M1 Informatics Project

Securing an event streaming platform for IoT - Apache Kafka

presented by

Nguyen Van Tien
Nguyen Thi Mai Phuong
Doan Thi Van Thao

Master 1 CRYPTIS

Instructors: Prof. Emmanuel Conchon
Dr. Mathieu Klingler

May 5, 2021

Contents

List of Figures

Chapter 1

Introduction

From the previous part of this document (1^{st} semester), we discussed about the concept of Kafka and Zookeeper. In terms of stream processing, Kafka is a messaging system that lets you publish and subscribe to streams of messages. In this way, it is similar to products like ActiveMQ, RabbitMQ, IBM's MQSeries, and other products. But it is not just a message broker. Apache Kafka can help you decouple different data pipelines and simplify software architecture to make it manageable. Kafka is used heavily in the big data space as a reliable way to ingest and move large amounts of data very quickly. There are serveral enterprises using Kafka such as Uber, Netflix, Activision, Spotify, Slack, Pinterest, Coursera and of course Linkendin. [9]. ZooKeeper is the default storage engine, for consumer offsets. However, all information about how many messages Kafka consumer consumes by each consumer is stored in ZooKeeper.

Today's markets are highly dynamic, with major change occurring randomly on a frequent basis. No organization can function without data these days. With huge amounts of data being generated every second from business transactions, sales figures, promotion strategies, customer behaviors, and stakeholders, data is the fuel that drives companies. But when the data volume is huge, enterprises have to face the most pressing challenges that is storing all these huge sets of data properly. However, as these data sets grow exponentially with time, it gets extremely difficult to handle.

A potential solution for companies is to outsource data storage, thus seeking third-party experts outside of the company. Data storage outsourcing is, however, placing its trust in a third-party – relying on their expertise, their resources, and their services. Moreover, storage outsourcing shares their resources with other tenants. This sharing helps to save on costs and makes the IT scalable, but it does mean that other companies will be using some of the same servers and same devices. This creates security concerns that need to be addressed, especially if the company has strict compliance requirements. While companies may trust a Storage Service Provider's (SSP) reliability and performance, they cannot be sure 100% that an SSP is not going to use the data for other purposes, especially when the data is highly valuable.

If our company is hiring a kafka server as the third party, it's necessary to encrypt all transmitted data as well as hide the "topic" name from server's acknowledgement.

Within the scope of this project, we are going to propose and implement a matching scheme which matches two different topic names requested from producer and consumer, into a common one that was an encrypted topic name stored in the kafka server.

The requirements for the accomplishment are:

- Successfully implementing a matching algorithm into a zookeeper project and export it as a jar library which could be used for other projects;
- Simulating two kafka clients (producer and consumer) in Java;
- Simulating communication between a producer, a consumer and multiple kafka servers.

For a more secure system, we propose encrypting all data stored in the 3^{rd} party server. These tasks will be completed in a future version of this project.

Chapter 2

Problem Analysis

2.1 Solution Proposals

In this section, we will propose some solutions to resolve two main problems:

- Hide the plain topic name from the server; and
- Only authorized clients could subscribe to a certain topic.

Let take a look at three following schemes.

2.1.1 Symmetric Key Encryption

To hide topic name from the server and allow only authorized clients to subscribe encrypted topic names, we can use a symmetric key encryption scheme. Assume that an organisation has their own secret key "K_c" and server will generate a unique server-side secret key "K_s" for each organisation. We propose following protocol:

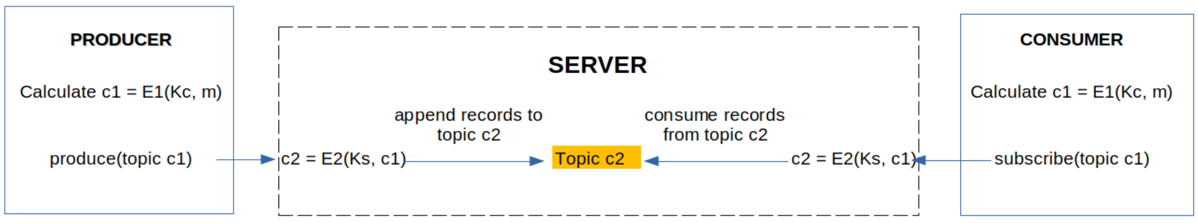


Figure 2.1: A symmetric key encryption scheme

The K_c is used to hide topic name from server. The K_s is secret and only known with server. With this model, only clients in a same organisation are able to access to the exact c2 topic. The problem is that each client need to be assigned an ID to be recognized in which organisation he is.

2.1.2 PKI based Scheme

Each organization has their own a key pair (public key, private key) = (K_P, k_s). Remind a public key model:

$$D(K_P, E(k_s, m)) = m$$

Let see how this system works:

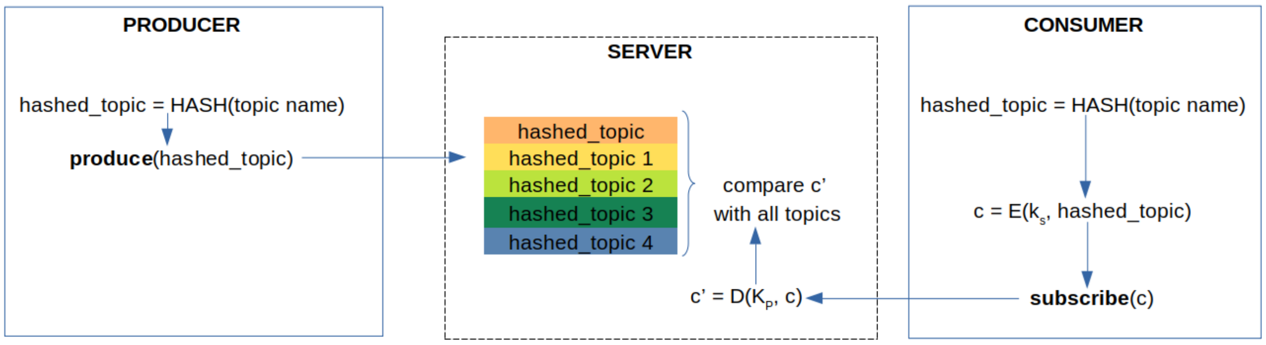


Figure 2.2: A PKI model

To hide plain topic name from the server, producers will create and produce records to a hashed topic name. A consumer will send a request to subscribe the encrypted of that hashed value using the secret key k_s of their organization. The server then calculate c' from c using the public key K_P and compare with all hashed topics of the server. If there exists a hashed topic = c', return that stream back to consumer.

With this proposal, the server need to know which organization a client is in for using the exact public key for the matching phase.

This scheme also holds a potential risk where all clients in an organization keep the secret key. The more people keep a same secret, the higher risk of leaking it. We need to a more efficient scheme where each client keeps their own secret.

2.1.3 RSA based Matching Scheme

The notion of proxy encryption was first introduced by Blaze et al. [7] in 1998. In RSA-based proxy encryption scheme as on Figure 2.3, a group of symbols *(IGen, UGen, UEnc, UDec, PEnc, PDec)* is used to denote the proxy encryption scheme, where:

- *IGen* is the master key generation algorithm which is identical to the key generation algorithm in the standard RSA. *IGen* needs only to be run once at the beginning of the system setup.
- *UGen* is the algorithm for generating the key pairs for the users and the proxy. For each user i, *UGen* takes the output of *IGen* and finds e_{i1}, e_{i2}, d_{i1}, and d_{i2} such that $e_{i1}e_{i2} \equiv e \mod \varphi(n)$ and $d_{i1}d_{i2} \equiv d \mod \varphi(n)$.

- *UEnc* is the algorithm for user encryption. For a message m, user i encrypts it using his encryption key $K_{uei} = e_{i1}$. The resulting ciphertext is $c = m^{e_{i1}}$.
- *PEnc* is the algorithm for proxy encryption. When the proxy receives a ciphertext c from user i, it re-encrypts it using the corresponding encryption key.
- *PDnc* is the algorithm for proxy decryption. Before sending the ciphertext to user j, the proxy decrypts it using the corresponding decryption key.
- *UDec* is the algorithm for user decryption. When a user j receives a ciphertext c' from the proxy, he decrypts it using his decryption key.

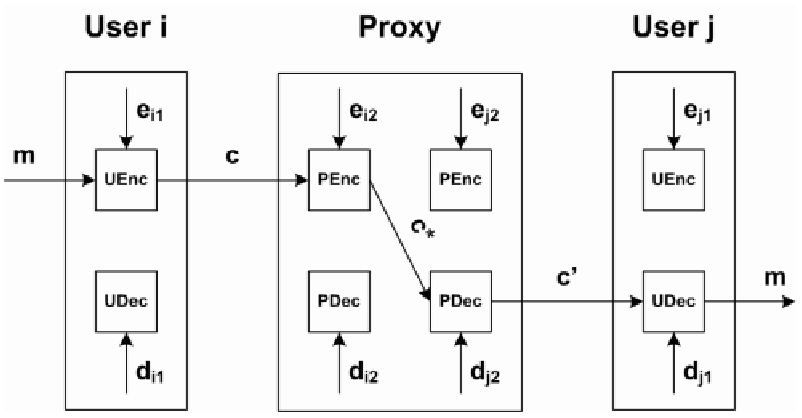


Figure 2.3: RSA-based Proxy Encryption Scheme.

A trusted key management server (KMS) controlled by the data owner is aplied to manage the keys. To be general, a producer is called client i and a customer is called client j. Suppose that client i pushed data in a untrusted server and client j wants to search for data from the same server without security loss. As stated on the name, the proxy is built based on RSA technique, where there are private and public keys. On the step of *UGen*, the set of e_{i1} , e_{i2}, d_{i1}, and d_{i2} are found. While (e_{i1}, d_{i1}) are kept by client i, (e_{i2}, d_{i2}) are kept by the server. If the user is removed from the system at a later stage, the KMS can send a instruction to the data server to remove the key pair (e_{i2}, d_{i2}) at the server side.

For this model, each client keep a private key for their own and the proxy (here, the zookeeper server) keep a corresponding secret key. Server will be not aware of any information about a client other than that secret key.

2.2 Security Analysis

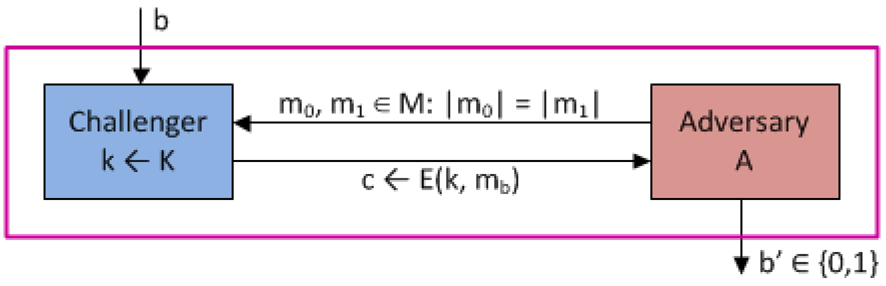
2.2.1 Semantic Security

In cryptography, a semantically secure cryptosystem is one where only negligible information about the plaintext can be feasibly extracted from the ciphertext. Specifically, any probabilistic, polynomial-time algorithm (PPTA) that is given the ciphertext of a certain message m (taken from any distribution of messages), and the message's length, cannot determine any partial information on the message with probability non-negligibly higher than all other PPTA's that only have access to the message length (and not the ciphertext). [1] This concept is the computational complexity analogue to Shannon's concept of perfect secrecy. Perfect secrecy means that the ciphertext reveals no information at all about the plaintext, whereas semantic security implies that any information revealed cannot be feasibly extracted.

Adversary's power: Adversary sees only one ciphertext (one time key).

Adversary's goal: Learn info about plaintext from ciphertext (sematic security).

Let E= (E,D) be a cipher over (K,M,C). For b=0,1 define EXP(b) as:



Definition 1. E is semantically secure if for all efficient A:

$$Adv_{SS}[A,E] := |PR[EXP(0) = 1] - PR[EXP(1) = 1]| \text{ is negligible.}$$

2.2.2 RSA based Matching Scheme is semantically secure

Definition 2. Let $E =$ (IGen, UGen, UEnc,UDec,P Enc, P Dec) be the proxy encryption scheme. E is said to be One-Way secure against any PPT attacker A if $Succ_{A,E}$ is negligible. $Succ_{A,E}$ is defined as follows:

$$Succ_{A,E} = \left[m = m' \;\middle|\; \begin{array}{c} \text{m(p, q, n, } \varphi\text{(n), e, d)} \leftarrow IGen(1^k), \\ (K_u, K_p) \leftarrow UGen(\varphi(n), e, d), \\ m \leftarrow A(K_p, n, m^{\epsilon}), \epsilon \in K \end{array} \right]$$

Loosely speaking, the proxy encryption scheme is one-way secure if by knowing the public parameter n, all the key pairs on the server side, ciphertexts encrypted under an authorised user's encryption key and any information can be derived from above, e.g. intermediate ciphertexts calculated using the server side keys, but without knowing any key pairs in the authorised user key pair set K_u, no PPT adversary can find the corresponding plaintext.

Lemma. *Under the RSA assumption, the proxy encryption scheme is OneWay secure against adversaries (Adv).* [10]

Proof. We will show that if Adv can break the proxy encryption scheme, i.e. $Succ_{A,E}$ is not negligible, then there is an attacker B who can solve the RSA problem with non-negligible probability.

Given an RSA ciphertext $c = m^e$ where the corresponding key pair is (e, d), the goal of B is to decrypt it, i.e. to find m. B can pick x pairs of random primes $\frac{n}{2} < (e_B, d_B)_i < n - 2^{161}$. The primes are relatively prime to $\varphi(n)$ because $\frac{\varphi(n)}{2} < (e_B, d_B)_i < \varphi(n)$. B then sends $c, n, (e_B, d_B)_i, i = 1, ..., x$ to Adv.

Adv can computes $c_1 = c^{e_{B1}}, c_2 = c_1^{d_{B1}}$. Next we will show that $c, c_1, c_2, n, (e_B, d_B)_i, i = 1, ..., x$ can correctly simulate adv's knowledge in the proxy encryption scheme. First we will show that c, c_1, c_2 are valid ciphertexts for the proxy encryption scheme. The ciphertexts are valid if there exists a d' such that $e_2^{d'} = m$, i.e. $ee_{B1}d_{B1}d' \equiv 1 \mod \varphi(n)$. Because e_{B1}, d_{B1} are relatively prime to $\varphi(n)$, we can always find y such that $e_{B1}d_{B1}y \equiv 1 \mod \varphi(n)$. Therefore there always exists $d \equiv dy \mod \varphi(n)$ such that $ee_{B1}d_{B1}d' \equiv ee_{B1}d_{B1}dy \equiv (ed)(e_{B1}d_{B1}y) \equiv 1 \mod \varphi(n)$. We also need to show that $(e_B, d_B)_i, i = 1, ..., x$ are valid server side key pairs, this can be easily proved using the similar method as above therefore is omitted.

Now with the message from B, Adv can find m with probability $Succ_{A,E}$ and returns the result to B. This means B can solve the RSA problem with non-negligible probability $Succ_{A,E}$, which contradicts the RSA assumption.

From these analysis, we will be adapting above mentioned RSA based matching scheme into a zookeeper server for hiding topic information from kafka server to reinforce the confidentiality of the exchanges while allowing authorized users to subscribe to its feeds.

Chapter 3

Implementation

In this chapter, we present the detailed steps to implement the encrypted matching scheme into the workflow of ZooKeeper and Kafka servers. In particular, the implementation process includes five main parts, which include:

- **section 3.1**: Develop encrypted matching scheme.
- **section 3.2**: Configure and implement encrypted matching schema to ZooKeeper.
- **section 3.3**: Configure Kafka sever.
- **section 3.4**: Implement encrypted matching scheme to action `publish` of Producer.
- **section 3.5**: Implement encrypted matching scheme to action `subscribe` of Consumer.

3.1 Develop encrypted matching scheme

After many examinations to choose a potential encrypted matching scheme for implementation, we decide to apply the RSA-based Proxy Encryption Scheme [3]. However, in this project, we have made some adaptable changes to fit the scheme with the goal of our project. The following is the encrypted matching schema that is applied.

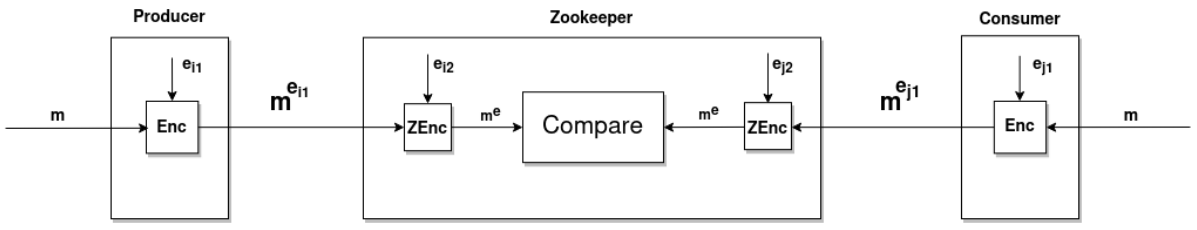


Figure 3.1: Encrypted matching scheme

In the RSA-based proxy encryption scheme, we have *IGen* and *UGen* (subsection 2.1.3), which are key generation algorithms. However, because of limited implementation time, we decided to skip the implementation of these two algorithms. To put it another way, in this implementation, we consider that ZooKeeper and all clients already have the required keys. Additionally, in this scope of the project, we only work with the topic's name so it is not essential to use decryption. Therefore, two algorithms for decryption *PDec* and *UDec* are also not implemented.

Function `Enc`

We developed function `Enc` based on algorithm `UEnc` of RSA-based Proxy Encryption Scheme. This function is used for producer and consumer encryption. With a message m, client i encrypts it using his encryption key $K_i = e_{i1}$. The resulting ciphertext is $c = m^{e_{i1}} \mod N$.

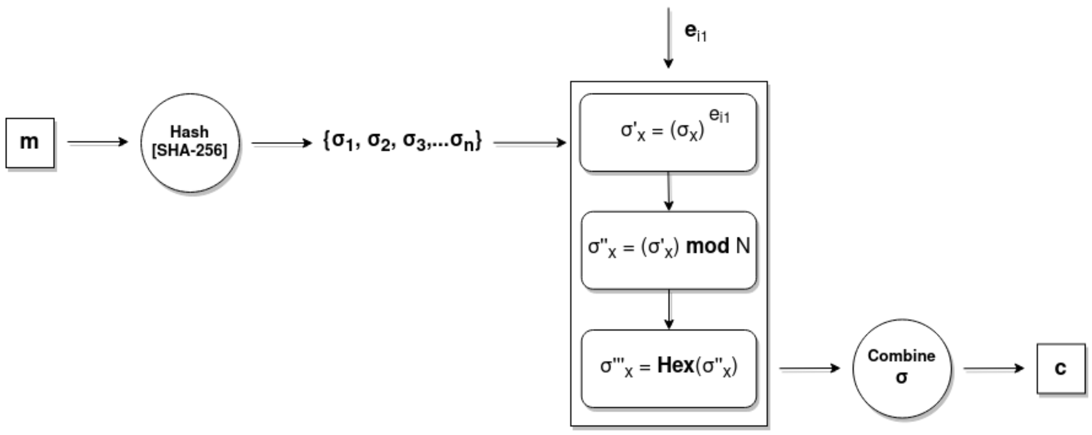


Figure 3.2: Client data encryption scheme

Listing 1 Function **Enc**

```java
public static String Enc(String str, int e) {
        try {
            MessageDigest digest = MessageDigest.getInstance("SHA-256");
            byte[] hash = digest.digest(str.getBytes(StandardCharsets.UTF_8));
            BigInteger[] Cwi = new BigInteger[hash.length];
            String hashString = "";
            for (int i = 0; i < hash.length; i++) {
                BigInteger big = BigInteger.valueOf(hash[i]).pow(e);
                Cwi[i] = big.mod(N);
                String tmp = Cwi[i].toString(16);
                while (tmp.length() < HEX_LENGTH) {
                    tmp = "0" + tmp;
                }
                hashString = hashString + tmp;
            }
            return hashString;
        } catch (NoSuchAlgorithmException exception) {
            exception.printStackTrace();
        }
        return "";
    }
```

In Figure 3.2 above, we chose to convert each value in the message into hexadecimal base, where each 4 bits in base 16 is corresponding to a σ_x''. However, in case N is large, the value of σ_x'' is also larger, it is necessary to change the number of bits in base 16 representing for each σ_x''.

Function ZEnc

In RSA-based proxy encryption scheme, PEnc is the algorithm for proxy encryption. When the proxy receives a ciphertext c from user i, it re-encrypts it by using the corresponding encryption key. Follow this algorithm, we create function ZEnc which is placed on ZooKeeper.

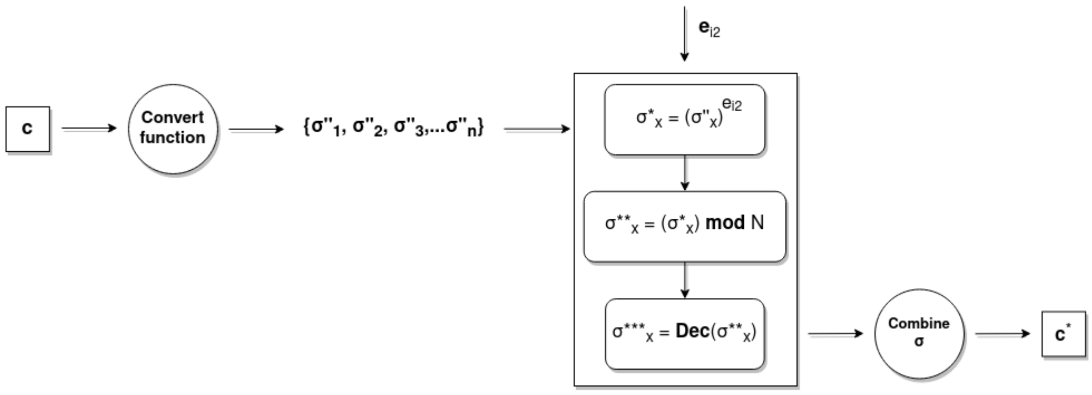


Figure 3.3: ZooKeeper data encryption scheme

Listing 2 Function **ZEnc**

```
public static String ZEnc(String str, int e) {
        int index = 0;
        List<BigInteger> Cwi = new ArrayList<>();
        try {
            while (index < str.length()) {
                String tmp = str.substring(index, Math.min(index + 4, str.length()));
                BigInteger tmp_int = new BigInteger(tmp, 16);
                Cwi.add(tmp_int);
                index += 4;
            }
            BigInteger[] Cwi_star = new BigInteger[Cwi.size()];
            String CwiStar = "";
            for (int i = 0; i < Cwi.size(); i++) {
                BigInteger bigCw = Cwi.get(i).pow(e);
                Cwi_star[i] = bigCw.mod(N);
                CwiStar = CwiStar + Cwi_star[i].toString(10);
                return CwiStar;
            }
        } catch (Exception exception) {
            exception.printStackTrace();
        }
        return "";
    }
```

Function `Compare`

If two clients send the same values into function `Enc` and ZooKeeper re-encrypts with the corresponding encryption keys, the output of the function `Enc` will always be the same, which is m^e.

Listing 3 Function **Compare**

```
public static Boolean ms_compare(String str1, String str2) {
        String prod_topic_mA = ZEnc(str1, EI2);
        String cons_topic_mA = ZEnc(str2, EJ2);
        return (prod_topic_mA.equals(cons_topic_mA));
    }
```

3.2 ZooKeeper

Download and configurations

Download

In this implementation, we use the version `Apache ZooKeeper 3.7.0 Source Release` that can download at `https://zookeeper.apache.org/releases.html`.

Configurations

To start ZooKeeper we need a configuration file. In folder `[zookeeper dir]/conf` we create file `zoo.cfg`.

Listing 4 Configuration of **zoo.cfg**

```
    tickTime = 2000
    initLimit = 10
    syncLimit = 5
    clientPort = 2181
```

The meanings for each of these fields:

- **tickTime**: the basic time unit in milliseconds used by ZooKeeper. It is used to do heartbeats and the minimum session timeout will be twice the tickTime.
- **initLimit**: is the timeouts ZooKeeper uses to limit the length of time the ZooKeeper servers in quorum have to connect to a leader.
- **syncLimit**: specifies the limits how far out of date a server can be from a leader.
- **clientPort**: the port to listen for client connections

Modify ZooKeeper's source code

Class MatchingScheme

This class represents the server part of encrypted matching scheme which includes two functions: `ZEnc` and `ms_compare`. Moreover, this class also keeps the required keys of function `ZEnc` and other constants in ZooKeeper such as paths link to topics or brokers.

Listing 5 Class `MatchingScheme`

```java
public class MatchingSchema {
    public static final String TOPICS_PATH = "/brokers/topics";
    public static final String BROKERS_PATH = "/brokers/ids";
    public static final BigInteger N = new BigInteger("3337");
    public static final int EI2 = 2173;
    public static final int EJ2 = 2467;

    private static String ZEnc(String str, int e) {...}
    public static Boolean ms_compare(String str1, String str2) {...}
}
```

We decided to place this class in package `org.apache.zookeeper.encryption`.

Function `getBrokersAddressList`

As we were known, a host can have many brokers that are functioning at the same time and every broker's information is stored in ZooKeeper. Therefore, we design this function in order to help the Producer and Consumer get the address of all available brokers and choose a compatible broker to connect with.

Listing 6 Function `getBrokersAddressList`

```java
public List<String> getBrokersAddressList() {
    List<String> brokersInfo = new ArrayList<>();
    try {
        List<String> ids = getChildren(MatchingSchema.BROKERS_PATH, false);
        for (String id : ids) {
            String brokerInfo = new String(getData(MatchingSchema.BROKERS_PATH + "/"
            ↪  + id, false, null));
            brokersInfo.add(brokerInfo);
        }
    } catch (Exception exception) {
        exception.printStackTrace();
    }
    return brokersInfo;
}
```

Function enc_match

In the scope of the project, we consider that the Producer's ciphertext generated by function Enc is enc_topic_name. Producers also use enc_topic_name as the topic name saved in Kafka brokers to publish messages. Meanwhile, the Consumer's ciphertext generated by function Enc will be called trapdoor. And Consumers will use the trapdoor to get the corresponding enc_topic_name from ZooKeeper.

The function enc_match that places on ZooKeeper is used to find the correct enc_topic_name in the list of existing topics with the consumer trapdoor. Moreover, to optimize the number of connections, this function not only returns the enc_topic_name but also the leader broker address of the first partition of the topic.

Listing 7 Function enc_match

```java
public String enc_match(String trapdoor) {
    try {
        trapdoor = trapdoor.substring(trapdoor.lastIndexOf('/') + 1);
        //get list topics
        List<String> topicsList = getChildren(MatchingSchema.TOPICS_PATH, false);
        for (String exist_topic : topicsList) {
            //get topic name
            exist_topic = exist_topic.substring(exist_topic.lastIndexOf('/') + 1);
            if (MatchingSchema.ms_compare(exist_topic, trapdoor)) {
                //get partitions
                String partitionspath = "/brokers/topics/" + exist_topic +
                ↪  "/partitions";
                List<String> partitionList = getChildren(partitionspath, false);
                //get broker leader address
                String partitionPath =
                ↪  partitionspath+"/"+partitionList.get(0)+"/state";
                String partitionInfo = new String(getData(partitionPath,false,null));
                JSONObject jsonStr = new JSONObject(partitionInfo);
                int brokerID = jsonStr.getInt("leader");
                JSONObject brokerInfo = new
                ↪  JSONObject(getBrokersAddressList().get(brokerID));
                String brokerAddress = brokerInfo.getString("host") + ":" +
                ↪  brokerInfo.getInt("port");
                //create response string
                String response_str = "{\"topic_name\":\"" + exist_topic +
                ↪  "\",\"brokerAddress\":\"" + brokerAddress+ "\"}";
                return response_str;
            }
        }
    } catch (Exception exception) {
        exception.printStackTrace();
        return exception.getMessage();
    }
    return "Topic not found";
}
```

In this implementation, we always get the broker address from the first available partition of the topic. To begin, we get partition details by path `/brokers/topics/[topic name]/partitions/0/state` and we have a JSON string whose structure is as below.

```
{"controller_epoch":47242, "leader":0, "version":1, "leader_epoch":0,
"isr":[0]}
```

In the further step, we use the value of field `leader` as the `broker_id` to get the broker information. The broker information is also presented as a JSON string.

```
{"features":{}, "jmx_port":-1, "port":9092, "host":"127.0.0.1",
"listener_security_protocol_map":{"PLAINTEXT":"PLAINTEXT"}, "version":5,
"endpoints":["PLAINTEXT://127.0.0.1:9092"], "timestamp":"1620339443318"}
```

From the broker information, we generate the broker address with two fields `host` and `port` (for example, we have broker address is `127.0.0.1:9092`). Finally, to finish the function, we create a JSON response including the topic name and broker address and return it to the client.

Build ZooKeeper

Whenever we change the source code of ZooKeeper, we should re-build the project by the following command with "superuser" permission:

```
~/[zookeeper dir]$ sudo  mvn clean install
```

And then, we will use the new `zookeeper-3.7.0.jar` (that includes our changed code) as a dependency of Consumer and Producer projects.

Start and manage ZooKeeper

To start ZooKeeper, we open command prompt and type this command:

```
~/[zookeeper dir]$ bin/zkServer.sh start
```

Now, we can run `zkCli` which is a very simple command-line which can be used to manage ZooKeeper storage by the following command:

```
~/[zookeeper dir]$ bin/zkCli.sh -server 127.0.0.1:2181
```

The full version of ZooKeeper after adapting encrypted matching scheme can be downloaded at `https://git.unilim.fr/nguyen132/ADL-zookeeper-3.7.0`.

3.3 Kafka server

With Kafka, we use binary version with Scala 2.13: `kafka_2.13-2.8.0.tgz` that can be downloaded at `https://kafka.apache.org/downloads`.

After extracting Kafka project, in the file `sever.properties`, we add `listeners` parameter and change value of `zookeeper.connect` as follows:

Listing 8 Configuration of **server.properties**

```
############################# Server Basics #############################
broker.id=0
######################### Socket Server Settings ########################
listeners = PLAINTEXT://127.0.0.1:9092
num.network.threads=3
num.io.threads=8
socket.send.buffer.bytes=102400
socket.receive.buffer.bytes=102400
socket.request.max.bytes=104857600
############################### Log Basics ##############################
log.dirs=/tmp/kafka-logs
num.partitions=1
num.recovery.threads.per.data.dir=1
####################### Internal Topic Settings  ########################
offsets.topic.replication.factor=1
transaction.state.log.replication.factor=1
transaction.state.log.min.isr=1
######################### Log Retention Policy ##########################
log.retention.hours=168
log.segment.bytes=1073741824
log.retention.check.interval.ms=300000
############################### ZooKeeper ###############################
zookeeper.connect=127.0.0.1:2181
zookeeper.connection.timeout.ms=18000
###################### Group Coordinator Settings #######################
group.initial.rebalance.delay.ms=0
```

Here we need to pay attention that Kafka can start only when ZooKeeper is running. Therefore, after starting Zookeeper, we should open another command prompt to start Kafka server with the following command:

```
~/[kafka dir]$ bin/kafka-server-start.sh config/server.properties
```

3.4 Producer

Producers are the components that create messages and push them to the topic partitions in brokers. With the objective of securing the event streaming in our project, we designed the Producer's workflow as Figure 3.4. The coding implementation has been done also based on this.

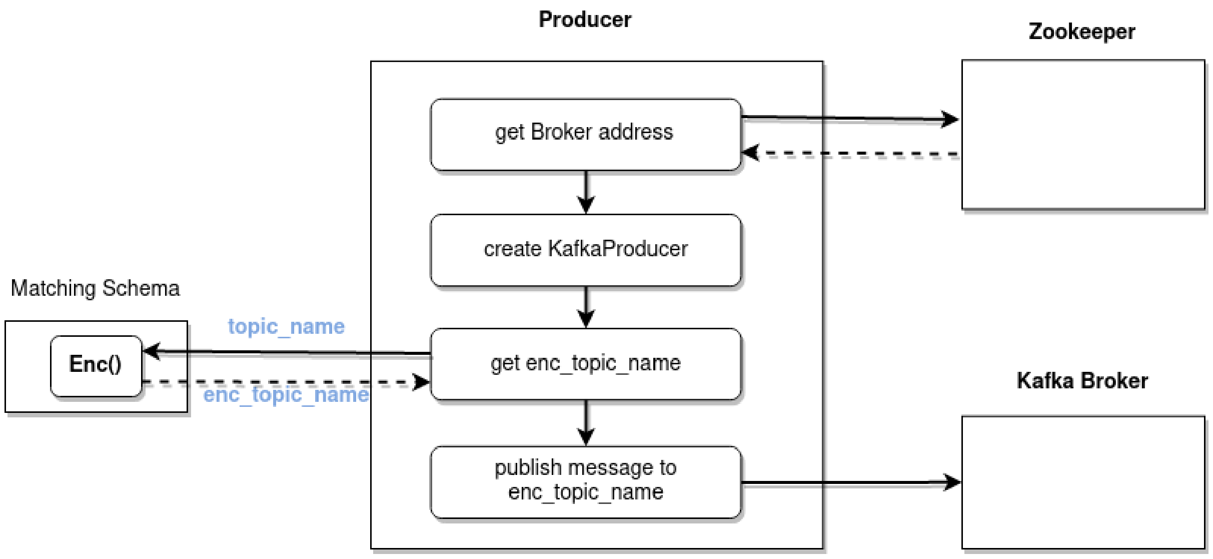


Figure 3.4: Workflow of Producer

Get broker's address

At the beginning, a Producer will send a request to ZooKeeper to get information on all available brokers at this moment. And then the Producer can choose one of them to connect with. However, in this implementation, we choose the first broker as the default option.

Listing 9 Get broker's address

```
    //get all available brokers
    ZooKeeper zk_client = new ZooKeeper(ZOOKEEPER_ADDRESS, 10000,null);
    List<String> brokersAddressList = zk_client.getBrokersAddressList();
    //using first broker
    JSONObject json = new JSONObject(brokersAddressList.get(0));
    String brokerAdress = json.getString("host")+":"+json.getInt("port");
    System.out.println("Connecting to "+brokerAdress+"...");
```

Create a Kafka Producer

Listing 10 Create a producer

```
//create producer
Properties props = new Properties();
props.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerAdress);
props.put(ProducerConfig.CLIENT_ID_CONFIG, "myProducer");
props.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG,
↪ LongSerializer.class.getName());
props.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG,
↪ StringSerializer.class.getName());
KafkaProducer<Long, String> producer = new KafkaProducer<>(props);
```

In this implementation, we decide to create the simplest Producer that only has the required configurations. And the broker address that we obtained from the previous step will be used for `BOOTSTRAP_SERVERS_CONFIG`.

Get encrypted topic name

As we shown in Figure 3.4, from a normal topic name, Producer will call function `Enc` of encrypted matching schema to get `enc_topic_name`. The encryption key of Producer is `EI1`.

Listing 11 Get encrypted topic name

```
//get encrypted topic name
String enc_topic_name = MatchingSchema.Enc("topic-2",MatchingSchema.EI1);
System.out.println("Publish to "+enc_topic_name);
```

Publish data

The Producer will use the `enc_topic_name` to publish messages. Here, we set up the Producer to publish the message about 10 times and after sending a message, he will wait 4 seconds to send the new one. For the message records, we decided to use the timestamp as the key and the record's content will have the same format `"Hello at ..."`.

Listing 12 Producer publishes data to topic

```java
//Publish message to topic
long time = System.currentTimeMillis();
try {
    for (long index = time; index < time + 10; index++) {
        ProducerRecord<Long, String> record = new
        ↪  ProducerRecord<>(enc_topic_name, index,"Hello at " + index);
        RecordMetadata metadata = producer.send(record).get();
        long elapsedTime = System.currentTimeMillis() - time;
        System.out.printf("Sent record: (key=%s value=%s) meta(partition=%d,
        ↪  offset=%d) time=%d\n", record.key(), record.value(),
        ↪  metadata.partition(), metadata.offset(), elapsedTime);
        Thread.sleep(4000);
    }
} finally {
    producer.flush();
    producer.close();
}
```

The full version of Producer after adapting encrypted matching scheme can be downloaded at `https://git.unilim.fr/nguyen132/ADL-producer`.

3.5 Consumer

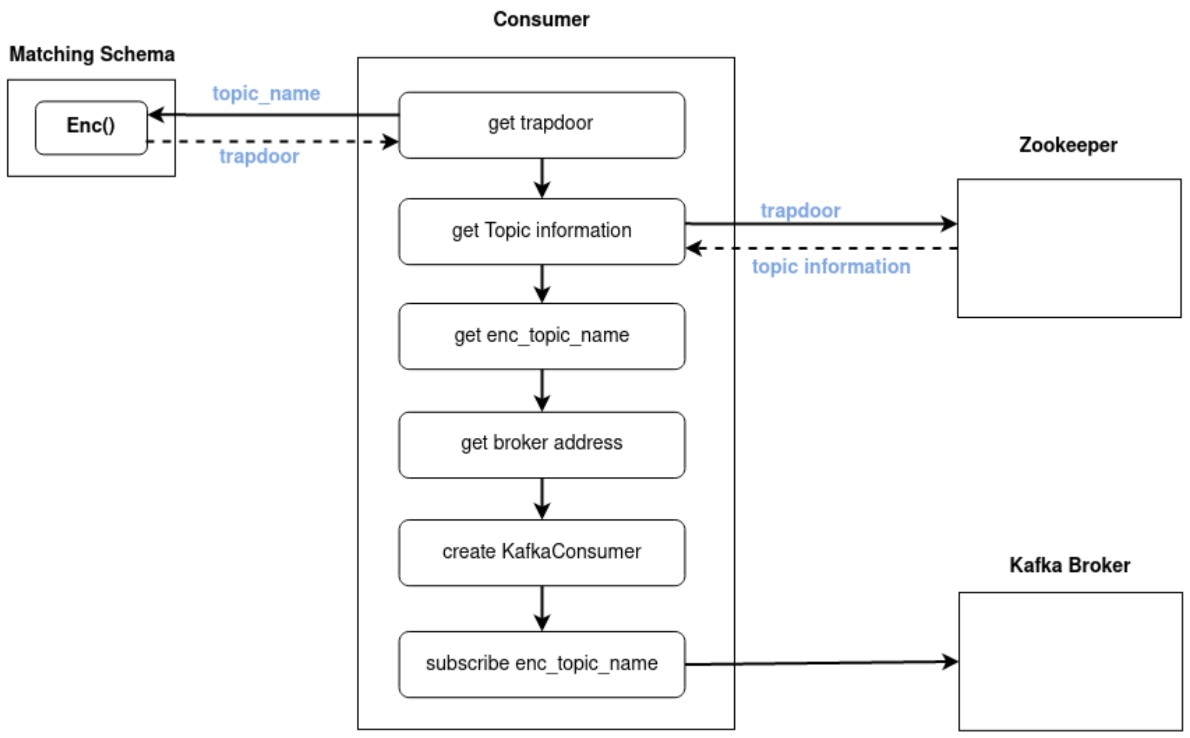


Figure 3.5: Workflow of Consumer

While Kafka Producers write to topics, Kafka consumers read from topics. The Consumer subscribes to one or more topics and reads the messages in the order in which they were produced. The Figure 3.5 presents the workflow of Consumer in the system that already implemented encrypted matching schema. And as same as Producer, we also follow the Consumer's workflow to finish the implementation.

The full version of Consumer after adapting encrypted matching scheme can be downloaded at `https://git.unilim.fr/nguyen132/ADL-consumer`.

Generate trapdoor

As the short disscussion in section 3.2, the `trapdoor` of Consumer is the ouput of function `Enc` and will be used to get the `enc_topic_name`. The key used to encrypt data of Consumer is `EJ1`.

Listing 13 Generate trapdoor

```java
    //generate trapdoor
    String enc_topic_name = MatchingSchema.Enc("topic-2", MatchingSchema.EJ1);
    System.out.println("Trapdoor is "+enc_topic_name);
```

Get topic name and broker's address from ZooKeeper

Listing 14 Get topic name and broker's address

```java
    //get topic info
    ZooKeeper zk_client = new ZooKeeper(ZOOKEEPER_ADDRESS, 10000, null);
    String topicInfo = zk_client.enc_match(enc_topic_name);
    //get topic name
    JSONObject json = new JSONObject(topicInfo);
    String topic_name = json.getString("topic_name");
    //get broker address
    String brokerAdress = json.getString("brokerAddress");
    System.out.println("Connecting to " + brokerAdress + "...");
```

We use the `trapdoor` to get the topic's information from ZooKeeper. The response is a Json string that includes topic name and broker's address. An example of the response:

```
{"topic_name":"058402c700ea051f...","brokerAddress":"127.0.0.1:9092"}
```

In the case that Zookeeper can not find the `enc_topic_name`, the response result will be:

```
Topic not found
```

Create a Consumer

Although it is possible to create Consumers which do not belong to any consumer group, but this is uncommon. Therefore, for this implementation, we will assume the Consumer is a part of a group named "groupConsumer".

Listing 15 Create a consumer

```java
//create consumer
Properties props = new Properties();
props.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, brokerAdress);
props.put(ConsumerConfig.GROUP_ID_CONFIG, "groupConsumer");
props.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG,
↪ LongDeserializer.class.getName());
props.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG,
↪ StringDeserializer.class.getName());
Consumer<Long, String> consumer = new KafkaConsumer<>(props);
```

Subscribe a topic

Once we created a consumer, the next is to subscribe to the `enc_topic_name`.

Listing 16 Consumer subscribes to a topic

```java
//subscribe topic
System.out.println("Subscribe topic "+topic_name);
consumer.subscribe(Collections.singletonList(topic_name));
int giveUp = 50;
int noRecordsCount = 0;
while (true) {
    final ConsumerRecords<Long, String> consumerRecords =
    ↪ consumer.poll(Duration.ofSeconds(1));
    if (consumerRecords.count() == 0) {
        noRecordsCount++;
        if (noRecordsCount > giveUp) break;
        else continue;
    }
    consumerRecords.forEach(record -> {
        System.out.printf("Received record: (key=%s value=%s) meta(partition=%d,
        ↪ offset=%d)\n", record.key(), record.value(), record.partition(),
        ↪ record.offset());
    });
    consumer.commitAsync();
}
consumer.close();
System.out.println("DONE");
```

Chapter 4

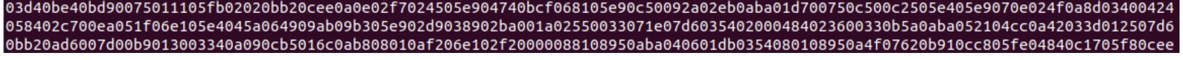
Experimental Results

As presented, the project consists in studying the functioning as well as the security of Apache Kafka, then adapting an encrypted matching scheme in order to hide the “topics” from the server with the aim of reinforcing the confidentiality of the exchanges while allowing authorized users to subscribe to their feeds.

For matching scheme, based on the security’s proofs indicated in section 2.2, we decided to implement the **RSA - Based Proxy Encryption Scheme** [3] in Zookeeper. Therefore, once a consumer wants to get messages, he will ask Zookeeper where his topic is to subscribe by sending his desired topic. The Encryption Scheme then does the matching between the encoded topic sent by the subscriber and the one encoded by the publisher.

In this chapter, we will show experimental results of adapting RSA - Based Proxy Encryption Scheme in Zookeeper and the data stream returned.

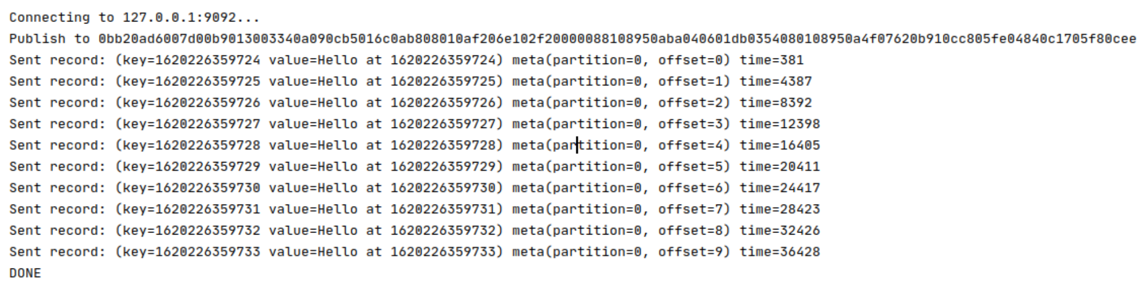
Roughly speaking, what we have done is firstly to interfere in topic-sending process of producers. Now instead of publishing an understandable string or cleartext, they will send its encrypted string. One example is on Figure 4.1, where the three hexadecimal strings are ciphertexts of plain texts: “topic-1”, “topic-2”, “topic-3”.



```
dvthao@dvthao:~/Desktop/Kafka/kafka_2.13-2.8.0$ bin/kafka-topics.sh --zookeeper localhost:2181 --list
03d40be40bd90075011105fb02020bb20cee0a0e02f7024505e904740bcf068105e90c50092a02eb0aba01d700750c500c2505e405e9070e024f0a8d03400424
058402c700ea051f06e105e4045a064909ab09b305e902d9038902ba001a02550033071e07d6035402000484023600330b5a0aba052104cc0a42033d012507d6
0bb20ad6007d00b9013003340a090cb5016c0ab808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805fe04840c1705f80cee
```

Figure 4.1: List of encrypted topic

The Figure 4.2 below will illustrate that under RSA - Based Proxy Encryption Scheme (presented in Figure 2.3), a ciphertext, i.e. `0bb20ad6007d00b9013003340a090cb5016c0a b808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805f e04840c1705f80cee`, is also the name saved in Kafka brokers to identify the topic, as well as the name returned to a customer when he wants to read messages from it.



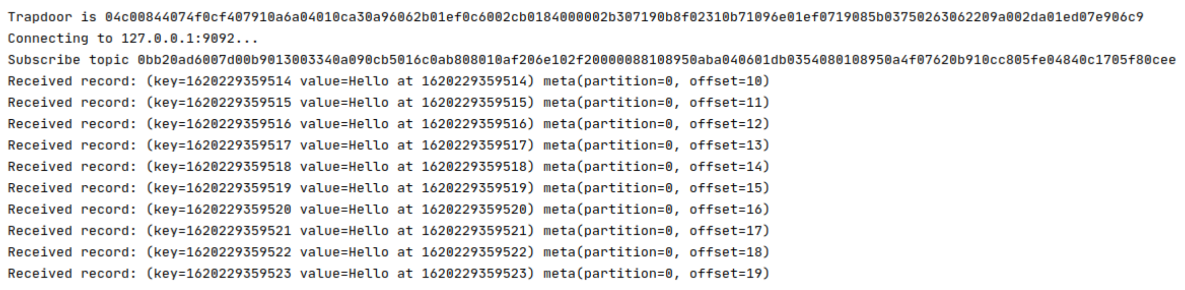
```
Connecting to 127.0.0.1:9092...
Publish to 0bb20ad6007d00b9013003340a090cb5016c0ab808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805fe04840c1705f80cee
Sent record: (key=1620226359724 value=Hello at 1620226359724) meta(partition=0, offset=0) time=381
Sent record: (key=1620226359725 value=Hello at 1620226359725) meta(partition=0, offset=1) time=4387
Sent record: (key=1620226359726 value=Hello at 1620226359726) meta(partition=0, offset=2) time=8392
Sent record: (key=1620226359727 value=Hello at 1620226359727) meta(partition=0, offset=3) time=12398
Sent record: (key=1620226359728 value=Hello at 1620226359728) meta(partition=0, offset=4) time=16405
Sent record: (key=1620226359729 value=Hello at 1620226359729) meta(partition=0, offset=5) time=20411
Sent record: (key=1620226359730 value=Hello at 1620226359730) meta(partition=0, offset=6) time=24417
Sent record: (key=1620226359731 value=Hello at 1620226359731) meta(partition=0, offset=7) time=28423
Sent record: (key=1620226359732 value=Hello at 1620226359732) meta(partition=0, offset=8) time=32426
Sent record: (key=1620226359733 value=Hello at 1620226359733) meta(partition=0, offset=9) time=36428
DONE
```

Figure 4.2: A producer sending records (1)

Each record is in the format: `(key=1620226359724 value=Hello at 1620226359724) meta(partition=0, offset=0) time=381`, where:

- `key` is the key included in the record to ensures that all messages generated with a given key will be written to the same single partition.
- `value` is record content;
- `partition` is the partition count;
- `offset` is a number assigned by Kafka brokers to received messages from producers;
- `time` is a timestamp, which is set to print after each 4000 milliseconds in Figure 4.2.

On the side of a customer, in case he makes a command to read records from the same topic on Figure 4.2, Zookeeper will get him subscribed with the results on Figure 4.3:



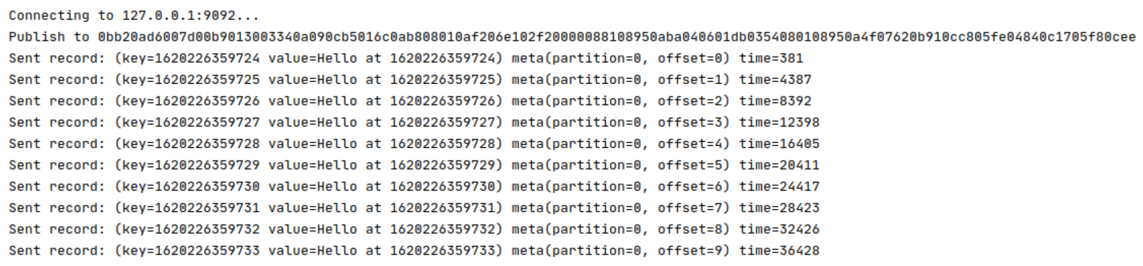
```
Trapdoor is 04c00844074f0cf407910a6a04010ca30a96062b01ef0c6002cb0184000002b307190b8f02310b71096e01ef0719085b03750263062209a002da01ed07e906c9
Connecting to 127.0.0.1:9092...
Subscribe topic 0bb20ad6007d00b9013003340a090cb5016c0ab808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805fe04840c1705f80cee
Received record: (key=1620229359514 value=Hello at 1620229359514) meta(partition=0, offset=10)
Received record: (key=1620229359515 value=Hello at 1620229359515) meta(partition=0, offset=11)
Received record: (key=1620229359516 value=Hello at 1620229359516) meta(partition=0, offset=12)
Received record: (key=1620229359517 value=Hello at 1620229359517) meta(partition=0, offset=13)
Received record: (key=1620229359518 value=Hello at 1620229359518) meta(partition=0, offset=14)
Received record: (key=1620229359519 value=Hello at 1620229359519) meta(partition=0, offset=15)
Received record: (key=1620229359520 value=Hello at 1620229359520) meta(partition=0, offset=16)
Received record: (key=1620229359521 value=Hello at 1620229359521) meta(partition=0, offset=17)
Received record: (key=1620229359522 value=Hello at 1620229359522) meta(partition=0, offset=18)
Received record: (key=1620229359523 value=Hello at 1620229359523) meta(partition=0, offset=19)
```

Figure 4.3: A consumer receiving records (1)

where each parameters in a `Received record` is corresponding to `record.key()`, `record.value()`, `record.partition()`, `record.offset())` respectively.

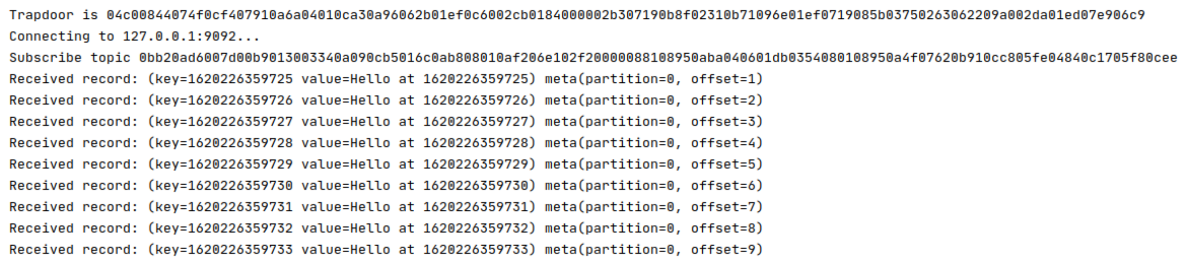
When there are many brokers, Zookeeper needs to indicate the location of the broker which contains the addressed topic. Figure 4.3 showed that `127.0.0.1:9092` is the broker's address returned by Zookeeper to the consumer. In addition, the `Trapdoor` is equivalent to $m^{e_{j1}}$ being searched topic by this customer.

As noted, event streaming is the practice of capturing data in real-time from event sources like databases, sensors, mobile devices, cloud services, and software applications in the form of streams of events [12]. Event streaming thus ensures a continuous flow and interpretation of data so that the right information is at the right place, at the right time. The example below will depict the flow of record streaming between a producer and customer subscribing on a same topic.



```
Connecting to 127.0.0.1:9092...
Publish to 0bb20ad6007d00b9013003340a090cb5016c0ab808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805fe04840c1705f80cee
Sent record: (key=1620226359724 value=Hello at 1620226359724) meta(partition=0, offset=0) time=381
Sent record: (key=1620226359725 value=Hello at 1620226359725) meta(partition=0, offset=1) time=4387
Sent record: (key=1620226359726 value=Hello at 1620226359726) meta(partition=0, offset=2) time=8392
Sent record: (key=1620226359727 value=Hello at 1620226359727) meta(partition=0, offset=3) time=12398
Sent record: (key=1620226359728 value=Hello at 1620226359728) meta(partition=0, offset=4) time=16405
Sent record: (key=1620226359729 value=Hello at 1620226359729) meta(partition=0, offset=5) time=20411
Sent record: (key=1620226359730 value=Hello at 1620226359730) meta(partition=0, offset=6) time=24417
Sent record: (key=1620226359731 value=Hello at 1620226359731) meta(partition=0, offset=7) time=28423
Sent record: (key=1620226359732 value=Hello at 1620226359732) meta(partition=0, offset=8) time=32426
Sent record: (key=1620226359733 value=Hello at 1620226359733) meta(partition=0, offset=9) time=36428
```

Figure 4.4: A producer sending records (2)

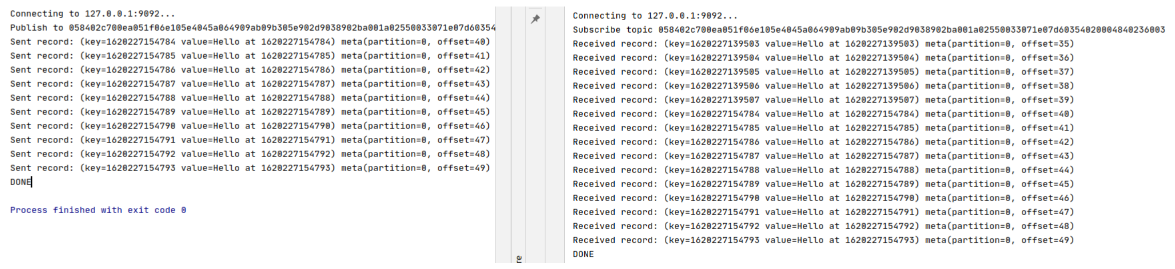


```
Trapdoor is 04c00844074f0cf407910a6a04010ca30a96062b01ef0c6002cb0184000002b307190b8f02310b71096e01ef0719085b03750263062209a002da01ed07e906c9
Connecting to 127.0.0.1:9092...
Subscribe topic 0bb20ad6007d00b9013003340a090cb5016c0ab808010af206e102f20000088108950aba040601db0354080108950a4f07620b910cc805fe04840c1705f80cee
Received record: (key=1620226359725 value=Hello at 1620226359725) meta(partition=0, offset=1)
Received record: (key=1620226359726 value=Hello at 1620226359726) meta(partition=0, offset=2)
Received record: (key=1620226359727 value=Hello at 1620226359727) meta(partition=0, offset=3)
Received record: (key=1620226359728 value=Hello at 1620226359728) meta(partition=0, offset=4)
Received record: (key=1620226359729 value=Hello at 1620226359729) meta(partition=0, offset=5)
Received record: (key=1620226359730 value=Hello at 1620226359730) meta(partition=0, offset=6)
Received record: (key=1620226359731 value=Hello at 1620226359731) meta(partition=0, offset=7)
Received record: (key=1620226359732 value=Hello at 1620226359732) meta(partition=0, offset=8)
Received record: (key=1620226359733 value=Hello at 1620226359733) meta(partition=0, offset=9)
```

Figure 4.5: A consumer receiving records (2)

In Figure 4.4, a publisher sent records of one same topic from offset value 0. However, this offset was not received by the customer on Figure 4.5 because he subscribed to the topic at the moment when the producer already streamed the first offset. Thus, the offsets gotten started from 1.

One benefit of this publish / subscribe architecture is that the data will not be loss when the customer already subscribed to a topic, but he stops his streaming in between. Whenever he continues his work, the record will be sent from the offset where he resumed as showed on Figure 4.6.



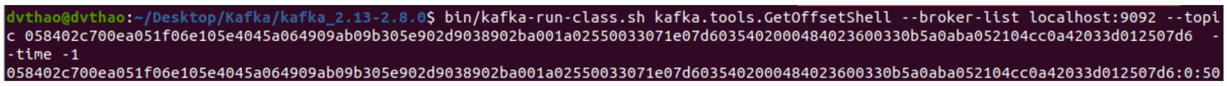
```
Connecting to 127.0.0.1:9092...
Publish to 058402c700ea051f06e105e4045a064909ab09b305e902d9038902ba001a02550033071e07d60354
Sent record: (key=1620227154784 value=Hello at 1620227154784) meta(partition=0, offset=40)
Sent record: (key=1620227154785 value=Hello at 1620227154785) meta(partition=0, offset=41)
Sent record: (key=1620227154786 value=Hello at 1620227154786) meta(partition=0, offset=42)
Sent record: (key=1620227154787 value=Hello at 1620227154787) meta(partition=0, offset=43)
Sent record: (key=1620227154788 value=Hello at 1620227154788) meta(partition=0, offset=44)
Sent record: (key=1620227154789 value=Hello at 1620227154789) meta(partition=0, offset=45)
Sent record: (key=1620227154790 value=Hello at 1620227154790) meta(partition=0, offset=46)
Sent record: (key=1620227154791 value=Hello at 1620227154791) meta(partition=0, offset=47)
Sent record: (key=1620227154792 value=Hello at 1620227154792) meta(partition=0, offset=48)
Sent record: (key=1620227154793 value=Hello at 1620227154793) meta(partition=0, offset=49)
DONE

Process finished with exit code 0
```

```
Connecting to 127.0.0.1:9092...
Subscribe topic 058402c700ea051f06e105e4045a064909ab09b305e902d9038902ba001a02550033071e07d6035402000484023600З
Received record: (key=1620227139503 value=Hello at 1620227139503) meta(partition=0, offset=35)
Received record: (key=1620227139504 value=Hello at 1620227139504) meta(partition=0, offset=36)
Received record: (key=1620227139505 value=Hello at 1620227139505) meta(partition=0, offset=37)
Received record: (key=1620227139506 value=Hello at 1620227139506) meta(partition=0, offset=38)
Received record: (key=1620227139507 value=Hello at 1620227139507) meta(partition=0, offset=39)
Received record: (key=1620227154784 value=Hello at 1620227154784) meta(partition=0, offset=40)
Received record: (key=1620227154785 value=Hello at 1620227154785) meta(partition=0, offset=41)
Received record: (key=1620227154786 value=Hello at 1620227154786) meta(partition=0, offset=42)
Received record: (key=1620227154787 value=Hello at 1620227154787) meta(partition=0, offset=43)
Received record: (key=1620227154788 value=Hello at 1620227154788) meta(partition=0, offset=44)
Received record: (key=1620227154789 value=Hello at 1620227154789) meta(partition=0, offset=45)
Received record: (key=1620227154790 value=Hello at 1620227154790) meta(partition=0, offset=46)
Received record: (key=1620227154791 value=Hello at 1620227154791) meta(partition=0, offset=47)
Received record: (key=1620227154792 value=Hello at 1620227154792) meta(partition=0, offset=48)
Received record: (key=1620227154793 value=Hello at 1620227154793) meta(partition=0, offset=49)
DONE
```

Figure 4.6: Data streaming between a producer and consumer

As we all know, each new record in the partition gets an offset number which is one more than its previous number. In order to control streaming flow, the latest offset still in the topic which has not been published is also updated by Kafka. Figure 4.7 indicated that the 50th offset will be published next into the server.



```
dvthao@dvthao:~/Desktop/Kafka/kafka_2.13-2.8.0$ bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topi
c 058402c700ea051f06e105e4045a064909ab09b305e902d9038902ba001a02550033071e07d603540200048402360033 0b5a0aba052104cc0a42033d012507d6  -
-time -1
058402c700ea051f06e105e4045a064909ab09b305e902d9038902ba001a02550033071e07d6035402000484023600330b5a0aba052104cc0a42033d012507d6:0:50
```

Figure 4.7: The latest offset in the topic

In other case when a customer wants to get data a topic which has not been created in Kafka brokers, Zookeeper will return a message as “non-existent topic”.

Chapter 5

Conclusion and Future Work

5.1 Conclusion

In the scope of the project, we successfully adapted an encrypted matching scheme called RSA - Based Proxy Encryption Scheme in Zookeeper system. The main objective of this implementation is to hide the "topics" from the Kafka server to reinforce the confidentiality of the exchanges while allowing authorized users to subscribe to their feeds. In the adapted scheme, the server supposed to be untrusted is not able to see the content of the topic, which is encoded. The meaning of the hiding process is to reduce the risk of revealing secretly valuable data. In case the record content is leaked to an intruder, he can not see the meaning of these numeral records without a topic name. However, we can deny that if the record is being ciphered, the system security will be improved, this part will be presented in subsection 5.2.1.

Besides of returning enciphered topic to a consumer, we made changes in order that Zookeeper also sends back the location of a Kafka broker who keeps data of the same topic.

In chapter 2, after performing problem analysis, we discussed different matching schemes as well as conducted the initial assessments on their security related to Kafka and Zookeeper systems. A proof of concept is presented in section 2.2 to evaluate the chosen solution.

Chapter 3 includes step-by-step implementation of adapting RSA - Based Proxy Encryption Scheme in Zookeeper, together with coding changes made in different classes of Zookeeper and Kafka.

5.2 Future Work

5.2.1 Record encryption

In the course of conducting this project, other questions arose. Indeed, we wondered if our program was more secure if we can encode not only the topic but also the records.

In fact, there is a possibility that the records can also be encrypted by RSA - Based Proxy Encryption. The initial idea is that, suppose a publisher, called user i, has record r, by using same method, he calculates $r^{e_{i1}}$ and publish to his topic in a Kafka server. From this moment, the Kafka server have no idea what the topic and record's contents are. When a consumer, called user j, wants to get record r, the server will compute $r^{e_{i1}*e_{i2}*d_{j2}} = r^{e*d_{j2}}$ and send to user i. User i now can use his private key to get the original message $r^{e*d_{j2}*d_{j1}} = r^{e*d} = r$ (see detail in Figure 2.3). The process is possible; however, this would make big changes on the Kafka server in order to calculate $r^{e*d_{j2}}$. Moreover, RSA-based encryption takes a long computation time and thus is for a not-too-large plaintext. In fact, RSA is usually only applied in digital signatures because before signing, plaintext has been hashed to a certain size. Therefore, it is necessary for another assessment on this initial idea as well as looking for a better solution.

5.2.2 Key auto-generation

As presented in chapter 2, the implemented proxy is built based on RSA technique, where there are private and public keys. On the step of *UGen*, the set of e_{i1} , e_{i2}, d_{i1}, and d_{i2} are found. While (e_{i1}, d_{i1}) are kept by client i, (e_{i2}, d_{i2}) are kept by the server. In a scope of a company, one independent team or department will use the same public key e, n, d, whereas each team member owns his different private key (e_{i1}, d_{i1}). For the moment we manually calculate these numbers for each client. In the ongoing studies, we would like to implement a master key generation algorithm to produce more secure parameters in the standard RSA.

5.2.3 Integrated PIR

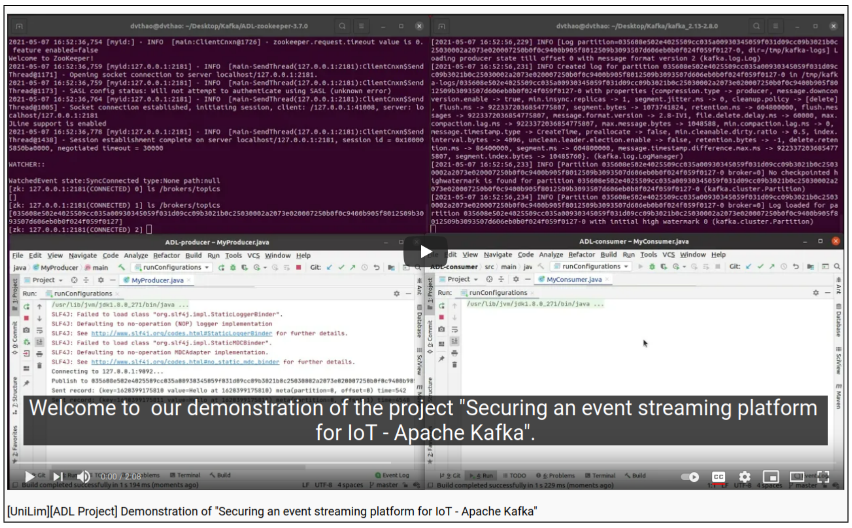
In the effort of improving our scheme's security, we look for investigating more in our matching scheme. The initial stage is to understand the private information retrieval (PIR) protocol, firstly applied in 1997 by Kushilevitz and Ostrovsky. PIR [5] allows a user to retrieve an element of a database without the owner of that database being able to determine which item is retrieved. In our scheme, we assume that authorised users are fully trusted. However, in reality an intruder can be an outsider or even a untrustworthy employee who can give access to an outsider. By integrating PIR, we expect to restrict the conspiracy of gathering direct or indirect information about the stored data.

Appendix

5.3 Demonstration video

The following is our video of the demonstration for this project. In the video, we present step by step to run our implementation, as well as show experimental results. Please consider to turn on the "Subtitle" of the video to get detail information.

Youtube video link: `https://youtu.be/xaq_OREwj3g`.



dvthao@dvthao: ~/Desktop/Kafka/ADL-zookeeper-3.7.0

```
2021-05-07 16:52:36,754 [myid:] - INFO  [main:ClientCnxn@1726] - zookeeper.request.timeout value is 0.
 feature enabled=false
Welcome to ZooKeeper!
2021-05-07 16:52:36,759 [myid:127.0.0.1:2181] - INFO  [main-SendThread(127.0.0.1:2181):ClientCnxn$Send
Thread@1171] - Opening socket connection to server localhost/127.0.0.1:2181.
2021-05-07 16:52:36,759 [myid:127.0.0.1:2181] - INFO  [main-SendThread(127.0.0.1:2181):ClientCnxn$Send
Thread@1173] - SASL config status: Will not attempt to authenticate using SASL (unknown error)
2021-05-07 16:52:36,764 [myid:127.0.0.1:2181] - INFO  [main-SendThread(127.0.0.1:2181):ClientCnxn$Send
Thread@1005] - Socket connection established, initiating session, client: /127.0.0.1:41008, server: lo
calhost/127.0.0.1:2181
JLine support is enabled
2021-05-07 16:52:36,778 [myid:127.0.0.1:2181] - INFO  [main-SendThread(127.0.0.1:2181):ClientCnxn$Send
Thread@1438] - Session establishment complete on server localhost/127.0.0.1:2181, session id = 0x10000
5850ba0000, negotiated timeout = 30000

WATCHER::

WatchedEvent state:SyncConnected type:None path:null
[zk: 127.0.0.1:2181(CONNECTED) 0] ls /brokers/topics
[]
[zk: 127.0.0.1:2181(CONNECTED) 1] ls /brokers/topics
[035608e502e4025509cc035a00930345059f031d09cc09b3021b0c25030002a2073e020007250b0f0c9400b905f8012509b30
93507d606eb0b0f024f059f0127]
[zk: 127.0.0.1:2181(CONNECTED) 2]
```

dvthao@dvthao: ~/Desktop/Kafka/kafka_2.13-2.8.0

```
[2021-05-07 16:52:56,229] INFO [Log partition=035608e502e4025509cc035a00930345059f031d09cc09b3021b0c
25030002a2073e020007250b0f0c9400b905f8012509b3093507d606eb0b0f024f059f0127-0, dir=/tmp/kafka-logs] L
oading producer state till offset 0 with message format version 2 (kafka.log.Log)
[2021-05-07 16:52:56,231] INFO Created log for partition 035608e502e4025509cc035a00930345059f031d09c
c09b3021b0c25030002a2073e020007250b0f0c9400b905f8012509b3093507d606eb0b0f024f059f0127-0 in /tmp/kafk
a-logs/035608e502e4025509cc035a00930345059f031d09cc09b3021b0c25030002a2073e020007250b0f0c9400b905f80
12509b3093507d606eb0b0f024f059f0127-0 with properties {compression.type -> producer, message.downcon
version.enable -> true, min.insync.replicas -> 1, segment.jitter.ms -> 0, cleanup.policy -> [delete]
, flush.ms -> 9223372036854775807, segment.bytes -> 1073741824, retention.ms -> 604800000, flush.mes
sages -> 9223372036854775807, message.format.version -> 2.8-IV1, file.delete.delay.ms -> 60000, max.
compaction.lag.ms -> 9223372036854775807, max.message.bytes -> 1048588, min.compaction.lag.ms -> 0,
message.timestamp.type -> CreateTime, preallocate -> false, min.cleanable.dirty.ratio -> 0.5, index.
interval.bytes -> 4096, unclean.leader.election.enable -> false, retention.bytes -> -1, delete.reten
tion.ms -> 86400000, segment.ms -> 604800000, message.timestamp.difference.max.ms -> 922337203685477
5807, segment.index.bytes -> 10485760}. (kafka.log.LogManager)
[2021-05-07 16:52:56,233] INFO [Partition 035608e502e4025509cc035a00930345059f031d09cc09b3021b0c2503
0002a2073e020007250b0f0c9400b905f8012509b3093507d606eb0b0f024f059f0127-0 broker=0] No checkpointed h
ighwatermark is found for partition 035608e502e4025509cc035a00930345059f031d09cc09b3021b0c25030002a2
073e020007250b0f0c9400b905f8012509b3093507d606eb0b0f024f059f0127-0 (kafka.cluster.Partition)
[2021-05-07 16:52:56,234] INFO [Partition 035608e502e4025509cc035a00930345059f031d09cc09b3021b0c2503
0002a2073e020007250b0f0c9400b905f8012509b3093507d606eb0b0f024f059f0127-0 broker=0] Log loaded for pa
rtition 035608e502e4025509cc035a00930345059f031d09cc09b3021b0c25030002a2073e020007250b0f0c9400b905f8
012509b3093507d606eb0b0f024f059f0127-0 with initial high watermark 0 (kafka.cluster.Partition)
```

ADL-producer – MyProducer.java

```
/usr/lib/jvm/jdk1.8.0_271/bin/java ...
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See http://www.slf4j.org/codes.html#StaticLoggerBinder for further details.
SLF4J: Failed to load class "org.slf4j.impl.StaticMDCBinder".
SLF4J: Defaulting to no-operation MDCAdapter implementation.
SLF4J: See http://www.slf4j.org/codes.html#no_static_mdc_binder for further details.
Connecting to 127.0.0.1:9092...
Publish to 035608e502e4025509cc035a00930345059f031d09cc09b3021b0c25030002a2073e020007250b0f0c9400b90
Sent record: (key=1620399175810 value=Hello at 1620399175810) meta(partition=0, offset=0) time=542
```

ADL-consumer – MyConsumer.java

```
/usr/lib/jvm/jdk1.8.0_271/bin/java ...
```

Welcome to our demonstration of the project "Securing an event streaming platform for IoT - Apache Kafka".
0:00 / 2:08
Build completed successfully in 1 s 194 ms (moments ago)
[UniLim][ADL Project] Demonstration of "Securing an event streaming platform for IoT - Apache Kafka"

Figure 5.1: Demonstration video

5.4 Participation evaluation form

As required, we would like to attach the paricipation evaluation form as follows:

Securing an event streaming platform for IoT						
Student		Organisation, coordination		Production		
Last name	First name	Coordination of meetings	Active participation in meetings	Code	Final Report	Preparation for the defense
NGUYEN	Van Tien	5	5	5	5	4
NGUYEN	Thi Mai Phuong	4	5	5	5	5
DOAN	Thi Van Thao	5	5	4	5	5

Figure 5.2: Participation evaluation form

Bibliography

[1] S. Goldwasser and S. Micali, *Probabilistic encryption how to play mental poker keeping secret all partial information*, Annual ACM Symposium on Theory of Computing, 1982.

[2] Neha Narkhede, Gwen Shapira and Todd Palino. *Kafka - The Definitive Guide.* O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472., July 2017.

[3] Changyu Dong, Giovanni Russello, and Naranker Dulay. *Shared and Searchable Encrypted Data for Untrusted Servers.* Department of Computing, Imperial College London, 180 Queen's Gate, London, SW7 2AZ, UK, 2008.

[4] Yang, Z., Zhong, S., Wright, R.N. *Privacy-Preserving Queries on Encrypted Data.* In: Gollmann, D., Meier, J., Sabelfeld, A. (eds.) ESORICS 2006. LNCS, vol. 4189, pp. 479–495. Springer, Heidelberg, 2006.

[5] Chor, B., Goldreich, O., Kushilevitz, E., Sudan, M.: *Private information retrieval.* In: FOCS, pp. 41–50, 1995.

[6] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, 2004.

[7] Blaze, M., Bleumer, G., Strauss. *Divertible Protocols and Atomic Proxy Cryptography.* In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 127–144. Springer, Heidelberg, 1998.

[8] S. Goldwasser and M. Bellare. *Lecture notes on cryptography.* Summer CourseLecture Notes at MIT, 1999.

Websites:

[9] https://stackshare.io/kafka

[10] https://link.springer.com/content/pdf/10.1007%2F978-3-540-70567-3_10.pdf

[11] *Apache Kafka Overview.* Cloudera Runtime 7.1.1, 2019-12-18.
https://docs.cloudera.com/runtime/7.1.1/kafka-overview/kafka-overview.pdf

[12] https://kafka.apache.org/uses

[13] https://data-flair.training/blogs/

[14] https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

[15] https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying

[16] https://data-flair.training/blogs/zookeeper-tutorial/

[17] https://zookeeper.apache.org/doc/r3.1.2/zookeeperOver.html

[18] https://hadoopabcd.wordpress.com/2015/05/02/zookeeper-distributed-coordination-service/

[19] https://techvidvan.com/tutorials/apache-zookeeper-tutorial/

[20] https://web.archive.org/web/20131209063307/http://wiki.apache.org/hadoop/ZooKeeper /PoweredBy

[21] https://engineering.fb.com/2018/07/19/data-infrastructure/location-aware-distribution-configuring-servers-at-scale/

[22] https://blog.twitter.com/engineering/en_us/topics/infrastructure/2018/zookeeper-at-twitter.html

[23] https://lucene.apache.org/solr/guide/6_6/solrcloud.html

[24] https://data-flair.training/blogs/zookeeper-znodes/

[25] https://www.edureka.co/blog/zookeeper-tutorial/

[26] https://www.linkedin.com/pulse/role-apache-zookeeper-kafka-malini-shukla/

[27] https://medium.com/@rinu.gour123/role-of-apache-zookeeper-in-kafka-monitoring-configuration-c5bd1a7e4226

[28] https://www.tutorialspoint.com/zookeeper/zookeeper_workflow.htm

[29] https://www.tutorialspoint.com/zookeeper/zookeeper_quick_guide.htm