

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

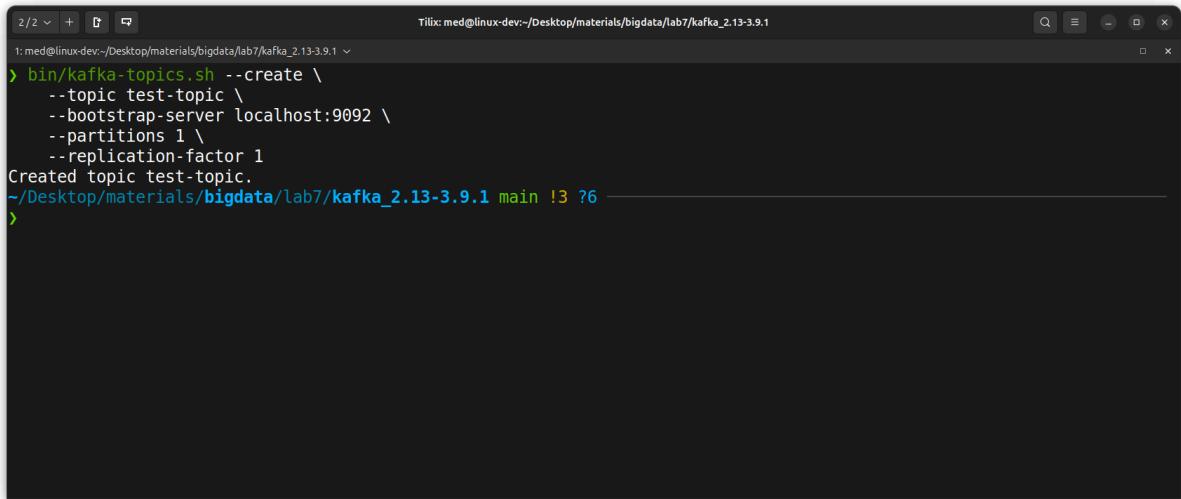
2.1 Kafka Setup & Topic Creation

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.



```
2/2 + 1 Tilix: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1
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

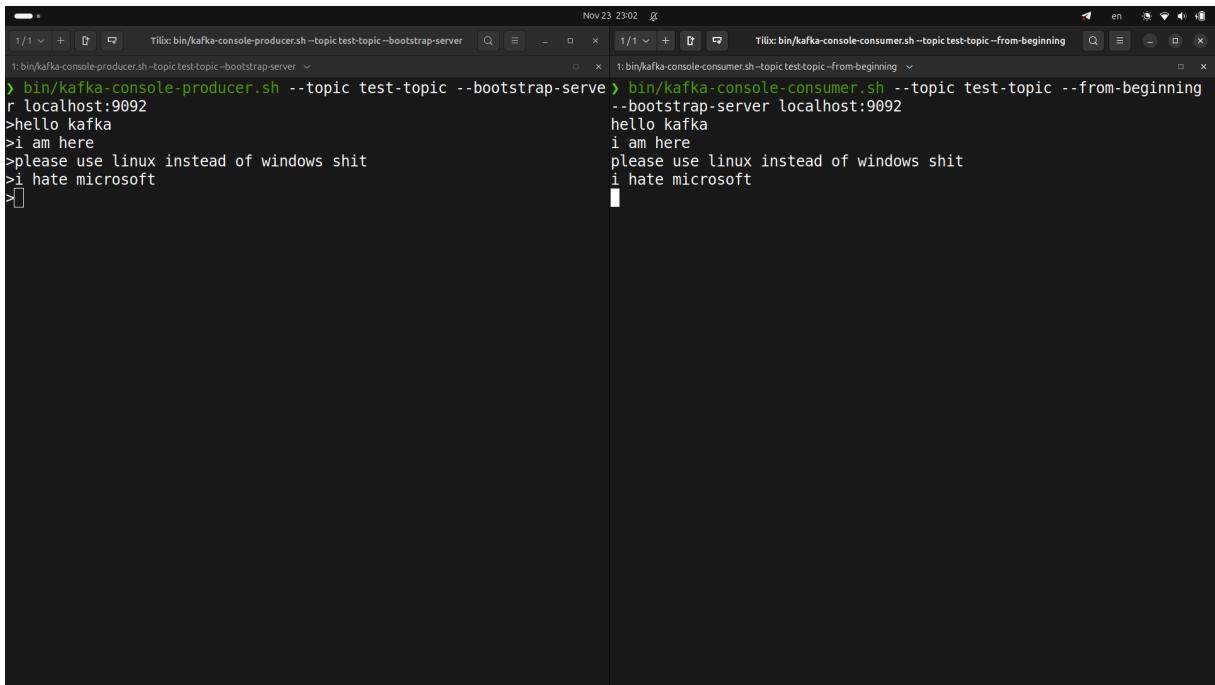
Instruction: detailed usage of the Console Producer to send messages and the Console Consumer to read them in real-time.

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.



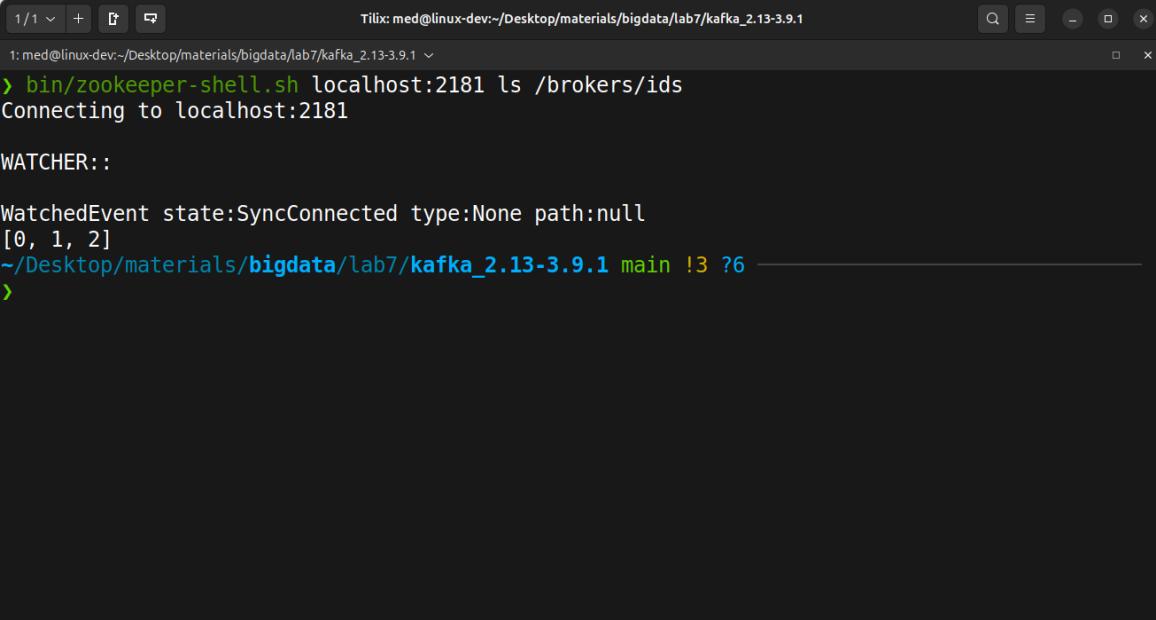
The screenshot shows two terminal windows side-by-side. The left window is titled 'Tilix: bin/kafka-console-producer.sh --topic test-topic --bootstrap-server' and contains the command 'bin/kafka-console-producer.sh --topic test-topic --bootstrap-server localhost:9092'. Below the command, several messages are being typed: '>hello kafka', '>i am here', '>please use linux instead of windows shit', and '>i hate microsoft'. The right window is titled 'Tilix: bin/kafka-console-consumer.sh --topic test-topic --from-beginning' and contains the command 'bin/kafka-console-consumer.sh --topic test-topic --from-beginning --bootstrap-server localhost:9092'. It immediately displays the same four messages as the producer window, showing they appear instantaneously in the consumer window.

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.



The screenshot shows a terminal window titled "Tilix: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1". The command entered is "bin/zookeeper-shell.sh localhost:2181 ls /brokers/ids". The output indicates that the cluster is connecting to localhost:2181 and shows the brokers' IDs: [0, 1, 2]. The path "/brokers/ids" is also listed.

```
1/1 + Tilix: med@linux-dev:~/Desktop/materials/bigdata/lab7/kafka_2.13-3.9.1
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 screenshot shows two terminal windows side-by-side. The left window, titled 'Tilix: python sensor_simulation.py', displays a stream of sensor data from various machinery. The right window, titled 'Tilix: python dashboard_monitor.py', shows the monitoring script processing this data and flagging specific instances of overheating.

Terminal 1 (Sensor Simulation):

```
> python sensor_simulation.py  
IoT Sensors started... Press Ctrl+C to stop.  
Sent: {'sensor_id': 'Bulldozer-02', 'temperature': 98.1, 'engine_rpm': 3530, 'timestamp': 1763936457.0383554}  
Sent: {'sensor_id': 'Crane-04', 'temperature': 105.0, 'engine_rpm': 3316, 'timestamp': 1763936458.2074995}  
Sent: {'sensor_id': 'Excavator-01', 'temperature': 78.7, 'engine_rpm': 4076, 'timestamp': 1763936459.2087634}  
Sent: {'sensor_id': 'Bulldozer-02', 'temperature': 96.5, 'engine_rpm': 2365, 'timestamp': 1763936460.212072}  
Sent: {'sensor_id': 'Crane-04', 'temperature': 84.4, 'engine_rpm': 424, 'timestamp': 1763936461.215364}  
Sent: {'sensor_id': 'Crane-04', 'temperature': 72.2, 'engine_rpm': 474, 'timestamp': 1763936462.218547}  
Sent: {'sensor_id': 'Bulldozer-02', 'temperature': 89.5, 'engine_rpm': 3636, 'timestamp': 1763936463.2213078}  
Sent: {'sensor_id': 'Bulldozer-02', 'temperature': 69.7, 'engine_rpm': 3062, 'timestamp': 1763936464.222618}  
Sent: {'sensor_id': 'Excavator-01', 'temperature': 107.7, 'engine_rpm': 3165, 'timestamp': 1763936465.2238686}  
Sent: {'sensor_id': 'Crane-04', 'temperature': 93.3, 'engine_rpm': 2195, 'timestamp': 1763936466.2262566}  
Sent: {'sensor_id': 'Bulldozer-02', 'temperature': 95.5, 'engine_rpm': 4183, 'timestamp': 1763936467.228427}  
Sent: {'sensor_id': 'Excavator-01', 'temperature': 74.8, 'engine_rpm': 2479, 'timestamp': 1763936468.230675}  
Sent: {'sensor_id': 'Crane-04', 'temperature': 95.9, 'engine_rpm': 2799, 'timestamp': 1763936469.233554}
```

Terminal 2 (Dashboard Monitor):

```
> python dashboard_monitor.py  
Monitoring System Active... Waiting for data.  
Normal: Bulldozer-02 at 98.1°C  
CRITICAL ALERT: Crane-04 is OVERHEATING! (105.0°C)  
Normal: Excavator-01 at 78.7°C  
Normal: Bulldozer-02 at 96.5°C  
Normal: Crane-04 at 84.4°C  
Normal: Crane-04 at 72.2°C  
Normal: Bulldozer-02 at 89.5°C  
Normal: Bulldozer-02 at 69.7°C  
CRITICAL ALERT: Excavator-01 is OVERHEATING! (107.7°C)  
Normal: Crane-04 at 93.3°C  
Normal: Bulldozer-02 at 95.5°C  
Normal: Excavator-01 at 74.8°C  
Normal: Crane-04 at 95.9°C
```

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