## 1. Chat Application using Kafka (Command Line - Java Example)

### Technologies:

Apache Kafka

Java (for simplicity)

Command Prompt for input/output

### Overview:

One producer sends messages.

Multiple consumers (or same client) receive and display messages.

All messages go through a Kafka **topic**.

### Step-by-Step Guide

#### 1. Set up Kafka

[Download Kafka](https://kafka.apache.org/downloads" \t "C:\\Users\\KIIT\\AppData\\Local\\Temp\\_new)

Start Zookeeper:

bin/zookeeper-server-start.sh config/zookeeper.properties

Start Kafka server:

bin/kafka-server-start.sh config/server.properties

1. Create a Kafka Topic

bin/kafka-topics.sh --create --topic chat-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

Java Chat Producer:

public class ChatProducer {

public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("key.serializer", "org.apache.kafka.common.serialization.StringSerializer");

props.put("value.serializer", "org.apache.kafka.common.serialization.StringSerializer");

Scanner scanner = new Scanner(System.in);

KafkaProducer<String, String> producer = new KafkaProducer<>(props);

while (true) {

System.out.print("You: ");

String message = scanner.nextLine();

producer.send(new ProducerRecord<>("chat-topic", "user1", message));

}

}

}

1. Java Chat Consumer

public class ChatConsumer {

public static void main(String[] args) {

Properties props = new Properties();

props.put("bootstrap.servers", "localhost:9092");

props.put("group.id", "chat-group");

props.put("enable.auto.commit", "true");

props.put("key.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

props.put("value.deserializer", "org.apache.kafka.common.serialization.StringDeserializer");

KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);

consumer.subscribe(Collections.singletonList("chat-topic"));

while (true) {

for (ConsumerRecord<String, String> record : consumer.poll(Duration.ofMillis(100))) {

System.out.println("Friend: " + record.value());

}

}

}

}

## 2. Chat Application using C# Windows Forms (Kafka)

### Technologies:

Apache Kafka

.NET (C# WinForms)

Confluent.Kafka NuGet package

### Overview:

GUI-based producer and consumer.

Kafka handles message distribution.

Each instance acts as a client.

Step-by-Step Setup

#### Install Confluent.Kafka

Install-Package Confluent.Kafka

1. ChatForm.cs (Producer & Consumer)

using System;

using System.Windows.Forms;

using Confluent.Kafka;

using System.Threading;

using System.Threading.Tasks;

namespace KafkaChatWinApp

{

public partial class ChatForm : Form

{

private IProducer<Null, string> producer;

private IConsumer<Null, string> consumer;

private CancellationTokenSource cts;

public ChatForm()

{

InitializeComponent();

var configProducer = new ProducerConfig { BootstrapServers = "localhost:9092" };

producer = new ProducerBuilder<Null, string>(configProducer).Build();

var configConsumer = new ConsumerConfig

{

BootstrapServers = "localhost:9092",

GroupId = Guid.NewGuid().ToString(),

AutoOffsetReset = AutoOffsetReset.Earliest

};

consumer = new ConsumerBuilder<Null, string>(configConsumer).Build();

consumer.Subscribe("chat-topic");

cts = new CancellationTokenSource();

Task.Run(() => ListenForMessages(cts.Token));

}

private async Task ListenForMessages(CancellationToken token)

{

while (!token.IsCancellationRequested)

{

var consumeResult = consumer.Consume(token);

Invoke(new Action(() => chatBox.AppendText($"Friend: {consumeResult.Message.Value}\r\n")));

}

}

private async void sendButton\_Click(object sender, EventArgs e)

{

var message = messageInput.Text;

await producer.ProduceAsync("chat-topic", new Message<Null, string> { Value = message });

chatBox.AppendText($"You: {message}\r\n");

messageInput.Clear();

}

protected override void OnFormClosing(FormClosingEventArgs e)

{

cts.Cancel();

consumer.Close();

base.OnFormClosing(e);

}

}

}

ANALYSIS:-

Here's a detailed **analysis** of both chat application implementations using Kafka—one in Java (command-line) and another in C# (Windows Forms GUI). We'll evaluate them on **architecture, usability, scalability, and practical use cases**, and also compare them to traditional messaging solutions.

**Architecture Overview**

Kafka-Based Chat Model

**Central Messaging Broker:** Kafka acts as the core messaging queue (topic-based).

**Producer:** Any chat client sending messages.

**Consumer:** Any chat client receiving messages (subscribed to Kafka topics).

**Kafka Topic:** Acts as the shared chatroom/channel.

Java Command-Line Chat Application

### Pros:

**Lightweight & Simple:** Very minimal dependencies and infrastructure.

**Good for Learning:** Excellent for understanding Kafka producer/consumer flow.

**Easily Testable:** Run multiple consumers in different terminals to simulate clients.

### Cons:

**No UI:** Limited usability for non-technical users.

**No User Authentication/Management:** All messages are from anonymous users.

**Concurrency/Threading:** No threading model; messages sent and received in different terminals.

**Not Scalable in UI/UX:** Not suited for deployment to end-users.

### Use Cases:

Prototyping Kafka message pipelines.

Educational projects or demos.

Backend chat logging where frontend is handled elsewhere.

C# Windows Forms Chat App (Kafka + GUI)

### Pros:

**User-Friendly Interface:** Suitable for real users, especially on Windows systems.

**Asynchronous Messaging:** Uses background threads to consume messages.

**Real-time Communication:** All clients can see messages in near real-time.

**Multiple Clients Support:** Each instance acts as a separate Kafka client.

### Cons:

**Limited UI Complexity:** Windows Forms is dated compared to WPF/MAUI or web-based UIs.

**No User Identity:** Messages are all "You" and "Friend", lacking true identity support.

**Error Handling:** Minimal error handling on network failure, Kafka broker down, etc.

**No Persistence:** Chat history isn't stored unless Kafka is configured for log retention.

### Use Cases:

Internal company chat tools.

POCs for scalable message-driven GUI apps.

Desktop-based IoT or alert systems using Kafka backend.

Kafka as a Backend for Chat Applications

### Strengths:

**High Throughput:** Kafka can handle thousands of messages per second.

**Scalability:** Add more clients without impacting performance much.

**Reliability:** Kafka ensures durable messaging with commit logs.

**Replay Capability:** Consumers can replay messages from specific offsets.

### Weaknesses:

**Not Built for Chat:** Kafka isn’t designed for low-latency messaging like WebSockets or SignalR.

**Latency:** Slight delay in message consumption may exist.

**No Presence/Typing Indicators:** Requires extra layers for real-time features.

**Overkill for Simple Apps:** Kafka is complex and heavyweight for basic chat systems.

## Kafka Chat vs Traditional Chat Tech

| **Feature** | **Kafka Chat** | **SignalR/WebSockets** | **MQTT (IoT) Chat** |
| --- | --- | --- | --- |
| Real-time Communication | Near-real-time | Real-time | Near-real-time |
| Scalability | High (distributed system) | Medium | High |
| Persistence | Yes (via Kafka log retention) | No (ephemeral) | Limited |
| Offline Message Support | Yes | No | Yes (with broker support) |
| Complexity | Medium-High | Low | Medium |
| Best Use Case | Distributed, scalable logging/chat | Real-time UIs, games | IoT, low bandwidth apps |