Big Data Systems Assignment 3

This assignment is due on Jan 17, 23:59. While individual hand-ins will be accepted, we strongly recommend that this assignment be solved in groups of maximum two students.

A well-formed solution to this assignment should include a PDF file with answers to all requirements posed along this assignment file. In addition, you must submit your code along with your written solution. Evaluation of the assignment will take both into consideration. Note that all assignments have to be submitted via Absalon in electronic format. It is your responsibility to make sure in time that the upload of your files succeeds. While it is allowed to submit scanned PDF files of your homework assignments, we strongly suggest composing the assignment using a text editor or LaTeX and creating a PDF file for submission. Email or paper submissions will not be accepted.

Learning Goals

This assignment targets the following learning goals:

- Further explore the virtual actor model by taking advantage of the guarantees provided by the actor framework Microsoft Orleans;
- Understand key concepts in distributed stream processing systems and implement a myriad of operators to fulfill a stream processing application.

Preliminaries

1. **Orleans Application.** As in previous assignments, you must build your solution based on the actor Orleans framework. Extensive documentation can be found on Orleans' website.
2. **Orleans Streams.** As in assignment 1, it is expected that you rely on *Orleans Streams* to enable asynchronous event processing across the operators of your stream processing application. In other words, the implemented operators (task of this assignment) should subscribe to defined streams in order to receive and process stream data. In the same line of reasoning, the output result of an operator should be published into a stream for downstream processing.
3. **Data Model.** In this assignment, we assume that the data model is a key-value pair associated with a timestamp. More specifically, each tuple consists of a key, a value and a timestamp indicating the event time (e.g., < key, value, timestamp >). In the case that a tuple is not keyed, a special symbol (e.g., empty string) can be used to indicate non-key tuples.
4. **Testing and Workload.** Three files (GPS, Photo, and Tag) compose the input data for this assignment (see *Dataset* below). A sample workload generator is embedded within the base code. It is expected that you make use of this sample to test your implementation and expand from it if necessary. Details including the data streams and classes involved in the workload are explained in the next blocks.

Dataset

The data set contains three data files: Photo, Like, and Tag. Their formats are described below.

1. **Photo**: Each line consists of five elements: photo_id, user_id, time, lat, long. The pair [photo_id] and [user_id] identify the photo as well as the user who posted it. Moreover, [time], [lat] and [long] stores the time and location (latitude and longitude) of the post.
2. **Like**: Each line consists of two integers: user_id, photo_id, denoting the user likes the Photo.
3. **Tag**: each line consists of two integers: photo_id, user_id, denoting the user is tagged in the photo.

Understanding the Code Base and Hints

In this programming task, we provide you with a starting code base along with a workload generator. We next explain some key classes and interfaces.

The data driver class (Client/DataDriver.cs) has a Run method with five parameters: photoStream, tagStream, gpsStream, rate and randSpan. The first three parameters are references to Photo, Tag and GPS streams respectively, while <rate> indicates the approximate number of tuples generated per second for Photo stream and <randSpan> indicates the timespan of randomization for out-of-order data streams. It is worthy to note that, once <rate> is given, the rates of the Tag and GPS streams are set correspondingly.

Some sample codes are provided in the codebase as well, including the method SampleClient (Client/Program.cs) and other classes and interfaces in the Grains and GrainInterfaces projects, respectively. You may find the following points helpful for you to complete your assignment.

Function passing: To the best of our knowledge, neither anonymous functions nor lambda functions are supported by Orleans runtime. One way to implement user-defined function is to create new grains that implement the specific function.

For example, IFilter is an interface for filter operators, and FilterGrain is an abstract grain that implements IFilter and IFilterFunction, with an unimplemented Apply method.

```
public abstract class FilterGrain<T> : Grain, IFilter, IFilterFunction<T>
{
        public abstract bool Apply(T e);
        // Implements the Process method from IFilter
        public Task Process(object e)
        {
            // If the function returns true, send the element to SinkGrain
            if (Apply((T)e))
            {
                    this.GrainFactory.GetGrain<ISink>(0,
"GrainStreamProcessing.GrainImpl.SinkGrain").Process(e);
            } // Otherwise, skip it
            return Task.CompletedTask;
        }
}
```

A new grain class can be created to extend FilterGrain and to implement the Apply method with the specific filter function.

```
public class LargerThanTenFilter : FilterGrain<long>
{
        // Implements the Apply method, filtering numbers larger than 10
        public override bool Apply(long e) {
             if (e > 10)
             {
                 return true;
             }
             else
             {
                 return false;
             }
        }
}
```

GetGrain with specific implementation : There are possibly multiple grain classes implementing the same grain interface. When getting reference to a grain object, the following method is available.

```
client.GetGrain<IFilter>(0,
"GrainStreamProcessing.GrainImpl.OddNumberFilter");
```

SourceGrain : A source grain has to be activated by calling its Init method. Thereafter, the OnActivateAsync method is invoked to subscribe to the stream. For each message arrived, the OnNextMessage method is invoked to process the message.

```
public Task Init()
{
     Console.WriteLine($"SourceGrain of stream {streamName} starts.");
     return Task.CompletedTask;
}
```

```
public override async Task OnActivateAsync()
{
     var primaryKey = this.GetPrimaryKey(out streamName);
     var streamProvider = GetStreamProvider("SMSProvider");
     var stream = streamProvider.GetStream<String>(primaryKey, streamName);

     // To resume stream in case of stream deactivation
     var subscriptionHandles = await stream.GetAllSubscriptionHandles();

     if (subscriptionHandles.Count > 0)
     {
         foreach (var subscriptionHandle in subscriptionHandles)
         {
             await subscriptionHandle.ResumeAsync(OnNextMessage);
         }
     }

     await stream.SubscribeAsync(OnNextMessage);
}

private Task OnNextMessage(string message, StreamSequenceToken
sequenceToken)
{
    Console.WriteLine($"Stream {streamName} receives: {message}.");
    //Add your logic here to process received data
    return Task.CompletedTask;
}
```

Programming Task - Implementing Operators

In this programming task, you are required to implement some building blocks of a stream processing system. Particularly, you are required to implement the following operators using the concept of Orleans Grains.

1. Source: Source operators are starting points for a stream processing application. They subscribe to input data streams and then forward the received data to the next processing operators.

2. Sink: Sink grains receive result data streams and persist them, e.g., using Kafka or a database management system.

3. FlatMap: A flatmap operator takes a single data stream and a user-defined function (UDF) as input. For each input tuple, the flatmap operator can generate one or more than one tuple, so its output should be a list of tuples.

4. Filter: A filter operator takes a single data stream and a UDF as input. If the UDF is evaluated as true, the input tuple will be outputted. Otherwise, no output will be generated.

5. WindowJoin: A window join operator takes two data streams and a window function as input, and continuously outputs joined tuples within each window instance. The two streams are joined on their keys. You can assume you have enough main memory to hold the input streams. Note that you should reason about how the timestamps of the output tuples should be set.

6. WindowAggregation: A window aggregation operator takes a single data stream, a window function and a generic aggregate function as input. For each window instance and a specific key, it outputs a value which is an aggregate of all the tuples with the same key in this stream.

Furthermore, your solution should provide a test for each operator and also a test embracing the whole dataflow, that is, composing the implemented operators into a dataflow.

Questions & Answers Regarding the Programming Task

1. What aggregate function should I use? As stated earlier, you can employ any aggregation function for testing.
2. What UDF should I implement? The last response also applies here. You can implement a UDF of your choice.
3. What is the query topology that I should rely on? You can define your own query topology. The result dataflow must embrace all the operators implemented.
4. Should I test the operators individually or as a dataflow? As stated earlier, both.
5. How can I represent the dataflow input in my solution? You should define the data structures and algorithms to handle it, since this is part of the assignment.
6. What continuous query should I set up for the tests? You are free to set up any query that matches what is expected by the assignment. Feel free to reach out to the lecturers for examples or additional comments.
7. Do I need to set up Kafka or a DBMS for sink operators? Not necessarily. You can either resort to a main memory DBMS, like SQLite, or simply a file that is durably stored in the disk.
8. Do operators implemented require out-of-order tuple processing support? No. However, we strongly suggest students to additionally provide this support.
9. Do operators implemented require distributed processing or partitioning? No. However, feel free to implement partitioning and document it appropriately in your report.

10. What type of processing should be supported? You can assume a tuple-at-time processing model for this assignment.

Instructions for Submission

In addition to the instructions presented in the beginning of this document, the following must be observed:

1) Report: The report should contain no more than 6 pages (single column, 10pt, 2cm margin on all sides). You can include some codes which are necessary for explanation in the report. Besides, your report should include the following contents:

a. Explain what APIs you have designed for each type of operator.
b. Explain what data structures and algorithms are used to implement each type of operator.
c. Briefly report how you test your implementation.
d. State the contribution of each group member.

2) Source code: A zip file containing all the source codes that you develop for this assignment. Please **DO NOT** include any other files, such as libraries, output files, Visual Studio generated files, and input dataset.

3) Single submission: Only a single submission per group is accepted. In other words, if your partner has submitted the assignment, you are free from submitting it.