CS876 Streaming Data Systems
Project Report

Transactional Semantics on AIR

Submitted by

IMT2020016 Shivankar Pilligundla

CONTENTS

1. Abstract

AIR currently doesn't support transactional semantics in the streaming context. So I have tried to come up with an implementation approach based on MorphStream. **[1]** The ultimate goal is that transactions should be processed in AIR with guaranteed ACID properties. It has multiple phases starting with **Transaction Collection** the splitting of incoming state transactions into atomic operations and constructs a Task Precedence Graph(TPG) that maps dependencies across the operations. Then **Mapping phase** where the TPG is now mapped to the AIR dataflows such that each operation is assigned a vertex in the most optimal way such that transactional semantics are ensured with minimal coordination overhead among the operations. Then **Execution Phase** where transactions are executed and coordinated using AIR's Message Passing Interface(MPI) in the required order based on the constraints and dependencies defined for each operation. Then the final **Result Phase** where computed results are written back into the memory. These phases ensure transactional semantics in AIR.

2. Problem Statement

Many emerging stream applications rely on the support of shared mutable states, where application states may be concurrently read and modified by multiple threads during stream processing. Those applications are challenging to be supported correctly and/or efficiently in today's mainstream stream processing engines (SPEs), such as Storm, Flink and Spark-Streaming. In response, transactional SPEs (TSPEs) have been recently proposed with built-in support of shared mutable states and have received much attention from academia and industry.

Different from conventional SPEs, TSPEs adopt transactional semantics during the processing of continuous data streams, where accesses to shared mutable states are modelled as state transactions. Subsequently, the concurrent process of state transactions must be scheduled to ensure some form of transactional semantics in TSPEs.

I have tried to implement this transaction semantics into AIR so that it can be used as a Transactional Stream Processing Engine. AIR is designed in C++ using Message Passing interface(MPI) and pthreads for multithreading. AIR implements a light-weight, dynamic sharding protocol (referred to as "Asynchronous Iterative Routing"), which facilitates a direct and asynchronous communication among all client nodes which avoids performance bottleneck due to communication using message passing interfaces. This is a really important functionality for TSPEs to perform efficiently. So, AIR can be effectively used as a TSPE by fine-tuning it further.

3. Approach

3.1 Transaction Collection Phase

The first step towards this is Transaction Collection phase. Given a transaction we should implement an efficient algorithm to split it into atomic operations based on state access and identify necessary dependencies among them. For now we are assuming that this step is handled out of the AIR system. So, there are external data generators that create the transactions.Let us first define types of dependencies that can exist among the operations:

- **Temporal Dependency (TD):** State accesses must follow the event sequence, leading to temporal dependencies among operations.
- **Parametric Dependency (PD):** There is a parametric dependency between two write operations when write value(v) of one operation depends on the execution of another operation.
- **Logical Dependency (LD):** To guarantee ACID, aborting one operation shall lead to aborting all operations of the same state transaction. This can be guaranteed during transaction commit by tracking a logical dependency among operations of the same transaction. Basically ensuring all or nothing property.

To identify the above mentioned dependencies among operations we can use the algorithm proposed in the Morphstream Paper. It is as follows:

Upon arrival, transactions are decomposed into atomic state access operations, which are the vertexes of the TPG, accordingly. During this decomposition, LDs can be identified among operations from the same transaction. However, TDs and PDs can not be identified immediately because the arrival of transactions may be out-of-order. To address this issue, we partition the TPG construction process into two steps during the stream processing phase and transaction processing phase, correspondingly.

To help identify TDs among operations, which may arrive out-of-order, all operations are inserted into key-partitioned sorted lists where the key is the targeting state of each operation. These lists are sorted by transaction timestamps. To help identify PDs during the next phase for each write operation
$Oi = Write(k, f(k1, k2, ...kn))$ we additionally maintain n proxy operations of Oi. Each “proxy operation” is a read operation, denoted as $(PO_i)^{kj}$, where $k_j \in k1, k2, ...kn$. inserted into $sortedLists$ of $k1, k2, ...kn$, correspondingly.

During the transaction processing phase we can identify TDs and PDs efficiently with the help of the constructed $sortedList$ and “proxy operations” during the stream processing phase. First, TDs can be identified in a straightforward way by iterating through operations inserted into each $sortedList$, i.e., $O1, O2, O5$ and $O3$, $O4$. Note that “proxy operations” are not involved in identifying TDs. Second, PDs can be identified according to the inserted “proxy operations”. In

particular, Oi is parametric dependent on the precedent write operation of "proxy operation" in the $sortedList$.

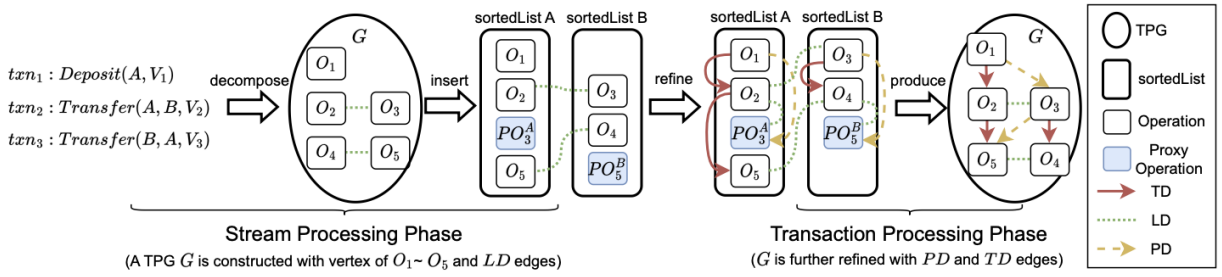
Let us take an example of following transaction to understand the algorithm better. For Bank application/Streaming Ledger which has deposit and transaction functionalities. Here 3 transactions as defined below:

- txn1: Deposit(A, v1)
- txn2: Transfer(A, B, v2)
- txn3: Transfer(B, A, v3)

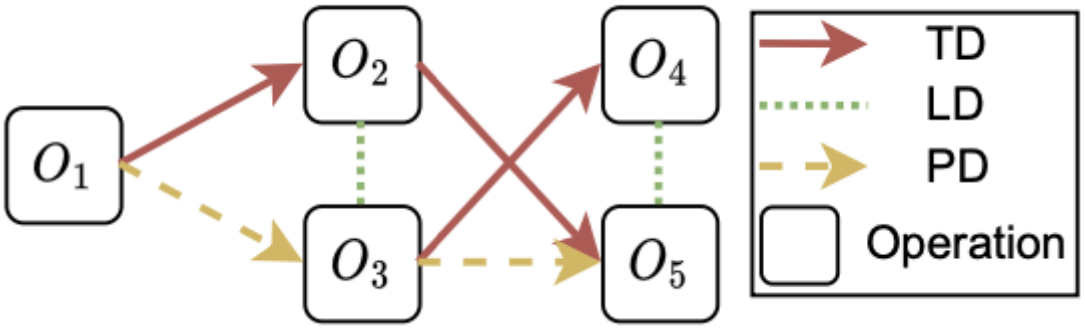
Now these transactions are split into 5 operations as shown in the below table:

TS	State Access Operation	Function (f)
1	$O_1 = Write_1(A, f_1(V_1))$	$f_1 : Read_1(A) + V_1$
2	$O_2 = Write_2(A, f_2(A, V_2))$	$f_2 : Read_2(A) - V_2 \quad \text{if} \quad Read_2(A) > V_2$
2	$O_3 = Write_2(B, f_3(B, A, V_2))$	$f_3 : Read_2(B) + V_2 \quad \text{if} \quad Read_2(A) > V_2$
3	$O_4 = Write_3(B, f_4(B, V_3))$	$f_4 : Read_3(B) - V_3 \quad \text{if} \quad Read_3(B) > V_3$
3	$O_5 = Write_3(A, f_5(A, B, V_3))$	$f_5 : Read_3(A) + V_3 \quad \text{if} \quad Read_3(B) > V_3$

The dependencies are now identified among these operations. The algorithmic workflow for this is pictorially represented below:

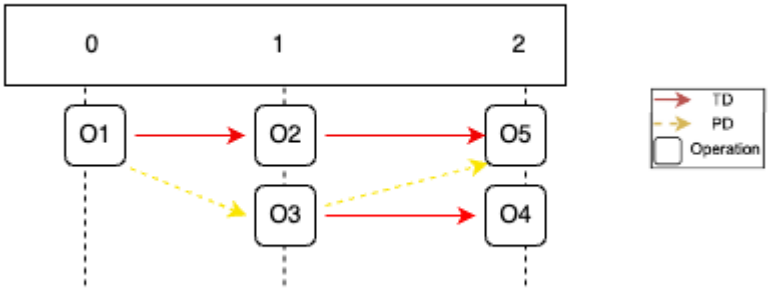


Once the dependencies are generated our next step is to create a Task Precedence Graph(TPG) which highlights the sequence of execution of operations to avoid anomalies due to their interdependencies. The below image shows the TPG for the above transaction example:



3.2 Mapping Phase

Now, this TPG should be mapped to AIR dataflows and vertices in an efficient way such that there is very minimal coordination overhead among operations. To ensure this we generate an auxiliary structure from TPG by the algorithm mentioned in **[2]**. The auxiliary structure is depicted in the below figure:



So temporal dependencies are tried to model into the same level. and multiple such levels are created with logical and parametric dependencies among different levels. Now each level is mapped to a rank/dataflow in the air. We assume that we have some fixed size grid of vertices in AIR which is big enough to accommodate the whole TPG.

AIR has some limitations due to its current implementation like it cannot send data/communicate with vertices of other rank but same tag and also it can only communicate with immediate next vertices in a rank not any arbitrary vertex which is more than one vertex away from current vertex. Keeping this into consideration auxiliary structure has to created. We can see from the auxiliary structure diagram that the dependencies can be supported by AIR.

To feed this into AIR, I have defined a fixed format for each operation as follows:

- op_id: The operation id.
- function: The processing logic of the operation
- tag: vertex tag to be mapped to
- sender_list: It is a list of pairs where is pair consists of rank, tag separated by _
- input_count: No of input events a vertex should wait for before release the operation.

Now as per the auxiliary structure there will worldsize no of files with each containing its respective operations separated by a ':' and each attribute of an operation separated by ','. The below image shows an example input_file:

```
TSPE_data > ≡ rank0.txt
  1    01,Read(A)+v1,1,[[1_1]],0:02,Read(A)-v2,2,,0:05,Read(A)+v3,3,,1
```

3.3 Execution Phase

We define 3 classes Node, Propogator, Aggregator that extend AIR Vertex. Node reads data from each file and passes it to the respective dataflow. Then each operation is executed as a part of Propogator Node where the operation logic is also implemented. Then there is an aggregator that aggregates the results and maintains a final state of the system.

Node starts with parsing the input file message and converting it to an event so that it can be serialized and sent to other vertices. The event is defined as shown in the figure below:

```
typedef struct EventNode {
    char op_id[5];
    char func[20];
    long int tag;
    long int cnt;
} EventNode;
```

Propogator has the logic as described in func. every propogator will only run its part of the operation in the message. so it will only run events that have tag matching its vertex tag. If any state modifications happen then it has to be incorporated into event for further transmission(more work has to be done here). There can be multiple approaches like sending the updated states over the messaging interface as events or allowing each vertex to maintain a statemap with timestamps generated by a synchronized clock(these timestamps will help for implementing rollback strategies later). Then propogator will wait for cnt no of events to hit its vertex until it completes its execution. After this it will access its sender list that is configured for each vertex and send the required events to its dependent propogators so they can start execution. Then a final Aggregator can be implemented which will aggregate and collect all the modified states and store them accordingly.

4. Evaluation & Results:

The implementation is ran on a basic TPG with 5 operations. O1, O2, O5 in rank 1 and O3, O4 rank 2. There is a dependency from O4 in rank 2 to O5 in rank1. The corresponding input files for Nodes are as follows:

```
TSPE_data > ≡ rank0.txt
  1    O1,Read(A)+v1,1,,0:O2,Read(A)-v2,2,,0:O5,Read(A)+v3,3,,1
```

```
TSPE_data > ≡ rank1.txt
  1    O3,Read(B)+v2,1,,0:O4,Read(B)-v3,2,[[0_3]],0
```

The mapped operations are printed out to the console as follows:

```
~/AIR/Release master !13 ?8 > make all
[  2%] Building CXX object CMakeFiles/AIR.dir/src/TSPE/Propogator.o
[  4%] Linking CXX executable AIR
[100%] Built target AIR
~/AIR/Release master !13 ?8 > mpirun -np 2 ./AIR TSPE

******************AIR (c) 2020 Uni.lu******************

AIR INSTANCE AT RANK 1/2 | MSG/SEC/RANK: 1 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 2/2 | MSG/SEC/RANK: 1 | AGGR_WINDOW: 10000ms
RANK: 0  TAG: 1
op_id:O1        function: Read(A)+v1     tag: 1  dependencies: 0
RANK: 1  TAG: 1
op_id:O3        function: Read(B)+v2     tag: 1  dependencies: 0
RANK: 0  TAG: 2
op_id:O2        function: Read(A)-v2     tag: 2  dependencies: 0
RANK: 1  TAG: 2
op_id:O4        function: Read(B)-v3     tag: 2  dependencies: 0
RANK: 0  TAG: 3
op_id:O5        function: Read(A)+v3     tag: 3  dependencies: 1
```

5. References

[1] Yancan Mao, Jianjun Zhao, Shuhao Zhang, Haikun Liu, and Volker Markl. 2023. MorphStream: Adaptive Scheduling for Scalable Transactional Stream Processing on Multicores. Proc. ACM Manag. Data. 1, 1, Article 59 (May 2023), 26 pages. https://doi.org/10.1145/3588913.
[2] Nikola S Nikolov and Alexandre Tarassov. 2006. Graph layering by promotion of nodes. Discrete Applied Mathematics 154, 5 (2006), 848–860.
[3] Asynchronous Iterative Routing (AIR). https://github.com/bda-uni-lu/AIR