1. FOG COMPUTING

Fog computing is a decentralized computing infrastructure, which brings data, computing, storage, and applications closer to the data source. Many people use the terms fog computing and edge computing interchangeably because both involve bringing intelligence and processing closer to where the data is created. This is often done to improve efficiency, though it might also be done for security and compliance reasons [1].

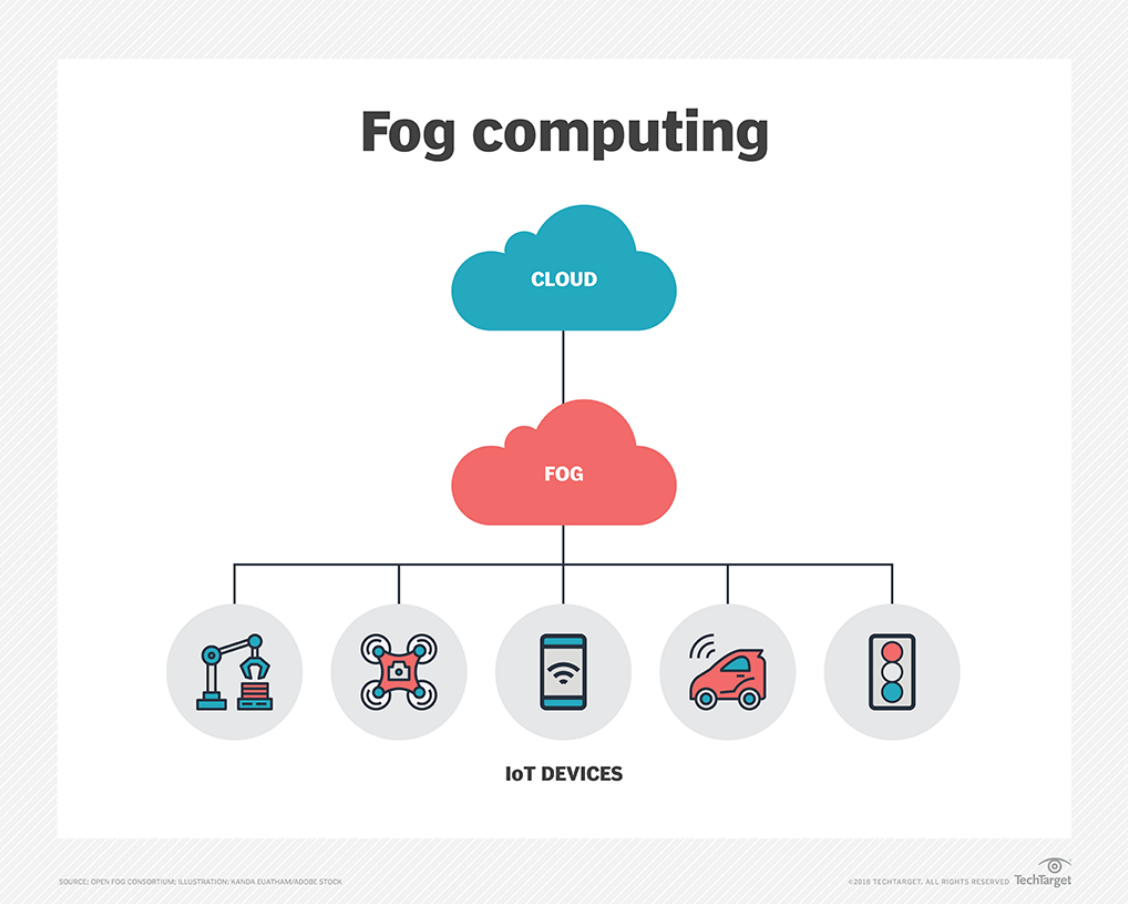

Figure 1: Fog computing architecture.

Fog computing does not tend to replace cloud computing, fogging enables short-term analytics to reduce centralized computing load, while the cloud performs resource-intensive, longer-term analytics. Fog servers are not always smaller than centralized cloud servers, but in fact, they sometimes are even more powerful. In this lab, we will simulate fog architecture through the local cluster in each group.

[1] https://www.techtarget.com/iotagenda/definition/fog-computing-fogging

2. KAFKA STREAM

Along with the data streaming platform, Apache Kafka also provides its build-in stream processing library, named "KStream", in order to rapidly create the data pipeline. KStream offers both stateless (each record is treated independently) and stateful (consider historical data) API but for simplicity, we only focus on stateless operations in this lab.

Similar to other streaming processing libraries, KStream also contains transform and terminate operations, where transform are *lazy evaluate* and triggered by terminate operations. Normally, KStream is used to process data retrieved from particular topics and load it again into Kafka under a different topic (ETL process). The following code reads each Environment record, assigns the time field to its key, transforms it from Environment datatype to String type then loads it to "Environment_String" topic.

```
import java.util.*;
import org.apache.kafka.streams.*;
import org.apache.kafka.streams.kstream.*
import org.apache.kafka.common.serialization.*;

public class SimpleKStream {
  public static void main(String args[]) {
    Properties conf = new Properties();
    conf.put(StreamsConfig.APPLICATION_ID_CONFIG, "stream-app");
    conf.put(StreamsConfig.BOOTSTRAP_SERVERS_CONFIG, "<machine 1/2/3 IP>:9092");
    conf.put(StreamsConfig.NUM_STREAM_THREADS_CONFIG, 1);
    conf.put(StreamsConfig.DEFAULT_KEY_SERDE_CLASS_CONFIG, Serdes.String().getClass().getName());
    conf.put(StreamsConfig.DEFAULT_VALUE_SERDE_CLASS_CONFIG, EnvSerde.class.getName());

    StreamsBuilder builder = new StreamsBuilder();
    KStream<String, Environment> env_stream = builder.<String, Environment>stream("<topic name 1>")
      .selectKey((k, v) -> v.time.toString())
      .mapValues(v -> String.format("Station: %s value=%s", v.station, v.value()));
    env_stream.to("<topic name 2>", Produced.with(Serdes.String(), new EnvSerde()));

    KafkaStreams streams = new KafkaStreams(builder.build(), conf);
    Runtime.getRuntime().addShutdownHook(new Thread() {
      public void run() {
        System.out.println("Shutting down...");
        stream.close();
      }
    });
    streams.start();
  }
}
```

Since Environment is the custom datatype, similar to producer and consumer, we have to create the custom serializer and deserializer (Serde in the context of KStream).

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

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

  @Override
  public Serializer<Environment> serializer() {
    return new EnvSerializer();
  }

  @Override
  public Deserializer<Environment> deserializer() {
    return new EnvDeserialzer();
  }

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

KStream provides almost every common transform function beside *selectKey, mapValues* such as *filter, flatmap,....* The return type of each function indicates the type of operations, the terminate function will return *void*. In case of transform function, if the return type is also *KStream* then it is a stateless operation and is stateful otherwise [2].

In case of transform operation, KStream provides the ability to integrate two streams of different datatype through either *join, leftJoin and outerJoin* API [3]. Records from two streams will be joined if they have the same key, hence the record key can not be null. The following code joins two streams of Environment and Integer datatype based on the time field (key) to create a new stream with String datatype.

```
// Config is the same
StreamsBuilder builder = new StreamsBuilder();
KStream<String, Environment> stream_1 = builder.<String, Environment>stream("<topic name 1>")
  .selectKey((k, v) -> v.time.toString());
KStream<String, Integer> stream_2 = builder.<String, Environment>stream("<topic name 2>")
  .selectKey((k, v) -> v.time.toString())
  .mapValues(v -> v.value);

KStream<String, String> join_stream = stream_1.join(stream_2, (left, right) -> {
  return String.format("station: %s value=%s", left.station, left.value + right);
}, JoinWindows.ofTimeDifferenceWithNoGrace(Duration.ofMillis(1000)),
// Each record waits record from other stream at most 1s
StreamJoined.with(Serdes.String(), new EnvSerde(), Serdes.Integer());
// StreamJoined.with(<Key datatype>, <stream 1 Value datatype>, <stream 2 Value datatype>)
join_stream.foreach((k, v) -> System.out.println(k + ": " + v));

KafkaStreams streams = new KafkaStreams(builder.build(), conf);
Runtime.getRuntime().addShutdownHook(new Thread() {
  public void run() {
    System.out.println("Shutting down...");
    stream.close();
  }
});
streams.start();
```

[2] KStream API: `https://kafka.apache.org/38/javadoc/org/apache/kafka/streams/kstream/KStream.html`
[3] `https://www.confluent.io/blog/crossing-streams-joins-apache-kafka/`

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 23/09/2023.

Suppose we have to collect environmental data from 3 types of sensors above. Assume that all earth sensors continuously push data to the earth station which has its own Kafka cluster independent of air station and water station (also have its own Kafka cluster). Since sensor data is uncertain and not reliable because of various reasons such as bad weather, out of battery, ... we have to preprocess it before sending it to the centralized server for further complex analysis. In particular, students are required to:

1. Reuse all the exercise results from the previous lab (topic, datatype, serializer, ...).
2. Each group should have at least 2 Kafka servers operating independently, one acts as a fog and the other as a cloud server.
3. Use KStream library to impute missing numeric data from each record at the fog node based on the following formula then send it to the centralized server.

$$v_n^i = avg(v_{0\to n-1}^i) + rand(-std_{0\to n-1}^i, std_{0\to n-1}^i) \quad \forall i$$

   Where $avg(v_{0\to n-1}^i)$ is the average value of all previous data point at i^{th} data field and $std_{0\to n-1}^i$ is the standard deviation of all previous data point at i^{th} field. Note that the avg and std must be calculated **incrementally**.

4. Use KStream library to join all 3 types of data to form a complete tuple (each record must contain data from all "air", "earth", and "water") then write to "Environment" topic at the centralized server.

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

- Compress all necessary files as **"Lab_3_<Student ID>.zip"**.
- Students must submit and demo their results before the end of class.