### **Topic: Message-brokers (like RabbitMq and Kalka)**

### **Assignment 1: Introduction to RabbitMQ/Kafka**

**Objective**: Understand the basics of message brokers and set up RabbitMQ or Kafka.

**Tasks**:

1. Install RabbitMQ or Kafka on your local machine or use a Docker container to set it up.
2. Create a simple producer and consumer:
   * For RabbitMQ: Use Python with the pika library.
   * For Kafka: Use Python with the confluent-kafka or kafka-python library.
3. Send a "Hello, World!" message from the producer to a queue (RabbitMQ) or topic (Kafka).
4. Consume the message with the consumer and display it in the console.

**Deliverable**: Submit the installation steps, Python code for producer/consumer, and a screenshot of the output.

### **Assignment 2: Fanout Exchange (RabbitMQ) / Publish-Subscribe (Kafka)**

**Objective**: Learn how to implement a publish-subscribe pattern.

**Tasks**:

1. Create a RabbitMQ fanout exchange or a Kafka topic.
2. Write a producer that sends messages like "Update 1", "Update 2", etc., to the exchange or topic.
3. Create two consumers:
   * Consumer 1 prints messages in uppercase.
   * Consumer 2 prints messages in lowercase.
4. Run the producer and both consumers simultaneously and observe the output.

**Deliverable**: Submit the code for the producer, consumers, and screenshots of both consumers' outputs.

### **Assignment 3: Task Queues (RabbitMQ) / Work Distribution (Kafka)**

**Objective**: Implement a task queue to distribute tasks among multiple workers.

**Tasks**:

1. Simulate a scenario where tasks (e.g., processing files) are distributed among multiple workers.
2. The producer sends task descriptions (e.g., "Task 1: Process file A", "Task 2: Process file B") to the queue (RabbitMQ) or topic (Kafka).
3. Write two workers that:
   * Pull tasks from the queue/topic.
   * Simulate task processing by printing the task and sleeping for a random number of seconds.
4. Ensure tasks are evenly distributed between workers.

**Deliverable**: Submit the producer, worker code, and screenshots showing the distribution of tasks among workers.

### **Assignment 4: Priority Queues (RabbitMQ) / Topic-Based Filtering (Kafka)**

**Objective**: Learn how to implement message prioritization or filtering based on topics.

**Tasks**:

1. For RabbitMQ:
   * Create a priority queue with different message priorities (e.g., "High", "Medium", "Low").
   * Send messages with different priorities to the queue.
   * Write a consumer that processes messages based on their priority.
2. For Kafka:
   * Create topics for different types of notifications (e.g., alerts, logs, metrics).
   * Write a producer to send messages to these topics.
   * Create consumers that subscribe only to specific topics and print the messages.

**Deliverable**: Submit the producer/consumer code, configuration files, and screenshots demonstrating prioritization or filtering.

### **Assignment 5: Real-Time Chat Application Simulation**

**Objective**: Build a real-time messaging system using RabbitMQ/Kafka.

**Tasks**:

1. Simulate a real-time chat application where users can join chat rooms.
2. Producer:
   * Simulate a user sending messages to a specific chat room.
3. Consumer:
   * Multiple consumers subscribe to the chat room and receive messages in real time.
4. Bonus:
   * Add a feature where users can join/leave chat rooms dynamically.
   * For Kafka, use partitioning to manage different chat rooms.

**Deliverable**: Submit the producer/consumer code and demonstrate the chat functionality with multiple simulated users.

### **Assignment 6: Monitoring and Scaling**

**Objective**: Learn to monitor message brokers and scale consumers.

**Tasks**:

1. Enable monitoring for RabbitMQ (using the management plugin) or Kafka (using tools like Kafka Manager or JMX metrics).
2. Simulate a high volume of messages from the producer.
3. Scale the number of consumers dynamically to handle the load.
4. Capture the effect of scaling on throughput and latency.

**Deliverable**: Submit screenshots of monitoring tools and observations on scaling behavior.

### **Bonus Assignment 1: Implementing Dead Letter Queues (RabbitMQ/Kafka)**

**Objective**: Handle failed message processing using dead letter queues.

**Tasks**:

1. Create a RabbitMQ queue or Kafka topic for processing tasks.
2. Simulate message failure:
   * For RabbitMQ: Use the x-dead-letter-exchange and x-dead-letter-routing-key arguments to send failed messages to a dead letter queue.
   * For Kafka: Set up a retry mechanism in the consumer to move messages to a "dead letter" topic after 3 failed attempts.
3. Write a script to process the messages in the dead letter queue/topic and log them for further analysis.

**Challenge**: Implement a retry mechanism with exponential backoff before sending a message to the dead letter queue.

**Deliverable**: Submit the producer, consumer, and dead letter handler code with logs showing failed messages.

### **Bonus Assignment 2: Implement Distributed Transaction Processing**

**Objective**: Use RabbitMQ or Kafka to implement distributed transactions.

**Tasks**:

1. Simulate a banking system with two microservices:
   * Service 1: Deduct money from the sender’s account.
   * Service 2: Add money to the receiver’s account.
2. Ensure that the transaction is processed atomically using:
   * For RabbitMQ: Use publisher confirms and acknowledgments.
   * For Kafka: Use the Exactly Once Semantics (EOS) feature.
3. Handle failures in either service and ensure the transaction is either completed or rolled back.

**Challenge**: Implement a compensation mechanism to roll back the transaction in case of a failure.

**Deliverable**: Submit the code, along with logs demonstrating successful and failed transactions with rollback.

### **Bonus Assignment 3: Event-Driven Microservices Architecture**

**Objective**: Design an event-driven system using RabbitMQ/Kafka for real-time order processing.

**Scenario**:

1. Simulate an e-commerce platform with three microservices:
   * **Order Service**: Sends an order event when a new order is placed.
   * **Inventory Service**: Listens to the order event and updates stock.
   * **Notification Service**: Listens to the order event and sends a confirmation email/SMS.
2. Use RabbitMQ exchanges or Kafka topics to decouple the services.
3. Implement retries and error handling for message delivery failures.

**Challenge**:

* Add a payment service that processes payment asynchronously and updates the order status only after successful payment.

**Deliverable**: Submit the code, architecture diagram, and logs showing the workflow.