1. APACHE KAFKA

1.1 Kafka overview

Apache Kafka [1] is one of the most popular distributed event streaming platforms used for building real-time data pipelines and streaming applications. Kafka is designed to handle massive volumes of data in a scalable and fault-tolerant manner, making it ideal for real-time analytics, data ingestion, and event-driven architectures. Kafka is a distributed publish-subscribe message queue system where data is organized into different topics that can be partitioned to leverage the power of parallel processing and also can be replicated across multiple brokers to ensure fault tolerance.

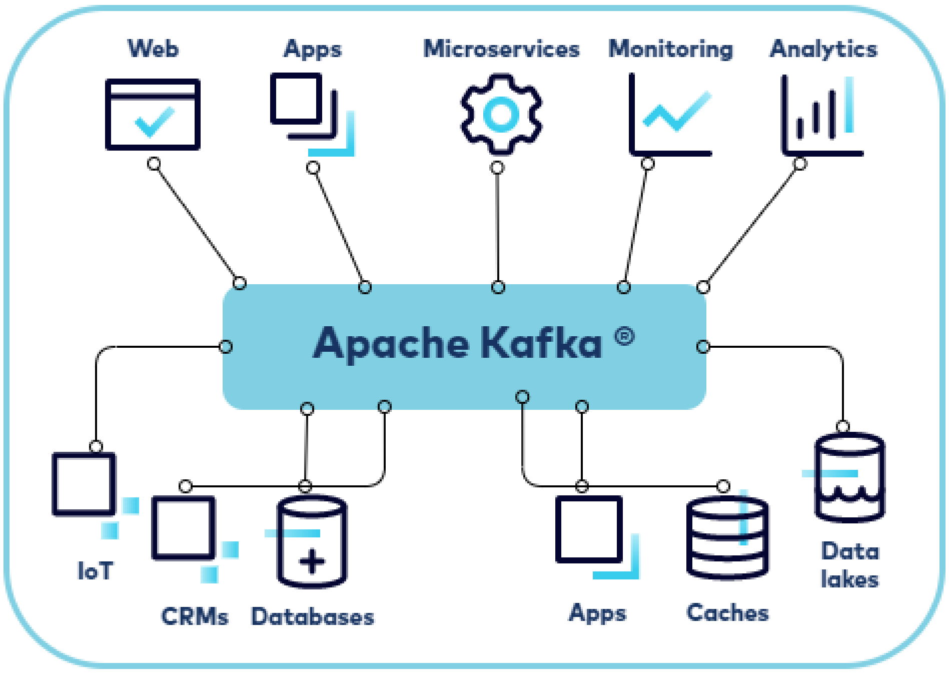


Figure 1: Apache Kafka use cases.

In Figure 1, suppose a large-scale company has multiple applications with massive amounts of data generating and moving across their system. Each application may have to interact with different data sources such as IoT, Databases, Data Lake, ... thus increasing the complexity of data pipelines leads to system error, data inconsistency, ... In this case, Kafka can be used as a centralized data broker where data sources take roles as producers and applications as consumers in order to separate data pipelines to make system communication and management easier. Kafka also reorganizes data in the physical hard disks for boosting ordered data access, making it one of the most powerful when dealing with streaming applications.

[1] `https://docs.confluent.io/kafka/introduction.html`

1.2 Setting up Kafka cluster

In this lab, we use the example of **3 Linux machines** to create a Kafka cluster, make sure to have at least 2 Linux machines (or VMs with bridge network) in each group to run Kafka in cluster mode.

*** Note:** Kafka can even be installed on Window machine since it only needs Java (JVM) environment but it might cause some unexpected errors.

*** Note:** All the following steps are based on the example of Ubuntu 24, students should modify it to match your Linux distro.

- **Step 1:** Install Java on all machines.

```
# If Java is not already installed, install it on all machines
$ sudo apt update -y
$ sudo apt upgrade -y
$ sudo apt install openjdk-11-jdk -y

# Verify Java installation
$ java -version
```

- **Step 2:** Download Apache Kafka.

```
# On each machine, download and extract the Kafka binaries
$ curl -O https://downloads.apache.org/kafka/3.8.0/kafka_2.13-3.8.0.tgz
$ tar -xvzf kafka_2.13-3.8.0.tgz

# From now on, we use the path of this one as $KAFKA_HOME
$ mv kafka_2.13-3.8.0 kafka
$ cd kafka
$ export KAFKA_HOME=$(pwd)
```

- **Step 3:** Configure the KRaft cluster. Kafka brokers in KRaft mode communicate directly with each other to manage the cluster metadata.

```
# Generate a unique cluster ID (only on one machine and use this ID across all brokers)
$ bin/kafka-storage.sh random-uuid
```

- **Step 4:** Configure the brokers. Find and replace these keys with following values located at *$KAFKA_HOME/config/kraft/server.properties* in all machines.

```
# Node ID: Each broker must have unique ID
node.id = 1 # This line appears only in machine 1 configuration
node.id = 2 # This line appears only in machine 2 configuration
node.id = 3 # This line appears only in machine 3 configuration

# Listeners: Specify the machine IP address for communication
listeners=PLAINTEXT://<current machine IP>:9092,CONTROLLER://<current machine IP>:9093

# Cluster metadata configuration: Set the path for storing metadata and the cluster ID generated earlier
process.roles=broker,controller
log.retention.hours=1024
controller.quorum.voters=1@<machine 1 IP>:9093,2@<machine 2 IP>:9093,3@<machine 3 IP>:9093
advertised.listeners=PLAINTEXT://<current machine IP>:9092
```

  After all, the final configuration file should look like this (example for machine 1):

```
node.id=1
listeners=PLAINTEXT://<machine 1 IP>:9092,CONTROLLER://<machine 1 IP>:9093
log.retention.hours=1024
controller.quorum.voters=1@<machine 1 IP>:9093,2@<machine 2 IP>:9093,3@<machine 3 IP>:9093
advertised.listeners=PLAINTEXT://<machine 1 IP>:9092
```

- **Step 4:** Format the storage on each broker.

```
# remove all previous data if exists
$ rm -rf /tmp/kraft-combined-logs

# On each machine, format the storage to initialize the metadata
$ bin/kafka-storage.sh format -t <cluster id> -c $KAFKA_HOME/config/kraft/server.properties
```

- **Step 5:** Setting firewall for Kafka (if firewall is enable).

```
# Allow Kafka Broker (PLAINTEXT) port 9092
$ sudo ufw allow 9092/tcp

# Allow Kafka Controller (KRaft mode) port 9093
$ sudo ufw allow 9093/tcp

# Reload the firewall
$ sudo ufw reload

# Check the firewall status
$ sudo ufw status
```

- **Step 6:** Start the Kafka brokers.

```
# Once the configuration is done and storage is formatted, you can start the brokers on all three machines
# All brokers must be started simultaneously
$ bin/kafka-server-start.sh $KAFKA_HOME/config/kraft/server.properties

# Verify the cluster
$ bin/kafka-topics.sh --list --bootstrap-server <machine 1 IP>:9092
```

**** Further reading:** Kafka standalone version (out of scope in this course).

Apache Kafka also provided a standalone version which only needs 1 machine to operate through Docker container. Docker and Docker Compose (optional but recommended) can be installed by following the official instructions [2].

- **Step 1:** Create a docker-compose.yml file.

```
version: "3"
services:
  kafka:
    image: "confluentinc/cp-kafka:7.4.0"
    container_name: kraft-kafka # Uses the Confluent-provided Kafka image (cp-kafka) that supports KRaft mode
    ports: # Exposes port 9092 for client communication and port 9093 for controller communication
      - '9092:9092'
      - '9093:9093'
    environment:
      KAFKA_BROKER_ID: 1
      # PLAINTEXT for client communication on port 9092
      # CONTROLLER for inter-broker controller communication on port 9093.
      KAFKA_LISTENERS: PLAINTEXT://0.0.0.0:9092,CONTROLLER://0.0.0.0:9093
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://<host ip>:9092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: CONTROLLER:PLAINTEXT,PLAINTEXT:PLAINTEXT
      KAFKA_INTER_BROKER_LISTENER_NAME: PLAINTEXT
      KAFKA_CONTROLLER_QUORUM_VOTERS: "1@localhost:9093" # Make this both broker and controller (single-node setup)
      KAFKA_PROCESS_ROLES: broker,controller # Defines the broker as both role of broker and controller for KRaft
          mode
      KAFKA_NODE_ID: 1
      KAFKA_LOG_DIRS: /var/lib/kafka/data
      KAFKA_METRIC_REPORTERS: io.confluent.metrics.reporter.ConfluentMetricsReporter
      KAFKA_CONFLUENT_SUPPORT_METRICS_ENABLE: "true"
      KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
      KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
      KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
      KAFKA_GROUP_INITIAL_REBALANCE_DELAY_MS: 0
    volumes:
      - ./kafka-data:/var/lib/kafka/data # Enable data persisted with Docker volume
```

- **Step 2:** Run Docker Compose.

```
# Start the Kafka services using docker-compose
$ docker-compose up -d

# This will pull the required Docker images for Kafka and start them in detached mode. To view logs, use
$ docker-compose logs -f kafka
```

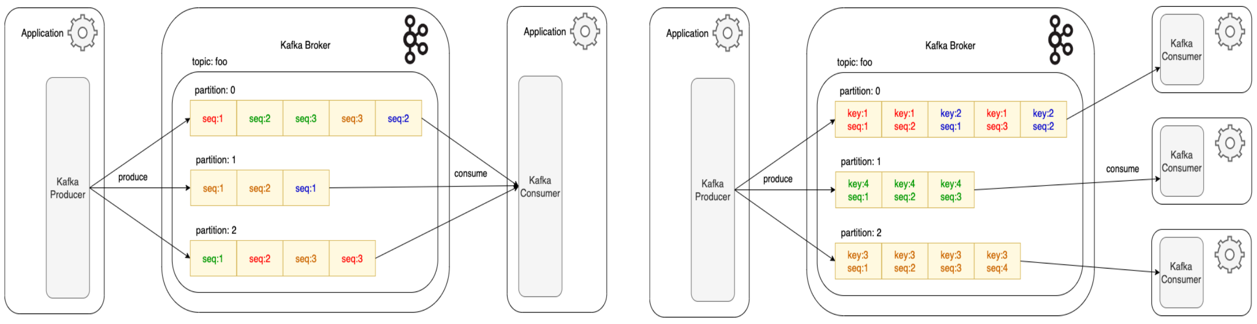
- **Step 3:** Testing Kafka. Once the Kafka service is up and running, you can use the Docker Kafka container to interact with the Kafka broker.

[2] `https://docs.docker.com/engine/install/ubuntu/`

```
# Use the following command to create a Kafka topic named test-topic
$ docker exec -it kraft-kafka kafka-topics --create --topic test-topic --partitions 1 --replication-factor 1 --
    bootstrap-server localhost:9092

# Check the existing topics by running
$ docker exec -it kraft-kafka kafka-topics --list --bootstrap-server localhost:9092

# Produce messages to the test-topic using the Kafka console producer
# After running the command, type a message and press Enter
$ docker exec -it kraft-kafka kafka-console-producer --topic test-topic --bootstrap-server localhost:9092

# To consume the message you just produced, run the following command
$ docker exec -it kraft-kafka kafka-console-consumer --topic test-topic --from-beginning --bootstrap-server
    localhost:9092

# This command will stop and remove the Kafka containers, but the images will remain for future use
$ docker-compose down
```

1.3 Managing topics and partitions

Kafka organizes data into different topics, each may have multiple partitions where data actually stores and replicates. Kafka Producer works independently, each producer specifies the topic, and data records will be placed at different partitions based on its own mechanism [3]. In contrast, Kafka Consumer works with groups for example in Figure 2, if consumer group contains only one consumer then it will receive data from all partitions, otherwise partitions will be assigned equally to all consumers within the same group to enable the power of parallelism [4].



(a) Group contains only one consumer **(b)** Group contains multiple consumers

Figure 2: Kafka topic and partition.

Topics and partitions can be created by **command line** or through **programming api** [5]. For simplicity, we can manage topic and partition using the command below.

```
# Specify --bootstrap-server with <IP BROKER 1>:<PORT>[,<IP BROKER 2>:<PORT>]
# Specify --topic with unique <TOPIC NAME>
$ bin/kafka-topics.sh --create --bootstrap-server <machine 1/2/3 IP>:9092 --topic <topic name>

# Specify --paritittions <NUM> and --replication-factor <NUM>
# Partition number and replication are both 1 by default
$ bin/kafka-topics.sh --create --bootstrap-server <machine 1/2/3 IP>:9092 --partitions 3 --topic <topic name>

# Specify --delete to delete topic and its data
$ bin/kafka-topics.sh --delete --bootstrap-server <machine 1/2/3 IP>:9092 --topic <topic name>

# Specify --describe to check topic configuration
$ bin/kafka-topics.sh --describe --bootstrap-server <machine 1/2/3 IP>:9092 --topic <topic name>
```

[3] https://learn.conduktor.io/kafka/producer-default-partitioner-and-sticky-partitioner/
[4] https://www.lydtechconsulting.com/blog-kafka-message-keys.html
[5] https://kafka.apache.org/38/javadoc/org/apache/kafka/clients/admin/package-summary.html

2. PRELIMINARIES

Apache Kafka provides build-in scripts for manipulating its producer and consumer for simple and fast testing in addition to Programming API written in *Python*, *Java* or *Scala*. For the sake of simplicity, this course will use mostly **Java** to demonstrate the material and students are required to use Java for any Kafka related tasks.

*** Note:** If using Java, make sure to install at least version `11` or newer.

*** Note:** If using Java, make sure to add classpath *-cp* pointing to Kafka "libs/*" when compiling and running (aka. when using *javac* and *java*).

2.1 Producer

Producer is a client application that continuously publishes events (data) to Kafka on a particular topic. The producer specifies the topics they will write to and also controls how events are assigned to partitions within a topic. Producer can be easily created by the command as below.

```
# Specify --bootstrap-server with <IP BROKER 1>:<PORT>[,<IP BROKER 2>:<PORT>]
# Specify --topic with unique <TOPIC NAME>
$ bin/kafka-console-producer.sh --topic <topic name> --bootstrap-server <machine 1/2/3 IP>:9092
```

With Programming API, a simple producer application can be created as below. But in fact, the producer may generate infinite streams of data, thus the sending logic should be wrapped in a *while (true)* loop.

```java
import java.util.*;
import org.apache.kafka.clients.producer.*;
import org.apache.kafka.common.serialization.*;

public class Producer {
  public static void main(String args[]) {
    // Producer mandatory config
    final Properties conf = new Properties();
    conf.setProperty(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, "<machine 1/2/3 IP>:9092");
    conf.setProperty(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class.getName());
    conf.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, EnvSerializer.class.getName());

    // Initialize producer and send data
    KafkaProducer<String, String> producer = new KafkaProducer<>(conf);
    String data = "Message from producer";
    ProducerRecord<String, String> record = new ProducerRecord<>("<topic name>", data);
    producer.send(record);

    // Close producer
    producer.flush();
    producer.close();
  }
}
```

2.2 Consumer

Consumer is a client application that reads events from Kafka. The only metadata retained on a per-consumer basis is the offset or position of that consumer in a topic. This offset is automatically and linearly increased when the consumer reads records but it can be defined and committed manually. The following script is used to create a consumer which dumps all records into the screen.

```
# Specify --bootstrap-server with <IP BROKER 1>:<PORT>[,<IP BROKER 2>:<PORT>]
# Specify --topic with unique <TOPIC NAME>
# Specify --from-beginning to consume all historical data
$ bin/kafka-console-consumer.sh --topic <topic name> --from-beginning --bootstrap-server <machine 1/2/3 IP>:9092
```

Consumer applications can be implemented using Java as below. Similar to the producer, consumer logic should be put inside a *while (true)* loop since it might have to process infinite streams of data generated at the producer side.

```java
import java.util.*;
import org.apache.kafka.clients.consumer.*;
import org.apache.kafka.common.serialization.*;

public class Consumer {
  public static void main(String args[]) {
    // Consumer mandatory config
    final Properties conf = new Properties();
    conf.put(ConsumerConfig.GROUP_ID_CONFIG, "<group name>");
    conf.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, "<machine 1/2/3 IP>:9092");
    conf.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class.getName());
    conf.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, EnvDeserialzer.class.getName());

    // Initialize consumer and poll data
    KafkaConsumer<String, String> consumer = new KafkaConsumer<>(conf);
    consumer.subscribe(List.of("<topic name>"));
    ConsumerRecords<String, String> records = consumer.poll(Duration.ofMillis(100));
    records.forEach(record -> {
        System.out.println(record.value().toStr());
    });

    // Close consumer
    consumer.close();
  }
}
```

Besides, consumers can reset to an older offset to reprocess data from the past or skip ahead and start to consume the most recent record and start consuming from "now". For example, add the following config.

```java
// To read all historical data
conf.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

// To read only current data
conf.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "latest");
```

2.3 Custom Serializer

Producer sends data to Kafka in the form of byte-stream through a TCP connection and is similar to consumer when receiving data from Kafka as depicted in Figure 3. Serializer converts meaningful data from producer into a byte array and deserializer converts it back to the original format before the consumer begins processing [6].

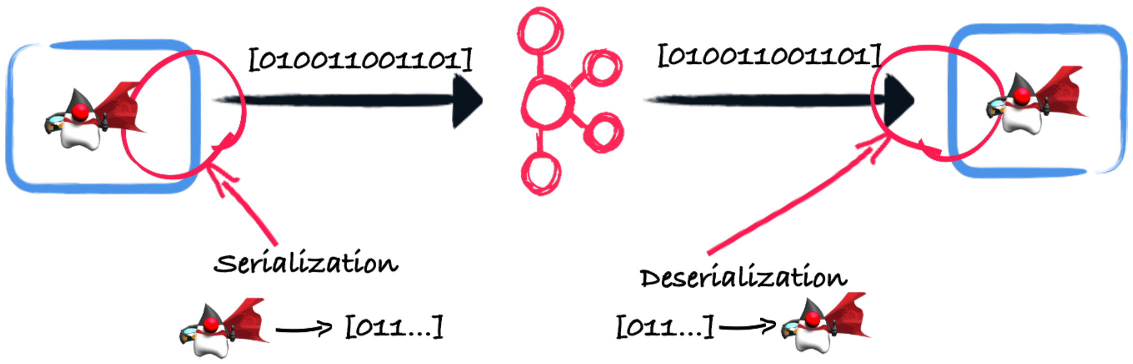


Figure 3: Kafka data serialization.

Suppose we have a dataset of environment with the structure as below where *time* indicates the timestamp collects that record, *type* is the type of data which is either "air", "earth" or "water" and *value* is the collected value corresponding to that type.

[6]https://quarkus.io/blog/kafka-serde/

```
import java.util.Date;

public class Environment {
  public Date time;
  public String type;
  public int value;
}
```

We can create our own serializer to convert Environment data into a byte array as below. Note that if using a custom serializer we have to specify it in producer config.

```
import Enviroment;
import java.util.Map;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import org.apache.commons.lang3.SerializationException;
import org.apache.kafka.common.serialization.Serializer;

public class EnvSerializer implements Serializer<Environment> {
  final SimpleDateFormat DF = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

  @Override
  public void configure(Map<String, ?> configs, boolean isKey) {
    // Do nothing, not necessary right now
  }

  @Override
  public byte[] serialize(String topic, Environment data) {
    try {
      if (data == null) {
        return null;
      }

      // Serialize string data to byte
      byte[] time = DF.format(data.time).getBytes("UTF8");
      byte[] station = data.station.getBytes("UTF8");
      int value = data.value;

      // Create byte buffer
      int len = Integer.BYTES + station.length + Integer.BYTES + time.length + Integer.BYTES;
      ByteBuffer buf = ByteBuffer.allocate(len);
      buf.putInt(station.length);
      buf.put(station);
      buf.putInt(time.length);
      buf.put(time);
      buf.putInt(value);

      return buf.array();

    } catch (Exception e) {
      throw new SerializationException("Error when serializing environment data.");
    }
  }

  @Override
  public void close() {
    // Do nothing, not necessary right now
  }
}
```

```
// Put in producer config
conf.setProperty(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, EnvSerializer.class.getName());
// Producer will be adjusted as follows
KafkaProducer<String, Environment> producer = new KafkaProducer<>(conf);
ProducerRecord<String, Environment> record = new ProducerRecord<>("topic-name", null);
```

We also have to create our own deserializer. The following codes demonstrate the deserializer for Environment data and similarly, and we also have to specify it in the consumer config.

```
import java.util.Map;
import java.nio.ByteBuffer;
import java.text.SimpleDateFormat;
import org.apache.commons.lang3.SerializationException;
import org.apache.kafka.common.serialization.Deserializer;

public class EnvDeserialzer implements Deserializer<Environment> {
  final SimpleDateFormat DF = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");

  @Override
  public void configure(Map<String, ?> configs, boolean isKey) {
    // Do nothing, not necessary right now
  }

  @Override
  public Environment deserialize(String topic, byte[] data) {
    try {
      if (data == null) {
        return null;
      }

      ByteBuffer buf = ByteBuffer.wrap(data);
      byte[] station = new byte[buf.getInt()];
      buf.get(station);
      byte[] time = new byte[buf.getInt()];
      buf.get(time);
      int value = buf.getInt();

      return new Environment(
        DF.parse(new String(time, ENCODING)), new String(station, ENCODING), value
      );

    } catch (Exception e) {
        throw new SerializationException("Error when deserializing environment data.");
    }
  }

  @Override
  public void close() {
    // Do nothing, not necessary right now
  }
}
```

```
// Put in consumer config
conf.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, EnvDeserialzer.class.getName());
// Consumer will be adjusted as follows
KafkaConsumer<String, Environment> consumer = new KafkaConsumer<>(conf);
ConsumerRecords<String, Environment> records = null;
```

2.4 Custom Partitioner

By default, producer does not care which partition where data should go into and it will be split equally in a round-robin fashion. But in some cases such as supporting analyzing tasks on consumer side, produce could manually specify the partition of each record or route it automatically through a custom partitioner. Suppose the environment topic has 3 partitions, the following partitioner code ensures "air" data will go to partition 0, "earth" to partition 1, and "water" to partition 2.

```
import Enviroment;
import java.util.Map;
import org.apache.kafka.common.Cluster;
import org.apache.kafka.clients.producer.Partitioner;

public class EnvPartitioner implements Partitioner {
  @Override
  public void configure(Map<String, ?> configs) {
    // Do nothing, not necessary right now
  }

  @Override
  public int partition(String topic, Object key, byte[] keyBytes, Object value, byte[] valueBytes, Cluster cluster) {
    Environemnt data = (Environment) value;
    if (data.type.equals("air")) return 0;
    else if (data.type.equals("earth")) return 1;
    else return 2;
  }

  @Override
  public void close() {
    // Do nothing, not necessary right now
  }
}
```

```
// Put in producer config
conf.setProperty(ProducerConfig.PARTITIONER_CLASS_CONFIG, EnvPartitioner.class.getName());
```

3. Exercises

Students are given real-world datasets collected from environmental sensors in "Dong Thap" province. Data is collected in a 1-minute interval from the beginning of 01/01/2023 to the end of 30/09/2023. In this lab, we are only interested in 3 files below.

- **AIR2308.csv:** monitor the air quality.
- **EARTH2308.csv:** monitor the status of agricultural land.
- **WATER2308.csv:** monitor the quality of water source.

Students must be careful (will have to preprocess if necessary) when dealing with real-world datasets since they may contain outlier, noises, null data, ... In particular, students are required to:

1. Create 3 classes corresponding to the structure format of "air", "earth" and "water".
2. Create 3 topics containing data of "Air", "Earth" and "Water". The partition of each topic should contain data collected from only 1 station. For example "air" data is collected from station "SVDT1" and "SVDT3" hence it corresponding topic should have 2 partitions similar to "earth" and "water".
3. Create a producer to send all these data **CONCURRENTLY** using the class datatype defined from task 1 to the corresponding topics.
4. Create consumers with the same group id, each will receive and dump data from different partitions independently.

*** Note:** This exercise requires students to create exactly 1 custom serializer, 1 custom deserializer, 1 custom partitioner along with 1 producer. The amount of consumers may vary depending on the total number of partitions.

**** Submission:** Works must be done individually.

- Compress all necessary files as **"Lab_1_<Student ID>.zip"**.
- Students must submit and demo their results before the end of class.