

Lab Report 7: Apache Kafka

Distributed Streaming IoT Pipeline

Student: Mohammed Seddik Lifa

Specialization: Master II: AI & Data Science

Date: November 23, 2025

1 Objective

The primary objective of this lab is to install and configure **Apache Kafka** as a distributed streaming platform on a Linux environment. The lab involves:

- Installing Kafka and Zookeeper (v3.9.1).
- Creating and managing Topics.
- Implementing the **Producer-Consumer** messaging pattern.
- Configuring a multi-broker cluster (3 nodes).
- Developing a Python-based **IoT Simulation** to stream and monitor sensor data for failure detection.

2 Implementation & Proof of Execution

2.1 Kafka Setup & Topic Creation

Instruction: Start the Zookeeper and Kafka server, then create a topic named `test-topic` with a single partition and replication factor.

Code Pattern Executed:

```
bin/kafka-topics.sh --create \  
  --topic test-topic \  
  --bootstrap-server localhost:9092 \  
  --partitions 1 \  
  --replication-factor 1
```

Execution Proof: The topic was successfully created. The screenshot below verifies the creation command execution.

A terminal window titled 'Tilix: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1'. The terminal shows the execution of the 'bin/kafka-topics.sh --create' command with the following flags: '--topic test-topic', '--bootstrap-server localhost:9092', '--partitions 1', and '--replication-factor 1'. The output of the command is 'Created topic test-topic.'. The prompt then changes to '~/.Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 main !3 ?6'.

```
1: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1
> bin/kafka-topics.sh --create \
  --topic test-topic \
  --bootstrap-server localhost:9092 \
  --partitions 1 \
  --replication-factor 1
Created topic test-topic.
~/.Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 main !3 ?6
>
```

Figure 1: Creating the 'test-topic' in Kafka

2.2 Producer & Consumer Messaging Pattern

Instruction: detailed usage of the Console Producer to send messages and the Console Consumer to read them in real-time.

Code Pattern Executed:

```
# Producer
bin/kafka-console-producer.sh --topic test-topic --bootstrap-server
localhost:9092

# Consumer
bin/kafka-console-consumer.sh --topic test-topic --from-beginning
--bootstrap-server localhost:9092
```

Execution Proof: Messages typed in the producer terminal appear instantly in the consumer terminal, demonstrating the streaming pipeline.

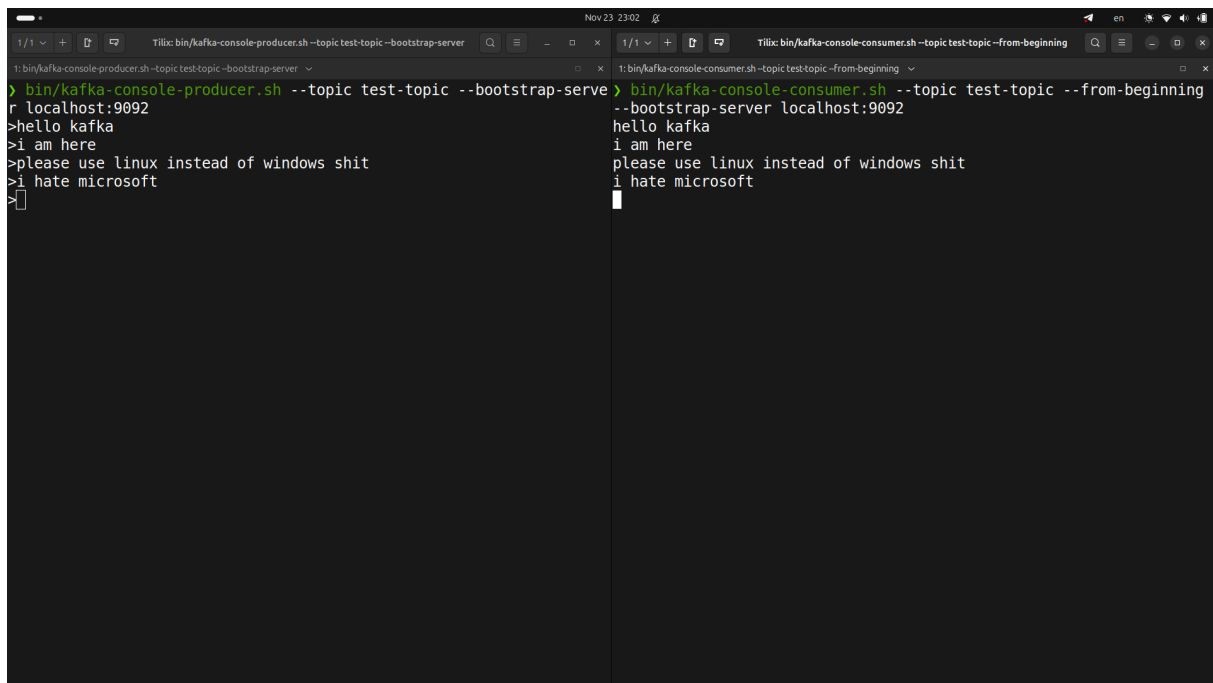
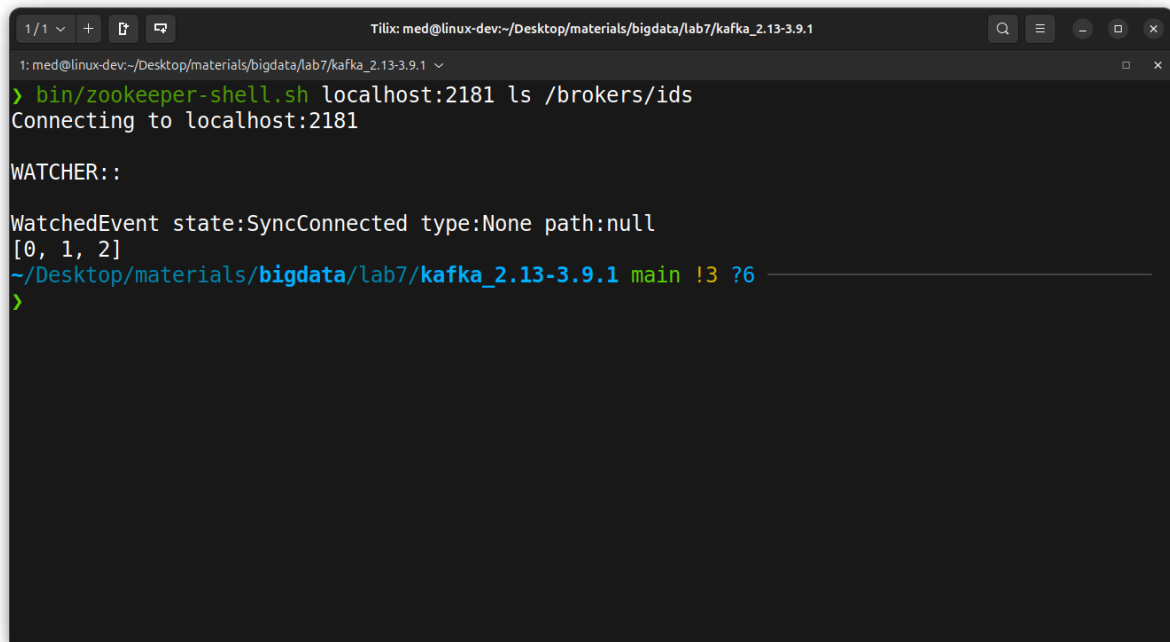


Figure 2: Real-time messaging pipeline (Producer vs Consumer)

2.3 Multi-Broker Cluster Configuration

Instruction: Simulate a distributed cluster by configuring three brokers (IDs: 0, 1, 2) on different ports (9092, 9093, 9094) and verifying them via Zookeeper.

Execution Proof: The Zookeeper shell confirms that all three broker IDs are registered and active in the cluster.

A terminal window titled 'Tilix: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1'. The prompt is '1: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 ~'. The user enters the command 'bin/zookeeper-shell.sh localhost:2181 ls /brokers/ids'. The terminal shows 'Connecting to localhost:2181' and then 'WATCHER::'. Below that, it displays 'WatchedEvent state:SyncConnected type:None path:null' and '[0, 1, 2]'. The prompt returns to '~/.Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 main !3 ?6' with a green cursor.

```
1: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 ~  
> bin/zookeeper-shell.sh localhost:2181 ls /brokers/ids  
Connecting to localhost:2181  
WATCHER::  
WatchedEvent state:SyncConnected type:None path:null  
[0, 1, 2]  
~/.Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1 main !3 ?6  
>
```

Figure 3: Verification of 3-Node Cluster [0, 1, 2] in Zookeeper

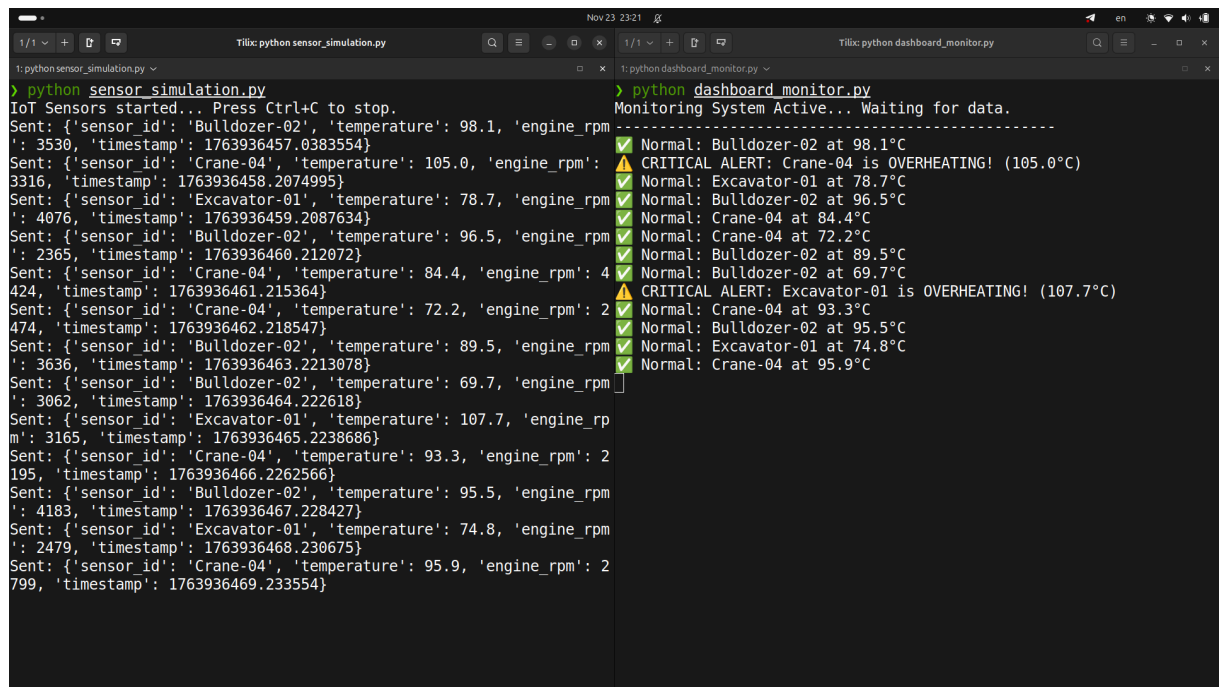
2.4 Project: IoT Sensor Simulation & Failure Detection

Instruction: Create a Python system where simulated heavy machinery (Bulldozers, Cranes) sends temperature data via Kafka. A monitor script must detect overheating (Temp > 100°C).

Code Logic (Python Consumer):

```
if data['temperature'] > 100:
    print(f"CRITICAL ALERT: {data['sensor_id']} is OVERHEATING!")
else:
    print(f"Normal: {data['sensor_id']} at {data['temperature']}")
```

Execution Proof: The Producer script streams random sensor data. The Consumer script successfully flags "Crane-04" and "Excavator-01" as **CRITICAL ALERT** when simulated temperatures exceed the threshold.



The image shows two terminal windows side-by-side. The left window, titled 'Tilix: python_sensor_simulation.py', shows the output of a Python script that generates sensor data. It lists various sensors like 'Bulldozer-02', 'Crane-04', and 'Excavator-01' along with their temperature and engine RPM values. The right window, titled 'Tilix: python_dashboard_monitor.py', shows the output of a monitoring script. It displays a series of status messages for each sensor, indicating whether the temperature is 'Normal' or 'CRITICAL ALERT'. For example, it shows 'Normal: Bulldozer-02 at 98.1°C' and 'CRITICAL ALERT: Crane-04 is OVERHEATING! (105.0°C)'. The status messages are preceded by green checkmarks for normal and yellow triangles for critical alerts.

Figure 4: IoT Simulation: Sensor Data Stream and Overheat Detection