Final project

Big Data Systems

Gilli Fjallstein - `gwn787`, Johan Pedersen - `rlh177`

January 17, 2022

Introduction

This project aims to explore how the virtual actor model, through Orleans, can be used in distributed stream applications. To do this we have implemented a series of operator grains and used them as templates to easily implement user defined functions (UDF) for that operator. Since we benefit from the guarantees of the Orleans streams we are able to have asynchronous in-order event processing throughout the entire pipeline. With this we implement an entire topology that process events in-order and write the result to a simple log file.

We will start by explaining the data types we use throughout the streams and operators. Then we will explain how the operators are built, what kind of API they expose and what algorithm they use to compute their results. After that we will explain how we test the entire data flow through a topology as well as the tests for the individual operators. Finally we will conclude with selected ideas about how we could improve our implementation and make it more general.

Datamodel

The data model used in the implementation is, as described in the assignment text, a tuple consisting of <Key, Value, Timestamp>. The key is a string used to identify properties in keyed operators. These are for the `WindowJoin` and `WindowAggregate` operators that can aggregate on any of their properties, defined by the key. This also means that the client is fully able to specify the property used as key for aggregating or joining, but must use the exact name, including case, of the property. As the keyed property is found via reflection, trying to join on non-existent properties are allowed but will result in no tuples being output. For the non-keyed operators we simply use an empty string in the tuple.

The Value part, is the actual tuple read from the GPS, Tag and Photo files. We represent these values with the abstract `DataTuple` class, and several sub classes representing the output from each of the files, and the special case of the aggregate operator producing only an aggregated value. We create the types by simply parsing the string output by `datadriver`, and having separate constructors for each sub type.

Each of the values (user_id, photo_id, lat, long) are represented by a list of the corresponding type. This representation is chosen to be as generic as possible, as the joining

of two streams can create tuples with arbitrarily many properties, thus representing them with lists allows for any given topology, even merging the output of multiple join-streams. Tuples without all properties, such as Tags having no Long/Lat, are represented by simply having the list be empty. The timestamp element the tuple is represented by a long, due to its size.

Since our `DataTuple` type is so general we will assume that the UDFs are aware of their positions in their topology, that is to say that they know which type will come in and what to output. This allows for a very general grain where the derived UDF is responsible for implementing the specifics needed.

Operators

We expect the operators the be initialised before the data flow starts. So we make sure that the client gives them which streams to listen and output to as parameters. As such most of our operators are generic enough to be used in any part of the overall topology and the same UDF can be present in two different places without having to make a new grain.

Each operator only has a single UDF defined for it, or two when processing lists are needed. In most cases these are just trivial computations so the output is obvious, making for easier testing and serves as a proof of concept. As for the windowed operations we chose to make the aggregate tuple-based and the window join time-based to showcase that both types of windows can be supported. Sadly we were not able to make them generic enough to switch between each mode, but this should be a relatively simple extension of the current operators.

The actual implementation of the grains all follow the same model. We define a generic abstract base class of the operator type, that implements the abstract `Apply(T e)` method defined in an interface, with an already known return type. Besides this abstract function, the base class has concrete implementations for `OnActivateAsync()` that handles subscribing to a user defined stream.

The user defined functions are the implemented by implementing a concrete class with the payload type as a type parameter, and overriding the `Apply(T e)` function with the desired functionality.

Filter

The filter operator is the simplest of all operators implemented as it is stateless and does not modify the incoming tuple. Thus the operator simply applies a user defined function the return a boolean value, and if the result is false the tuple is not sent to the next operator. The only detail to notice an implementation of both operators taking both a list of tuples and a single tuple. The FlatMap and WindowJoin operators outputs a list of tuples, and Filter process these by looking at each tuple in the list and filtering the individually. When filtering a tuple that has more than one of a property, a result of merging tuples, we always look at first in the list. The implemented UDF example filters tuples based on the *user_id* being equal to or larger than 10.

FlatMap

Similar to the Filter operator, the FlatMap operator is stateless. However it is a little more involved as the map function may produce one or more tuples and thus outputs a list of tuples. Just like the Filter operator we also implements UDFs for both incoming lists and single tuples, but the output will be a flat list of tuples. The implemented UDF is a simple function that maps over the tuples and increases the longitude by 10.

WindowJoin

The WindowJoin operator is different from the former two operators as it is stateful. We need to keep two windows, one for each stream and for this we use a dictionary keyed with the *user_id* as string and the incoming tuple as value. As is it now the operator does not use reflection to get property from the key in the tuple, but could easily be extended like in the AggregateJoin. That means right know the operator is hardcoded to join on the *user_id*. With the operator we can really utilize the implementation of the payload as an object where all properties are list, as the WindowJoin operator can create many different kinds of payloads.

The grain itself is relatively simple, it assumes that the UDF know what the type of each stream is. This makes it possible for the UDF to do the processing needed for each stream in `OnNextMessage1` and `OnNextMessage2`. This is where the tuples are added to their respective dictionaries, always using the first *user_id* found in the tuple. It also keeps track of the start time of the window and if it minus the timestamp from an incoming event

exceeds an the window size then the result gets computed. The result is all the merge tuples generated from all GPS and Photo values where their dictionary keys intersect.

Currently the slide of the window is hardcoded to be the same size as the window size. Thus we have what is called a tumbling window function. This could be extended but we had to prioritise other things in this project. As we mentioned earlier the window that is returned is simply all values where the keys intersect, as such we simply clear the dictionaries afterwards. This means that we don't have the need to do the probe-and-purge method used in the slides.

WindowAggregate

The aggregate window function implemented using a tuple based sliding window, with a default window size of 6 and and default slide of 1. This is achieved by the tuple-at-a-time processing implemented by the operator, using a serially increasing integer id as key in a dictionary to store in incoming tuple. After storing the tuple we check the size of the dictionary against the size of the sliding window, and if the dictionary count is larger we remove the earliest arrived tuple. This way we ensure the size of the dictionary is never larger than the sliding window size.

After handling the incoming tuple with regards to the sliding window we can apply the provided UDF on the incoming tuple. For the aggregate function we have implemented an *Average* function that takes the average of the longitude on tuples in the sliding window with the same user id. If the incoming stream is the product of a WindowJoin stream, meaning it contains multiple values of some property, we consider each value as a separate tuple when applying the aggregate function. For the average longitude it means using the sum of the longitudes and dividing by the total number of longitudes found.

The key on which we aggregate on is specified in the string in the data model. By using reflection we can get the property specified as key and use the value to find matches in the dictionary storing tuples in the sliding window.

After applying the function we return a simple version of the *DataTuple*, the type representing the payload in tuples, containing nothing the but the aggregated value and the key on which the values were aggregated.

Topology and tests

While the streaming infrastructure fully supports out-of-order processing the individual operators have not been extended to do so. For us to implement it we would need to introduce watermarks from in the source grains as well as extend the rest of the operators to handle them. For example we would need purge the other cached stream when receiving a watermark in the window join operator. The aggregate operator would also need to wait for watermarks before being sure the computation includes all values.

In our proof of concept implementation we chose to do implement a simple topology that uses every operator and outputs to the sink grain which simply writes the result to a file. Furthermore we have sample clients to test the individual operator directly.

The query topology in our client goes as follow `Source -> WindowJoin -> FlatMap -> Filter -> Sink`. This has all three sources write to a WindowJoin operator where tuples will be joined on their `user_id`s. From this a list of tuples is sent to the FlatMap operator where we map over the list, increasing the longitudes of the tuples. This streams a list to the Filter operator which filters *user_id*s smaller that 10 and lastly the Aggregate operator takes the average of longitudes in the window, keyed on the *user_id*s. Lastly the sink grain will process each tuple received by writing it to the *Log.txt* file fount in the *Client* folder.

For testing the individual operators we have extended the *SinkGrain* be making it abstract and implementing to concrete classes, one writing to the file and one writing to console. To run these tests one must uncomment lines 26-29 in the `RunMainAsync()` function in *Client/Program.cs* to run the static functions activating the static functions executing the tests. In the code handed in only the client streaming the full topology is activated by default. As the input data files are not included in the source code, the project will not run but it expects file *GPS*, *Tag* and *Photo* in the *Client* folder.

Conclusion

Our project works well at showcasing how the virtual actor framework can be used together with stream based asynchronous event processing to implement a distributed and persistent stream query application. The implementation has all functionality asked for, and overall works well when setup correctly.

However there are many improvements that can be made. Some of the more obvious ones are for example; Generalising how our operators know which streams to subscribe and output to further, and having a more robust implementation with error handling. This would benefit the ease of creating more dynamic topologies. It should also be possible to extend the operators to support more data types and their processing in a more dynamic and generic way. But the biggest improvement would be supporting out-of-order events using watermarks. It would take a varying amount of work for the different operators, it is mostly the windowed operators that would need nontrivial implementations. These changes are very feasible and some of them were event attempted but we were not able to finish them in time.

Contributions

Both authors has contributed evenly, and has been involved in the whole project, although working on individual parts.