Project 2

Stream Processing System — Report

Abril Cano Castro - Emanuele Cimino - Jose Luiz

0.1 Purpose

The aim of this project is to write a stream processing system, starting from a message flow of `<key, value>` pairs.

0.2 Sensor-data producer

To begin, we implemented a simple Kafka producer that creates the initial topic (if needed) and configures the required properties.

Table 1: Kafka Producer Properties: meaning and purpose.

Property	**Meaning**	**Effect / Why Used**
Connection & Serialization		
`bootstrap.servers`	List of Kafka brokers to connect to.	The producer contacts these brokers to discover the cluster and send data.
`key.serializer`	Class to serialize the message key.	Converts the key into bytes before sending (here: `StringSerializer`).
`value.serializer`	Class to serialize the message value.	Converts the value into bytes before sending (here: `StringSerializer`).
Exactly-Once Semantics (EOS) / Reliability		
`enable.idempotence`	Enables idempotent producer (no duplicates).	Guarantees no duplicate messages even on retries.
`acks`	Number of replicas that must acknowledge a message.	`all` waits for all in-sync replicas (strongest delivery guarantee).
`retries`	Maximum number of retries on failure.	`Integer.MAX_VALUE` retries indefinitely until success.
Throughput / Performance		
`linger.ms`	Time to wait before sending a batch.	Allows messages to accumulate (20 ms) for larger batches and better throughput.
`batch.size`	Maximum batch size in bytes for one partition.	32 KB per batch → fewer network requests.

Property	Meaning	Effect / Why Used
`delivery.timeout.ms`	Total time allowed for message delivery.	120 s before giving up, including retries.
`request.timeout.ms`	Timeout for a single request to a broker.	30 s; if the broker does not respond, retry.
`transaction.timeout.ms`	Max allowed time for an open transaction.	180 s; the broker aborts if the transaction is not completed.

0.3 Sensor data consumer and output producer

Here we list the properties for the consumer of produced data and the producer of aggregated values, after the transformations performed with Akka.

Table 2: Kafka Consumer and Producer Properties used in the system.

Property	Meaning	Effect / Why Used
Consumer Properties		
`bootstrap.servers`	List of Kafka brokers to connect to.	Tells the consumer which brokers to contact to fetch data.
`group.id`	Identifier of the consumer group.	Allows multiple consumers to share the same topic load, coordinating partitions among them.
`key.deserializer` / `value.deserializer`	Classes used to deserialize key/value from bytes to objects.	Here: `StringDeserializer` converts byte[] to String.
`auto.offset.reset`	What to do if there is no committed offset.	`earliest` starts from the beginning of the topic; `latest` starts from new messages.
`enable.auto.commit`	Enables automatic offset commits by the consumer.	Disabled (`false`) to allow transactional commits together with producer output.
`isolation.level`	Controls which records are visible to consumers in case of transactions.	`read_committed` reads only messages from committed transactions (EOS support).
`max.poll.records`	Maximum number of records returned in a single poll.	Limits batch size per poll (e.g., 500).
`max.poll.interval.ms`	Maximum delay between two `poll()` calls before the consumer is considered dead.	300 s by default; prevents rebalancing if processing takes too long.
Producer Properties		
`bootstrap.servers`	List of Kafka brokers to connect to.	Producer uses these brokers to send data.
`key.serializer` / `value.serializer`	Classes to serialize key/value to byte[].	Here: `StringSerializer` converts String to byte[].

Property	Meaning	Effect / Why Used
`enable.idempotence`	Enables idempotent producer (no duplicates).	Required for exactly-once semantics and safe retries.
`acks`	Number of replicas that must acknowledge a message.	`all` = wait for all in-sync replicas (strongest durability).
`retries`	Maximum number of retries on failure.	`Integer.MAX_VALUE` = retry indefinitely.
`linger.ms`	Time to wait before sending a batch.	Accumulates more messages (20 ms) → better batching and throughput.
`batch.size`	Max batch size per partition.	32 KB → fewer network requests.
`compression.type`	Compression algorithm for batches.	`lz4` gives fast compression and lower network usage.
`delivery.timeout.ms`	Total allowed time to deliver a message.	120 s including retries, before failure.
`request.timeout.ms`	Timeout for a single request to the broker.	30 s; if exceeded, a retry is triggered.
`transaction.timeout.ms`	Max allowed time for an open transaction.	180 s; broker aborts if transaction not finished.

In fact, this class also initializes cluster sharding, which is responsible for creating and routing messages to the `SensorActorSupervisor` actors that perform the aggregate calculations. It also creates a main `AggregateSink` actor, which collects all the results and publishes the aggregated data to the appropriate Kafka topic.

0.4 Akka actors

Table 3: Overview of Akka actors used for data aggregation.

Actor	Responsibility	Notes
`AggregateSink`	Collects the results produced by all sensor actors and acts as the final aggregation point.	Publishes the aggregated data to the correct Kafka topic.
`SensorActorSupervisor`	Supervisor actor for each sensor type. Manages window state, triggers aggregation, and forwards results to the sink.	Instantiated and sharded by Akka Cluster Sharding.
`TempActor`	Processes temperature readings.	Maintains a sliding/tumbling window for temperature data and performs aggregation (e.g., average).
`HumActor`	Processes humidity readings.	Similar to `TempActor` but for humidity data.

Actor	Responsibility	Notes
`WindActor`	Processes wind readings.	Can compute average/max wind speed over time windows.
`AirActor`	Processes air quality readings.	Aggregates AQI or pollutant levels over time.
`SensorMessageExtractor`	Extracts `entityId` and `shardId` from incoming messages.	Required by Akka Cluster Sharding to route messages to the correct actor instance.

Finally, we implemented a simple consumer for the aggregated data.