Recovering a crashed consumer

Both of the consumers in our application use **XREAD** to get new messages from their streams. When using **XREAD**, the consumer needs to remember the last ID that it received in a response from Redis, then re-use that ID in subsequent calls to **XREAD** to get the next message.

As our averages consumer simply reads messages and prints data from their payloads, it only needs to store the last message ID received between calls to **XREAD**.

The aggregating consumer needs to store a few items to maintain state between calls to **XREAD**:

- The last message ID that is received.
- The name of the stream it is reading from. This is required as it is working through a set of streams, each representing a time partition of the overall dataset. The stream name changes over time.
- The current sum of all temperatures read for the hour it is working on.
- The number of messages seen for that hour.

Once it reaches the end of the hour, it then simply divides the sum of all temperatures seen for the hour by the number of messages seen to get an average value. If the consumer crashes, it needs to be able to recover state when re-starting, so that it can resume reading the stream partition it was working on in the right position with the same working values for calculating the hourly average.

Both consumers in the application use Redis to persist their state to a hash every time they read a message from a stream. This ensures that if the consumer crashes and comes back up, it can resume processing from where it left off.

Let’s try this out by restarting the consumer processes while they are working, so that we can see how they recover.

Start the consumers from the beginning of the dataset again:

```
python stream_consumers.py temps:20250101
```

You should see consumption starting from the beginning of **temps:20250101**:

```
$ python stream_consumers.py temps:20250101

agg: Starting aggregating consumer in stream temps:20250101 at message 0.

avg: Starting averages consumer in stream temps:averages at message 0.

avg: Average temperature for 2025/01/01 at 0 was 41F (3600 observations).

avg: Average temperature for 2025/01/01 at 1 was 64F (3600 observations).

avg: Average temperature for 2025/01/01 at 2 was 83F (3600 observations).
```

After the consumers have produced two to three hourly average messages, simulate a crash by stopping them with **Ctrl+C**.

Before re-starting the consumers, let's use **redis-cli** to take a look at their stored state beginning with the aggregating consumer:

```
127.0.0.1:6379> HGETALL aggregating_consumer_state

1) "current_stream_key"

2) "temps:20250101"

3) "last_message_id"

4) "1735702767000-0"

5) "current_hourly_total"

6) "61857"

7) "current_hourly_count"

8) "2368"
```

As we see, the aggregating consumer's stored stage includes the name of the stream partition it was working on, the ID of the last message that it read and the other items required to resume calculating the hourly average when execution restarts.

We can also look at the stored state for the averages consumer, which only needs to store the last message ID for the **temps:averages** stream that it reads:

```
127.0.0.1:6379> HGETALL averages_consumer_state

1) "last_message_id"

2) "1557107738024-0"
```

Next, restart the consumers, this time without specifying a stream partition name at the command line:

```
python stream_consumers.py
```

This starts the consumers from their previously saved states. You should see work continue from where it left off, in this case from message **1735702767-0** in hour 3 of the 1st of January 2025:

```
$ python stream_consumers.py

agg: Starting aggregating consumer in stream temps:20250101 at message 1735702767-0.

avg: Starting averages consumer in stream temps:averages at message 1557107738024-0.

avg: Average temperature for 2025/01/01 at 3 was 25F (3600 observations).

avg: Average temperature for 2025/01/01 at 4 was 64F (3600 observations).

...
```

Similarly, the averages consumer restarts from where it left off in the **temps:averages** stream, and won't report any results that it had already seen.

☐ Lesson

Course overview

☐ Lesson

Environment setup

Advanced consumer group management

☐ Lesson

Managing pending messages

☐ Assessment

Quiz 1 | Redis streams in production

☐ Lesson

Consumer recovery & poison-pill messages

☐ Assessment

Quiz 2 | Redis streams in production

☐ Lesson

The XAUTOCLAIM command

Performance and memory management

Redis

Cloud

Software

Pricing