



Developing a Chatting Application Using Apache Kafka

Minh Duc Pham

BACHELOR'S THESIS

March 2023

Degree Programme in Software Engineering

ABSTRACT

Tampereen ammattikorkeakoulu
Tampere University of Applied Sciences
Degree Programme in Software Engineering

PHAM, MINH DUC
Developing a Chatting Application Using Apache Kafka

Bachelor's thesis 28 pages
March 2023

Apache Kafka is a popular distributed streaming platform that has gained immense popularity in recent years due to its high performance and scalability. This thesis presents the use of Apache Kafka in developing a chatting application. The primary objective of this thesis was to design and implement a real-time messaging system using Apache Kafka, focusing on its scalability and fault-tolerance capabilities. The objective was to make the system able to handle large volumes of messages while maintaining an acceptable response time.

The development process involved designing the system architecture, configuring Kafka and ZooKeeper, and conducting tests. The tools and technologies that were used in the practical work for the thesis are Apache Kafka, ZooKeeper, Spring Boot, ReactJS and JMeter.

The proposed chat application utilized the publish-subscribe messaging pattern provided by Kafka. This thesis also discusses the architecture and design of the chat application and how the features of Kafka, such as message replication, can help in ensuring data consistency and fault tolerance.

Along the development phase, JMeter was adopted for load testing and performance measurement. After evaluating the proposed chat application, Apache Kafka has proven to be an effective solution for building a durable and scalable messaging system.

Key words: distributed systems, Kafka, Spring Boot, chat application

CONTENTS

1	INTRODUCTION	5
2	BACKGROUND AND OBJECTIVE.....	6
2.1	History of chatting application	6
2.2	Project background and purpose	7
2.3	Challenges and solutions	8
3	TECHNOLOGIES	10
3.1	JavaScript programming language.....	10
3.2	ReactJS framework.....	10
3.3	Java programming language.....	11
3.4	Spring Boot	12
3.5	Apache Kafka.....	12
3.6	Theory of leader election.....	14
3.7	Apache ZooKeeper	14
4	TECHNICAL IMPLEMENTATION.....	16
4.1	Software architecture	16
4.2	Kafka configuration	18
4.3	Message model.....	19
4.4	Producer and consumer implementation.....	20
4.4.1	Producer implementation.....	20
4.4.2	Consumer implementation.....	21
4.5	Testing	23
4.5.1	JMeter configuration	23
4.5.2	Results	24
5	CONCLUSIONS AND DISCUSSION	26
	REFERENCES	27

ABBREVIATIONS AND TERMS

HTTP	Hypertext Transfer Protocol used for transmitting data over the internet.
WebSocket	A protocol used for real time, bidirectional communication between a web browser and a server.
POST	A type of HTTP request method used to submit data to a server.
API	Application Programming Interface.
Cluster	A logical grouping of processes in a distributed system working together to achieve a common goal.

1 INTRODUCTION

Chat applications have been important means of communication throughout the 21st century. With a chatting application, people can communicate and interact with each other in real time despite of geographical location. Therefore, real-time messaging is a critical feature of any chat application, as users expect instant delivery of their messages.

One method of developing chat applications is to implement restful API so clients can send HTTP requests to the server to deliver or retrieve messages. However, this solution has major disadvantage as clients will not be notified when new messages are available. Another alternative is using WebSocket protocol to create connections between clients and servers. Nevertheless, if WebSocket sessions fail, messages will be lost.

To address this problem, Apache Kafka can be used in developing chat systems. Kafka is a suitable option for large-scale message processing applications because it provides superior throughput, built-in partitioning, replication, and fault-tolerance than the majority of messaging systems (Apache Kafka n.d.). As a result, Apache Kafka performs effectively as an alternative to more conventional message brokers such as ActiveMQ and RabbitMQ. A message broker acts as a central hub for receiving, routing, and delivering messages from one system to another. Twitter is an example of a popular message-processing application that uses Apache Kafka. At Twitter, Kafka is used for real-time processing of tweets, user interactions, and other data streams (Su 2018).

In this thesis, the use of Apache Kafka is proposed to develop a real-time chat messaging. The availability and scalability features of Kafka are leveraged to ensure that messages are not lost even in the event of failure. The chat application is implemented using WebSocket protocol to provide full-duplex and real-time communication between users. The proposed solution ensures that messages are delivered to users in real time and without the need of polling. This resolution provides a solid foundation to develop a real-time chat application that distributes messages in acceptable time and is highly reliable.

2 BACKGROUND AND OBJECTIVE

2.1 History of chatting application

CompuServe's CB Simulator, which was released in 1980, was largely regarded as the first specialized online chat service (Barot & Oren 2015, 21). However, it was not until the rise of the internet and personal computers during the 1990s that new chat applications were more widely accessible to the public. According to Barot and Oren (2015, 21), chat applications at this time, such as ICQ, AOL Instant Messenger, and MSN Messenger, were rising in popularity among public consumers and helped to popularize instant messaging as a form of communication.

In the early 2000s, due to the rise of social media platforms, companies started to develop new chat applications that could be integrated into these platforms. For instance, Facebook launched its chat feature in 2008, allowing users to communicate with each other in real time while logged into the social network's account.

In the late 2000s and early 2010s, chatting applications were gaining more popularity among the public due to the advancement of mobile technology and smartphones (Innoinstant 2022). WhatsApp, for example, was launched in 2009 and quickly became one of the most popular chat applications in the world. As of 2021, WhatsApp had 2.2 billion active users (Iqbal 2023).

Today, chat applications are an essential part of modern communication. Not only can they send messages but also location information and pictures. Moreover, with chatting applications, users can now make audio or video call. The chat applications continue to evolve, with new features and technologies being introduced to enhance the user experience and improve the security and reliability of these applications.

2.2 Project background and purpose

Chat applications have grown in popularity in recent years as they provide an easy and convenient way for people to communicate instantaneously with each other. Initially, an online chat application was a software application that enabled users to send and receive text over the internet (Tarud 2021). As the internet has improved, chat applications have evolved over time from basic text-based messaging systems to more sophisticated applications that support multimedia content and advanced features such as group messaging, file sharing, and voice/video calls.

The rise of smartphones and mobile computing has contributed to the popularity of chat applications, as users can now get access to their chat applications from anywhere with their mobile devices. As a result, Chat applications are widely used for variety of purposes including personal communication between friends and family, professional communication between colleagues, and customer service between customers and businesses.

Chat applications have also become crucial in the digital workplace since they allow remote teams to communicate and collaborate effectively. Due to the COVID-19 pandemic - forcing many organizations to adopt remote work, the use of chat applications has become even more widespread.

However, building a chat application that can handle high traffic and large volumes of data in real time presents significant challenges. Developers need to consider factors such as reliability, scalability, and performance when designing and implementing a chat application. There are various messaging technologies available to address these challenges, including RabbitMQ and Apache Kafka. Both of these technologies are open-source messaging systems that can be used for chat applications, but they have some differences in their design and capabilities that may make them more suitable for certain specific use cases.

RabbitMQ supports different messaging patterns, including direct exchange, topic exchange, and fanout exchange, while Kafka uses a publish-subscribe mes-

saging pattern. In a chat application, where users need to subscribe to chat channels and receive messages in real time, the publish-subscribe pattern of Kafka may be more suitable. Compared to Kafka, RabbitMQ, while still fast and reliable, may have slightly higher latency due to its more complex architecture. Moreover, Kafka can handle high volumes of data and has excellent scalability, making it a natural fit for chat applications requiring high throughput and real-time processing of large volumes of messages.

While RabbitMQ and Apache Kafka are compatible with chat applications, Kafka may be more appropriate if low-latency messaging, high throughput, and real-time processing of messages are required. On the other hand, RabbitMQ may be more suitable if a more complex messaging pattern is needed. Therefore, after careful analysis and evaluation, Apache Kafka was chosen as the messaging platform for the proposed real-time chat application.

For the development of the proposed chat application, Spring Boot and ReactJS were selected. The event-driven architecture of Spring Boot facilitated the handling of real-time messaging, while ReactJS simplified the development of the implemented application's front end. However, there are different frameworks or technologies that could be used with Apache Kafka to build the chat application. Examples of the programming languages that Kafka supports are Go, Python, C/C++, and many more.

2.3 Challenges and solutions

The proposed chat application faces several challenges that need to be addressed to ensure its success. One of the primary challenges is real-time messaging. Messages should be delivered instantly and without delay. This requires a robust messaging system that can handle real-time communication between users. If using HTTP protocol, there will be delays in delivering data as HTTP needs to finish a complete process including a request from the client and a response from the server to the corresponding request (Figure 1). Moreover, as shown in Figure 1, the client needs to send a request to the server to receive

messages with an HTTP connection. In order to overcome this problem, WebSocket protocol can be used to enable real-time communication between users. With WebSocket, a connection between the client and the server is established after the client makes a request to the server and receives a response from the server (Figure 1). By using WebSocket protocol, the need for polling can be eliminated and the chat application can deliver messages instantly to users.

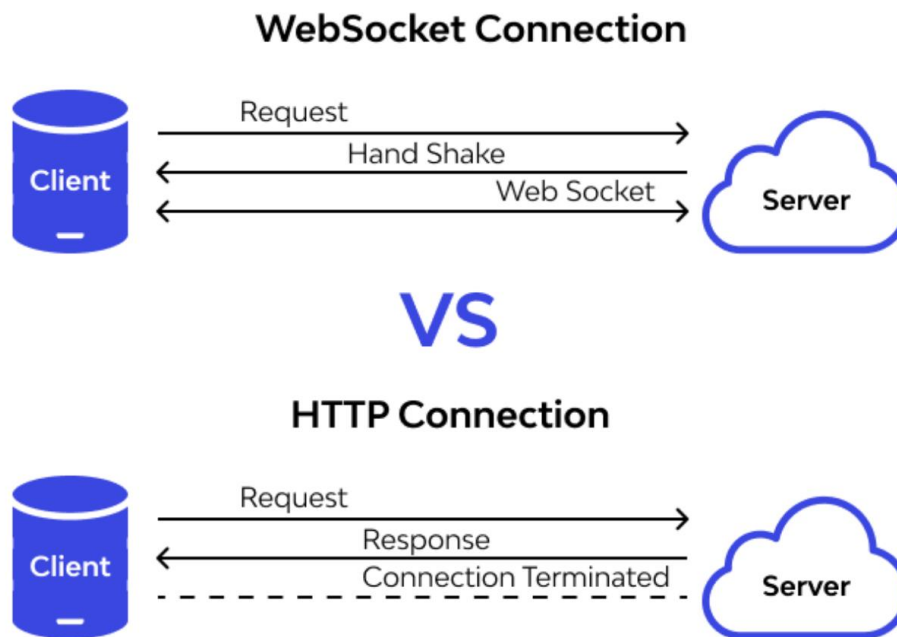


FIGURE 1. WebSocket protocol versus HTTP protocol (Wallarm n.d.).

Another critical challenge is availability. In the event of failure, the application must guarantee that messages are not lost as well as users can continue communicating without interruption. One of the features of Kafka is the replication factor which enables replicating data through different nodes - one node is the leader, and it can replicate data to multiple nodes. This feature enables the application to be fault tolerant.

The next challenge is scalability. The application must handle high traffic and large volumes of data. As the number of users escalates, it may be difficult to route data from clients to servers and vice versa. To address this challenge, Kafka topics and partitions can be used to store and distribute data among users, allowing the application to scale horizontally as the traffic increases. Kafka serves as a cluster of message brokers to distribute data accordingly and avoid a single point of failure.

3 TECHNOLOGIES

3.1 JavaScript programming language

JavaScript is a high-level, interpreted programming language used for creating dynamic, interactive web pages and web applications (Tomar & Dangi 2021, 1). It was originally invented by Brendan Eich at Netscape, in collaboration with Sun Microsystems, in 1995 and has since become one of the most widely used programming languages on the web (McFarland 2008, 2).

According to Keith (2006, 1), web browsers were basic programs that could display hypertext pages until the birth of JavaScript. The features of JavaScript allow developers to add interactivity to web pages, such as displaying pop-up windows, validating user input, and responding to user actions such as mouse clicks and keyboard input. In addition, JavaScript is used for creating complex web applications, browser games, and mobile applications.

JavaScript is a versatile language that can be used in conjunction with HTML and CSS to create rich, interactive web experiences. Today, JavaScript is maintained and developed by ECMA, and it is constantly evolving with new features being added regularly.

3.2 ReactJS framework

ReactJS is a popular open-source and free JavaScript framework for building dynamic user interfaces (Fedosejev 2015, 2). It was developed by Facebook (currently Meta) and is now maintained by a community of developers. ReactJS allows developers to create reusable UI (user interface) components and efficiently controls how the UI is updated in response to user input or changing data.

One of the main advantages of ReactJS is its use of virtual DOM (Document Object Model), which allows web browsers to update and render UI components

faster. The framework also offers declarative programming model, helping developers easily manage complex UI components as well as saving time.

ReactJS has gained widespread adoption in the web development community and is used by many tech giants including Facebook, Netflix, and Airbnb. The framework's success is partly because of its adaptability, simplicity, and compatibility with other popular front-end libraries and framework. According to a survey among more than 70,000 developers conducted by Stack Overflow (2022), ReactJS is the second most popular technologies, only behind NodeJS.

3.3 Java programming language

Java is a class-based object-oriented programming language which was first released by Sun Microsystems in 1995 (Savitch 2018, 49). Since then, it has become one of the most widely used programming languages in the world, with applications ranging from desktop and web applications to mobile and embedded systems.

Java's popularity can be attributed to its scalability, security features, and support for distributed computing. Java is designed to be platform-independent, and its philosophy is "Write once, run anywhere" (WORA) (Sufyan 2022, 2). This feature is possible because Java code is compiled into an intermediate form called bytecode, which is executed by the Java Virtual Machine (JVM) on different platforms. Furthermore, Java's object-oriented features make it an ideal language for building complex applications. By encapsulating data and behavior objects, Java allows developers to manage and maintain large codebases with ease.

In present day, Java has a large and active community of developers due to it being an open-source language. After acquiring Sun Microsystem in 2009, Oracle Corporation has since continued to develop and support Java.

3.4 Spring Boot

Spring Boot is an open-source Java framework which helps to ease and accelerate the process of developing a web application (Spring n.d.). It was developed by Pivotal Software, the same company behind the well-known Spring Framework.

One of the key benefits of Spring Boot is that it simplifies the setup and configuration that needed in a Java application. By using Spring Boot, programmers can focus on writing code rather than having to configure the application. This is possible because Spring Boot uses a convention-over-configuration approach. It also provides a set of starter dependencies that developers can use to quickly add functionalities to their applications.

With Spring Boot, web application development is intended to be efficient and flexible. It offers developers a collection of pre-configured modules that they can quickly incorporate into their projects. These modules, among others, cover topics including security, web access, and web services.

Overall, Spring Boot is a strong and adaptable framework that can assist developers building highly scalable web applications since Spring Boot is compatible with microservices architecture. Developers can use Spring Boot's auto-configuration feature to automatically configure their services based on the dependencies they use. This reduces the amount of boilerplate code that needs to be written, making it more convenient to develop and maintain microservices.

3.5 Apache Kafka

Apache Kafka is an open-source distributed streaming platform that was originally developed by LinkedIn in 2011 (Garg 2013, 22). LinkedIn's initial purpose was to design a system that could handle large volumes of data in real-time and provide a scalable, fault-tolerant platform to store and process data.

Kafka is built around the concept of a distributed commit log, which allows data to be processed and stored in real time across a cluster of machines (Thein 2014, 9478). Consequently, Kafka can process millions of events per second. Kafka is used by many big companies worldwide, including Box, Goldman Sachs, Cisco, and many more (Apache Kafka n.d.). At LinkedIn, 7 trillion messages are being processed every day by Kafka (LinkedIn Engineering n.d.).

Apache Kafka is also highly flexible and can be suitable for various use cases. In addition to serving as messaging system to send data between servers, Kafka can be used as a storage system for storing massive amount of data, or as a streaming platform for processing real-time data (Wu, Shang, Peng & Wolter 2020, 207).

Figure 2 illustrates the overall architecture of Apache Kafka. Moreover, as can be seen in Figure 2, producers publish messages to a topic, which may be handled by multiple brokers, and consumers listen for messages from that topic. By implementing the theory of leader election, Kafka decides which brokers will handle the read-and-write operation for each topic partition. Additionally, Kafka uses ZooKeeper as a service registry for Kafka's message brokers. Leader election theory and Apache ZooKeeper will be explained further in the next part.

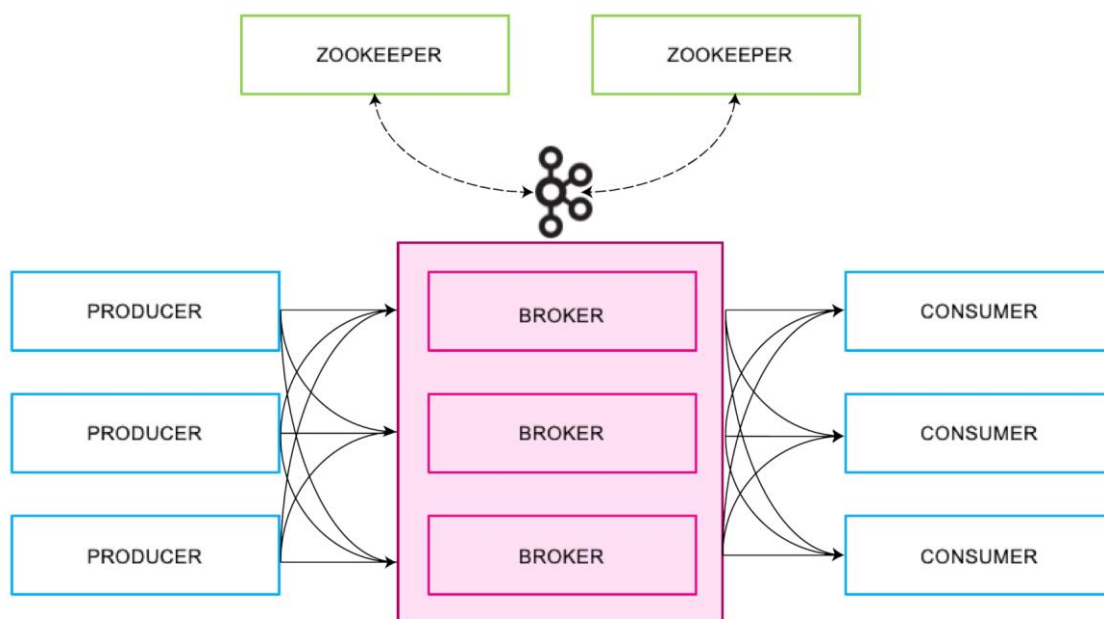


FIGURE 2. Kafka architectures with ZooKeeper (Splunk 2021).

3.6 Theory of leader election

Leader election is a fundamental problem in distributed computing, where a group of nodes must agree on a single leader node that will coordinate the actions of the group (Dolev, Israeli & Moran 1997, 424). The purpose of leader election is to ensure that there is a single point of control within the system, allowing for efficient decision-making and coordination (EffatParvar et al. 2010, V2-6).

Leader election can have a challenging problem because nodes may fail or go offline at anytime, leading to network partitions and inconsistencies in the state of the system. Therefore, the leader election theory must be designed to be fault-tolerant and scalable.

Kafka's leader election algorithm is a core component of its distributed architecture (Manish & Singh 2017, 29). It is based on the principles of distributed consensus, which ensures that an agreement is reached among a group of nodes over a decision and that the cluster can continue to operate in the presence of failures (Wang et al. 2015, 1654). In Kafka, a node is typically referred to a message broker, which is an instance of the Kafka server that runs on a physical or virtual machine.

Kafka uses ZooKeeper coordination service to maintain a consistent overview of the cluster membership and determine which broker is the leader for each partition. When a leader broker goes offline, ZooKeeper will detect the failure and trigger a leader election process. Because of the ZooKeeper coordination service, the remaining brokers in the cluster are able to exchange information and settle to an agreement on which broker should be the new leader.

3.7 Apache ZooKeeper

Apache ZooKeeper is a distributed coordination service that provides a centralized infrastructure for managing configuration, synchronization, and naming services in a distributed system (Apache ZooKeeper, n.d.). It is an open-

source project maintained by the Apache Software Foundation. Additionally, ZooKeeper is designed to be fault-tolerant and scalable.

ZooKeeper coordination service is one of the most significant features of ZooKeeper as it allows to manage state of the system even in the presence of failure, which is achieved through a consensus protocol. Kafka uses ZooKeeper as a registry for brokers as well as for monitoring and detecting failure. Each time a broker goes down, Kafka is informed by ZooKeeper and redistributes data accordingly to the remaining brokers in the cluster. By using ZooKeeper to manage metadata and coordinate cluster operations, Kafka is able to provide fault tolerance and high reliability, ensuring that messages are always available.

4 TECHNICAL IMPLEMENTATION

4.1 Software architecture

The chat application's client-side allows users to create an HTTP request containing their messages, usernames, and timestamp to the API endpoint. In Kafka, the pub/sub (publish/subscribe) model is a messaging pattern where publishers (also known as producers) send messages to a Kafka topic, and subscribers (also known as consumers) receive messages from that topic. Figure 3 illustrates the overall architecture of the implemented application. Upon receiving users' messages, the server passes these messages to the Kafka producer, which publishes them to the Kafka topic (Figure 3). Kafka topic is a category or feed name to which messages are published by Kafka producers and from which Kafka consumers can consume these messages. As can be seen in Figure 3, Kafka consumers listen to the messages and broadcast them to the WebSocket topic.

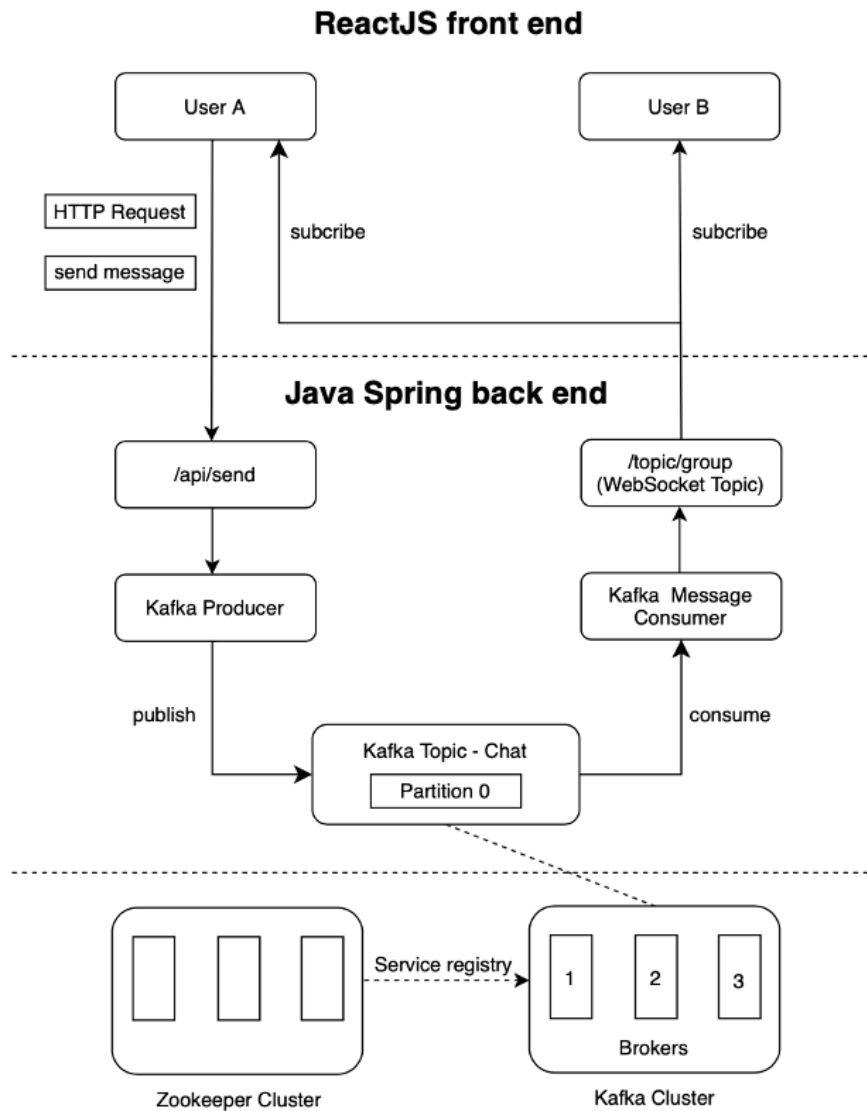


FIGURE 3. Chat application architecture.

Kafka topics are managed by Kafka brokers, which register themselves with the ZooKeeper coordination service (Figure 3). Each partition has only one leader broker responsible for read and write operations, while the other brokers act as followers and replicate data in case of failure. Moreover, Spring Boot's event-driven architecture simplifies application's logic by handling all HTTP requests sent to API endpoints with preconfigured annotations (Picture 1).

```

@Autowired
private KafkaTemplate<String, Message> kafkaTemplate;

@PostMapping(value = "/api/send", consumes = "application/json", produces = "application/json")
public void sendMessage(@RequestBody Message message) {
    LocalDateTime myDateObj = LocalDateTime.now();
    DateTimeFormatter myFormatObj = DateTimeFormatter.ofPattern("dd-MM-yyyy HH:mm:ss");

    message.setTimestamp(myDateObj.format(myFormatObj));

    try {
        kafkaTemplate.send(KAFKA_CHAT_TOPIC, message).get();
    } catch (InterruptedException | ExecutionException e) {
        throw new RuntimeException(e);
    }
}

```

PICTURE 1. Handling POST requests with Spring Boot.

4.2 Kafka configuration

In the implemented application, only one topic was created – chat topic. A Kafka topic is a category or feed name representing a stream of data in Kafka. It is a logical container that Kafka producers use to publish messages and Kafka consumers use to consume these messages. A topic is identified by a unique name, and it can be divided into multiple partitions to enable parallelism and provide fault tolerance.

A Kafka partition is a subset of a topic that contains an ordered and immutable sequence of messages. However, there is no global ordering between Kafka partitions in a topic. Each partition is replicated across multiple brokers in a Kafka cluster to provide fault tolerance and high availability. The number of partitions in a topic determines the maximum parallelism of message consumption by Kafka consumers. Kafka uses a partitioning scheme to distribute messages across partitions based on the provided message key or a hash of the message key if not provided. In the chat topic of the implemented application, the number of partitions was configured to one as the application has one group chat only. In order to have multiple groups chat, the number of partitions in the chat topic can be increased and each group chat can have a message key so that the producer can produce the messages to the desired partition.

4.3 Message model

A message model is the structure of a message that can be exchanged between services. It defines how data is organized and communicated between different components. Moreover, a message model specifies the structure and content of the messages being exchanged, including the data element, data types, and any additional metadata.

Picture 2 presents the message model of the implemented application, which includes the sender, content, and timestamp of the message. In addition, the message model defines the payload that will be exchanged between the client and server.

```
public class Message {
    private String sender;
    private String content;
    private String timestamp;

    public Message() {}

    public void setTimestamp(String timestamp) { this.timestamp = timestamp; }

    public void setSender(String sender) { this.sender = sender; }

    public void setContent(String content) { this.content = content; }

    public String getTimestamp() { return timestamp; }

    public String getSender() { return sender; }

    public String getContent() { return content; }

    @Override
    public String toString() {
        return "Message{" +
            "sender='" + sender + '\'' +
            ", content='" + content + '\'' +
            ", timestamp='" + timestamp + '\'' +
            '}';
    }
}
```

PICTURE 2. Message model.

4.4 Producer and consumer implementation

In Kafka, producers publish messages to a Kafka topic without knowing which consumers, if any, will receive them. Consumers who subscribe to one or more topics will receive all messages published on those topics. This decoupling of producers and consumers allows for flexible and scalable communication between different parts of the distributed system. This feature also allows the application to hide system logic from the client side.

In the publish-subscribe model, Kafka brokers act as intermediaries between producers and consumers. Producers send messages to brokers, which then distribute the messages to all subscribed consumers. To achieve fault tolerance in the chat application, three Kafka brokers have been initialized and the replication factor set to three, meaning data will be copied to two other brokers and one broker acts as a leader in control of read and write operations. This ensures that if the leader broker goes down, messages are still available and not lost.

4.4.1 Producer implementation

Picture 3 demonstrates how to configure Kafka producer with Spring Boot. To configure a Kafka producer, as can be seen in Picture 3, a variable called “configs” was created. This variable is a key-value pairs container that can be passed to the Kafka producer. The first key-value pair that was added is the bootstrap server addresses. This pair contains a list of Kafka addresses that the producer will use to establish initial connection to the entire Kafka cluster.

Kafka allows users to use basically any Java object as a key and any Java object of any type as a value. However, for each type of key and value, users need to tell the Kafka library how to serialize that object to a binary format so it can be sent over the network. Therefore, in Picture 3, key serializer class and value serializer class were set to String serializer and Json serializer respectively.

In addition, **kafkaTemplate bean**, a type of Java Bean, was created to perform operations such as sending messages to Kafka topic (Picture 3). A Java Bean is

a reusable software component corresponding to a set of conventions for properties, events, and methods.

```

@EnableKafka
@Configuration
public class KafkaProducerConfig {
    private static final String BOOTSTRAP_SERVERS = "localhost:9092,localhost:9093,localhost:9094";

    @Bean
    public ProducerFactory<String, Message> producerFactory() {
        Map<String, Object> configs = new HashMap<>();

        configs.put(ProducerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ProducerConfig.KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class);
        configs.put(ProducerConfig.VALUE_SERIALIZER_CLASS_CONFIG, JsonSerializer.class);

        return new DefaultKafkaProducerFactory<>(configs);
    }

    @Bean
    public KafkaTemplate<String, Message> kafkaTemplate() {
        return new KafkaTemplate<>(producerFactory());
    }
}

```

PICTURE 3. Producer configuration.

Next, an endpoint was needed for handling messages and publishing them to the Kafka topic (Picture 1). When a user creates a Post request to **api/send**, **kafkaTemplate bean** configured earlier (Picture 3) is then injected and sends messages to the Kafka topic (Picture 1).

4.4.2 Consumer implementation

Likewise when implementing producers, we need to create consumers configuration which has Kafka addresses, consumer groups and deserializer to deserialize data from Kafka topic. Two beans that were configured are **consumerFactory** and **kafkaListenerContainerFactory** (Picture 4).

```

@EnableKafka
@Configuration
public class KafkaConsumerConfig {
    private static final String BOOTSTRAP_SERVERS = "localhost:9092,localhost:9093,localhost:9094";
    private static final String consumerGroup = "group-chat-1";

    @Bean
    public ConsumerFactory<String, Message> consumerFactory() {
        Map<String, Object> configs = new HashMap<>();

        configs.put(ConsumerConfig.BOOTSTRAP_SERVERS_CONFIG, BOOTSTRAP_SERVERS);
        configs.put(ConsumerConfig.KEY_DESERIALIZER_CLASS_CONFIG, StringDeserializer.class);
        configs.put(ConsumerConfig.VALUE_DESERIALIZER_CLASS_CONFIG, JsonDeserializer.class);
        configs.put(ConsumerConfig.GROUP_ID_CONFIG, consumerGroup);
        configs.put(ConsumerConfig.AUTO_OFFSET_RESET_CONFIG, "earliest");

        return new DefaultKafkaConsumerFactory<>(configs, new StringDeserializer(), new JsonDeserializer<>(Message.class));
    }
}

@Bean
public ConcurrentKafkaListenerContainerFactory<String, Message> kafkaListenerContainerFactory() {
    ConcurrentKafkaListenerContainerFactory<String, Message> factory = new ConcurrentKafkaListenerContainerFactory<>();
    factory.setConsumerFactory(consumerFactory());

    return factory;
}

```

PICTURE 4. Consumer configuration.

Consequently, `@KafkaListener` annotation can now be used to consume messages from a topic. In Picture 5, after consuming messages, `simpMessagingTemplate.convertAndSend()` will convert and broadcast messages to the Web-Socket topic.

```

@Component
public class MessageConsumer {
    private static final String KAFKA_TOPIC = "chat";
    private static final String GROUP_ID = "group-chat-1";

    @Autowired
    SimpMessagingTemplate simpMessagingTemplate;

    @KafkaListener(
        topics = KAFKA_TOPIC,
        groupId = GROUP_ID
    )
    public void listen(Message message) {
        System.out.println("Consumer is part of consumer group " + GROUP_ID);
        System.out.println("Incoming message: " + message);

        simpMessagingTemplate.convertAndSend(" /topic/group", message);
    }
}

```

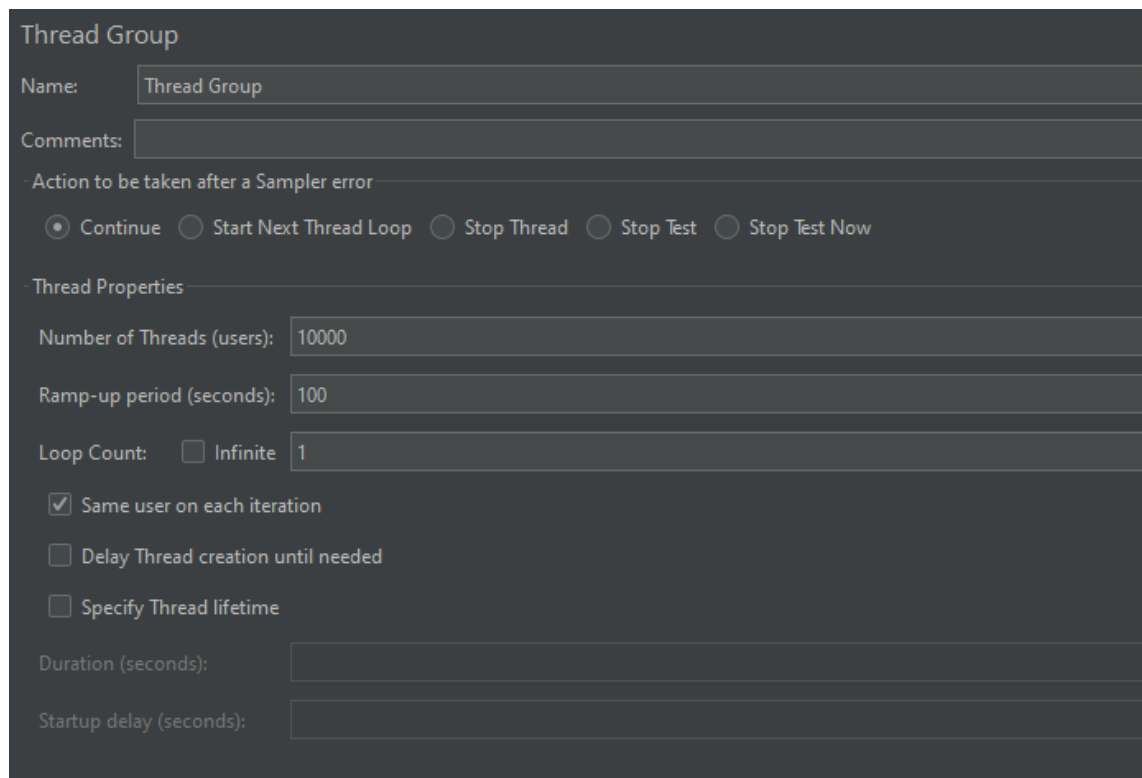
PICTURE 5. Consuming messages.

4.5 Testing

In order to test the chat application, JMeter was used. JMeter is a popular open-source tool for load testing and performance testing of applications (Halili 2008, 16). Load testing is the process of simulating user traffic to an application to determine how well it performs under different loads. JMeter allows users to create test plans that simulate many users sending requests to an application and measure the response time as well as throughput.

4.5.1 JMeter configuration

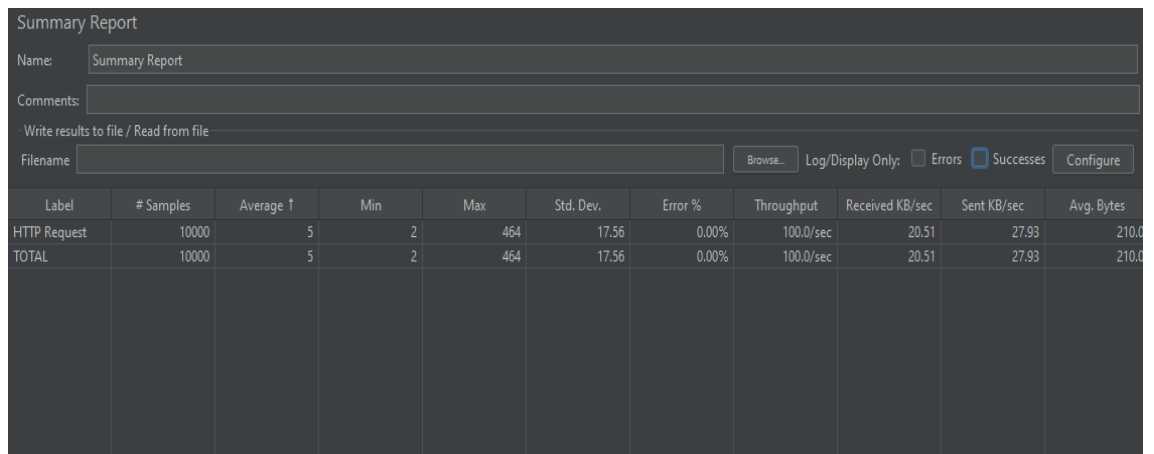
To simulate high traffic, 10,000 users were added to make HTTP requests to **api/send** endpoint, where the messages would be handled and sent to the Kafka topic (Picture 6). The HTTP requests contained a username and message content. The ramp-up period was configured to 100 seconds, meaning 10,000 requests would be completed in 100 seconds.



PICTURE 6. Thread set up in JMeter for testing.

4.5.2 Results

Picture 7 presents the report that was created by JMeter, including the number of samples being processed, min/max elapsed time, error percentage and throughput rate. The throughput rate of the chat application is 100 requests per second which is 6,000 requests per minute meaning the application can handle a large number of users concurrently. In addition, no requests failed during the test which indicates that the system is highly reliable even with high traffic. A log message was also added to the Kafka listener to print out the number of messages being consumed which was 10,000 (Picture 8).



Summary Report

Name: Summary Report

Comments:

Write results to file / Read from file

Filename: Log/Display Only: Errors Successes

Label	# Samples	Average t	Min	Max	Std. Dev.	Error %	Throughput	Received KB/sec	Sent KB/sec	Avg. Bytes
HTTP Request	10000	5	2	464	17.56	0.00%	100.0/sec	20.51	27.93	210.0
TOTAL	10000	5	2	464	17.56	0.00%	100.0/sec	20.51	27.93	210.0

PICTURE 7. Summary report of the test created by JMeter.

```

Number of consumed message: 9994
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 9995
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 9996
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 9997
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 9998
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 9999
Incoming message: Message{sender='Duc', content='hello', timestamp='29-03-2023 13:07:50'}
Number of consumed message: 10000

```

PICTURE 8. Log messages showing number of consumed messages.

Finally, the reliability of the system was experimented. To simulate system's failure, the port that the leader broker ran on was shut down during the load testing. Before the event of failure, there were three Kafka brokers online and Broker 2

was the leader (Picture 9). Moreover, all three brokers were “in sync replica” (in Picture 9 denoted as “ISR”) meaning the data was available to all brokers.

```
D:\Resources\kafka_2.12-2.2.0>.bin\windows\kafka-topics.bat --describe --bootstrap-server localhost:9092 --topic chat
Topic:chat      PartitionCount:1      ReplicationFactor:3      Configs:segment.bytes=1073741824
      Topic: chat      Partition: 0      Leader: 2      Replicas: 0,2,1 Isr: 2,0,1
```

PICTURE 9. Description for chat topic configuration before failure.

After shutting down port 9094 (the port that Broker 2 was running on), the system automatically elected a new leader, which was Broker 0, and updated the status of the data availability by showing that only Broker 0 and Broker 1 were in sync replicas (Picture 10). Overall, the test results indicate that the chat application, by using Kafka, is fault-tolerant and can handle a large number of concurrent users. These tests were designed in order to make an assessment of the functionalities of the chat application under real-world scenarios.

```
D:\Resources\kafka_2.12-2.2.0>.bin\windows\kafka-topics.bat --describe --bootstrap-server localhost:9092 --topic chat
Topic:chat      PartitionCount:1      ReplicationFactor:3      Configs:segment.bytes=1073741824
      Topic: chat      Partition: 0      Leader: 0      Replicas: 0,2,1 Isr: 0,1
```

PICTURE 10. Description for chat topic configuration after failure.

5 CONCLUSIONS AND DISCUSSION

In this thesis, the use of Apache Kafka as a messaging platform for developing a real-time chat application was explored. The proposed chat application was designed and implemented using Kafka. Additionally, the performance and scalability of the implemented application were evaluated through load testing with JMeter.

The load testing and performance analysis indicated that the system could handle a large number of users and messages with high throughput. In the test, 10,000 concurrent users were simulated to send and receive messages simultaneously. It is observed that the chat application was able to process 1,000 messages per second with a throughput rate of 100 requests per second. However, it is noteworthy that the tests could not cover all of the real-world scenarios such as the size of the messages and the number of group chats. These factors would negatively impact the performance of the proposed application since it would increase the time to process each message.

Overall, the results prove that Apache Kafka is a reliable and high-performing messaging platform for developing real-time chat applications. The architecture and features of Kafka are suitable for this use case, and the load testing result showed that Kafka could handle large volumes of messages with high throughput.

In conclusion, the development of a chat application using Kafka has demonstrated its effectiveness in building a scalable and reliable messaging system. The chat application provided a solution that could handle high levels of users traffic even in the event of catastrophes – by using Kafka topics and ZooKeeper coordination service, the system can quickly recover from failures. The performance of the proposed application can be improved through proper configuration and optimization. Further research can focus on developing more features such as file sharing, multiple chat channels, and message encryption. Apache Kafka has created a profound foundation for future enhancements of the chat application.

REFERENCES

Apache Kafka. N.d. Powered By. [website]. Read on 15.03.2023.
<https://kafka.apache.org/powered-by>

Apache Kafka. N.d. Use Cases. [website]. Read on 15.03.2023.
https://kafka.apache.org/uses#uses_messaging

Apache Zookeeper. N.d. Welcome to Apache ZooKeeper. Read on 18.03.2023.
<https://zookeeper.apache.org>

Barot, T. & Oren, E. 2015. Guide to Chat Apps. Tow Center for Digital Journalism. Columbia University. Reports.

Dolev, S., Israeli, I. & Moran, S. 1997. Uniform dynamic self-stabilizing leader election. IEEE Transactions on Parallel and Distributed Systems. 8 (4), 424-440.

EffatParvar, M.R., Yazdani, N., EffatParvar, M., Dadlani, A. & Khonsari, A. 2010. Improved algorithms for leader election in distributed systems. 2010 2nd International Conference on Computer Engineering and Technology. 2, V2-6–V2-10.

Fedosejev, A. 2015. React. js essentials. Packt Publishing Limited.

Garg, N. 2013. Apache Kafka. 1st edition. Birmingham: Packt Publishing.

Halili, E. H. 2008. Apache JMeter. Birmingham: Packt Publishing.

Innoinstant. 2022. The Past, Present and Future of Chat Applications. [blog]. Released on 20.02.2022. Read on 22.03.2023. <https://blog.innoinstant.com/evolution-of-instant-messaging-apps/>

Iqbal, M. 2023. WhatsApp Revenue and Usage Statistics. Released on 08.02.2023. Read on 10.03.2023. <https://www.businessofapps.com/data/whatsapp-statistics/>

Keith, J. 2006. Dom Scripting: Web Design with JavaScript and the Document Object Model. Berkely, CA: Apress.

LinkedIn Engineering. N.d. Kafka Ecosystem at LinkedIn. Read on 15.03.2023.
<https://engineering.linkedin.com/teams/data/data-infrastructure/streams/kafka>

Manish, K. & Singh, C. 2017. Building data streaming applications with Apache Kafka : designing and deploying enterprise messaging queues. 1st edition. Birmingham: Packt.

McFarland, D. S. 2008. JavaScript. 1st edition. Beijing: Pogue Press/O'Reilly.

Savitch, W. 2018. Java. 8th edition. Harlow, United Kingdom: Pearson Education Limit.

Splunk. 2021. Monitoring Kafka Performance with Splunk. [blog]. Released on 08.06.2021. Read on 17.03.2023. https://www.splunk.com/en_us/blog/devops/monitoring-kafka-performance-with-splunk.html

Spring. N.d. Why Spring?. Read on 15.03.2023. <https://spring.io/why-spring>

Stack Overflow. 2022. Developer Survey. Read on 15.03.2023. <https://survey.stackoverflow.co/2022/#overview>

Su. P. 2018. Twitter's Kafka adoption story. [blog]. Released on 28.11.2018. Read on 22.03.2023. https://blog.twitter.com/engineering/en_us/topics/insights/2018/twitters-kafka-adoption-story

Sufyan, B. U. 2022. Mastering Java: a Beginner's Guide. 1st edition. Boca Raton: CRC Press.

Tarud. J. 2021. How Chat Apps Are Evolving. [website]. Updated on 12.03.2021. Read on 17.03.2023. <https://www.koombea.com/blog/chat-apps-evolving/>

Thein, K. M. M. 2014. Apache Kafka: Next generation distributed messaging system. International Journal of Scientific Engineering and Technology Research. 3 (47), 9478–9483.

Tomar, R. & Dangi, S. 2021. JavaScript: Syntax and Practices. Milton: CRC Press LLC.

Wallarm. N.d. WebSocket vs HTTP: How are these 2 Different?. [website]. Read on 16.03.2023. <https://www.wallarm.com/what/websocket-vs-http-how-are-these-2-different>

Wang, G. 2015. Building a Replicated Logging System with Apache Kafka. Proceedings of the VLDB Endowment. 8 (12), 1654–1655.

Wu, H., Shang, Z., Peng, G., & Wolter, K. 2020. A reactive batching strategy of apache kafka for reliable stream processing in real-time. Proceedings - International Symposium on Software Reliability Engineering. ISSRE. 2020, 207–217.