Exercise 1 - Getting started with Kafka

The demo implementation (and further down the line all other implementation) has been done in a GitHub repository: `https://github.com/buehler/mcs-event-driven-systems/`.

Inside the `./demo` directory, there is a small demo of Kafka using Docker Compose and dotnet. The consumer and producer are implemented in C# using the Confluent.Kafka library. The producer sends one message each second with the unix timestamp and the topic is called `clock`. To run the application, go into the `./demo` directory and run `docker compose up`. It includes and configures Kafka from the root directory and also fires up the producer and consumer (and builds them first if necessary). If any changes are made to the code of the producer and/or consumer, you'll need to rebuild the docker files using `docker compose build`.

Task 2 - Experiments with Kafka

All code for the experiments regarding exercise 2 are exclusively under /exercise1

This means all commands shown below must be run from the corresponding path. ### Basic setup Set the correct KAFKA_ADVERTISED_HOST_NAME in docker-compose.yml (exercise1/docker/)

1. Producer Experiments

Start docker with the producer profile (run from /experiments1/docker)

```
$  docker compose --profile producer up -d
```

There are many two classes for different testing purposes. We did not change the pom file all the time. The poms therefore contain the following as a reminder:

```
<manifest>
   <mainClass>read_EDPO_T1_E1.md</mainClass>
</manifest>
```

To run the services the correct main class must be stated in the command. #### Batch Size & Processing Latency

Uses this main class for this test: ProducerExperimentBatchSizeClicksProducer.java

Note: Must be run from the exercise1/producerTests/target subfolder.

```
$  java -cp pubsub-producer-1.0-SNAPSHOT-jar-with-dependencies.jar com.experiments.ProducerExperimentBat
```

Experiment Setup

- Batch Sizes tested:
  - `1024 bytes`
  - `4096 bytes`
  - `16384 bytes`
  - `65536 bytes`
  - `262144 bytes`
  - `1048576 bytes`
- Total Messages: Each batch size sends `50,000` messages.
- Kafka Producer Settings:
  - `linger.ms = 5`: Ensures a consistent delay before sending messages.
  - `buffer.memory = 32MB`: Default Kafka buffer memory.

Metrics Measured

- Duration (ms): Total time taken to send all messages.
- Throughput (msgs/sec): The number of messages sent per second.
- Average Latency (ms): The average time it takes for a message to be sent and acknowledged by Kafka.

Experiment Results

Batch Size	Messages	Duration (ms)	Throughput	Avg Latency (ms)
1024	50000	1320	37878.79	607.96
4096	50000	559	89445.44	57.59
16384	50000	421	118764.85	6.94
65536	50000	354	141242.94	3.94
262144	50000	411	121654.50	8.18
1048576	50000	365	136986.30	18.46

Observations Throughput: - Throughput increases significantly as the `batch.size` grows. - Larger batch sizes reduce the overhead of network communication by allowing more messages to be sent in a single request, thus improving efficiency. - Maximum throughput is achieved at a batch size of `65536`, with 141242.94 messages/sec.

Latency Trade-off: - Average latency per message decreases and becomes negligible at batch sizes up to `65536`. - However, at larger batch sizes (e.g., `262144` and `1048576`), the average latency increases again to 8.18 ms and 18.46 ms respectively. This is due to the delays introduced by waiting to fill larger batches, despite the gains in throughput.

Key Insights for our project Small Batch Sizes: - When the `batch.size` is too small (e.g., `1024`), both throughput and latency suffer due to frequent network communication and higher per-message overhead. –> We don't want that.

Medium Batch Sizes: - Moderate batch sizes (e.g., `16384 - 65536`) are balanced between throughput and latency. –> Seems suitable for us.

Large Batch Sizes: - While large batch sizes (e.g., `262144` or `1048576`) deliver very high throughput, they slightly compromise latency. –> We prefer immediate message delivery. —

Load Testing Uses this main class for this test: ProducerExperimentLoadTestClicksProducer.java

Note: Must be run from the exercise1/producerTests/target subfolder.

```
$  java -cp pubsub-producer-1.0-SNAPSHOT-jar-with-dependencies.jar com.experiments.ProducerExperimentLoa
```

Experiment Setup

- Base producer properties are loaded from a `producer.properties` configuration file.
- A fixed batch size of `16384` is used for each experiment.
- Up to 5 concurrent producers are tested.
- Total messages to send per experiment: `8,000,000`.
- `acks:1`
- `retries:0`
- Kafka Topic Setup:
  - Single partition and replication factor of one for all topics.
- Experiment Iteration:
  - For each producer count (1 to 5):
    - Producers execute message sending concurrently.
    - Each producer sends its share of the total messages.
    - Experiment duration, throughput, latency, and resource usage are measured and recorded.

Metrics Measured See table.

Experiment Results

Batch Size	Producers	Messages Sent	Duration (ms)	Throughput (msg/sec)	Avg Latency (ms)	Message Drop	Dropped (%)	Avg CPU (%)	Avg Memory (MB)
16384	1	7900776	6075	1300512.76	30.39	99385	1.24	63.79	2281.52
16384	2	7704802	4484	1718287.69	177.64	295198	3.69	89.91	2224.71
16384	3	7249508	4797	1511258.70	422.96	750490	9.38	51.77	2534.69
16384	4	6914848	4377	1579814.48	492.23	1085152	13.56	37.50	1354.18
16384	5	6622923	4202	1576135.89	601.33	1377077	17.21	64.35	1871.41

Observations Throughput: - Maximum throughput was achieved with `2 producers`, reaching 1,718,287.69 messages/sec. - Increasing the number of producers beyond 2 caused slight inconsistencies in throughput performance.

Latency: - As the number of producers increased, average latency per message increased significantly. - Lowest latency observed with 1 producer: 30.39 ms. - Highest latency observed with 5 producers: 601.33 ms.

Message Drops: - Message drop rate (%) increased as the number of producers increased: - 1 producer: 1.24% dropped messages. - 5 producers: 17.21% dropped messages.

Resource Usage: - CPU Usage: CPU usage peaked at 89.91% with 2 producers, but fluctuated under heavier producer loads. - Memory Usage: Memory usage varied between 1354.18 MB and 2534.69 MB, with no clear linear trend observed across producer counts.

Key Insights for our project

- The number of high volume producer must match the number of brokers. If there are to many producers, the broker can get overwhelmed which results in message drop.

---

1. Consumer Experiments

Start docker with the producer profile

```
$  docker compose --profile producer up -d
```

To run the services the correct main class must be stated in the command. #### Consumer Lag & Data Loss Risks

Note: Must be run from the /exercise1/consumerTests/target subfolder.

```
$  java -jar target/pubsub-consumer-1.0-SNAPSHOT-jar-with-dependencies.jar
```

Experiment Setup

- Kafka Consumer-Producer Configuration:
  - Topic: `click-events` with a single partition.
  - Producer: Sends messages at different rates: 500, 1000, and 1500 messages/second.
  - Consumer: Simulates processing delays at intervals of 0 ms, 200 ms, 400 ms, and 600 ms.
  - Producer and Consumer run concurrently for a fixed test duration of 5 seconds.
- Testing Process:
  1. Delete all existing topics to start fresh for each experiment.
  2. Create a new topic and verify its readiness.
  3. Configure the producer to send messages at different rates.
  4. Measure consumer performance under various processing delays.
  5. Compute and aggregate metrics for each producer rate and delay combination.

- Duration:
  - Each test combination (rate x delay) runs for 5000 ms (5 seconds).

Metrics Measured See table.

Experiment Results

Messages/Sec	Processing Delay (ms)	Messages Processed	Avg Lag	Cumulative Lag
500	0	1934	0	1768
500	200	1836	0	1622
500	400	1834	0	1830
500	600	1968	1	2435
1000	0	4178	1	7167
1000	200	3671	1	5895
1000	400	3586	1	6587
1000	600	2000	4	9223
1500	0	204880	90	18590332
1500	200	163781	136	22301112
1500	400	48947	365	17906927
1500	600	2000	694	1389148

Observations

- Messages Processed:
  - At lower producer rates (500 msg/s), the consumer successfully handled most messages, even under higher processing delays.
  - At higher producer rates (1500 msg/s), the number of messages processed drastically reduced, especially with delays of 400 ms and 600 ms.
- Lag Behavior:
  - Average lag remains low for smaller producer rates (500, 1000 msg/s) but increases significantly at 1500 msg/s, especially under higher delays.
  - Cumulative lag dramatically spikes at 1500 msg/s with significant delays (e.g., 22301112 cumulative lag with 200 ms delay).
- Impact of Processing Delay:
  - At 500 msg/s and 1000 msg/s, delays up to 400 ms show minimal impact on lag and messages processed.
  - At 1500 msg/s, processing delays (400 ms and 600 ms) cause severe drops in messages processed and high lag values.

Key Insights for our project For our application, we have to keep in mind, that the production rate and consumption delay have to be balanced to guarantee stable balance between high throughput and low lag.

High producer rates, coupled with significant processing delays, overload the consumer. This indicates the need for tuning Kafka consumer properties or introducing additional partitions and consumers to handle higher workloads.

Therefore we conclude: - Scale horizontally by increasing the number of partitions and adding parallel consumers if higher producer rates or longer processing delays are necessary. - Optimize the consumer processing logic to reduce processing time per message, further minimizing the downstream lag.

---

3. Fault Tolerance & Reliability

Broker Failures & Leader Elections Start docker with the fault profile

```
$  docker compose --profile fault up -d
```

Run test from

```
../faultToleranceTest
```

```
$  mvn test -Dtest=com.experiments.KafkaFailoverDockerTest
```

Experiment Setup This experiment was conducted to observe the behavior of a Kafka cluster during a broker failover scenario. The setup consisted of a Kafka cluster with three brokers, and a topic configured with three partitions and a replication factor of three. The test involved managing graceful and ungraceful shutdowns, monitoring leader elections, and assessing the cluster's recovery performance. Key aspects of the setup include the following:

- Test Topic: `test-replication-topic`
- Producers: Three concurrent producer threads sending messages to the topic.
- Consumers: One consumer thread consuming messages from the topic.
- Failure Simulation: The Docker container `kafka1` was stopped to simulate broker failure, and later restarted for recovery.

Producer Settings (selection)

- `acks`: `all`
- `retries`: 3
- `retry.backoff.ms`: 500
- `request.timeout.ms`: 15000
- `metadata.max.age.ms`: 1000
- `linger.ms`: 5
- `max.in.flight.requests.per.connection`: 5
- `delivery.timeout.ms`: 30000
- `reconnect.backoff.ms`: 500
- `reconnect.backoff.max.ms`: 10000

Consumer Settings (selection)

- `auto.offset.reset`: `latest`
- `enable.auto.commit`: `false`
- `max.poll.interval.ms`: 60000
- `fetch.max.wait.ms`: 500
- `max.poll.records`: 500
- `session.timeout.ms`: 10000
- `heartbeat.interval.ms`: 1000
- `request.timeout.ms`: 40000
- `fetch.min.bytes`: 1
- `reconnect.backoff.ms`: 1000
- `reconnect.backoff.max.ms`: 10000

Metrics Measured

- Broker Kill Time (ms): Time taken to simulate a broker failure.
- Leader Election Time (ms): Time taken for the Kafka cluster to elect new leaders for the partitions previously managed by the failed broker.
- Broker Restart Time (ms): Time taken to restart the stopped broker and rejoin the cluster.

- Consumer Lag (Messages): The difference between the total number of messages produced and consumed throughout the duration of the experiment.
- Producer Recovery Time (ms): Time taken by producers to return to normal operation after the broker restarts.
- Consumer Recovery Time (ms): Time taken by consumers to start receiving messages after the broker is restarted.
- Total Produced Messages: Total number of messages produced during the experiment.
- Total Consumed Messages: Total number of messages consumed during the experiment.
- Partitions Revoked and Assigned: The number of partition reassignment events observed during the consumer's rebalancing process.

Experiment Results The experiment results are summarized in the following table:

Metric	Value
Test Name	Kafka Broker Failover Test
Topic Name	test-replication-topic
Broker Kill Time (ms)	10
Leader Election Time (ms)	26
Broker Restart Time (ms)	3294
Consumer Lag (Messages)	719
Producer Recovery Time (ms)	33329
Consumer Recovery Time (ms)	33329
Total Produced Messages	13574
Total Consumed Messages	12855
Test Successful	true
Last Produced Before Failure	2025-03-02 18:10:31.305
First Produced After Recovery	2025-03-02 18:11:04.634
Last Consumed Before Failure	2025-03-02 18:10:31.312
First Consumed After Recovery	2025-03-02 18:11:04.641
Revoked Partitions Count	3
Assigned Partitions Count	3

Observations

1. During the broker failover, all partitions transitioned to new leaders. The new leaders chosen for the partitions were as follows:

- Partition 0: New leader = 2
- Partition 1: New leader = 2
- Partition 2: New leader = 3

2. The Kafka producers and consumer did not resume operation until the broker (`kafka1`) was restarted. Following the successful restart of the broker:

- The first message production after recovery was captured at 2025-03-02 18:11:04.634.
- The first message consumption after recovery was captured at 2025-03-02 18:11:04.641.

3. Several disconnection and connection failure warnings were logged by the producers and consumers while attempting to reconnect to the unavailable broker (`kafka1`). Examples include:

- *`Connection to node 1 (/192.168.1.173:9092) could not be established. Node may not be available.`*
- Multiple failures were observed across the producer threads (`producer-1`, `producer-2`, `producer-3`) and the consumer thread.

4. The cluster resumed normal operations only after `kafka1` was successfully restarted, with new leaders continuing to manage partitions. This behavior indicates that while brokers transitioned leadership effectively, the client applications (producers, consumers) required the restarted broker to continue their workflows.
5. Successfully restarting `kafka1` reintroduced it to the cluster:

- Restart logs showed that the broker was started successfully.
- The broker was able to fully recover and rejoin the cluster.

Key Insights for our project This behavior emphasizes that while the Kafka cluster ensures no data loss and maintains partition leadership upon broker failover, the client applications (producers and consumers) must handle the detection of broker failures and react accordingly, if waiting for the broker is no option.