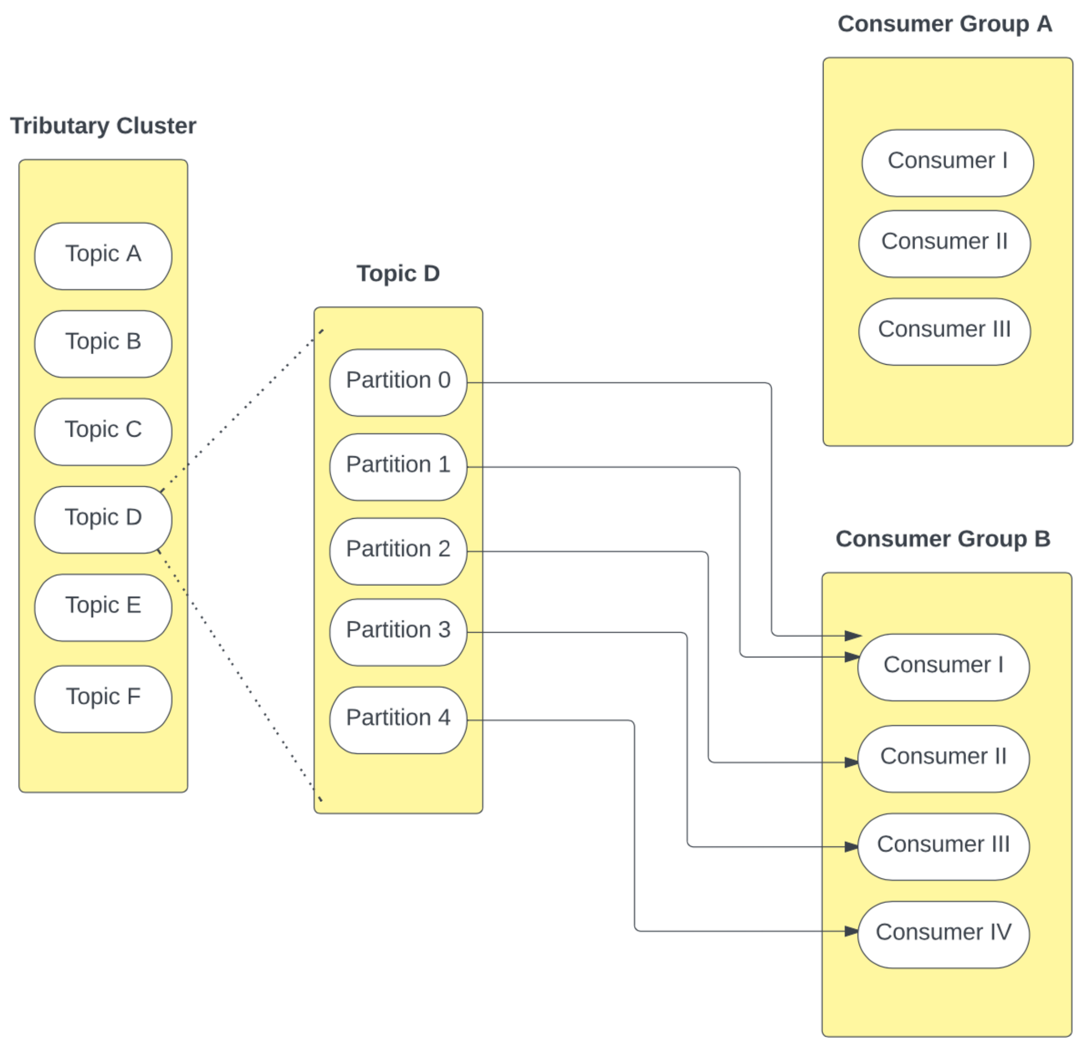


Tributary Cluster

STREAM PROCESSING EVENT DRIVEN ARCHITECTURE

Biswas Simkhada | z5476570 | COMP6841 | 01/11/2024
GitHub Link: https://github.com/Biswas57/Tributary-Cluster

Introduction

Modern applications heavily rely on event-driven architectures (EDA) to support microservices that process data efficiently and at scale. This reliance on EDA is the modern backbone of communication between different services and software. However, with the growing complexity of EDA systems comes the need to address critical security challenges that may compromise data integrity, confidentiality, and system resilience. This report examines the security measures integrated into the Tributary Cluster project, highlighting the application of tokenisation, hashing, and encryption methods.

Abstract

The objective of this report is to explain the function and reasoning behind security strategies implemented in the Tributary Cluster library; a simplified event-streaming framework inspired by Apache Kafka. Furthermore, it will also explore a specific case where the lack of Access Control Authentication led to the breach of large amounts of sensitive customer data that underwent the Kafka Broker pipeline. The report discusses the vulnerabilities that arise in event-driven systems and how specific security practices, including Role-Based Access Control (RBAC) with tokenisation, SHA-256 hashing for administrator authentication, and RSA encryption for secure message handling, can minimise these vulnerabilities. This security report provides an analysis of each security feature, reflects on the areas for potential improvement and an overall on the current standpoint of the Tributary Cluster library’s current security efficacy.

Background: What is Event-Driven Architecture and Stream Processing?

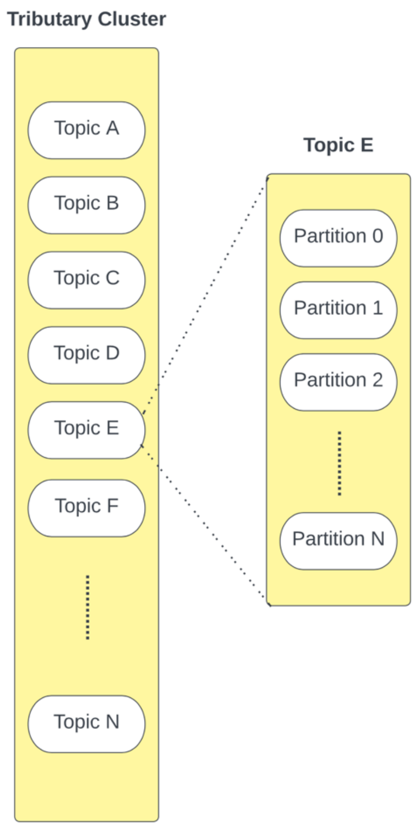
In simple terms, the basic idea of EDA systems is that they handle the communication of data between two main types of objects – event sources and event handlers, known as Producers and Consumers. Stream processing is a method of processing data in real time used in event-driven systems that can serve various purposes. For example, a stream processor may create specific events for detection by a downstream application in the pipeline. It may also perform real-time functions, such as alerting a stockbroker when a stock hits an all-time high or calculating the percentage change in a stock's price over the past year.

In more complex terms, however, Event-Driven Architecture rests on the ability of producer and consumer components in the system to share data asynchronously in a stream-like channel, which is the reason for this project being named 'Tributary Cluster'. Thus, stream processing is the concept that focuses on allowing for data to move through a data pipeline to be prcessed concurrently, bidirectionally and in real time. This is opposed to batch processing where data is collected and stored to be operated upon later. The Tributary Cluster library enhances the traditional in-memory stream EDA, which often relies on a single message queue shared by multiple consumers causing bottlenecks and delays, incorporating elements of batch processing in certain scenarios such as replayability of event logs and delayed and parallel event consumption. By adopting a log-based approach, The sophisticated stream processing EDAs overcome limitationg in other event driven platform such not allowing message playback and memory storage as seen in single message queue brokers. This improvement enhances data storage, replayability, and adds greater functionality to message processing, enabling more efficient and safe handling of messages/events.

How does the Tributary Cluster work?

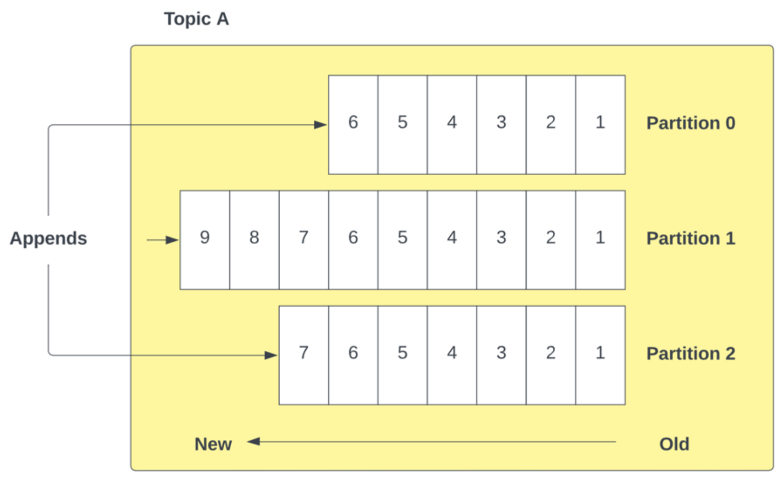
1. Structure

A Tributary Cluster is composed of several topics. Each topic serves as a l ogical container for related events, like a database table or a folder in a file system. For instance, one topic might handle image-related events while another manages video-related events.

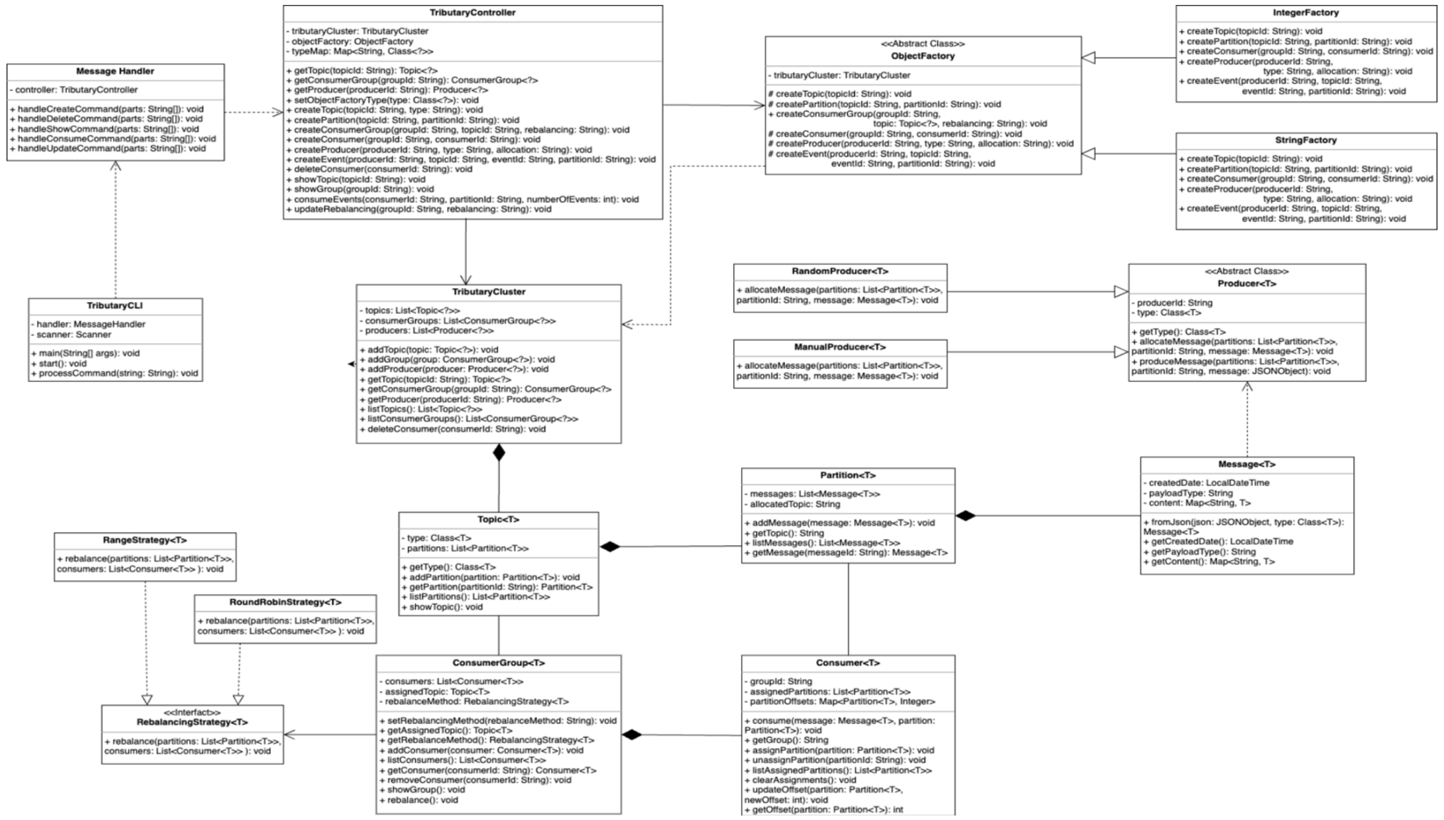


Within each topic, there are multiple partitions. These partitions act as queues where new messages are appended at the end. This division helps distribute the processing load and ensures parallel data handling.

A single data unit within a Tributary Cluster is called a message, orevent. For example, if a user sends a request to update their profile, a message is generated and sent to a specific partition within the relevant topic, such as "user profiles." Messages can optionally include a key to determine the specific partition they should be placed in.



Topics in the Tributary Cluster library are parameterised on a generic type, ensuring that all event payloads within that topic stick to the defined type.



Above is a UML diagram of the Tributary Cluster's structure prior to the addition of key Security features.

1.1. Message Lifecycle: Simple Example

Consider a user updating their profile. This action triggers an event generated by the producer for the "user profiles" topic, containing the updated profile data. The event is sent to the Tributary Cluster, which assigns it to a partition. The producer specifies whether the message should be assigned to a partition randomly or via a key. Consumers then sequentially consume the events within their assigned partitions, in order or appending.

1.2. Message Structure

Each message in the Tributary Cluster is a comprehensive package of information comprising of the:

- **Header/ Id:** Contains a unique identifier for referencing the message.
- **Payload Type:** Defines the data type of the message payload, supporting generics to allow flexibility (e.g. Integer, String, Byte).
- **Key:** Used for partitioning message content, used when consumers must reference specific datathat are store under keys in messages.

- **Value:** The actual content or payload of the message, which holds the relevant information pertinent to the topic.

1.3. Producers

Producers are entities responsible for creating and sending messages to the Tributary topics. They are designed with two types of allocation methods; random producers rely on the system to assign messages to partitions randomly, promoting an even distribution of load. Alternatively, manual producers use a specified key to determine the exact partition for message placement, providing control over message routing. Once a producer is established with a particular allocation method, this method remains constant.

1.4. Consumers and Consumer Groups

1.4.1. Consumers

Consumers process messages from partitions in the order they are produced and track consumed messages. Each partition can only be consumed by one consumer at a time, but a consumer can handle multiple partitions.

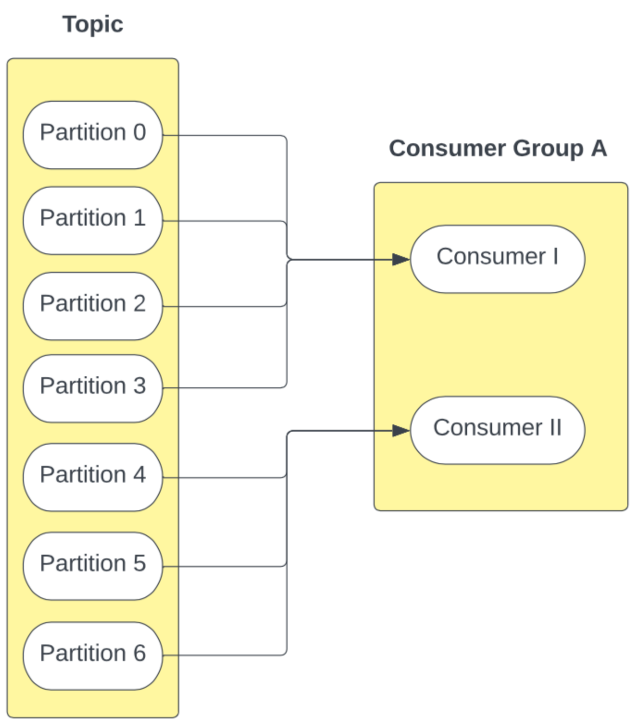
1.4.2. Consumer Groups

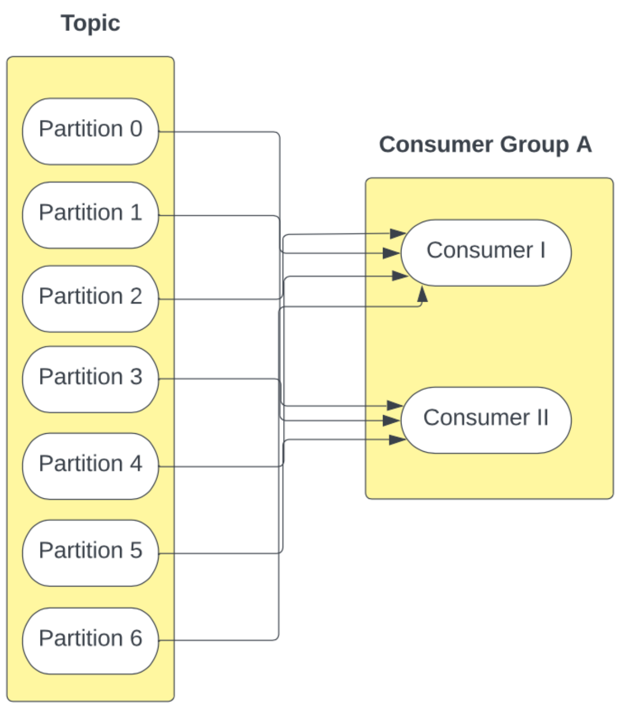
A consumer group consists of multiple consumers capable of processing all partitions in a topic. Multiple consumer groups can be assigned to the same topic, each group handling all partitions independently. When a consumer group starts, its consumers begin processing messages from the first unconsumed event in their partitions. Each message is consumed once, except during controlled replays.

1.4.3. Consumer Rebalancing

In the Tributary Cluster, consumer groups are essential components responsible for processing messages from partitions within a topic. Each consumer within a group is assigned one or more partitions from which it consumes messages sequentially. However, in dynamic systems where consumers can be added or removed due to scaling requirements or failures, the distribution of partitions among consumers can become unbalanced. This is where the rebalancing feature becomes crucial.

Rebalancing strategies dynamically reassign partitions when consumers are added or removed from a group. If a partition is reassigned, the new consumer continues from the last consumed message. The Tributary Cluster supports bother Range and Round Robin Rebalancing Methods.
Range Rebalancing is where partitions are divided into continuous ranges and each consumer is assigned a range of partitions. If the total number of partitions does not divide evenly among consumers, the consumers at the beginning of the list has one extra partition added to their contiguous range.

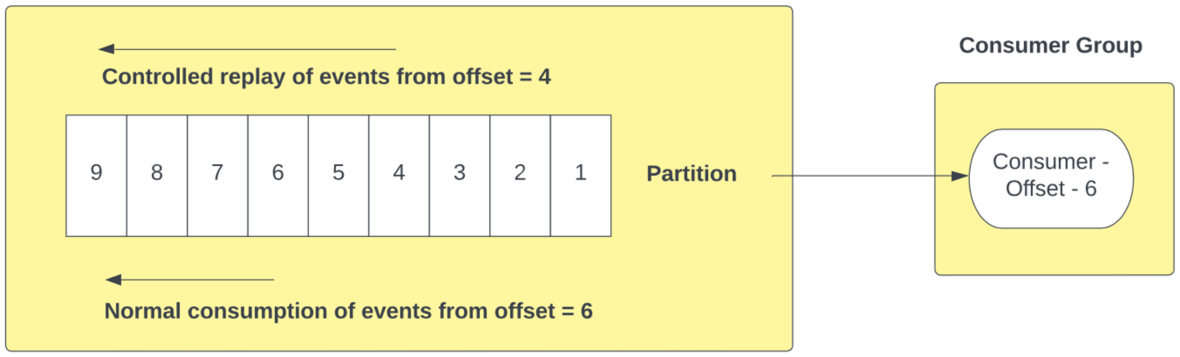




Round robin rebalancing is where partitions are assigned like cards dealt in a game, ensuring that every consumer is given a partition in order of their positiion in the consumer group to ensure even distribution. If there are extra partitions remaining, the consumers at the beginning of the list are the first to receive extra partitions.

1.5. Replay Messages

One of the most powerful aspects of event streaming is the ability to replay messages that are stored in the queue. The way this can occur is via a controlled replay, which is done from a message offset in a partition. Messages from that point onwards are streamed through the pipeline, until the most recent message at the latest offset is reached.


Controlled replay of events from offset = 4
9
8
7
6
5
4
3
2
1
Partition
Normal consumption of events from offset = 6
Consumer Group
Consumer - Offset - 6

For example, a consumer starting at offset 6 and consuming up to offset 9 can trigger a replay from offset 4, consuming messages 4, 5, 6, 7, 8, and 9 again.

1.6. Important Design Considerations

The design of the Tributary Cluster places a strong emphasis on thread safety, asynchronous methods, usability and type safety, specifically, it ensures thread safety and type safety across user threads by utilising Java Generics and Concurrency techniques.

In the Tributary Cluster, producers and consumers ran concurrently, so I needed robust mehtods to keep data consistent. I used Java's concurrency tools, to handle multiple threads. This approach lets me produce and consume events in parallel without the hassle of managing threads manually. I synchronised critical sections of the code to prevent race conditions, ensuring that shared resources like topics and partitions are accessed safely to achieve high throughput and responsiveness while maintaining data integrity across all operations.

I also made extensive use of Java generics to support different types of event payloads, giving us flexibility in the types of data the system can handle. This design choice allows developers to define topics that work with specific data types like Integer, String, or custom objects. By abstracting type handling through a TypeHandlerFactory, users can easily incorporate new data types by implementing the corresponding handlers.

On top of this, one of the major achievements was turning the project into a usable Java library. This means other developers can easily integrate the Tributary Cluster into their own projects. By designed the system with a modular approach, breaking it down into distinct components and structured the code with clear APIs and thorough documentation, it made the codebase easier to maintain and extend in the future.

2. Interface

The Tributary Cluster is a library enabling engineers to build event-driven systems. Public classes for developers go into the api folder, while internal classes are in the core folder. The library functions like a toolkit, providing components for building their own event-driven applications. The Command Line Interface (CLI) application, described below, showcases the library's functionality.

3. Command Line Interface

The CLI allows interaction with the Tributary Cluster through commands for creating, modifying, and interacting with the system. Developers can modify or add new commands as needed.

To run usability tests on this library I needed to develop a way to interact with tributaries, producers, and consumers via a command line interface. To do so, I wrote a wrapper class called TributaryCLI that allows users to input commands that create, modify, and interact with a tributary cluster system. This class is in a separate package to the api/core packages of my library, as it shouldn't be a part of ther library that other engineers developing their event-driven systems would use. The specific commands that can be ran on the CLI are found in this project’s GitHub Repository

Security: Why is Security Important for the Tributary Cluster?

Event-driven architectures let producers and consumers share data in real time and concurrently through event streams. But even though these systems are super efficient, they can run into security issues like unauthorised access, data interception, and message tampering. One such case of this happening is the 2021 Security Breach of a famous Stream Processing EDA known as Apache Kafka, which 80% of Fortune 100 companies use for their data processing, through a visualiser and management application called Kafdrop.

Misconfigured Kafdrop UI Security Breach

In 2021, it was found that many organisations had incorrectly setup their Kafdrop interfaces without securing it with authentication on the backend, which led the exposure of their entire Kafka clusters to the internet. Kafdrop is a massive open-source UI for managing Kafka clusters, which is widely used because it provides an easy way to view and manage Kafka topics, partitions, and consumers.

The problem was that Kafdrop, by default, doesn't enforce authentication. So, when companies deployed it without proper security measures, they left their Kafka clusters wide open. Unauthorised users could access sensitive data, manage topics, and even delete data and disrupt service by interfering with consumers. Due to these misconfigurations in setting up Kafdrop, sensitive information like customer transactions, medical records, and internal system traffic was exposed publicly. Attackers could tap into real-time traffic, revealing secrets, authentication tokens, and other access details that could allow them to infiltrate companies' cloud environments on platforms like AWS, IBM, and Oracle.

For the Tributary Cluster, this breach highlights how crucial security is in stream processing EDAs. Even if the core Kafka cluster is secure, tools like Kafdrop can become weak links if they’re not properly configured. So, the system’s core components need proper authentication and encryption. Without these, EDAs are vulnerable to this type of attack. The trust and reliability of these systems can really take a hit, especially when they're dealing with sensitive data.

Implementation: What are Security Considerations for the Tributary Cluster?

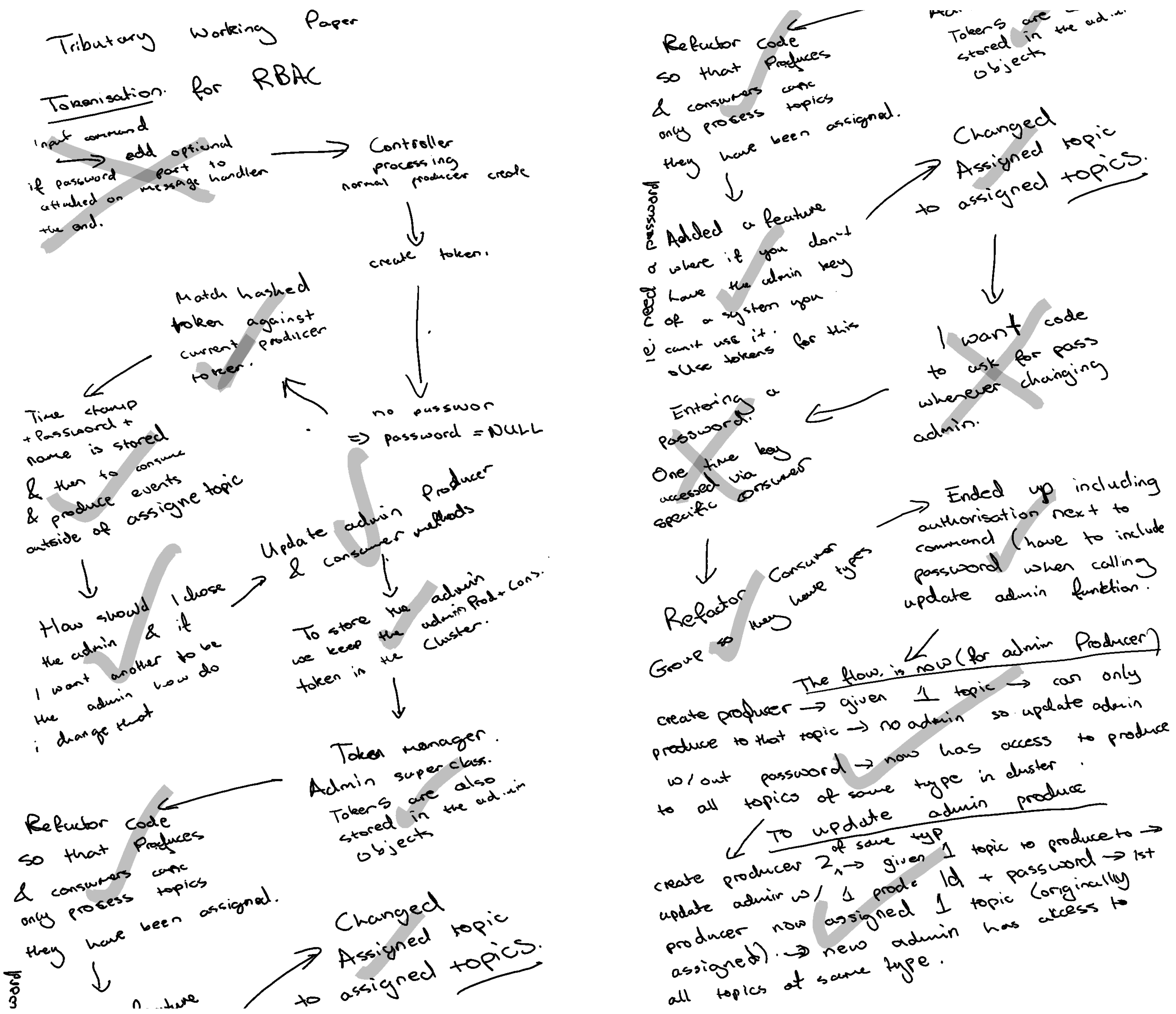
Traditional EDAs often don't have much authentication and secure data transer, which leaves them open to breaches that can expose data These two points of concern will be the focus of what my security implementation will focus around. Without proper security measures, the reliability of event-driven systems can really take a hit, especially when they're dealing with sensitive data.

Access Control (AC)
In Tributary Cluster, producers and consumers communicate with the cluster continuously. Without proper AC, any client could be allowing to read from or write to any topic within the system. As users start implementing the Tributary Cluster on a larger scale and for various types of data, a haphazard approach to securing the platform won't really work. It's crucial that all clients attempting to read or write to the cluster are properly authenticated for the topics they're interacting with. This ensures that improper access to sensitive data is prevented.

Data Security
Just like authentication are vital, ensuring the security of data as it's transmitted over the network is also of utmost importance. So, data-in-transit, whether from producer to the cluster or read operations by consumers, should be protected through sometype of unreadable layering to protect sensitive information. Specifically, data should be encrypted when communicated between client applications (producers and consumers) and the Tributary Cluster. By implementing encryption for data-in-transit, we safeguard against interception and eavesdropping, ensuring that sensitive information remains confidential and intact during transmission.

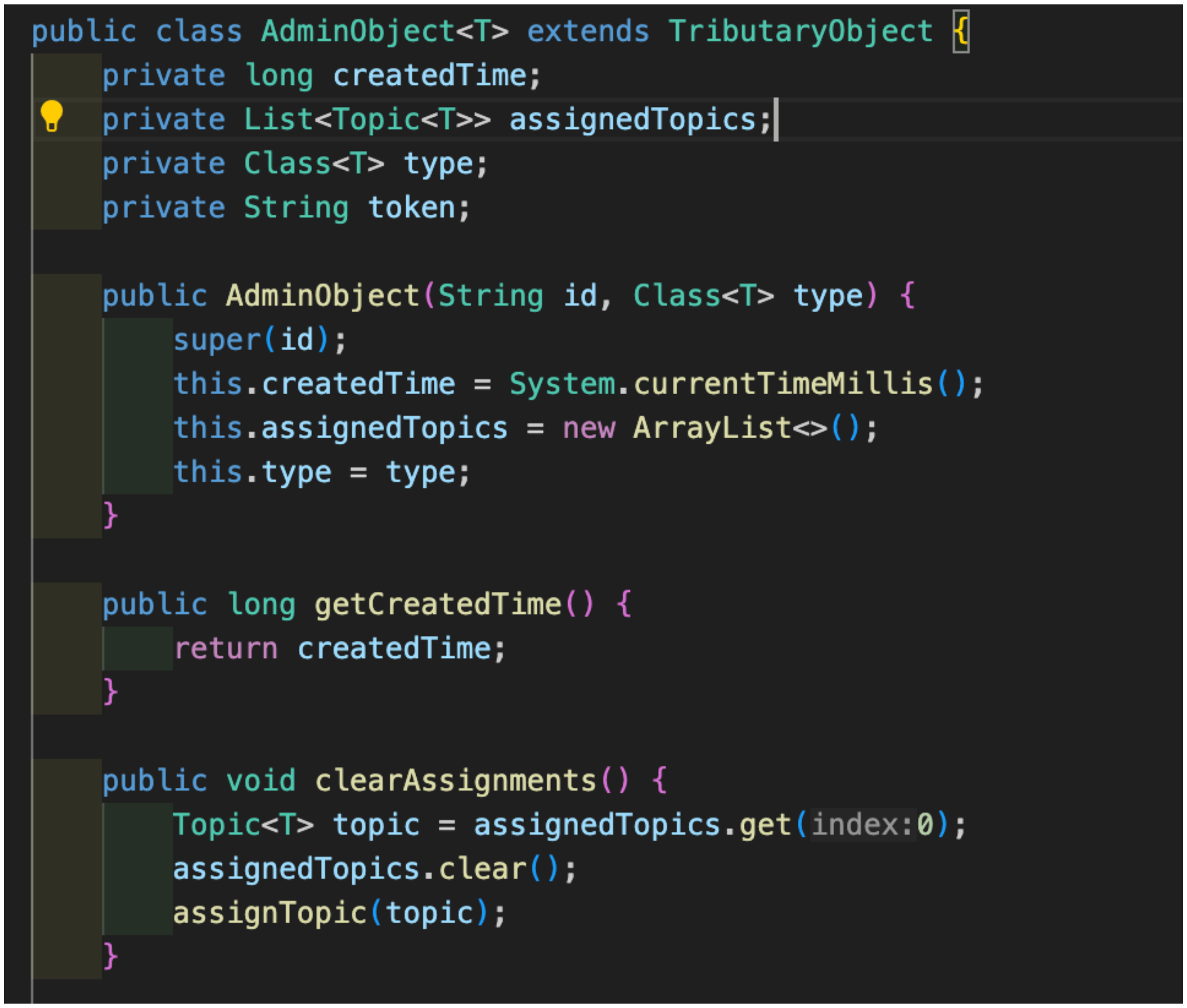
Role-Based Access Control (RBAC)

RBAC is implemented in Tributary Cluster to ensure that only authorised entities can produce or consume events. This was a complex process that required extensive planning. Initially, I explored various approaches to ensure that only authorised entities could produce or consume events, recognising that unrestricted access would cause a security risk. The general process needed significant brainpower to integrate without compromising performance. This is documented in my “brainstorm and jot down ideas” workflow seen below using ticks and crosses to show which features I did and didn’t end up implementing.


Tributary Working Paper
Tokenisation. for RBAC
Input command → add optional port to message handler
if password attached on the end.
Controller processing normal producer create
create token.
no passwor ⇒ password = NULL
Match hashed token against current producer token.
Time stamp + Password + name is stored & then to consume & produce events outside of assigne topic
How should the admin & if I want another to be the admin how do i change that
I chose
Update admin Producer & consumer methods
To store the admin we keep the token in the admin Prod + Cons. Cluster.
Token manager. Admin super class. Tokens are also stored in the admin object
Refactor code so that Produces & consumers can only process topics they have been assigned.
Changed Assigned topic to assigned topics.
Added a feature where if you don't have the admin key of a system you can't use it. Use tokens for this
ie: need a password
I want code to ask for pass whenever changing admin.
Entering a password. One time key accessed via specific consumer
Refactor Consumer Group so they have types
Ended up including authorisation next to command (have to include password when calling update admin function.
The flow is now (for admin Producer)
create producer → given 1 topic → can only produce to that topic → no admin so update admin w/ out password → now has access to produce to all topics of same type in cluster.
To update admin produce
create producer 2 of same typ → given 1 topic to produce to → update admin w/ 1 prod. Id + password → 1st producer now assigned 1 topic (originally assigned). → new admin has access to all topics of same type.

I began by defining the roles of the producers and consumer groups and their permissions, ensuring that each entity's access was restricted to only a single topic initially. The backbone of the solution came through with tokenisation, which allowed me to issue unique tokens for authentication, effectively managing access to specific topics and partitions. To strengthen security, I added hashing algorithms to make intercepted tokens difficult to reverse-engineer or tamper with. This led to me design the process to validate tokens and permissions before granting access for certain producers or consumers to become admins.

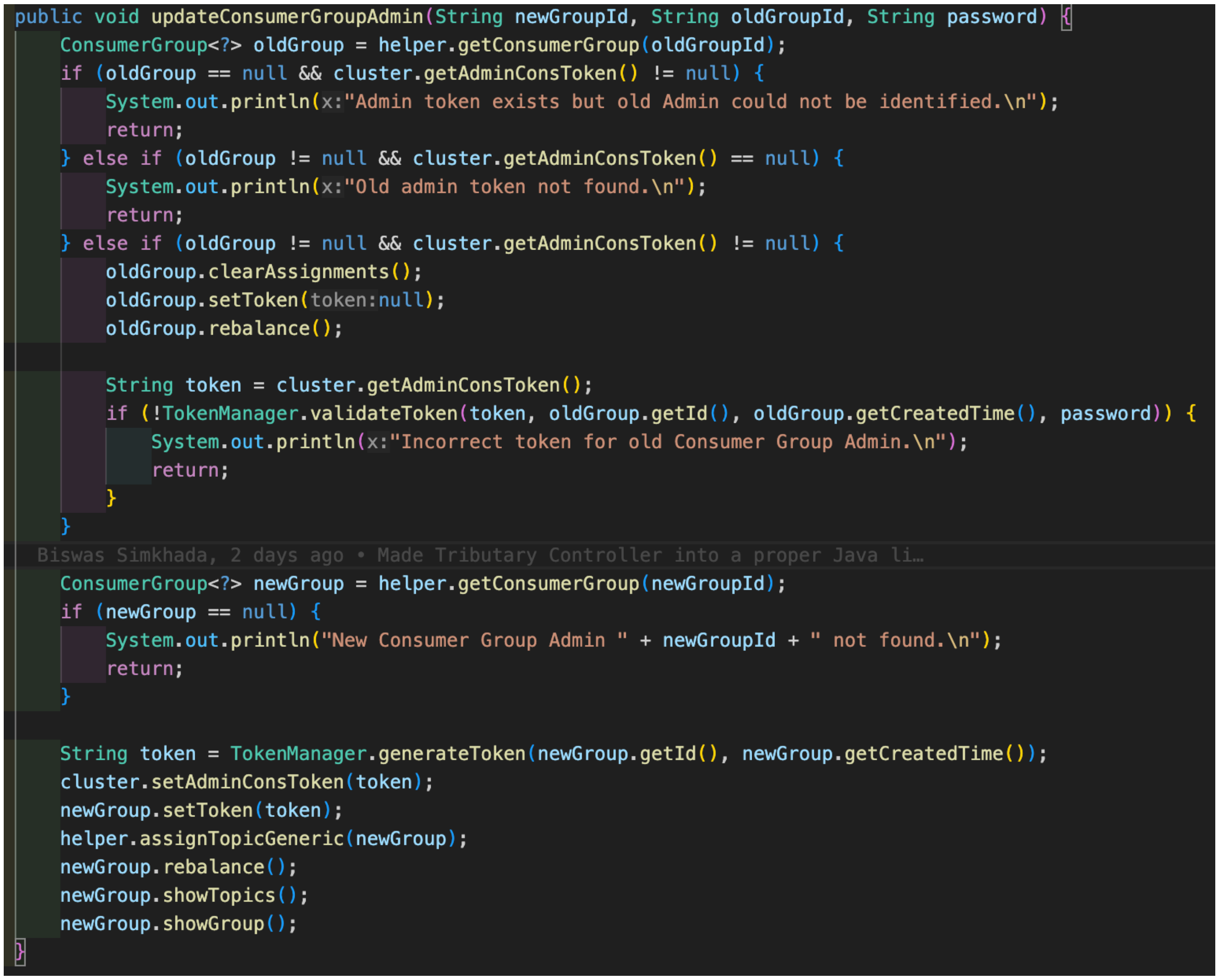
Figure 1 – Establishing a Heirarchy with Admin Objects



```java
public class AdminObject<T> extends TributaryObject {
    private long createdTime;
    private List<Topic<T>> assignedTopics;
    private Class<T> type;
    private String token;

    public AdminObject(String id, Class<T> type) {
        super(id);
        this.createdTime = System.currentTimeMillis();
        this.assignedTopics = new ArrayList<>();
        this.type = type;
    }

    public long getCreatedTime() {
        return createdTime;
    }

    public void clearAssignments() {
        Topic<T> topic = assignedTopics.get(index:0);
        assignedTopics.clear();
        assignTopic(topic);
    }
```

Estabilishing a heirarchy in the EDA which elavated the Producer and Consumer Group classes meant that authorised producers and consumer groups could assume administrative roles with the capacity to manage and modify access to specific topics and partitions.

Figure 2 & 3 – Update Producer and Consumer Group Admin Authorisation

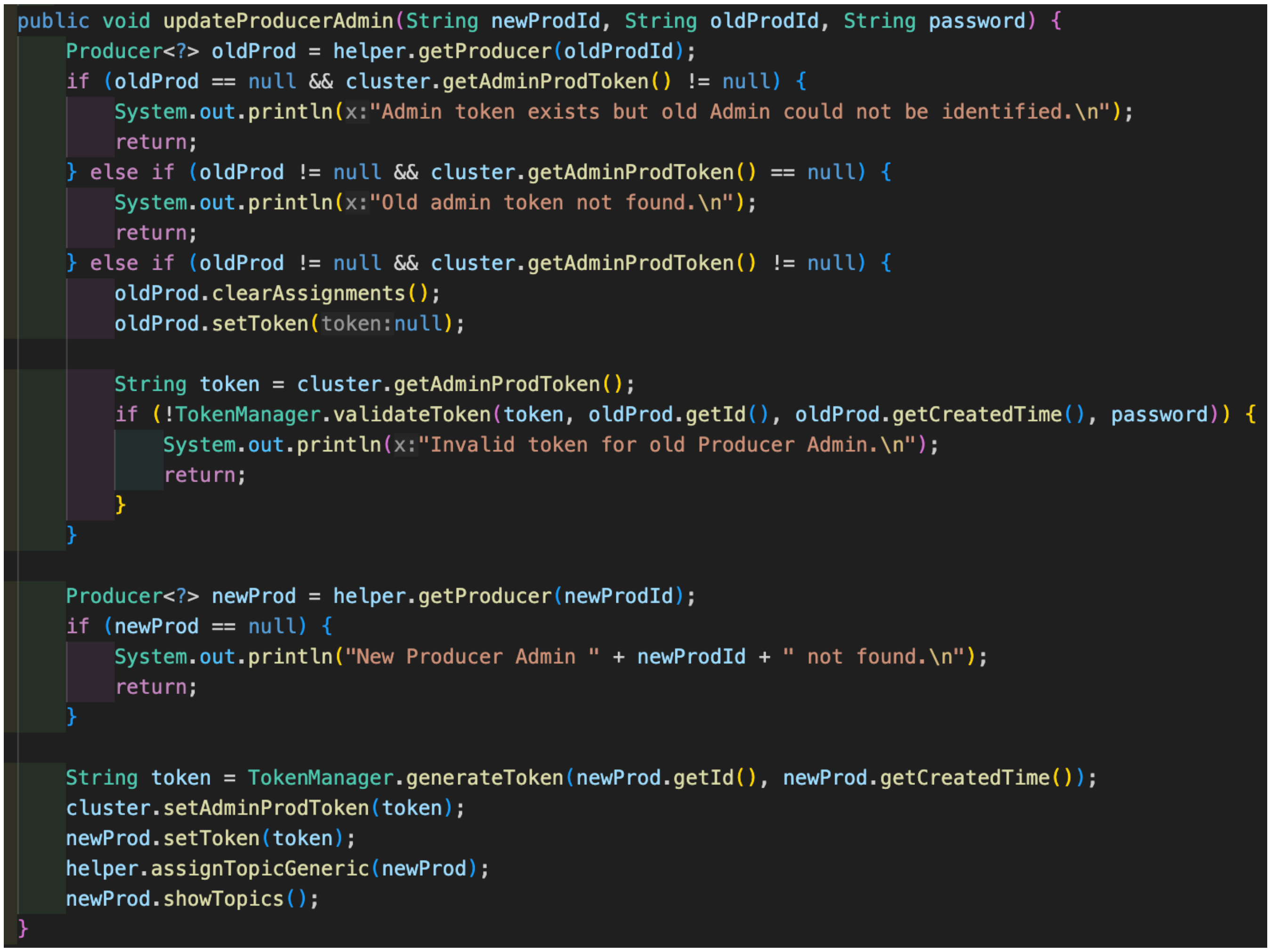


```java
public void updateConsumerGroupAdmin(String newGroupId, String oldGroupId, String password) {
    ConsumerGroup<?> oldGroup = helper.getConsumerGroup(oldGroupId);
    if (oldGroup == null && cluster.getAdminConsToken() != null) {
        System.out.println(x:"Admin token exists but old Admin could not be identified.\n");
        return;
    } else if (oldGroup != null && cluster.getAdminConsToken() == null) {
        System.out.println(x:"Old admin token not found.\n");
        return;
    } else if (oldGroup != null && cluster.getAdminConsToken() != null) {
        oldGroup.clearAssignments();
        oldGroup.setToken(token:null);
        oldGroup.rebalance();

        String token = cluster.getAdminConsToken();
        if (!TokenManager.validateToken(token, oldGroup.getId(), oldGroup.getCreatedTime(), password)) {
            System.out.println(x:"Incorrect token for old Consumer Group Admin.\n");
            return;
        }
    }
  Biswas Simkhada, 2 days ago • Made Tributary Controller into a proper Java li…
    ConsumerGroup<?> newGroup = helper.getConsumerGroup(newGroupId);
    if (newGroup == null) {
        System.out.println("New Consumer Group Admin " + newGroupId + " not found.\n");
        return;
    }

    String token = TokenManager.generateToken(newGroup.getId(), newGroup.getCreatedTime());
    cluster.setAdminConsToken(token);
    newGroup.setToken(token);
    helper.assignTopicGeneric(newGroup);
    newGroup.rebalance();
    newGroup.showTopics();
    newGroup.showGroup();
}
```

The update consumer group and producer admin class are the front facing authorisation methods that allow us to access that extra level of security by restricting which entities have unlimited access and further securing them by adding the feature of tokenisation-based authentication using the TokenManager class.

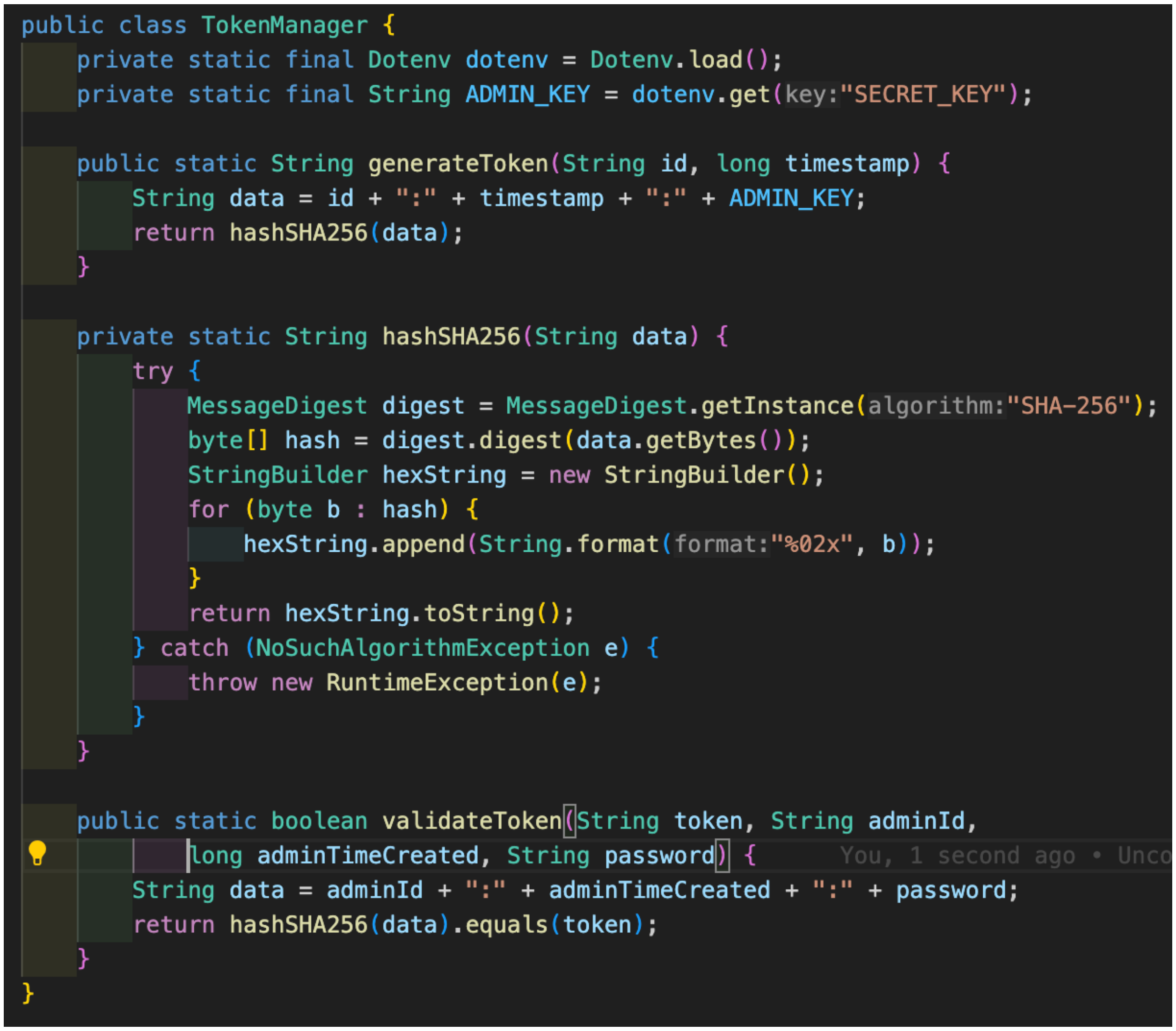
Specifically, they use the validate token class if a previous admin existed. Because of the risk of potential attackers attempting to gain control of the Tributary system, I don just compare the stored admin tokens to the token in the Producer or Consumer group class. Instead, I actively gather the Id of the old admin, the time it was created and the password entered by the use to validate the token using the validateToken method in the TokenManager class.

When assigning the new admin, the generateToken function is called and the new token is stored both in the Tributary Cluster admin variable and the new Admin’s token variable. This process is identical for both the Consumer Group class and the Producer class, except that the Consumer group is rebalanced with the partitions from all the topics after the group gains admin permissions and rebalanced with their originally assigned topic after they are removed.



```java
public void updateProducerAdmin(String newProdId, String oldProdId, String password) {
    Producer<?> oldProd = helper.getProducer(oldProdId);
    if (oldProd == null && cluster.getAdminProdToken() != null) {
        System.out.println(x:"Admin token exists but old Admin could not be identified.\n");
        return;
    } else if (oldProd != null && cluster.getAdminProdToken() == null) {
        System.out.println(x:"Old admin token not found.\n");
        return;
    } else if (oldProd != null && cluster.getAdminProdToken() != null) {
        oldProd.clearAssignments();
        oldProd.setToken(token:null);

        String token = cluster.getAdminProdToken();
        if (!TokenManager.validateToken(token, oldProd.getId(), oldProd.getCreatedTime(), password)) {
            System.out.println(x:"Invalid token for old Producer Admin.\n");
            return;
        }
    }

    Producer<?> newProd = helper.getProducer(newProdId);
    if (newProd == null) {
        System.out.println("New Producer Admin " + newProdId + " not found.\n");
        return;
    }

    String token = TokenManager.generateToken(newProd.getId(), newProd.getCreatedTime());
    cluster.setAdminProdToken(token);
    newProd.setToken(token);
    helper.assignTopicGeneric(newProd);
    newProd.showTopics();
}
```

Figure 6 – Tokenisation Manager Class

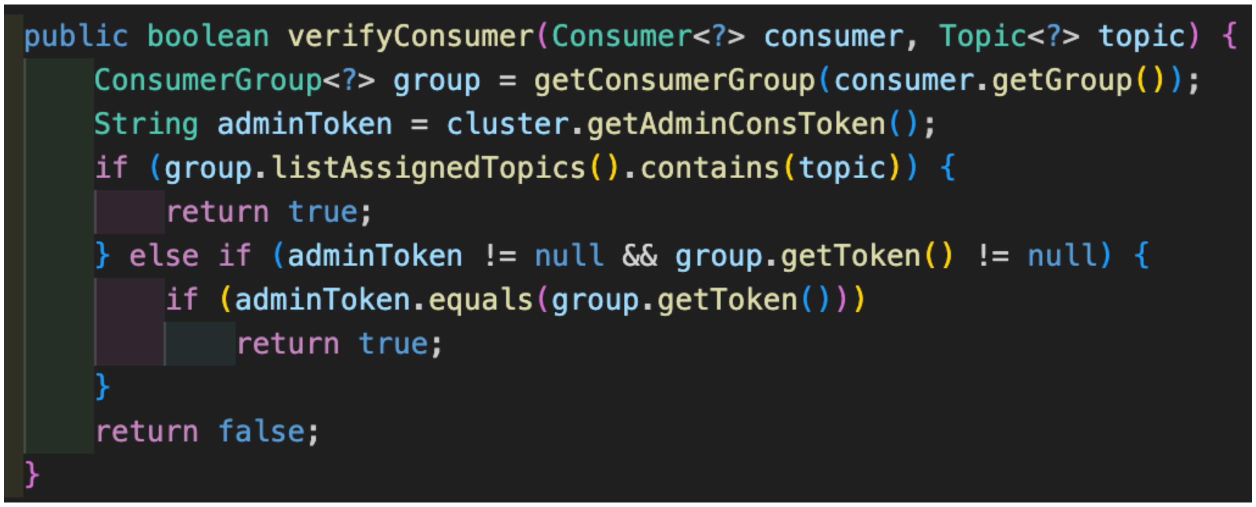


```
public class TokenManager {
    private static final Dotenv dotenv = Dotenv.load();
    private static final String ADMIN_KEY = dotenv.get(key:"SECRET_KEY");

    public static String generateToken(String id, long timestamp) {
        String data = id + ":" + timestamp + ":" + ADMIN_KEY;
        return hashSHA256(data);
    }

    private static String hashSHA256(String data) {
        try {
            MessageDigest digest = MessageDigest.getInstance(algorithm:"SHA-256");
            byte[] hash = digest.digest(data.getBytes());
            StringBuilder hexString = new StringBuilder();
            for (byte b : hash) {
                hexString.append(String.format(format:"%02x", b));
            }
            return hexString.toString();
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException(e);
        }
    }

    public static boolean validateToken(String token, String adminId,
            long adminTimeCreated, String password) {       You, 1 second ago • Unco
        String data = adminId + ":" + adminTimeCreated + ":" + password;
        return hashSHA256(data).equals(token);
    }
}
```

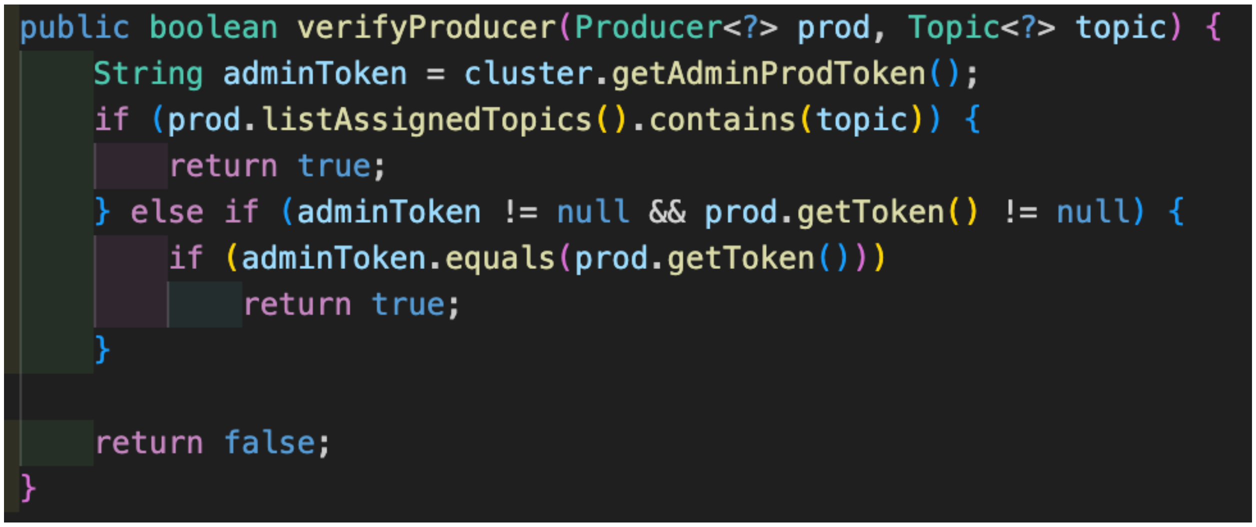
The TokenManager class implementation I have used here in combination with SHA256 hashing keeps the token for tokenisation-based authorisation extremely secure. Administrator roles in the Tributary Cluster are safeguarded through password hashing using the SHA-256 algorithm in the TokenManager class. This practice ensures that sensitive credentials are not stored in plaintext and are resistant to brute-force attacks.

I chose not to implement the SHA-256 algorithm myself and instead demonstrated my mathematical programming ability with the RSA encryption I used. This is shown later in this security report.

Figure 4 & 5 – Verify Topic Assignment or Admin Permissions



```
public boolean verifyConsumer(Consumer<?> consumer, Topic<?> topic) {
    ConsumerGroup<?> group = getConsumerGroup(consumer.getGroup());
    String adminToken = cluster.getAdminConsToken();
    if (group.listAssignedTopics().contains(topic)) {
        return true;
    } else if (adminToken != null && group.getToken() != null) {
        if (adminToken.equals(group.getToken()))
            return true;
    }
    return false;
}
```



```
public boolean verifyProducer(Producer<?> prod, Topic<?> topic) {
    String adminToken = cluster.getAdminProdToken();
    if (prod.listAssignedTopics().contains(topic)) {
        return true;
    } else if (adminToken != null && prod.getToken() != null) {
        if (adminToken.equals(prod.getToken()))
            return true;
    }

    return false;
}
```

The verify Consumer and Producer methods demonstrate how the system verifies permissions when a consumer group admin attempts to consume from a topic or when a producer admin tries to create an event for an unassigned topic. If the topic is already assigned to the consumer group or producer, the access is verified and granted. Otherwise, the admin permissions are checked to confirm whether the entity is authorised to perform the requested action.

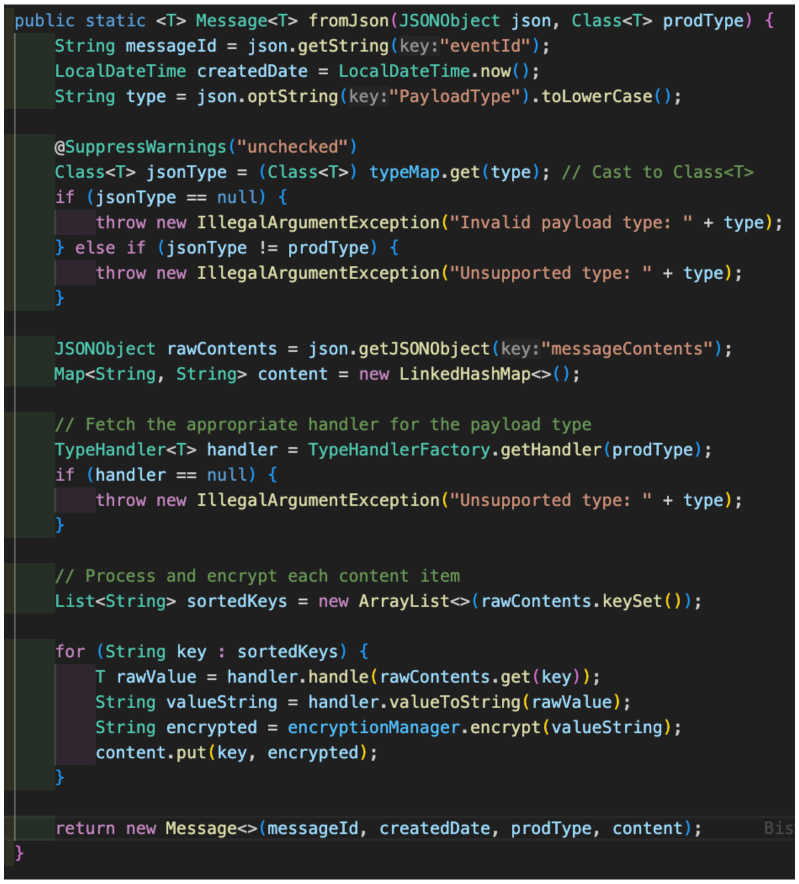
This ensures tokens are validated against stored tokens before any read or write operations are permitted or the producers' or consumer groups' access lists are checked, preventing unauthorised users from accessing the event streams.

Ultimately, incorporating tokenisation, hashing, and a robust authorisation layer established an effective RBAC system that secured the Tributary Cluster using authorisation of topic access during message production and consumption.

Encryption for Secure Data Transfer

To secure data transmission in the Tributary Cluster, I implemented RSA encryption. This decision was driven by RSA's compatibility with different data types. I specifically used RSA to encrypt string and converted all the data types to string be it is much more scalable that way. RSA’s asymmetric encryption uses a public key for encryption and a private key for decryption, ensuring that only authorised entities can access transmitted data. This encryption approach enhanced the overall security of the Tributary Cluster while being about to efficiently handle data with type safety.

Figure 1 – Encryption Flow (Message Creation)



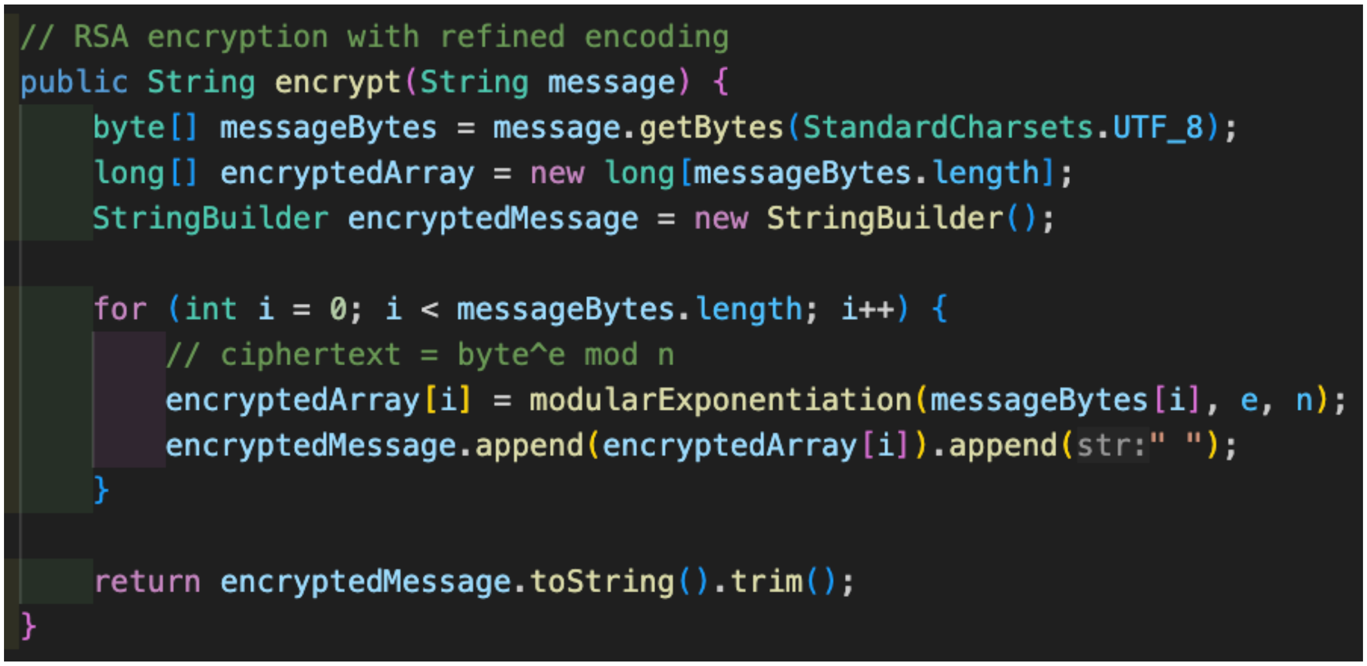
```
public static <T> Message<T> fromJson(JSONObject json, Class<T> prodType) {
    String messageId = json.getString(key:"eventId");
    LocalDateTime createdDate = LocalDateTime.now();
    String type = json.optString(key:"PayloadType").toLowerCase();

    @SuppressWarnings("unchecked")
    Class<T> jsonType = (Class<T>) typeMap.get(type); // Cast to Class<T>
    if (jsonType == null) {
        throw new IllegalArgumentException("Invalid payload type: " + type);
    } else if (jsonType != prodType) {
        throw new IllegalArgumentException("Unsupported type: " + type);
    }

    JSONObject rawContents = json.getJSONObject(key:"messageContents");
    Map<String, String> content = new LinkedHashMap<>();

    // Fetch the appropriate handler for the payload type
    TypeHandler<T> handler = TypeHandlerFactory.getHandler(prodType);
    if (handler == null) {
        throw new IllegalArgumentException("Unsupported type: " + type);
    }

    // Process and encrypt each content item
    List<String> sortedKeys = new ArrayList<>(rawContents.keySet());

    for (String key : sortedKeys) {
        T rawValue = handler.handle(rawContents.get(key));
        String valueString = handler.valueToString(rawValue);
        String encrypted = encryptionManager.encrypt(valueString);
        content.put(key, encrypted);
    }

    return new Message<>(messageId, createdDate, prodType, content);        Bis
}
```

This fromJson method is how Messages are created in Tributary Cluster and is the step before the encryption of message content.

The message information is taken from a JSON file, where it isn’t typed. It is then converted into a string by the TypeHandler class.

For each value under a key in messageContents, said value is encrypted and then stored in a Hashmap with a non-encrypted key.

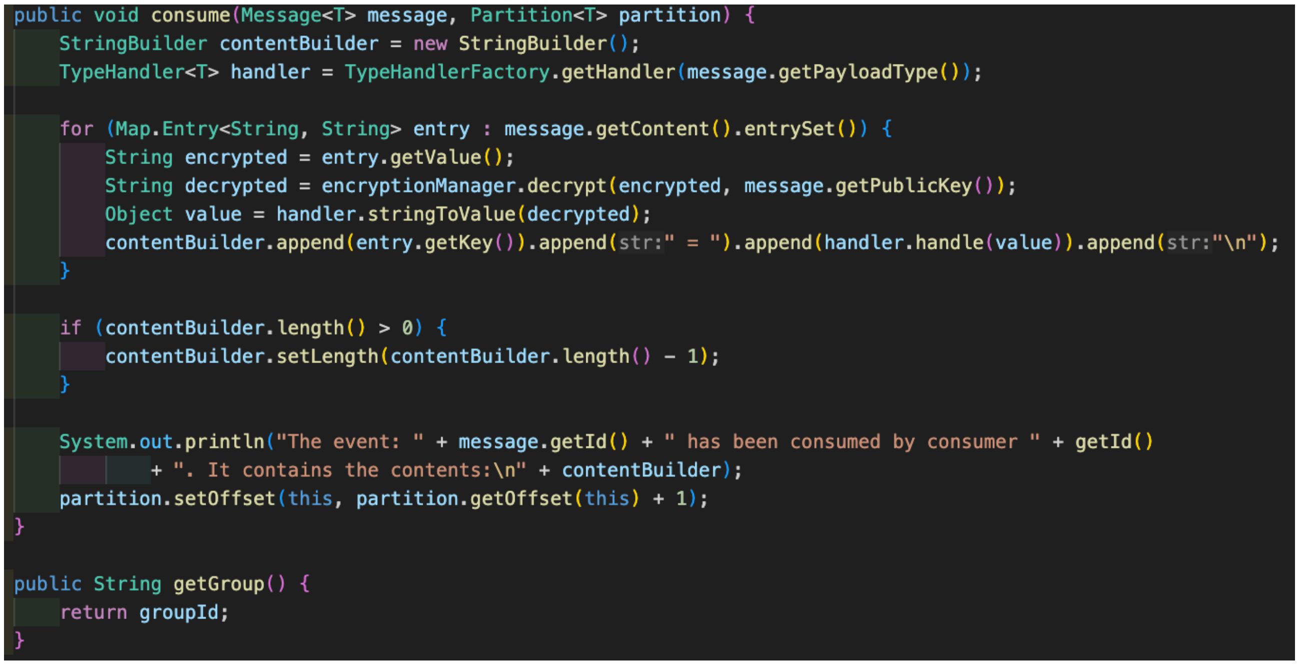
Figure 2 - Encryption Flow (Message Content Encryption)



```java
// RSA encryption with refined encoding
public String encrypt(String message) {
    byte[] messageBytes = message.getBytes(StandardCharsets.UTF_8);
    long[] encryptedArray = new long[messageBytes.length];
    StringBuilder encryptedMessage = new StringBuilder();

    for (int i = 0; i < messageBytes.length; i++) {
        // ciphertext = byte^e mod n
        encryptedArray[i] = modularExponentiation(messageBytes[i], e, n);
        encryptedMessage.append(encryptedArray[i]).append(str:" ");
    }

    return encryptedMessage.toString().trim();
}
```

The encryption process uses RSA to convert strings into an encrypted format. For uniformity, I chose to use UTF-8 byte strings, which ensures that each encrypted segment outputs as an 8-digit long value. This simplifies the decryption process as the string can be split every 8 characters to recover the original content.

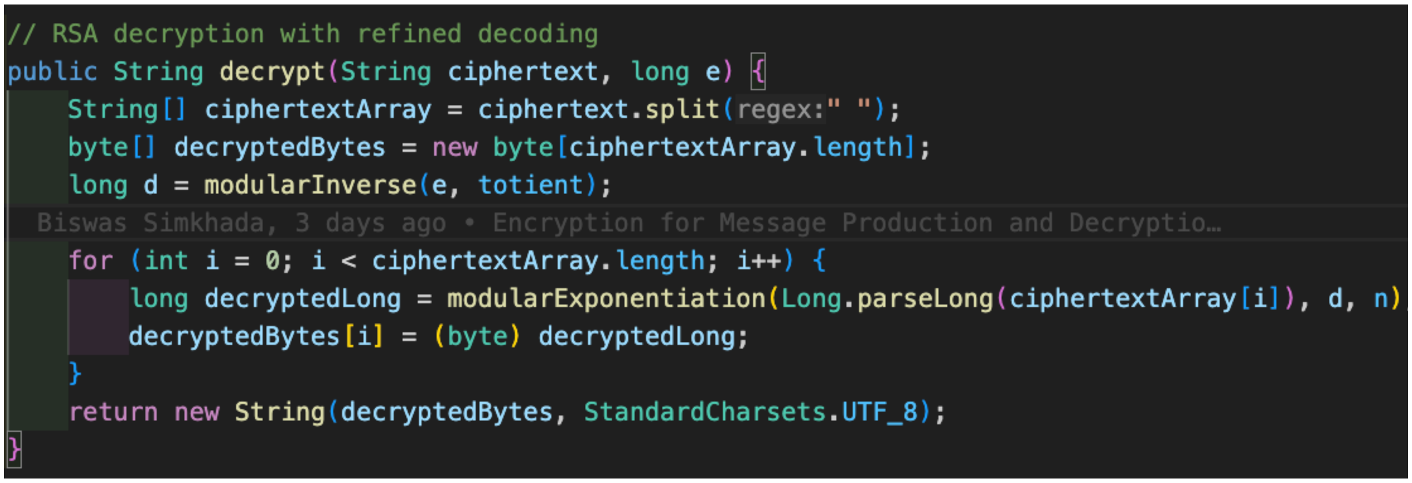
Figure 3 – Encryption Flow (Message Consumption)



```java
public void consume(Message<T> message, Partition<T> partition) {
    StringBuilder contentBuilder = new StringBuilder();
    TypeHandler<T> handler = TypeHandlerFactory.getHandler(message.getPayloadType());

    for (Map.Entry<String, String> entry : message.getContent().entrySet()) {
        String encrypted = entry.getValue();
        String decrypted = encryptionManager.decrypt(encrypted, message.getPublicKey());
        Object value = handler.stringToValue(decrypted);
        contentBuilder.append(entry.getKey()).append(str:" = ").append(handler.handle(value)).append(str:"\n");
    }

    if (contentBuilder.length() > 0) {
        contentBuilder.setLength(contentBuilder.length() - 1);
    }

    System.out.println("The event: " + message.getId() + " has been consumed by consumer " + getId()
            + ". It contains the contents:\n" + contentBuilder);
    partition.setOffset(this, partition.getOffset(this) + 1);
}

public String getGroup() {
    return groupId;
}
```

The consume method takes in a specified message from a partition, decrypting and processing the message content. It uses a TypeHandler to convert string representations back to their original types. Each entry in the message content is decrypted using the encryptionManager and re-converted from string format using the handler, appending the result to a String Builder. Once decryption and type handling are complete, the method prints out the processed content, indicating that the message has been successfully encrypted and decrypted back to the original content. This shows the full lifecycle of the RSA encryption process.

Figure 4 - Decryption Flow (Message Content Decryption)



```
// RSA decryption with refined decoding
public String decrypt(String ciphertext, long e) {
    String[] ciphertextArray = ciphertext.split(regex:" ");
    byte[] decryptedBytes = new byte[ciphertextArray.length];
    long d = modularInverse(e, totient);
 Biswas Simkhada, 3 days ago • Encryption for Message Production and Decryptio…
    for (int i = 0; i < ciphertextArray.length; i++) {
        long decryptedLong = modularExponentiation(Long.parseLong(ciphertextArray[i]), d, n)
        decryptedBytes[i] = (byte) decryptedLong;
    }
    return new String(decryptedBytes, StandardCharsets.UTF_8);
}
```

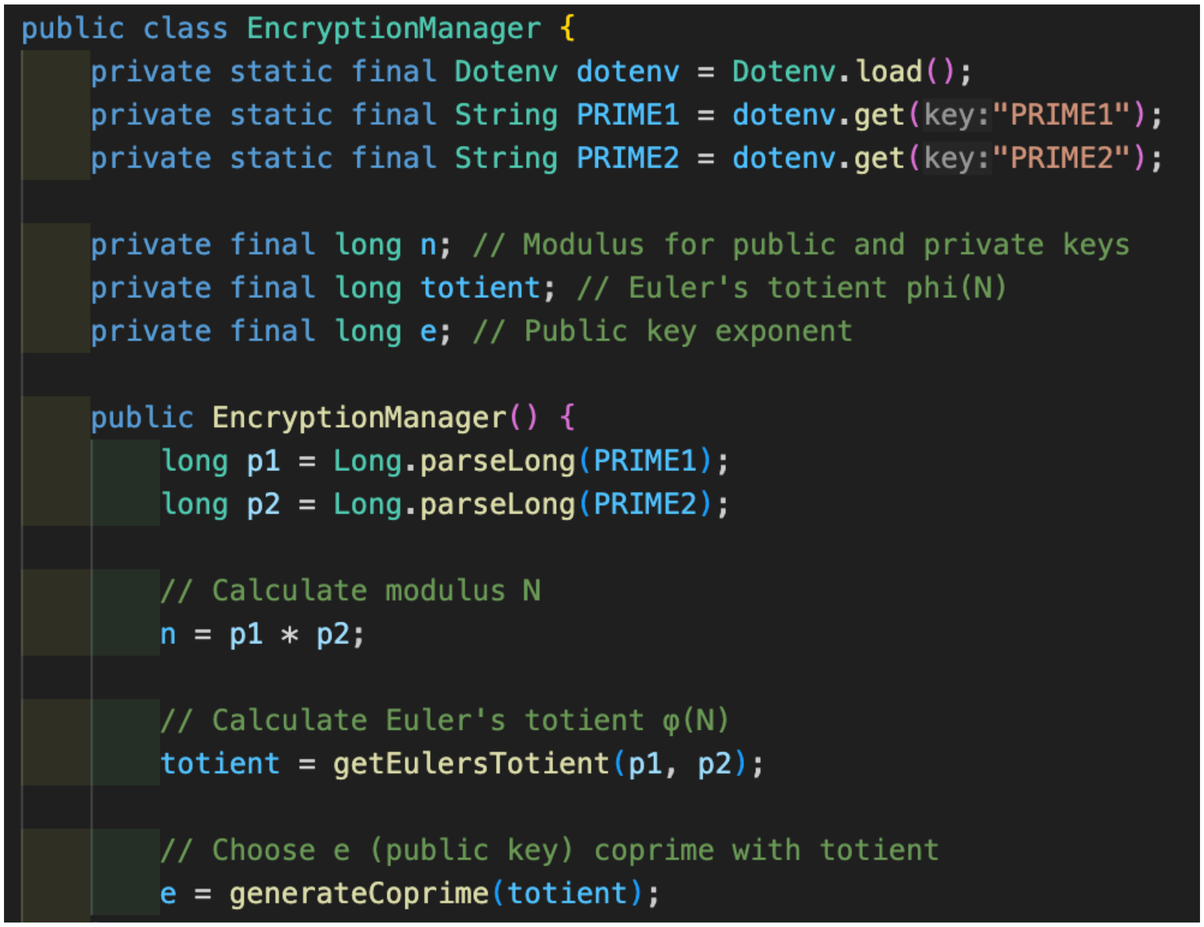
The decrypt method splits the ciphertext string into an array, processes each encrypted long value using modular exponentiation, and converts them back to bytes using the calculated modular inverse of e with respect to the totient. The decrypted bytes are then compiled into a string using UTF-8 encoding, returning the original plaintext message. Most importantly, the method ensures that messages are recovered and reverted to their original form.

To ensure that this entire process works with any payload that can be stored in a JSON file (for now), I designed an extensive testing suite covering a range of scenarios. This suite includes tests for basic encryption-decryption cycles, edge cases involving emojis, large payloads, and numeric values, as well as tests for integer boundary values and multi-part strings. The goal was to validate that the encryption and decryption functions maintain data integrity across different data types and content sizes. By incorporating these comprehensive tests, I confirmed that the RSA-based encryption that I coded in the Tributary Cluster is reliable for all types of payloads that might be encountered during real-world use cases.

Maths Behind RSA

The process of RSA encryption involves extensive use of modular arithmetic. In my implementation, I converted strings into a numeric format that is then encrypted and stored in the message content. Below are the steps I followed to implement RSA, including the mathematical procedures I used to generate the public encryption key, private decryption key, and the processes for encryption and decryption

Figure 1 – Generating Modulus and Eulers Totient



```java
public class EncryptionManager {
    private static final Dotenv dotenv = Dotenv.load();
    private static final String PRIME1 = dotenv.get(key:"PRIME1");
    private static final String PRIME2 = dotenv.get(key:"PRIME2");

    private final long n; // Modulus for public and private keys
    private final long totient; // Euler's totient phi(N)
    private final long e; // Public key exponent

    public EncryptionManager() {
        long p1 = Long.parseLong(PRIME1);
        long p2 = Long.parseLong(PRIME2);

        // Calculate modulus N
        n = p1 * p2;

        // Calculate Euler's totient φ(N)
        totient = getEulersTotient(p1, p2);

        // Choose e (public key) coprime with totient
        e = generateCoprime(totient);
```

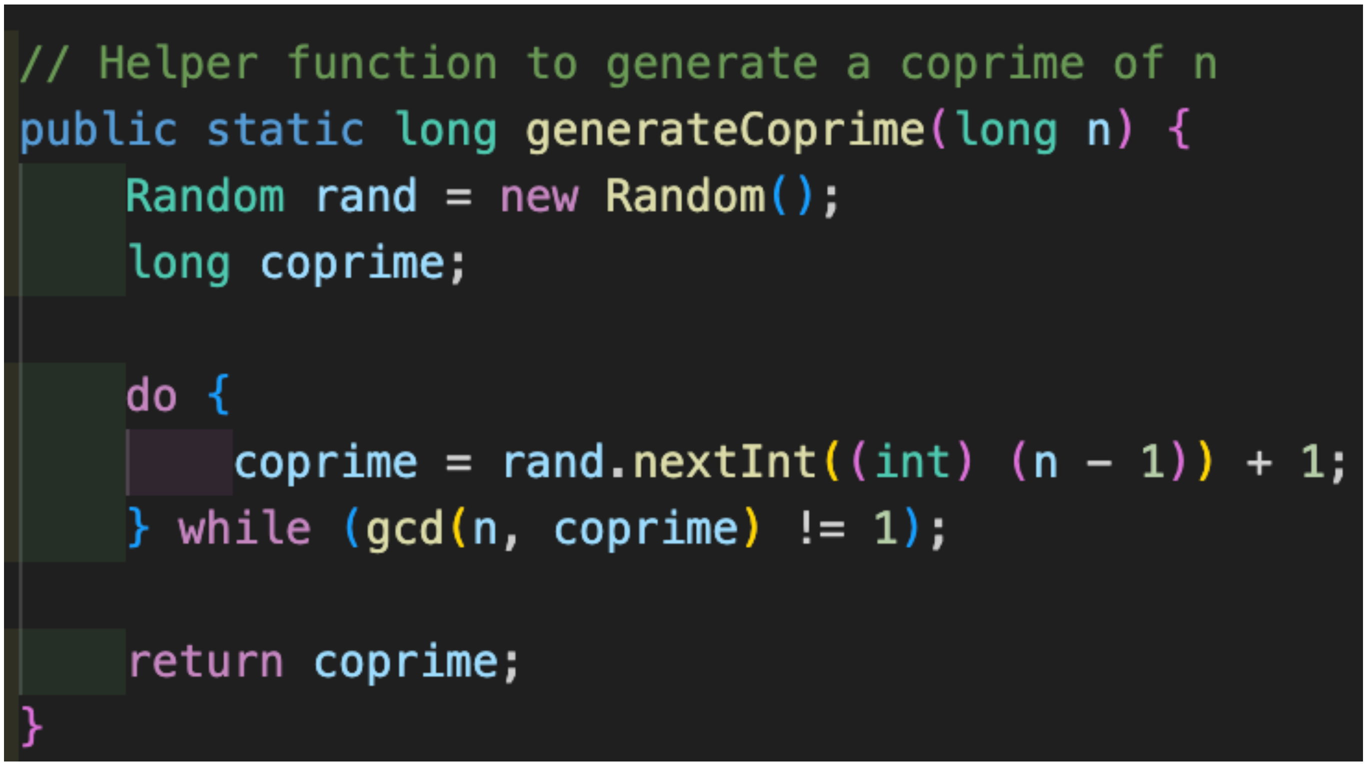
First, I chose two primes p and q and calculate the modulus n. I selected two large prime numbers, p and q, because their product creates a foundation that's hard to factor, which is essential for security. The size of n determines the range of data we can encrypt securely.

$$n = p \times q$$

Then we calculate the euler's totient m, which is basically a mathematical trick that helps ensure only specific numbers will work as encryption keys.

$$\phi(n) = (p-1) \times (q-1)$$

Figure 2 – Generating Public Encryption Key e



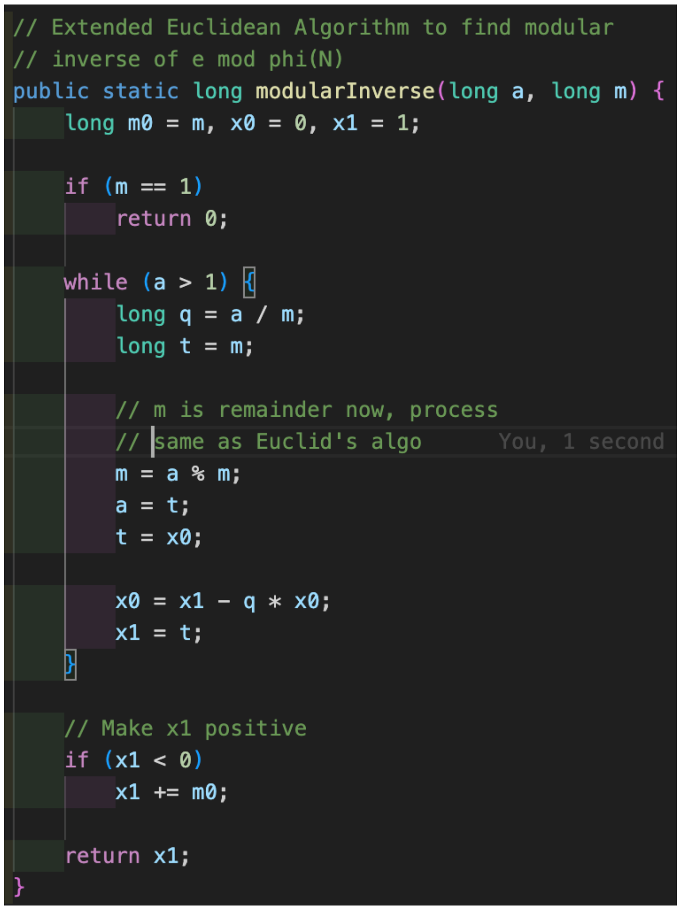
```java
// Helper function to generate a coprime of n
public static long generateCoprime(long n) {
    Random rand = new Random();
    long coprime;

    do {
        coprime = rand.nextInt((int) (n - 1)) + 1;
    } while (gcd(n, coprime) != 1);

    return coprime;
}
```

Encryption key e must be a coprime of the totient because it creates a foundation that's hard to factor, there has an increased factor of message security.

I've used a RNG for generate the coprime because then it is ahrder to guess, this public key is also stored to generate the decryption key.

‘

Figure 3 – Generating Private Decryption Key d



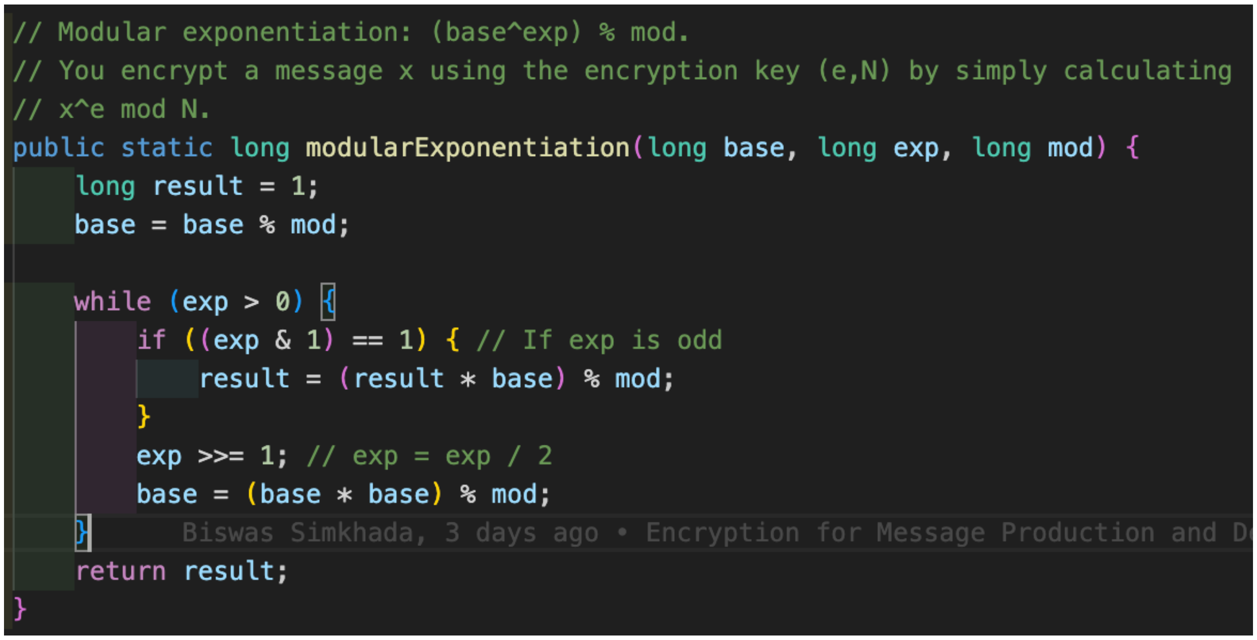
```
// Extended Euclidean Algorithm to find modular
// inverse of e mod phi(N)
public static long modularInverse(long a, long m) {
    long m0 = m, x0 = 0, x1 = 1;

    if (m == 1)
        return 0;

    while (a > 1) {
        long q = a / m;
        long t = m;

        // m is remainder now, process
        // same as Euclid's algo        You, 1 second
        m = a % m;
        a = t;
        t = x0;

        x0 = x1 - q * x0;
        x1 = t;
    }

    // Make x1 positive
    if (x1 < 0)
        x1 += m0;

    return x1;
}
```

I used extended version of the euclidian algorithm that we learnt from MATH1131 to find the modular inverse of $e \bmod \phi(n)$. A video demonstation to find said inverse using the algorithm can be found here.

Essentialy, d is the inverse of $e \bmod \phi(n)$ if and only if:

$$d \times e \equiv 1 \bmod \phi(N)$$

Figure 4 – Encryption and Decryption using Modular Exponention



```
// Modular exponentiation: (base^exp) % mod.
// You encrypt a message x using the encryption key (e,N) by simply calculating
// x^e mod N.
public static long modularExponentiation(long base, long exp, long mod) {
    long result = 1;
    base = base % mod;

    while (exp > 0) {
        if ((exp & 1) == 1) { // If exp is odd
            result = (result * base) % mod;
        }
        exp >>= 1; // exp = exp / 2
        base = (base * base) % mod;
    }      Biswas Simkhada, 3 days ago • Encryption for Message Production and De
    return result;
}
```

Both Encryption and Decryption in Tributary Cluster uses Modular Exponention. In encryption with each base representation of a message, x, the user is trying to encrypt, RSA uses this formula.

$$x^e \bmod n$$

During decryption, when convert an encrypted long value, y, back to its original string/value, RSA uses this formula.

$$y^d \bmod n$$

As can be seen, they both use the same formula only with differing exponents and bases in their modular exponention. The modular inverse used to calculate the private decryption key allows this to happen

Conclusion

In the end, coding up RSA and RBAC in the Tributary Cluster was a huge part of my enjoyment in this project. By integrating token-based authentication and using RSA's asymmetric encryption, I made sure that only everything transfer of data in my system and at least some level of basic security behind it.

Diving into the math behind RSA was interesting. Generating the modulus and Euler's totient for the encryption keys. Picking two large prime numbers p and q to calculate $n = p \times q$. Then, calculating the totient, $\phi(\mathrm{n}) = (\mathrm{p} - 1) \times (\mathrm{q} - 1)$ for key generation. Best part was the throwback I had to MATH1131 using the extended Euclidean algorithm to compute the modular inverse

Overall, integrating RSA encryption into the Tributary Cluster taught me a lot about securing data transfer while keeping things efficient and type safe. Working through the encryption and decryption processes and integrating RBAC gave me a better understanding of how to protect information in real-world applications., Keeping data safe while building scalable data processing systems showed me just how important security is in event-driven architectures.

References

Nahum, Dotan. "Misconfigured Kafdrop Puts Companies' Apache Kafka Completely Exposed." *Security Boulevard*, 6 Dec. 2021, https://securityboulevard.com/2021/12/misconfigured-kafdrop-puts-companies-apache-kafka-completely-exposed/.

Fitzpatrick, Scott. "Security Considerations for Data Stream Processing." *CyberArk Developer*, CyberArk Software Ltd. 28 Feb. 2021, https://developer.cyberark.com/blog/security-considerations-for-data-stream-processing/

"Learn How RSA Works." *Security Engineering Lecture Slides 2024*, SECedu, OpenLearning, https://www.openlearning.com/secedu/courses/security-engineering-lecture-slides-2024/activities/learnhowrsaworks/

Best Friends Farm. "Extended Euclidean Algorithm and Inverse Modulo Tutorial." *YouTube*, 21 Aug. 2013, https://www.youtube.com/watch?v=0b6YgIJRkFg