Written Report

Cloud Computing Technologies

Practical project : Kafka Cluster Configuration

Jan Wojdylak

Milan, 29.06.2023

1. Project objectives

The primary objective of this project is to successfully install and configure Kafka cluster using docker compose, while focusing on preserving key non-functional properties, such as fault tolerance and high availability. Furthermore, the project aims to enable advanced security features like channel encryption and authentication. Achieved functionalities are shown by developed java applications which play a role as consumer and producer.

2. Kafka introduction

Kafka is an open-source distributed streaming platform developed by the Apache Software Foundation. It serves as a highly scalable and fault-tolerant messaging system designed for handling real-time data streams. At its core, Kafka operates as a publish-subscribe messaging system, where producers publish messages to specific topics, and consumers subscribe to those topics to receive and process the messages. The most common kafka use cases are log aggregation and analytics, messaging and communication, Internet of Things. Kafka operates on the concept of topics, partitions, and replication to provide a scalable and fault-tolerant messaging system.

Topics - logical category to which messages are published by producers. Each topic is uniquely identified by its name, allowing producers to publish messages to the relevant topic and consumers to subscribe to and consume messages from the desired topic.

Partitions - within a Kafka topic, the data is divided into partitions. A partition is an ordered and immutable sequence of records. Each partition represents a linearly ordered log of messages. When a message is published to a topic, Kafka assigns it to a specific partition based on the message key or a partitioning strategy defined by the producer. Partitioning enables parallelism and scalability in Kafka, allowing multiple consumers to process messages concurrently.

Replication - Kafka employs replication to ensure fault tolerance and high availability. Each partition in a topic can have multiple replicas, where one replica acts as the leader and the others as followers. The leader replica handles all read and write operations for the partition, while the follower replicas replicate the data from the leader. Replication provides redundancy and allows for failover in case of leader failures. If the leader replica becomes unavailable, one of the followers is automatically elected as the new leader to ensure continuity of data availability.

3. Fault tolerance and high ability

In order to fully ensure fault tolerance and high ability, Kafka should use multiple **brokers** to store replicas. Brokers are individual instances or nodes that form the Kafka cluster. Replication ensures that even if a broker fails, the data remains available and accessible. To coordinate all this flow data between consumers, producers, and brokers, Kafka leverages **Apache ZooKeeper.** It is a distributed coordination service that acts as a centralized repository for storing metadata and coordinating the interactions between the various components of a Kafka cluster. ZooKeeper maintains a consistent view of the cluster's state and ensures reliable coordination among brokers, producers, and consumers. In the context of Kafka, ZooKeeper works closely with the **Kafka controller,** which is a specialized broker that is responsible for managing the overall state and metadata of the cluster. It monitors the health of brokers, assigns leaders to partitions, and handles administrative tasks. ZooKeeper assists the controller by providing a reliable and consistent source of metadata, such as broker and partition information. It helps maintain the stability and availability of the Kafka cluster by coordinating tasks such as leader election, partition reassignment, and cluster membership changes. The combination of ZooKeeper and the Kafka controller enables fault tolerance and high availability in Kafka. ZooKeeper ensures that the cluster remains in a consistent and coordinated state, while the controller manages the dynamic aspects of the cluster's operation.

4. Fault tolerance and HA implementation

To achieve a Kafka cluster with fault tolerance and high availability, I made the decision to create a cluster using Docker Compose with three Kafka brokers and three ZooKeeper servers.
With three Kafka brokers, the cluster can handle failures without data loss or service interruption. For a topic with three partitions and three replications, each partition is replicated across multiple brokers. Three replications means that each partition has two additional copies distributed across the brokers. If one broker fails, the other replicas of the partition on different brokers can take over, ensuring continuous availability of data. The cluster can tolerate the failure of up to two brokers while maintaining data accessibility.

To ensure fault tolerance with Zookeeper I decided to create 3 servers. It allows for a majority to be established (quorum) when at least two servers are functioning properly. This ensures that the system can tolerate the failure of one server while maintaining a consistent view of the cluster's metadata.

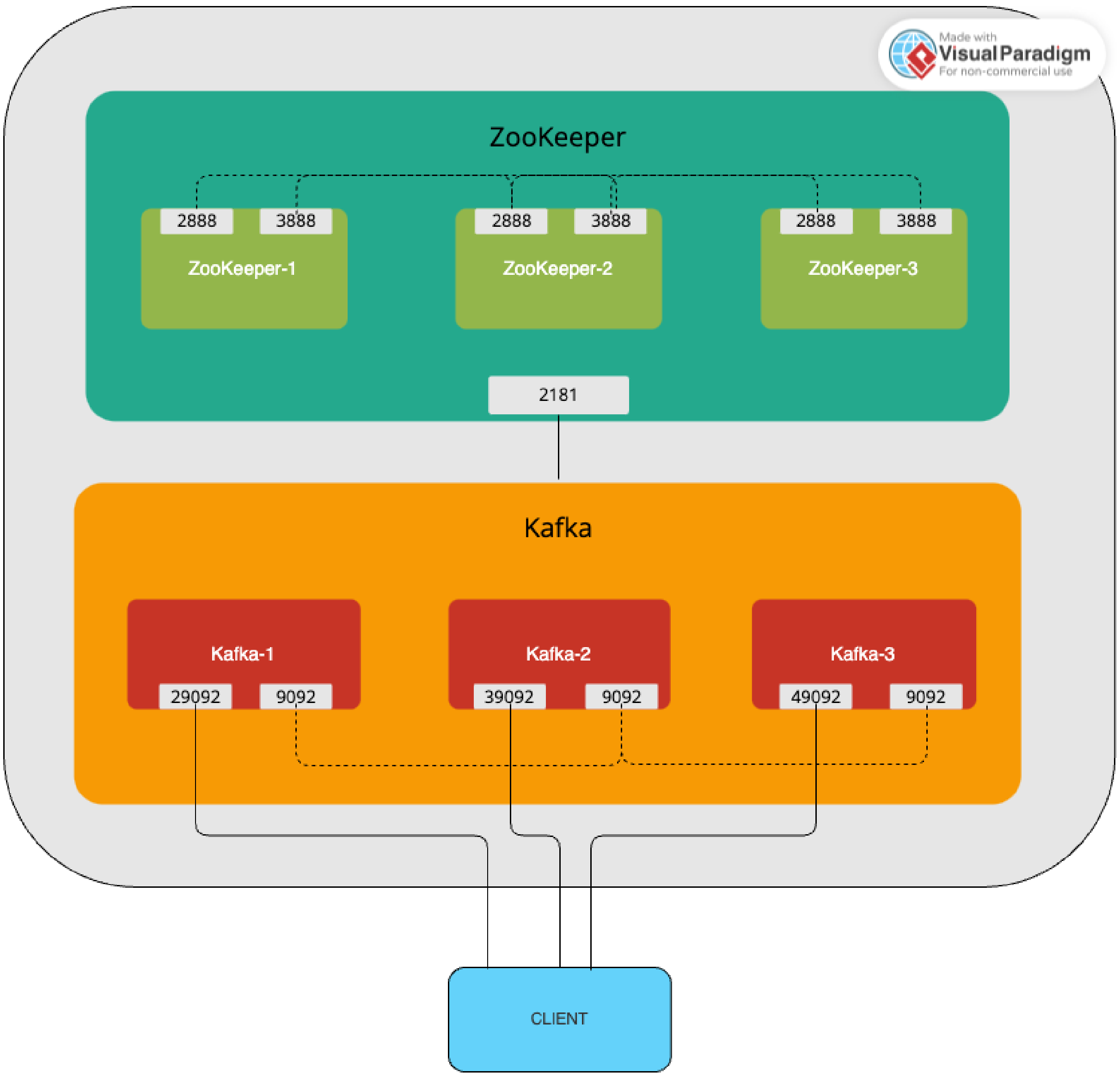


Figure 1. Kafka cluster

To define clusters on Figure 1. I used docker-compose to manage all containers

```
version: '2'
services:
  zookeeper:
    image: confluentinc/cp-zookeeper:latest
    container_name: zookeeper-1
    hostname: zookeeper-1
    environment:
      ZOOKEEPER_SERVER_ID: 1
      ZOOKEEPER_CLIENT_PORT: 2181
      ZOOKEEPER_TICK_TIME: 2000
      ZOOKEEPER_SERVERS:
      zookeeper-1:2888:3888;zookeeper-2:2888:3888;zookeeper-3:2888:3888
    ports:
      - 22181:2181

  kafka:
    image: confluentinc/cp-kafka:latest
    container_name: kafka-1
    hostname: kafka-1
    depends_on:
      - zookeeper
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT: zookeeper:2181
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka-1:9092,
      EXTERNAL://localhost:29092
    ports:
      - 29092:9092
```

Listing 1. Zookeeper server and Kafka broker configuration

On listing 1. there are sample configurations for Zookeeper and Kafka containers. In the final configuration, there are three Zookeepers servers and three Kafka brokers, but their configuration is similar to the one shown on listing 1.

Here is a breakdown of the configuration.
ZooKeeper:
Each ZooKeeper node has a unique server ID `ZOOKEEPER_SERVER_ID` and runs on a specified client port `ZOOKEEPER_CLIENT_PORT`.
The `ZOOKEEPER_TICK_TIME` parameter specifies the length of a single tick in milliseconds, which affects ZooKeeper's session timeouts and heartbeats.
The `ZOOKEEPER_SERVERS` configuration parameter is used to specify the ZooKeeper ensemble or cluster in Apache Kafka. `Zookeeper-1:2888:3888` means that the first port (2888) is used for follower communication, where the server connects to other servers in the ensemble. The second port (3888) is used for leader election, where servers coordinate and elect a leader.
`Ports: 22181:2181` means that container exposes port 2181, which is mapped to host port 22181. This allows it to connect to ZooKeeper from outside the Docker network.

Kafka:
Kafka brokers are configured to connect to the ZooKeeper ensemble `KAFKA_ZOOKEEPER_CONNECT`.
The `KAFKA_BROKER_ID` parameter assigns a unique ID to each Kafka broker.
Advertised listeners `KAFKA_ADVERTISED_LISTENERS` are defined to enable both internal and external communication with Kafka.
`Ports: 29092:9092` exposing port 9092 and mapped to host port 29092.

5. Security

Ensuring the security of data in Kafka is essential due to various reasons. Multiple consumers may need access to specific data, requiring data security from unauthorized consumers. Unwanted consumers can disrupt existing consumers, necessitating authorization security. Additionally, unauthorized users may delete topics in the cluster. Kafka security encompasses three components: **encryption**, **authentication**, and **authorization**.

- Encryption safeguards data exchange between brokers and clients, ensuring it is shared only in an encrypted format.
- Authentication allows authorized applications with unique credentials to publish or consume messages.
- Authorization controls access to prevent data pollution.

Apache Kafka offers different security models, including:

- PLAINTEXT - not recommended for high-security environments,
- SSL - providing encryption and authentication,
- SASL - enabling user authentication, it can use mechanisms like GSSAPI, PLAIN, SCRAM, OAUTHBEARER, and Delegation Tokens which offer various authentication and security capabilities.

In the project, my focus was placed on encryption and authentication using SSL. Security was configured both between the Kafka broker and the client, as well as between the ZooKeeper server and the Kafka broker. SSL was utilized to establish a secure communication channel, ensuring that data transmitted between these components remained confidential and protected against unauthenticated access.

6. Security implementation

SSL utilizes cryptographic certificates to establish secure communication channels between the Kafka brokers and clients. I decided to set up two way authentication, which means that both the Kafka broker and the client validate each other's identities using their respective SSL certificates. I generated and obtained certificates from a trusted Certification Authority (CA). To enable SSL, I created a keystore and a truststore. The keystore holds the private key and the server certificate, while the truststore contains the CA's public key and other trusted certificates.
These steps in details:

1. Generating CA

```
openssl req -new -x509 -keyout ca-key -out ca-cert -days 365
```

2. Creating Trustore

```
keytool -keystore kafka.zookeeper0.truststore.jks -alias ca-cert -import
-file ca-cert
```

3. Creating Keystore

```
keytool -keystore kafka.zookeeper0.keystore.jks -alias zookeeper0 -validity
365 -genkey -keyalg RSA -ext SAN=dns:localhost
```

4. Creating certificate signing request (CSR)

```
keytool -keystore kafka.zookeeper0.keystore.jks -alias zookeeper0 -certreq
-file ca-request-zookeeper0
```

5. Signing the CSR

```
openssl x509 -req -CA ca-cert -CAkey ca-key -in ca-request-zookeeper0 -out
ca-signed-zookeeper0 -days 365 -CAcreateserial
```

6. Importing the CA into Keystore

```
keytool -keystore kafka.zookeeper0.keystore.jks -alias ca-cert -import -file
ca-cert
```

7. Import the signed certificate into Keystore

```
keytool -keystore kafka.zookeeper0.keystore.jks -alias zookeeper0
-import -file ca-signed-zookeeper0
```

Step 1 to generate CA is done only once, steps 2-7 were done for each zookeeper server, Kafka broker and also for Kafka producer and consumer. These commands use openSSL which is a cryptographic library and toolkit used for generating and managing SSL certificates, and keytool which is a key and certificate management tool provided with the Java JDK. In order to fully implement encryption and authentication along with the use of generated keystores and trustores, I needed to create some changes in the docker-compose file.

```
 kafka-1:
    environment:
      KAFKA_BROKER_ID: 1
      KAFKA_ZOOKEEPER_CONNECT:
zookeeper-1:2181,zookeeper-2:2181,zookeeper-3:2181
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka-1:9092,
EXTERNAL://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:SSL,EXTERNAL:SSL
      KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_SSL_KEYSTORE_LOCATION:
/etc/kafka/secrets/kafka.kafka0.keystore.jks
      KAFKA_SSL_TRUSTSTORE_LOCATION:
/etc/kafka/secrets/kafka.kafka0.truststore.jks
      KAFKA_SSL_KEYSTORE_PASSWORD: password
      KAFKA_SSL_TRUSTSTORE_PASSWORD: password
      KAFKA_SSL_PASSWORD: password
      KAFKA_SECURITY_PROTOCOL: SSL
      SECURITY_PROTOCOL: SSL
      KAFKA_SSL_ENDPOINT_IDENTIFICATION_ALGORITHM: " "
      KAFKA_CONFLUENT_HTTP_SERVER_LISTENERS: " "
      KAFKA_SSL_KEYSTORE_FILENAME: 'kafka.kafka0.keystore.jks'
      KAFKA_SSL_KEYSTORE_CREDENTIALS: credentials.txt
      KAFKA_SSL_TRUSTSTORE_FILENAME: 'kafka.kafka0.truststore.jks'
      KAFKA_SSL_TRUSTSTORE_CREDENTIALS: credentials.txt
      KAFKA_SSL_KEY_CREDENTIALS: credentials.txt
      KAFKA_SSL_CLIENT_AUTH: required
      KAFKA_SSL_PROTOCOL: TLSv1.2
```

Listing 2. Security configuration

`KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:SSL,EXTERNAL:SSL` - defining that both ports, internal and external requires SSL authentication
`KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL`
- informs that INTERNAL port is used for internal brokers communication
To establish SSL encryption and authentication each container must has access to keystore and trustore.

The `KAFKA_SSL_KEYSTORE_LOCATION` and `KAFKA_SSL_TRUSTSTORE_LOCATION` specify the paths to the keystore and truststore files. The corresponding passwords for these stores are set using `KAFKA_SSL_KEYSTORE_PASSWORD` and "KAFKA_SSL_`TRUSTSTORE_PASSWORD` The `KAFKA_SSL_PASSWORD` represents the password for SSL authentication. However, these variables were not enough to establish connection and I had to provide
`KAFKA_SSL_TRUSTSTORE_FILENAME and KAFKA_SSL_TRUSTSTORE_CREDENTIALS.`
`KAFKA_SSL_CLIENT_AUTH: required` - important for enabling two way authentication.

In order to enable container access to the keystore and trustore, which I generated on my local pc, I used the volume section.

This configuration is very similar for each node - in Kafka brokers one part is dedicated for ssl communication with other Kafka brokers, and the other one is used for secure connection with Zookeeper.

7. Producer, consumer and tests

To test the Kafka cluster, I utilized Java Spring Boot applications as producers and consumers. Spring Boot provides convenient and efficient tools for developing microservice-based applications that can easily integrate with the Kafka system.
For creating the topic I used Kafka broker shell. In my application I wanted to simulate sending temperature measurements, so I created the topic “temperature” with three partitions and the replication factor equals three. The following command is:

```
kafka-topics --create --topic temperature --replication-factor 3
--partitions 3 --bootstrap-server localhost:9092 --command-config
/etc/kafka/secrets/client-ssl.properties
```

Since all connections are secure, I needed to provide client credentials.
To check if topic was created successfully I used command:

```
kafka-topics --describe --bootstrap-server localhost:9092
--command-config /etc/kafka/secrets/client-ssl.properties
```

the returned message is:

```
Topic: temperature      TopicId: 6hZmHy4rTDumZyWRNr8YFg PartitionCount: 3
ReplicationFactor: 3    Configs:
        Topic: temperature      Partition: 0    Leader: 3       Replicas: 3,2,1 Isr: 3,2,1
        Topic: temperature      Partition: 1    Leader: 1       Replicas: 1,3,2 Isr: 1,3,2
        Topic: temperature      Partition: 2    Leader: 2       Replicas: 2,1,3 Isr: 2,1,3
```

For testing fault tolerance I turned down a broker with a leader partition. In the consumer node I got information that the broker is not responding and a new coordinator was elected:

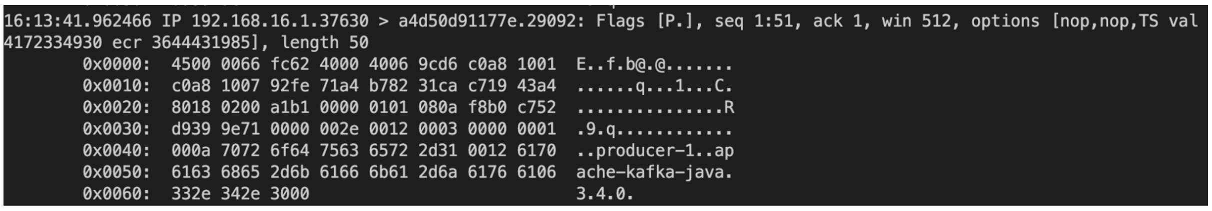
```
[Consumer clientId=consumer-app-3, groupId=app] Node 2147483646 disconnected.
[Consumer clientId=consumer-app-3, groupId=app] Connection to node 2147483646
(localhost/127.0.0.1:29092) terminated
[Consumer clientId=consumer-app-3, groupId=app] Group coordinator localhost:29092 (id:
2147483646 rack: null) is unavailable
[Consumer clientId=consumer-app-3, groupId=app] Discovered group coordinator
localhost:39092 (id: 2147483645 rack: null)
```

Broker at port 29092 stopped responding and a new coordinator was elected on port 39092.

To test the fault tolerance of ZooKeeper, I intentionally turned off one of the ZooKeeper servers in the cluster. Despite the failure of one server, the system continued to operate seamlessly. This is because ZooKeeper relies on a quorum-based approach, where a majority of servers must be available for the system to function properly. In this case, even with one server down, the remaining servers formed a quorum and maintained the consistency and availability of the ZooKeeper service.

To test authentication in my setup, I used a non-authenticated client to establish a connection with the system. As expected, the connection was unsuccessful, as the client lacked the necessary credentials to authenticate itself. This test validates the effectiveness of the authentication mechanism in preventing unauthenticated access to the system.

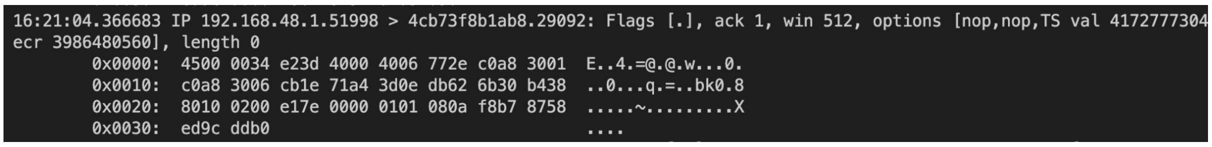
For testing encryption, I employed the use of tcpdump, a network packet analyzer. When I sent data without any encryption, I could observe that the communication channel was open, and the data transmitted was visible in plain text:



```
16:13:41.962466 IP 192.168.16.1.37630 > a4d50d91177e.29092: Flags [P.], seq 1:51, ack 1, win 512, options [nop,nop,TS val
4172334930 ecr 3644431985], length 50
        0x0000:  4500 0066 fc62 4000 4006 9cd6 c0a8 1001  E..f.b@.@.......
        0x0010:  c0a8 1007 92fe 71a4 b782 31ca c719 43a4  ......q...1...C.
        0x0020:  8018 0200 a1b1 0000 0101 080a f8b0 c752  ...............R
        0x0030:  d939 9e71 0000 002e 0012 0003 0000 0001  .9.q............
        0x0040:  000a 7072 6f64 7563 6572 2d31 0012 6170  ..producer-1..ap
        0x0050:  6163 6865 2d6b 6166 6b61 2d6a 6176 6106  ache-kafka-java.
        0x0060:  332e 342e 3000                           3.4.0.
```

Figure 3. Packet from unsecured connection

However, when I enabled data encryption, using protocols such as SSL/TLS, the tcpdump analysis revealed that all send packet were unreadable:



```
16:21:04.366683 IP 192.168.48.1.51998 > 4cb73f8b1ab8.29092: Flags [.], ack 1, win 512, options [nop,nop,TS val 4172777304
ecr 3986480560], length 0
        0x0000:  4500 0034 e23d 4000 4006 772e c0a8 3001  E..4.=@.@.w...0.
        0x0010:  c0a8 3006 cb1e 71a4 3d0e db62 6b30 b438  ..0...q.=..bk0.8
        0x0020:  8010 0200 e17e 0000 0101 080a f8b7 8758  .....~.........X
        0x0030:  ed9c ddb0                                ....
```

Figure 3. Packet from unsecured connection

8. Conclusion

The successful configuration of the Kafka cluster with SSL encryption and authentication, along with its high availability and fault tolerance mechanisms, highlights the robustness of the system. With the deployment of multiple brokers and ZooKeeper servers, the cluster can withstand failures and ensure continuous operation. The fault tolerance feature allows the cluster to maintain its functionality even when individual brokers or ZooKeeper servers experience issues.