CS5032: P1 WeatherVane

Word count: 1480

Overview

In this practical the task is to create an interactive weather station critical system that will replicate the process of real-time, live weather information reporting. Within Python Redis and InfluxDB will be utilised to complete this task. Redis will communicate between each of the different components: Streamer, Ingester and Processor. While InfluxDB will provide endpoints for a Flask API allowing users to query the database to retrieve varying information. Additionally, InfluxDB will be used as a means for the Ingester and Processor to store the raw data, and weather reports respectively.

Code Line Meaning	Shorthand
Streamer (streamer.py) <line_range>	(S <line_range>)
Ingester (ingester.py) <line_range>	(I <line_range>)
Processor (processor.py) <line_range>	(P <line_range>)
API (flask_test.py) <line_range>	(A <line_range>)

Design

Redis Offline:

When Redis goes down I designed my solution to utilise queues. Queues operate on a First-In-First-Out basis allowing for items queued first to leave first also. Queues also lend to ordered delivery of the data. Using this mechanism will allow for data meant to be sent to instead be added to a queue, meaning that even if Redis is offline the data is not lost and can be reconciled once Redis is back online. Queues are also a thread-safe structure, allowing for thread-safe operations to take place. Furthermore, within the Ingester I opted for a similar approach with messages to Processor for example, to notify the Processor when to begin processing the data within the InfluxDB instance.

Redis Channel Structure:

When communicating using publish and subscribe over Redis channels are used to determine the end points of communication.

Process	Channel	Pub/Sub	Function
Streamer	weather_channel:data:<batch_index>:<hour>	Pub	This channel is utilised to send the data from Streamer to Ingester, for the <batch_index> and <hour>
	weather_channel:request:*	Sub	This listens for requests to resend batches of data that could have been lost in transit to Ingester. The rest of the

			channel name contains the <batch_index> and <hour> to send to Ingester. After which the Publish channel in Streamer is utilised
Ingester	weather_channel:data:*	Sub	Listens for the streamed data from Streamer. Also includes the <batch_index> and <hour>. Data is then stored
	weather_channel: processor:<hour>	Pub	Sends notification to Processor to signal that it can begin producing the weather reports
	weather_channel:request: <batch_index>:<hour>	Pub	This is utilised to send to Streamer if any batches are missing for a particular hour
Processor	Weather_channel: processor:*	Sub	Listens and waits until receiving notification message from Ingester, signalling to begin

Architecture

Redis offline:
When Redis is offline the Streamer is made aware when attempting to send streamed data to the Ingester (S 40-57). This results in the data to be queued, waiting to be flushed which is performed periodically (S 59-81). When this occurs the Streamer sleeps periodically, and adds the data meant to be sent to a pending data queue (S 80). In a separate thread the pending data in the queue is attempts to be sent periodically (S 59-81).

When attempting to send the pending data from Streamer, if it fails the thread will sleep stopping busy-waiting occurring (S 78-81). Busy waiting in this context is the concept that after a failed attempt to send the pending data, there is no need to immediately try again as this could be a pointless operation that increases overall operational overhead and leads to the infinite checking until the connection is regained.

However, within the Ingester when requesting the missing batches for an hour, if no connection error occurs the process continues until all the requests are sent (I 141-161). It is not necessary to always halt in case of a failure, but when one does occur busy waiting should be avoided. Thus, if a request sent from Ingester to Streamer results in a Redis connection error then the process is paused (I 159-161).

Testing

Scenario	Outcome / Method to solve
Redis is down before the streaming of any data [exps/exp1.csv] *	**Streamer:** Aware of **Redis** connection loss, queues data to be sent whilst also attempting to reconnect **Ingester:** Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect (I 33-34) **Processor:** Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect In separate thread, periodically any pending data is flushed to **Ingester**. This means when the **Redis** connection is back online the queued data is sent to **Ingester**
Redis goes down in-between hours of streaming data [exps/exp1.csv] *	
Redis goes down in the middle of streaming data for an hour [exps/exp2.csv] *	**Streamer:** In exp2.csv the **Streamer** publishes the first 2 batches, after which the **Redis** connection is lost. This is recognised by the **Streamer** and the remaining data is queued (S 80). The **Streamer** attempts to reconnect periodically. The pending data thread attempts to send data periodically, once **Redis** is back online data is sent to **Ingester** **Ingester:** Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect (I 33-34) **Processor:** Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect (P 243-245)
Redis goes down when attempting to send the requests to **Streamer** for batches that are lost/missing i.e. didn't make it to **Ingester**	**Streamer**: Receives requests and sends the specified batch index and hour back to Ingester (S 94 – 113) **Ingester**: When making a request for a specific batch, if the Redis connection is down the process will pause periodically (I 157-159). Similarly, if the same batch number request is sent out twice in a row then it will pause further as for this to occur, the request must not have been published successfully (I 201-203) **Processor**: N/A
Redis goes down after **Ingester** has received all streamed data for an hour, but before notifying the **Processor** to begin performing the analytical query operations	**Streamer:** At this point the **Streamer** is waiting to send the next hour of data, at which point if **Redis** is still down it will function as shown in exp2.csv (I 57) **Ingester:** Aware of **Redis** connection loss and adds the notify **Processor** message to a pending messages queue that is flushed periodically. Once **Redis** is back online the notification message to the **Processor** will be sent (I 76-78) **Processor:** Continues to wait until the message is received. Once **Redis** is back online the message will be received, and normal operation can resume (P 243 – 245)

Redis is down over the course of multiple streaming hours	**Streamer:** The Streamer continues to queue data to be sent, and once the connection is established again it sends all the data to ensure no loss, and the process reconciles (I 34-37) **Ingester**: Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect (I 33-34) **Processor:** Aware of **Redis** connection loss, timeout for a period of time then attempts to reconnect (P 243-245)
InfluxDB goes down when **Ingester** sends cached data to database	**Streamer:** N/A Ingester: Queues data to be sent to DB, and periodically in separate thread the pending data is sent (I 80-87) Processor: N/A
InfluxDB goes down when **Processor** is attempting to query the database to generate the weather report information	**Streamer:** N/A **Ingester:** N/A **Processor:** Should queue all data to be sent to the database, when it fails and start a timeout attempting to send again, until the connection is re-established (P 223-225).
InfluxDB goes down when API is attempting to retrieve the information from database	**API:** it shows that there is no data available
No outages in anything [exps/whole_run.csv] *	**Streamer:** Streams all data for an hour to Ingester every 5 minutes (every 1 minute for ease of time) and repeats for all 24 hours. While also every periodically determining if there is any pending data in the queue to send **Ingester:** Receives all date for an hour and caches it, then sends all data to the **InfluxDB** database. Following a notification message is sent to **Processor** to communicate all data for that hour has been sent successfully to the **InfluxDB** database **Processor:** Waits to receive the notification message and once received, begins to query the database on the specific queries given in the specification. These results are stored in a different measurement of the same **InfluxDB** database

Conclusion

In conclusion, I found this practical to be an interesting insight and provided valuable experience. Initially, I found it was hard to plan out fully the architecture as there were so many scenarios with multiple moving parts needing to be dependable. However, this made for an interesting challenge to complete.

Next time, if I was to redo this practical I would make use of drawings to understand the process instead of writing out scenario tables similar to above. This could have more represented complex problems in an easier fashion. Furthermore, I would write down all possible scenarios that could happen and curate the architecture to solve those problems compared to doing this after already partially implementing multiple sections.