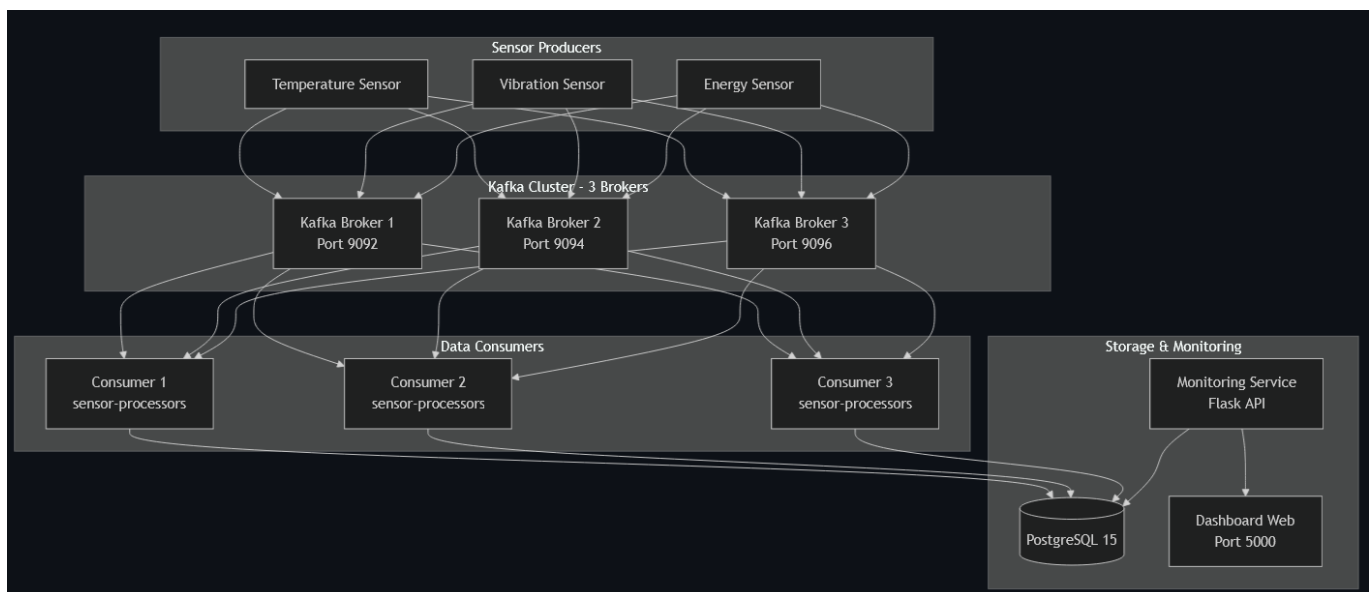
2. Arquitetura do Sistema

2.1 Visão Geral

O sistema implementa uma arquitetura de microserviços distribuídos composta por:

1. **Camada de Produção:** 3 produtores de sensores simulados
2. **Camada de Mensageria:** Cluster Kafka com 3 brokers
3. **Camada de Processamento:** 3 consumidores para processamento paralelo
4. **Camada de Persistência:** Banco de dados PostgreSQL
5. **Camada de Monitoramento:** Serviço de monitoramento com dashboard web

2.2 Diagrama de Arquitetura



2.3 Componentes Detalhados

2.3.1 Produtores de Sensores

Implementação: `src/producers/sensor_producer.py`

Produtor unificado configurável por variáveis de ambiente que simula diferentes tipos de sensores:

- **Tipos suportados:** temperature, vibration, energy
- **Geração de dados:** Valores baseados em distribuições realísticas com ruído
- **Simulação de anomalias:** Injeção probabilística de valores críticos
- **Configuração por ambiente:** Flexibilidade para criar múltiplas instâncias
- **Health monitoring:** Reporta status e métricas para o sistema de monitoramento

Características técnicas:

- Intervalo de amostragem: **3-10** segundos (configurável)
- Formato de mensagem: JSON estruturado
- Particionamento: Round-robin para distribuição uniforme
- Configuração de producer: `acks=all`, `retries=3`
- Compressão: Snappy para redução de payload

2.3.2 Cluster Kafka

Configuração:

- 3 brokers em modo KRaft
- Tópico **sensor-data**: 3 partições, replication factor 3
- Tópico **alerts**: 2 partições, replication factor 3

Garantias de entrega:

- Producers: **acks=all** (confirmação de todas as réplicas)
- Consumers: **enable.auto.commit=true** com offset tracking
- Replicação: Min In-Sync Replicas (ISR) configurado

Alta disponibilidade:

- Sistema tolerante a até 2 falhas simultâneas de 3 brokers
- Eleição automática de líder de partição
- Rebalanceamento automático de carga

2.3.3 Consumidores de Dados

Implementação: `src/consumers/sensor_consumer.py`

Consumidor unificado que processa todos os tipos de sensores:

- **Consumer group:** **sensor-processors** para balanceamento automático
- **Processamento:** Detecção de anomalias baseada em thresholds
- **Geração de alertas:** Publicação de alertas no tópico dedicado
- **Persistência:** Armazenamento de leituras e alertas no PostgreSQL
- **Rebalanceamento:** Callbacks para tracking de atribuição de partições

Características técnicas:

- Session timeout: 30s
- Heartbeat interval: 10s
- Auto commit interval: 5s
- Processamento por lote otimizado
- Tratamento robusto de exceções

2.3.4 Banco de Dados PostgreSQL

Schema principal:

- sensor_readings: Todas as leituras processadas
- alerts: Alertas gerados por anomalias
- consumer_health: Estado e métricas dos consumidores
- producer_health: Estado e métricas dos produtores
- rebalancing_events: Histórico de rebalanceamentos

2.3.5 Sistema de Monitoramento

Backend: Flask API REST (`src/monitoring/monitoring_service.py`)

Endpoints principais:

- `/api/system-status`: Status geral do sistema
- `/api/consumer-health`: Saúde e atribuições dos consumidores
- `/api/producer-health`: Saúde e throughput dos produtores
- `/api/partition-assignment`: Distribuição de partições
- `/api/real-time-metrics`: Métricas de throughput
- `/api/recent-alerts`: Alertas e eventos recentes

Frontend: Dashboard HTML5 responsivo com:

- Auto-refresh a cada 5 segundos
- Visualização de status em tempo real
- Indicadores de saúde dos componentes
- Tracking de rebalanceamentos
- Histórico de alertas

2.4 Fluxo de Dados

1. **Geração:** Sensores geram leituras a cada 3-10 segundos
2. **Publicação:** Dados enviados para tópico Kafka `sensor-data` com round-robin
3. **Distribuição:** Kafka distribui mensagens entre partições
4. **Consumo:** Consumer group processa mensagens em paralelo
5. **Deteção:** Algoritmo identifica anomalias baseado em thresholds
6. **Alertas:** Anomalias publicadas no tópico `alerts`

- 7. **Persistência:** Dados e alertas armazenados no PostgreSQL
- 8. **Visualização:** Dashboard consulta banco e exibe métricas em tempo real

3. Implementação

3.1 Tecnologias Utilizadas

Componente	Tecnologia	Versão	Justificativa
Message Broker	Apache Kafka	3.8+	Líder de mercado, alta performance, KRaft mode
Linguagem	Python	3.11+	Ecossistema rico, bibliotecas Kafka maduras
Kafka Client	confluent-kafka-python	Latest	Performance superior ao kafka-python
Banco de Dados	PostgreSQL	15	Confiabilidade, JSONB para flexibilidade
Web Framework	Flask	Latest	Simplicidade para APIs REST
Containerização	Docker	20.10+	Padrão da indústria, com Compose V2 integrado
Orquestração	Docker Compose V2	Plugin	Integrado ao Docker CLI, sintaxe moderna

3.2 Estrutura de Diretórios

```
.
├── src/
│   ├── producers/
│   │   ├── __init__.py
│   │   └── sensor_producer.py      # Producer unificado
│   ├── consumers/
│   │   ├── __init__.py
│   │   └── sensor_consumer.py     # Consumer unificado
│   └── monitoring/
│       ├── __init__.py
│       ├── monitoring_service.py  # Backend Flask
│       └── templates/
│           └── dashboard.html     # Frontend web
├── docker/
│   ├── Dockerfile.producer      # Imagem dos sensores
│   ├── Dockerfile.consumer      # Imagem dos consumidores
│   └── Dockerfile.monitoring    # Imagem do monitoring
├── config/
│   └── init.sql                 # Schema do PostgreSQL
├── docker-compose.yml           # Orquestração completa
├── Makefile                     # Automação de comandos
├── requirements.txt             # Dependências Python
└── .env                         # Configurações
```

└─ README.md	# Documentação GitHub
└─ RELATORIO.md	# Este documento

3.3 Configuração e Deployment

3.3.1 Variáveis de Ambiente

Todas as configurações importantes são externalizadas via variáveis de ambiente:

```
# Environment variables for the factory monitoring system
# Database Configuration
POSTGRES_DB=factory_monitoring
POSTGRES_USER=factory_user
POSTGRES_PASSWORD=factory_pass
DATABASE_URL=postgresql://factory_user:factory_pass@postgres:5432/factory_
monitoring

# Kafka Configuration
KAFKA_CLUSTER_ID=MkU30EVbNTcwNTJENDM2Qk
KAFKA_BROKERS=kafka1:29092,kafka2:29092,kafka3:29092
KAFKA_EXTERNAL_BROKERS=localhost:9092,localhost:9094,localhost:9096
SENSOR_TOPIC=sensor-data
ALERT_TOPIC=alerts
CONSUMER_GROUP=sensor-processors

# Application Configuration
LOG_LEVEL=INFO
SENSOR_INTERVAL_SECONDS=5
TEMPERATURE_WARNING_THRESHOLD=35.0
TEMPERATURE_CRITICAL_THRESHOLD=40.0
VIBRATION_WARNING_THRESHOLD=5.0
VIBRATION_CRITICAL_THRESHOLD=7.0
ENERGY_ANOMALY_THRESHOLD=20.0

# Development Configuration
DEBUG=false
ENABLE_METRICS=true
```

3.3.2 Comandos de Operação

Sistema gerenciado através de Makefile para automação:

```
make all          # Build e start completo
make start        # Infraestrutura apenas
make with-monitoring # Sistema completo
make stop         # Parar todos os serviços
make clean        # Limpar containers e volumes
make logs-consumers # Ver logs dos consumidores
make health       # Verificar saúde dos serviços
```

3.4 Aspectos de Implementação

3.4.1 Tratamento de Erros

O sistema implementa múltiplas estratégias para lidar com falhas:

- **Retry policies:** Configuração `retries=3` nos produtores e consumidores Kafka para retentativas automáticas em caso de falhas transitórias
- **Exception handling:** Blocos try-catch abrangentes com logging detalhado, permitindo que componentes continuem operando mesmo após erros pontuais
- **Graceful degradation:** Componentes não falham completamente - registram erros e continuam processando outras mensagens
- **Timeouts configurados:** Session timeout (30s) e heartbeat interval (10s) para detecção rápida de falhas
- **Error logging:** Logs estruturados com níveis (INFO, WARNING, ERROR) para debugging e análise post-mortem

3.4.2 Performance

- **Batching:** Mensagens processadas em lotes quando possível
- **Compression:** Snappy compression nos producers
- **Connection pooling:** Pool de conexões para database
- **Async I/O:** Operações assíncronas onde aplicável

3.4.3 Monitoramento e Observabilidade

- **Health checks:** Endpoints HTTP para verificação de saúde
- **Heartbeats:** Produtores e consumidores reportam status periodicamente
- **Metrics:** Contadores de mensagens, throughput, latência
- **Event tracking:** Rebalanceamentos e eventos do sistema registrados

4. Simulações de falha

4.1 Cenários de Teste

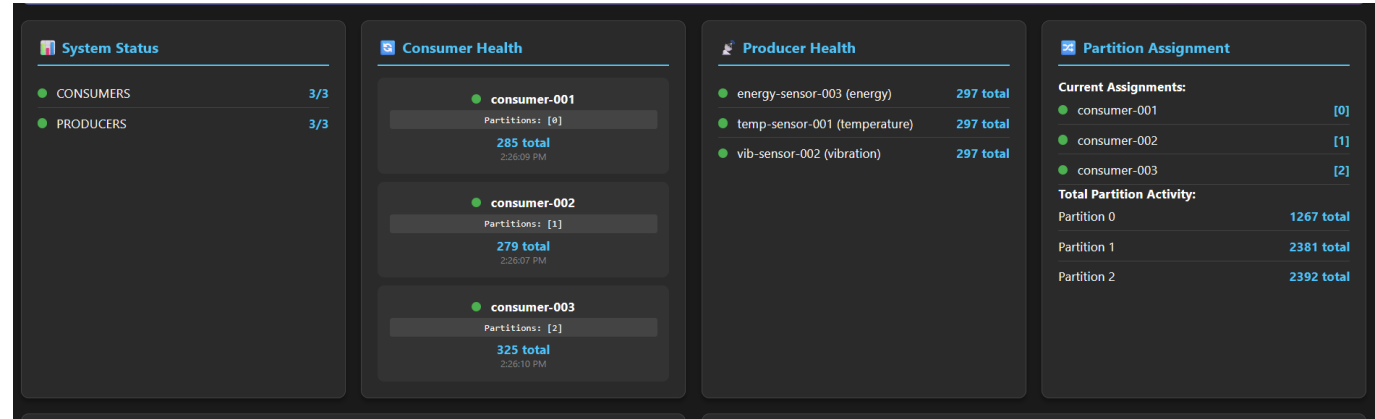
4.1.1 Consumers falharam

Ao derrubar dois dos três containers de consumidores, podemos observar através da interface de monitoramento que o sistema continua funcionando, com todos os dados dos sensores produtores sendo processados normalmente. O consumidor restante é automaticamente designado para consumir todas partições do cluster pelo group coordinator.

Colocando sistema completo em execução:

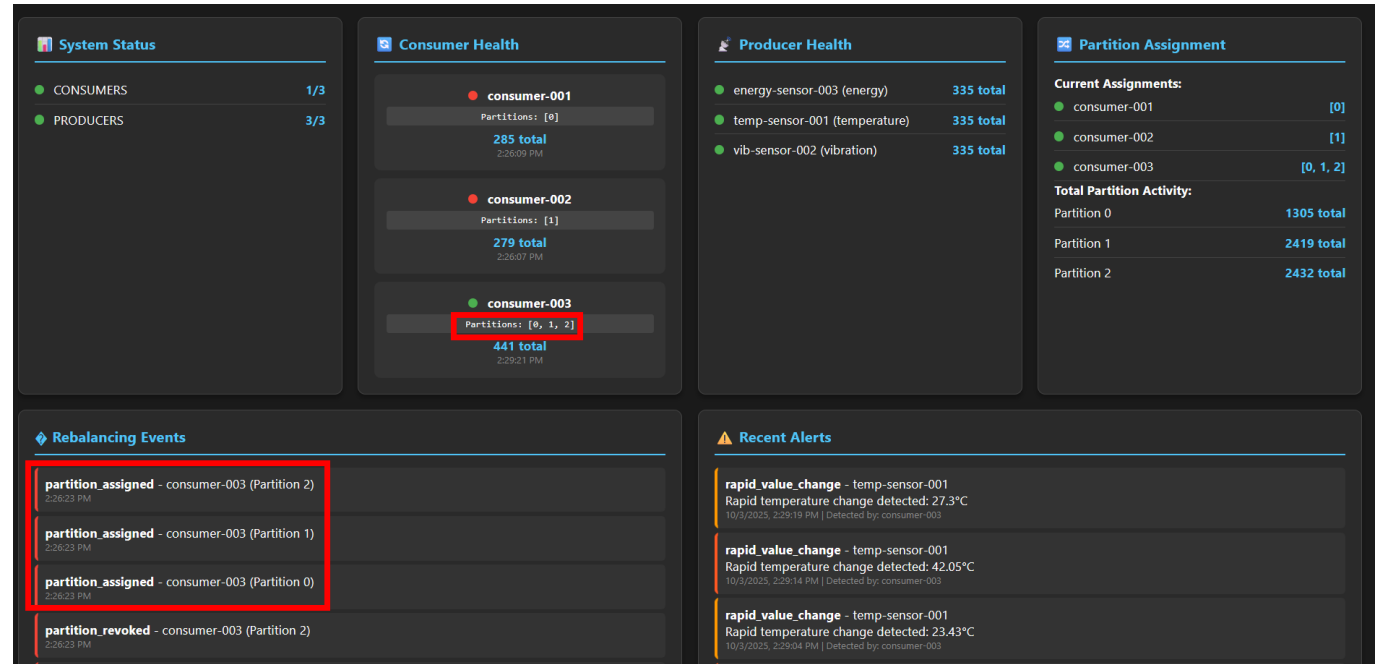
```
make all
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
dd010978947e	ti-monitoring	"python -m src.monit..."	2 minutes ago	Up 2 minutes (healthy)	0.0.0.0:5000→5000/tcp, [::]:5000→5000/tcp	monitoring
95cd98eeca48	ti-consumer-2	"python src/consumer..."	11 minutes ago	Up 8 minutes		consumer-2
ce879a48ca81	ti-consumer-1	"python src/consumer..."	11 minutes ago	Up 8 minutes		consumer-1
t1b7f9480e3	ti-energy-sensor	"python src/producer..."	11 minutes ago	Up 8 minutes		energy-sensor
fa94094e15af	ti-vibration-sensor	"python src/producer..."	11 minutes ago	Up 8 minutes		vibration-sensor
897ed120e3bd	ti-temperature-sensor	"python src/producer..."	11 minutes ago	Up 8 minutes		temperature-sensor
6eac1cacaeb9	ti-consumer-3	"python src/consumer..."	11 minutes ago	Up 8 minutes		consumer-3
ca84798b785a	provectuslabs/kafka-ui:latest	"/bin/sh -c 'java --..."	11 minutes ago	Up 8 minutes	0.0.0.0:8080→8080/tcp, [::]:8080→8080/tcp	kafka-ui
d91b37ceefb2	confluentinc/cp-kafka:7.5.0	"/etc/confluent/dock..."	11 minutes ago	Up 8 minutes (healthy)	9092/tcp, 0.0.0.0:9096-9097→9096-9097/tcp, [::]:9096-9097→9096-9097/tcp	kafka3
bab9d7b2bf1	confluentinc/cp-kafka:7.5.0	"/etc/confluent/dock..."	11 minutes ago	Up 8 minutes (healthy)	9092/tcp, 0.0.0.0:9094-9095→9094-9095/tcp, [::]:9094-9095→9094-9095/tcp	kafka2
0b60252e517c	postgres:15-alpine	"docker-entrypoint.s..."	11 minutes ago	Up 8 minutes (healthy)	0.0.0.0:5432→5432/tcp, [::]:5432→5432/tcp	postgres
43b75af01c5c	confluentinc/cp-kafka:7.5.0	"/etc/confluent/dock..."	11 minutes ago	Up 8 minutes (healthy)	0.0.0.0:9092-9093→9092-9093/tcp, [::]:9092-9093→9092-9093/tcp	kafka1



Derrubando dois consumers:

```
docker kill consumer-1 consumer-2
```



4.1.2 Brokers falharam

Similarmente, ao derrubar um dos três containers de kafka broker (não podemos derrubar dois pois o kafka precisa de consenso da metade para eleição de líder), podemos observar através da interface de monitoramento, ou do Kafka UI, que o sistema continua funcionando com o mesmo throughput. Os brokers restante automaticamente se organizam para assumir a partição do broker que caiu.

Restaurando consumers:


```
docker start consumer-1 consumer-2
```

Topics / **sensor-data**

Overview Messages Consumers Settings Statistics

Partitions
3

Replication Factor
3

URP •
0

In Sync Replicas •
9 of 9

Partition ID	Replicas	First Offset
0	2, 3, 1	0
1	3, 1, 2	0
2	1, 2, 3	0

Derrubando um broker:

```
docker kill kafka1
```

Topics / **sensor-data**

Overview Messages Consumers Settings Statistics

Partitions
3

Replication Factor
3

URP •
3

In Sync Replicas •
6 of 9

Partition ID	Replicas	First Offset
0	2, 3, 1	0
1	3, 1, 2	0
2	1, 2, 3	0